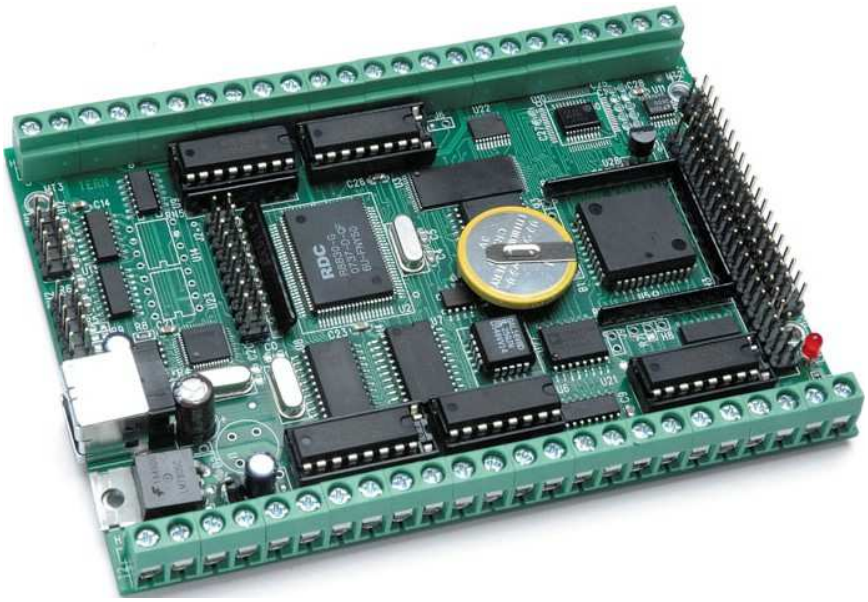
TDU ™

40 MHz Controller based on the 16-bit 188ES
with



70+ TTL and high voltage I/Os, USB, ADC, and DAC

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

TDU, NT-Kit, and ACTF are trademarks of TERN, Inc.
Am188ES and Am186ES are trademarks of Advanced Micro Devices, Inc.
Paradigm C/C++ is a trademark of Paradigm Systems.
Microsoft, Windows 95/98/2000/ME/NT/XP are trademarks of Microsoft Corporation.

Version 1.01

February 14, 2013

© 1998-2012 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 ***TinyDrive-USB™ (TDU)*** is designed for industrial control applications that require ADC, DAC, solenoid drivers and protected high-voltage inputs.

The ***TDU*** is C/C++ programmable with a 16-bit, 40 MHz CPU(x188). It supports 35 high-voltage I/O lines, 24 TTL I/O pins, 11 ch. 12-bit ADC(P2543) or 8 ch. 16-bit +-10V ADC (AD7606), an 8 ch. 16-bit DAC, a high speed USB port, 3 RS-232/RS-485, a real-time clock, battery, watchdog timer, PWM, 3 timer/counters, 512-byte EEPROM, up to 512KB SRAM, and 512KB ROM/Flash.

A true bipolar, simultaneous sampling ADC(AD7606/7607, 16/14-bit) can be installed. It can accept ±10V or ±5V true bipolar analog signals while sampling at throughput rates up to 100+ kSPS for all 8 analog inputs. Each analog input contains antialising filter, sample-and-hold and clamp protection tolerant up to ±16.5V. With 1M ohm analog input impedance, a 7000V ESD rating, and sustaining up to ±10 mA input current, the analog inputs are designed to survive in a rough industrial environment. A serial ADC(P2543, 11 ch., 12-bit, 0-5V, 10KHz) can be installed to replace the AD7606.

The 35 high-voltage I/O lines, routed to screw terminals, include 7 inputs, 14 outputs, and 14 hardware-configurable inputs or outputs. The inputs can take up to 35V DC. The outputs are capable of sinking 350mA at 50V per line, and they can drive solenoids or relays. A real-time clock (RTC72421) provides calendar information. Two DMA-driven serial ports support RS-232 communication, at up to 115,200 baud. The optional third UART SCC2691 can be configured as either RS-232 or RS-485, supporting 8-bit or 9-bit RS-485 networking.

An USB 1.1/2.0 slave USB-B port can be installed with a USB stack chip (FT232H, FTDI). No USB specific firmware programming is required. Data transfer rate can go up to 4 MB/sec with D2xx USB software driver.
A 82C55 chip provides 24 TTL I/Os. A supervisor chip (691) with a watchdog is on-board. An 8 ch. 16-bit DAC(LTC2600) can provide 0-5V analog voltage outputs. Precision reference can be installed for the DAC and the ADC(P2543).

A switching regulator can be installed to supports power-off mode, allowing µA-level power consumption.

The TDU supports TERN 8-bit expansion boards, including ACE, or UR8. An optional TD-Pack including a 16x2 LCD, 8x2 keys is available. The 48 screw terminals can be replaced with pin headers for OEM applications.

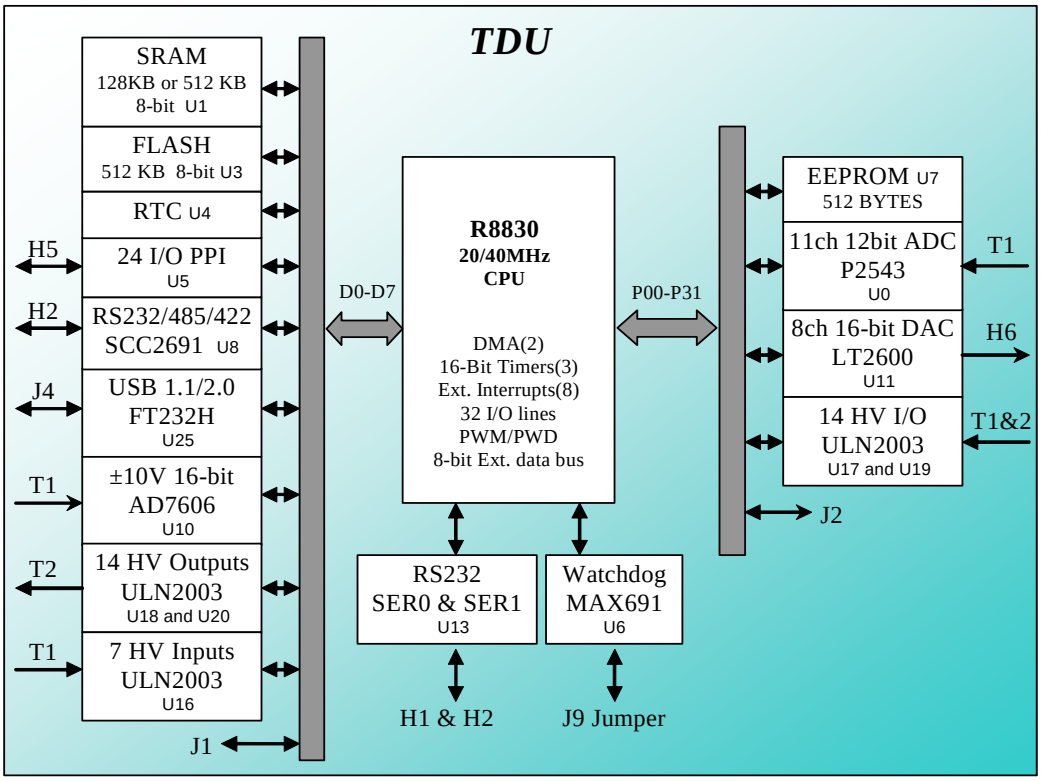


Figure 1.1 Functional block diagram of the TDU

1.2 Features

Standard Features

- Dimensions: 4.8 x 3.4 x 0.5 inches
- Power input: +9V to +12V unregulated DC with standard linear regulator
- 16-bit CPU (188ES), Intel 80x86 compatible
- 128KB SRAM, 512KB Flash
- 512-byte serial EEPROM
- 2 high-speed PWM outputs and Pulse Width Demodulation
- 32 I/O lines from Am188ES
- 6 external interrupt inputs, 3 16-bit timer/counters
- 2 serial ports from the 188ES support 8-bit or 9-bit asynchronous communication
- On-board +5V regulator
- Supervisor chip (691) for power failure, reset and watchdog
- 24 bi-directional I/O lines from 82C55
- 35 high-voltage I/Os, all routed to screw terminals
- Interface for LCD, keypad

Optional Features:

- Power input: +9V to +30V unregulated DC with optional switching regulator
- 512KB SRAM
- 8 ch. 16-bit parallel ±10V ADC (AD7606) or 8 ch. 14-bit parallel ±10V (AD7607) or 11ch. 12-bit serial 0-5V ADC (P2543).
- 8 ch. 16-bit DAC (LTC2600)
- SCC2691 UART (on-board) supports 8-bit or 9-bit networking
  UART comes with RS232 (default), 485 or 422 drivers
- USB 1.1/2.0 (FT232H)
- Real-time clock RTC72423 and lithium coin battery
- Precision reference, 20 PPM/°C, 2.5/4.0/5V
- TD-Pack box, 16x2 LCD and 8x2 keypad

1.3 Physical Description

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

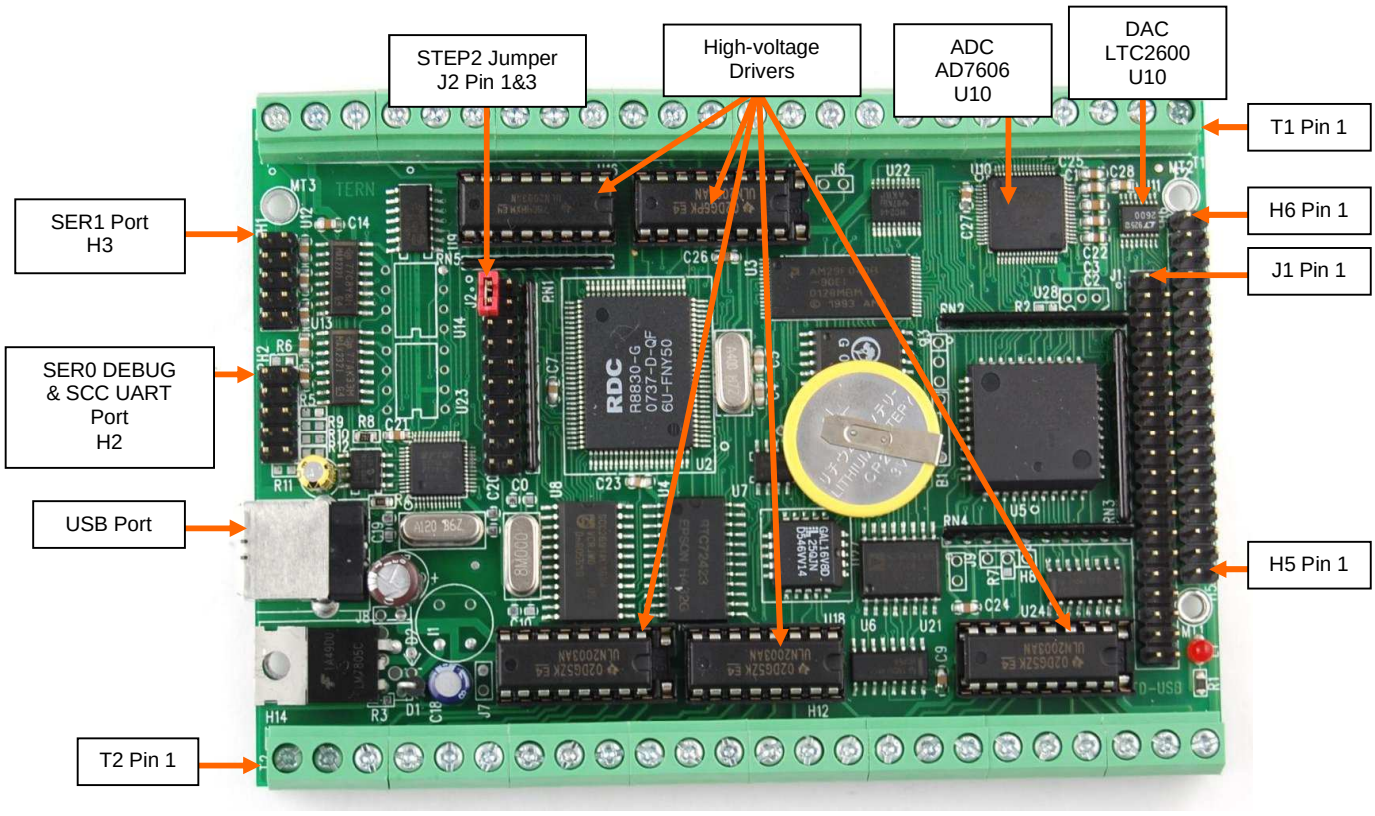


Figure 1.2 Physical layout of the TDU

1.4 TDU Programming Overview

An "ACTF Boot Loader" resides in the top protected sector of the 512KB on-board flash chip (29F040). At power-on/reset, the ACTF Utility will check the STEP 2 jumper (J2 pins 1&3). If the STEP 2 jumper is installed, the "jump address" located in the on-board serial EEPROM will be read and the CPU will jump to that address for immediate execution. A DEBUG kernel (*already pre-programmed at the factory*) can be downloaded and programmed into the flash starting at address 0xE0000. Using the ACTF Utility, the "GE0000 <enter>" command will set the jump address to 0xE0000. The command will also run the DEBUG kernel, preparing the TDU for communication with the Paradigm C/C++ IDE for downloading and debugging applications. The following diagrams show the procedure for programming the TDU. Steps include preparing the TDU for debugging, debugging the TDU, standalone field test, and production.

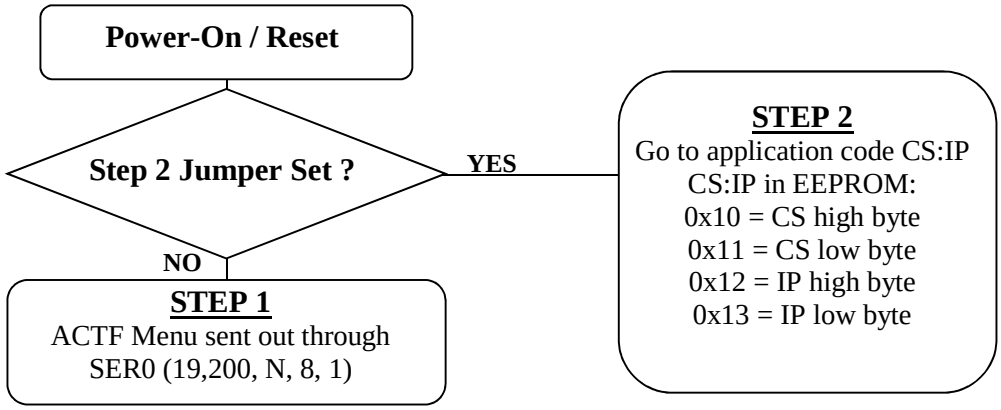


Figure 1.2 Flow Chart of ACTF Operation

By default, the DEBUG kernel has been loaded into the ACTF flash at the factory for your convenience. You may proceed directly to STEP 1: Debugging.

Preparation for Debugging:

This had already been done at the factory! You may proceed to STEP 1: Debugging. This step is only required if you have completed STEP 3 and would like to return to STEP 1.

● Connect the TDU (SER0, H2) to PC (COMx) via serial debug cable provided with the EV-P/DV-P. Using the Windows "Hyper Terminal", create a serial link based on 19,200, 8 bits, 1 stop, no parity.
● Power on the TDU *WITHOUT* the STEP 2 jumper installed (J2 pins 1 & 3). The ACTF text MENU should be sent out via serial port to "Hyper Terminal".
● Use the "D <enter>" command to initiate download. Select Transfer -> Send File, and select \tern\186\rom\lo_ee512.hex. Use the "G04000 <enter>" command to execute this script.
● Select Transfer -> Send File to select \tern\186\rom\af_0_115.hex. This is the debug kernel. Use the "GE0000 <enter>" command to set jump address and execute the debug kernel. The LED will blink twice and remain on.
● Set the STEP 2 jumper (J2 pins 1 & 3). The TDU is now ready to communicate with the Paradigm C/C++ IDE for debugging and application development.

Step 1: Debugging:

● Launch the Paradigm C/C++ IDE. Select File -> Open. Chose the project file \tern\186\samples\TDU\tdu.pdl.
● Use samples within the "tdu.pdl" project to create application. Download, run, and debug application.

Step 2: Standalone Field Test:

● After completing STEP 1, by default, your application resides in the battery-backed SRAM starting at address 0x08000.
● Remove STEP 2 jumper and setup Hyper Terminal link with the TDU. (Open Windows "Hyper Terminal" program. Set for 19,200, 8 bits, 1 stop, no parity).
● At power-on, ACTF menu will be sent to Hyper Terminal. Use the "G08000 <enter>" command to execute application. Set STEP 2 jumper (H2 pins 1&2). At every power-on/reset, application at 0x08000 will execute.
● Complete STANDALONE FIELD TEST. If return to STEP 1 is required, remove STEP 2 jumper and use the "GE0000 <enter>" command to run debug kernel to prepare to setup for communication with Paradigm C/C++ IDE.

Step 3: Production:

The DV-P Kit is required for this step.
If you do not have the DV-P Kit, visit http://tern.com/devkit.htm for upgrade details.

● Refer to the ACTF technical manual, found in the \tern_docs\manuals directory. Here you will find details on generating an ACTF downloadable HEX file based your application.
● Remove the STEP 2 jumper and create serial link using Hyper Terminal (19,200, N, 8, 1). At power-on/reset, you will see the ACTF menu at Hyper Terminal. Use the "D <enter>" command to initiate download process. Select Transfers -> Send File, and select \tern\186\rom\re\l_29f40r.hex.
● This file will erase the flash and prepare the flash to accept ACTF downloadable application HEX file. Use
the "G04000 <enter>" command to run script. Flash will be ready for application.
● Select Transfer -> Send File to select your ACTF downloadable application HEX file. Upon completion, use the "G80000 <enter>" command to execute application. This command also sets the jump address to point your application in flash. Set STEP 2 jumper (J2 pins 1&3). At power-on/reset application will execute.

There is no ROM socket on the TDU. The user's application program must reside in the SRAM (starting at address of 0x08000 by default based on \tern\186\config\186.cfg) for debugging in STEP 1, reside in the battery-backed SRAM for standalone field testing in STEP 2, and finally be programmed into the on-board 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. From the ACTF Utility, use the command "G80000 <enter>" to point to the user's application code in the flash. The STEP 2 jumper must be installed for every production-version board.

1.5 Minimum Requirements for TDU System Development

1.5.1 Minimum Hardware Requirements

- PC or PC-compatible computer with serial COMx port that supports 115,200 baud
- TDU controller
- Serial debug cable (RS232; DB9 connector for PC COM port and IDC 2x5 connector for controller)
- Center negative wall transformer (+9V 500 mA)

1.5.2 Minimum Software Requirements

- TERN Installation CD – EV-P or DV-P
- PC software environment: Windows 95/98/2000/ME/NT/XP/VISTA/7

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 Kit, you can program and debug the TDU in Step One and Step Two, but you cannot run Step Three. In order to generate an application ROM/Flash file, make production version ROMs, and complete the project, you will need the Development Kit (DV-P).

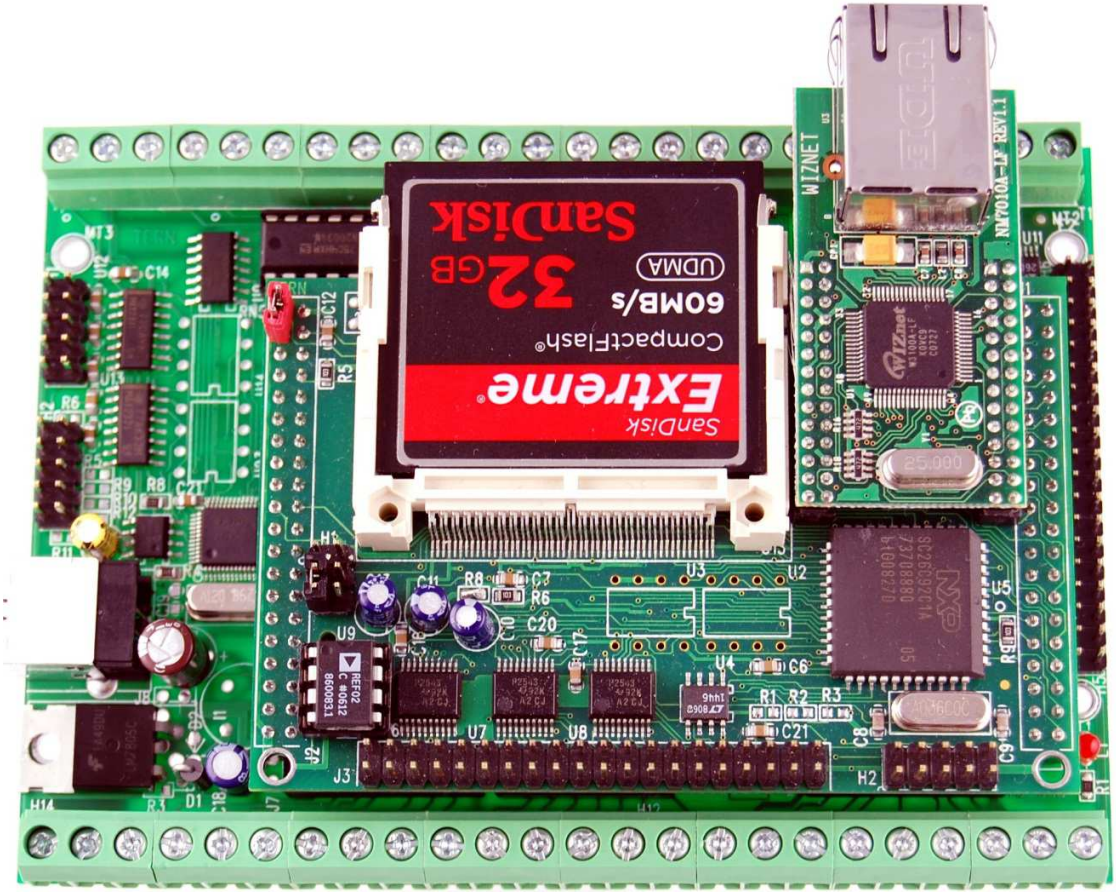


Figure 1.3 TDU with ACE + CompactFlash + Ethernet + 33 channels of 12-bit ADC

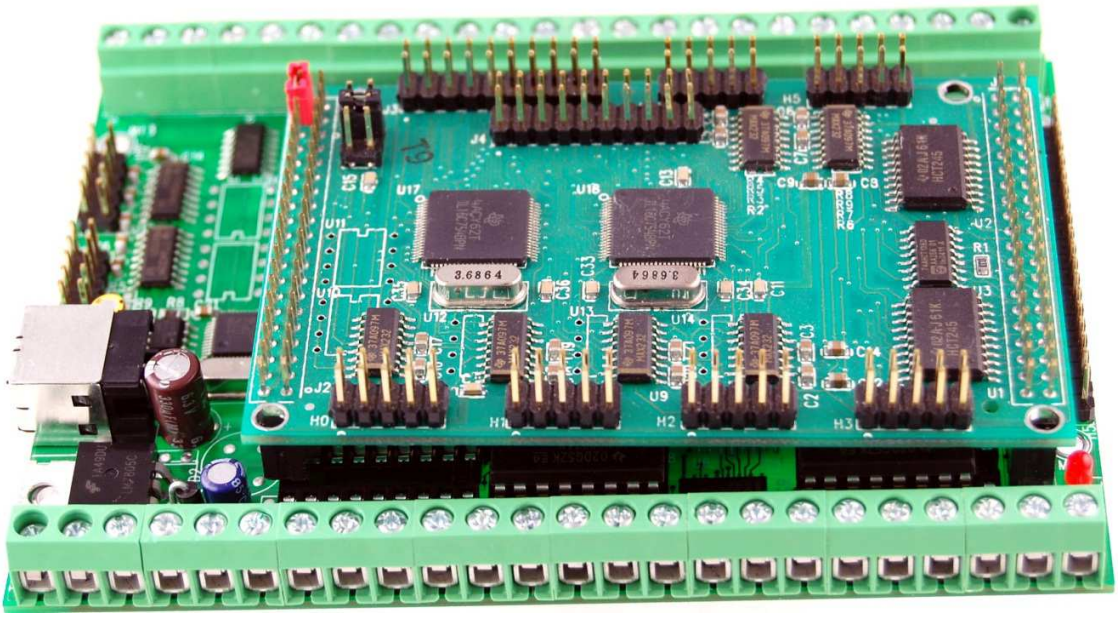


Figure 1.4 TDU with UR8 + 8 UART Ports

Chapter 2: Installation

2.1 Software Installation

Please refer to the "development kit pro.pdf" technical manual on the TERN installation CD, under tern_docs\manual\ development kit pro.pdf, for information on installing software.

2.2 Hardware Installation

Hardware installation consists primarily of connecting the microcontroller to your PC and powering on the device.

Overview

- Connect PC-IDE serial cable:
  For debugging (STEP 1), place IDE connector on SER0 (H2) with red edge of cable at pin 1. This DEBUG cable is a 10-pin IDE to DB9 cable, made by TERN (See Appendix D).
- 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

2.2.1 Connecting to the PC

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

The controller communicates through SER0 by default. Install the 5x2 IDE connector on the SER0 5x2 pin header. ***IMPORTANT:*** *Note that the* ***red*** *side of the cable must point to pin 1 of the SER0 header*. The DB9 connector should be connected to one of your PC's COM Ports (COM1 or COM2).

2.2.2 Powering-on the TDU

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 0xE0000. (Default setting in factory)
3) The EEPROM is set to jump address of 0xE0000. (Default setting in factory)

Connect +9-12V DC to the DC power terminal. The DC 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.

2.2.3 Connecting the TDU

The proper connections required to debug the board (through Paradigm software). J2 (SER 0) is a 5x2 pin header on the VE232.

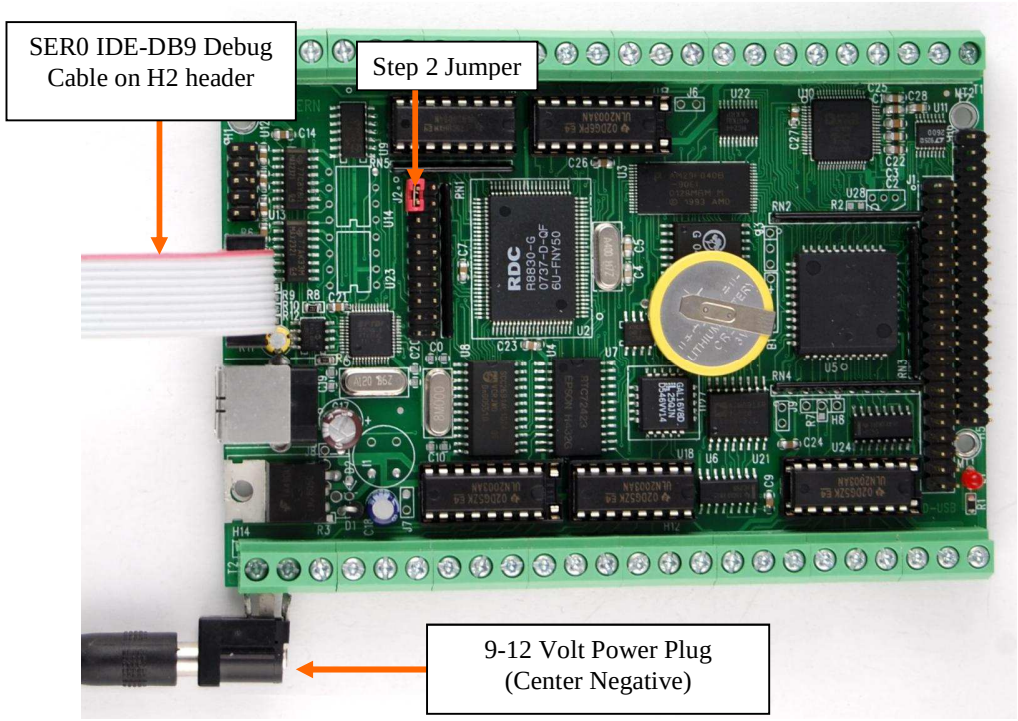


Figure 2.1: TDU with Debug Cable (SER0), Power Plug, and Step 2 Jumper shown

NOTE: Remember to watch for the "**double blink**" of the LED. This indicates the **Debug Kernel** has been loaded with the **jump address** pointing to it. This is mandatory to commence downloading code through the Paradigm environment.

Chapter 3: Hardware

3.1 188ES - Introduction

The 188ES is based on industry-standard x86 architecture. The 188ES controllers are higher-performance, more integrated versions of the 80C188 microprocessors. In addition, the 188ES has new peripherals. The on-chip system interface logic can minimize total system cost. The 188ES 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 188ES – Features

3.2.1 Clock

Due to its integrated clock generation circuitry, the 188ES 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 J8 pin 1 (named CLK), default 40 MHz. The CLKOUTB signal is routed to J8 pin 2.

CLKOUTA remains active during reset and bus hold conditions. The TDU 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 23.

3.2.2 External Interrupts and Schmitt Trigger Input Buffer

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

/INT0, used by SCC2691 UART
/INT1, used by AD7606
/INT2, J2 pin 20, available for use
/INT3, used by AD7606
/INT4, J2 pin 8, available for use
INT5 = P12, used as output for LED/EE/HWD
INT6 = P13, used by FT232H USB
/NMI = J2.17

Six external interrupt inputs, /INT0-4 and /NMI, are buffered by Schmitt-trigger inverters (U9), 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.

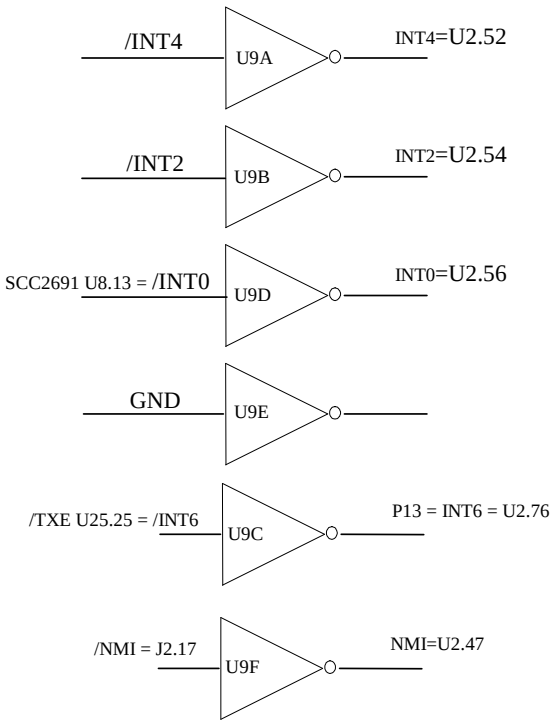


Figure 3.1 External interrupt inputs with Schmitt-trigger inverters

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

3.2.3 Asynchronous Serial Ports

The 188ES CPU has two asynchronous serial channels: SER0 and SER1. Both asynchronous serial ports support the following:

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

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

The optional external SCC2691 UART is located in the U8 socket. For more information about the external UART SCC2691, please refer the data sheet on the TERN CD, under tern_docs\parts\scc2691.pdf.

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 = ULN2003 U17 pin 16 > T1 pin 10 (IO2=U17 pin 3)
Timer0 input = P11 = ADC U0 pin 16 & EE U7 pin 5
Timer1 output = P1 = J2.12 = ULN2003 U17 pin 17 > T1 terminal 9 (IO1=U17 pin 2)
Timer1 input = P0 = ULN2003 U17 pin 15 > T1 terminal 11 (IO3=U17 pin 4)

These two timers can be used to count or time external events, or they can generate non-repetitive or variable-duty-cycle waveforms. A 10 K pullup resistor is required for P0 used as Timer1 input.

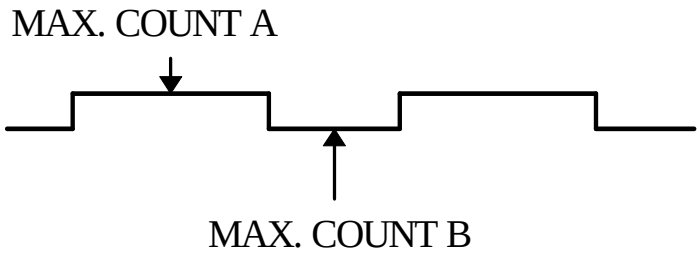
Timer2 is not connected to any external pin. It can be used as an internal timer for real-time coding or time-delay applications. It can also prescale timer 0 and timer 1 or be used as a DMA request source.

The maximum rate at which each timer can operate is 10 MHz, since each timer is serviced once every fourth clock cycle. Timer output takes up to six clock cycles to respond to clock or gate events. See the sample programs ***timer0.c*** and ***ae_cnt0.c*** in the **`tern\186\samples\ae`** directory.

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 secondary maximum count registers for variable duty cycle output. Using both the primary and secondary maximum count registers lets the timer alternate between two maximum values.



Pulse Width Demodulation can be used to measure the input signal's high and low phases on the /INT2=J2 pin 20. See Section 8.2 of the Am188ES User's Manual on the TERN CD, under the amd_docs\am186es directory.

3.2.6 Power-save Mode

The TDU is an ideal module for low power consumption applications. The power-save mode of the 188ES 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 TDU has a VOFF signal routed to J8 pin 1. 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 1/64 second, 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. More details are available in the sample file ***poweroff.c*** in the **`tern\186\samples\ae`** sub-directory.

3.3 188ES PIO lines

The 188ES 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***	***TDU Pin No.***	***TDU Initial***
P0	Timer1 in	Input with pull-up	T1.11=IO3 (U17.15)	Input with pull-up
P1	Timer1 out	Input with pull-down	T1.9=IO1 (U17.17)	Input with pull-down
P2	/PCS6/A2	Input with pull-up	U4 pin 2	RTC Select
P3	/PCS5/A1	Input with pull-up	U8 pin 14	SCC Select
P4	DT/R	Normal	J2 pin 3	Input with pull-up (STEP2)
P5	/DEN/DS	Normal	T1.12=IO4 (U17.14)	Input with pull-up
P6	SRDY	Normal	T1.13=IO5 (U17.13)	Input with pull-down
P7	A17	Normal	U3 pin 6	A17
P8	A18	Normal	U3 pin 9	A18
P9	A19	Normal	T2.3=O21 (U19.8)	A19
P10	Timer0 out	Input with pull-down	J2 pin 12	Input with pull-down
P11	Timer0 in	Input with pull-up	U7 EE pin 5	Input with pull-up
P12	DRQ0/INT5	Input with pull-up	U7 EE pin 6	Output for LED/EE/HWD
P13	DRQ1/INT6	Input with pull-up	U9 pin 10 /TXE	Input with pull-up (USB)
P14	/MCS0	Input with pull-up	T1.14=IO6 (U17.12)	Input with pull-up
P15	/MCS1	Input with pull-up	T1.15=IO7 (U17.11)	Input with pull-up
P16	/PCS0	Input with pull-up	J1 pin 19	/PCS0
P17	/PCS1	Input with pull-up	U27 pin 1	PAL Select
P18	CTS1/PCS2	Input with pull-up	J2 pin 19	Input with pull-up
P19	RTS1/PCS3	Input with pull-up	T2.9=O15 (U19.2)	Input with pull-up
P20	RTS0	Input with pull-up	U11 pin 7	DAC Select
P21	CTS0	Input with pull-up	J2 pin 5	Input with pull-up
P22	TxD0	Input with pull-up	H2 pin 3 /TXD0	TxD0
P23	RxD0	Input with pull-up	H2 pin 5 /RXD0	RxD0
P24	/MCS2	Input with pull-up	T2.7=O17 (U19.4)	Input with pull-up
P25	/MCS3	Input with pull-up	T2.6=O18 (U19.5)	Input with pull-up
P26	UZI	Input with pull-up	T2.5=O19 (U19.6)	Input with pull-up*
P27	TxD1	Input with pull-up	H1 pin 3 /TXD1	TxD1
P28	RxD1	Input with pull-up	H1 pin 5 /RXD1	RxD1
P29	/CLKDIV2	Input with pull-up	T2.4=O20 (U19.7)	Input with pull-up*
P30	INT4	Input with pull-up	J2 pin 8 /INT4	Input with pull-up (ET)
P31	INT2	Input with pull-up	J2 pin 20 /INT2	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

Four external interrupt lines are not shared with PIO pins:

INT0 = used by SCC2691 UART
INT1 = used by AD7606
INT3 = used by AD7606
/NMI = J2.17

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

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

```
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 C function in the library **ae_lib** can be used to initial 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,

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

Signal	Pin	Function
P0	Timer1 in	U17.15 high voltage driver
P1	Timer1 out	U17.17 high voltage driver
P2	/PCS6	U4 RTC72423 chip
P3	/PCS5	U8 SCC2691 UART chip select
P4	/DT	STEP 2 Jumper
P5	/DEN/DS	U17.14 high voltage driver
P6	SRDY	U17.13 high voltage driver
P7	A17	Address line 17 for SRAM and Flash
P8	A18	Address line 18 for SRAM and Flash
P9	A19	U19.8 high voltage driver input
P10	Timer0 out	U17.16 high voltage driver
P11	Timer0 in	Shared with U0 P2543 ADC and U7 24C04 EE data input *The ADC and EE data output can be tri-state, while disabled*
P12	DRQ0/INT5	LED, U7 serial EE clock, Hit watchdog and DAC LTC2600
P13	DRQ0/INT5	/TXE pin on USB FT232H U9
P14	/MCS0	U17.12 high voltage driver
P15	/MCS1	U17.11 high voltage driver
P17	/PCS1	PAL U27 chip select
P19	RTS1/PCS3	U19.2 high voltage driver input
P20	RTS0	DAC LT2600 U11 chip select
P22	TxD0	Default SER0 debug
P23	RxD0	Default SER0 debug
P24	/MCS2	U19.4 high voltage driver input
P25	/MCS3	U19.5 high voltage driver input
P26	UZI	U19.6 high voltage driver input
P27	TxD1	SER1 RS232 port
P28	RxD1	SER1 RS232 port
P29	/CLKDIV2	U19.7 high voltage driver input

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 to 15. The system clock is 25 ns (or 50 ns), giving a clock speed of 40 MHz (or 20 MHz). Details regarding this can be found in the Software chapter, and in the Am188ES 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 Am188ES 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	Available
0x0100-0x01ff	/PCS1	PAL U27	PPI, ADC, HV
• 0x0100-0x010f	/PPI	U5 pin 7	PPI
• 0x0110-0x011f: Read	/I	U22 pin 1 & 19	74HC244
• 0x0110-0x011f: Write	/V	U24 pin 14	74HC259
• 0x0120-0x012f: Read	/CV	U10 pin 9&10	AD7606 AD converstion
• 0x0120-0x012f: Write	/L	U21 pin 14	74HC259
• 0x0130-0x013f	/AD	U10 pin 13	AD7606 chip select
• 0x0140-0x014f: Read	/RDU	U25 pin 26	FT232H USB read
• 0x0140-0x014f: Write	WRU	U25 pin 27	FT232H USB write
0x0200-0x02ff	/PCS2	J2 pin 19 = CTS1	Available
0x0300-0x03ff	/PCS3	J2 pin 10 = RTS1	Available (shared w/U19)
0x0400-0x04ff	/PCS4		Reserved
0x0500-0x05ff	/PCS5	U8 pin 14	UART, SCC2691
0x0600-0x06ff	/PCS6	U4 pin 2	RTC 72423

3.4.2 74HC259

The 74HC259 8-bit decoder latch provides eight additional output lines for the TDU. The U21 74HC259 is mapped in the I/O address space 0x0120. The U24 74HC259 is mapped in the I/O address space 0x0110. You may access this device by using the following code. The output of U21 drives the high voltage driver U18. The output of U24 drives the high voltage driver U20. For pin locations and details refer to the schematic at the end of this manual.

```
outportb(0x0120 + i, val); // U21  i = output pin, val = 0/1 to set or reset latch.
outportb(0x0110 + i, val); // U24  i = output pin, val = 0/1 to set or reset latch.
```

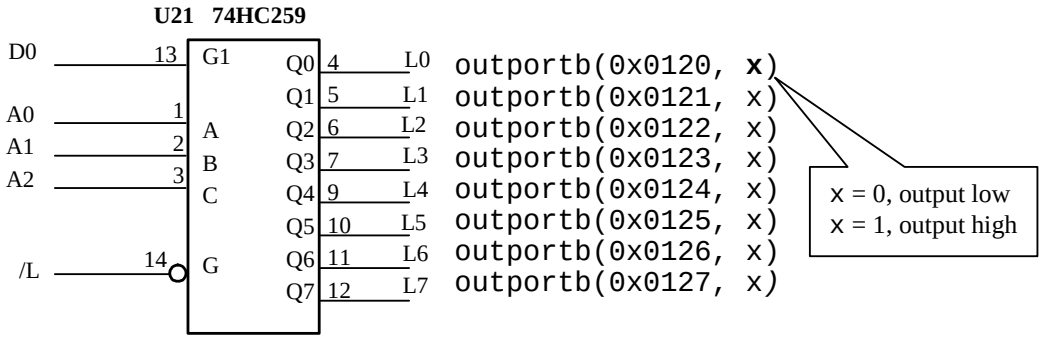


Figure 3.2 74HC259 diagram with corresponding outport addresses

3.4.3 Programmable Peripheral Interface (82C55A)

U5 PPI (82C55) is a low-power CMOS programmable parallel interface unit for use in microcomputer systems. It provides 24 I/O pins that may be individually programmed in two groups of 12 and used in three major modes of operation.

In MODE 0, the two groups of 12 pins can be programmed in sets of 4 and 8 pins to be inputs or outputs. In MODE 1, each of the two groups of 12 pins can be programmed to have 8 lines of input or output. Of the 4 remaining pins, 3 are used for handshaking and interrupt control signals. MODE 2 is a strobed bi-directional bus configuration.

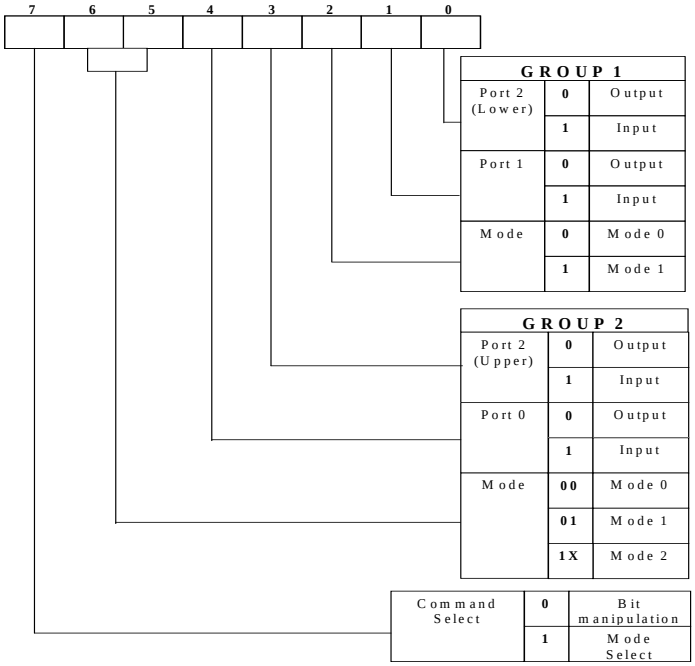


Figure 3.3 Mode Select Command Word

The TDU maps U5, the 82C55/uPD71055, at base I/O address 0x0100.

The Command Register = 0x0103; Port 0 = 0x0100; Port 1 = 0x0101; and Port 2 = 0x0102.

The following code example will set all ports to output mode:

```
outportb(0x0103,0x80); /* Mode 0 all output selection. */
outportb(0x0100,0x55); /* Sets port 0 to alternating high/low I/O pins. */
outportb(0x0101,0x55); /* Sets port 1 to alternating high/low I/O pins. */
outportb(0x0102,0x55); /* Sets port 2 to alternating high/low I/O pins. */
```

To set all ports to input mode:

```
outportb(0x0103,0x9f);     /* Mode 0 all input selection. */
```

You may read the ports with:

```
inportb(0x0100); /* Port 0 */
inportb(0x0101); /* Port 1 */
inportb(0x0102); /* Port 2 */
```

This returns an 8-bit value for each port, with each bit corresponding to the appropriate line on the port.

3.4.4 Real-time Clock RTC72423

If installed, the 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()*** . Refer to the data sheet on the TERN installation CD under tern_docs\parts\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**.

3.4.5 UART SCC2691

The UART SCC2691 (Signetics, U8) is mapped into the I/O address space at 0x0500. 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 the data sheet on the TERN CD, tern_docs\parts\scc2691.pdf. The SCC2691 on the TDU may be used as a network 9-bit.

3.5 Other Devices

A number of other devices are also available on the TDU. 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 TDU 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 TDU. 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 /RST. This automatic assertion of /RST may recover the application program if something is wrong. After the TDU 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 188ES has an internal watchdog timer. This is disabled by default with **ae_init()**.

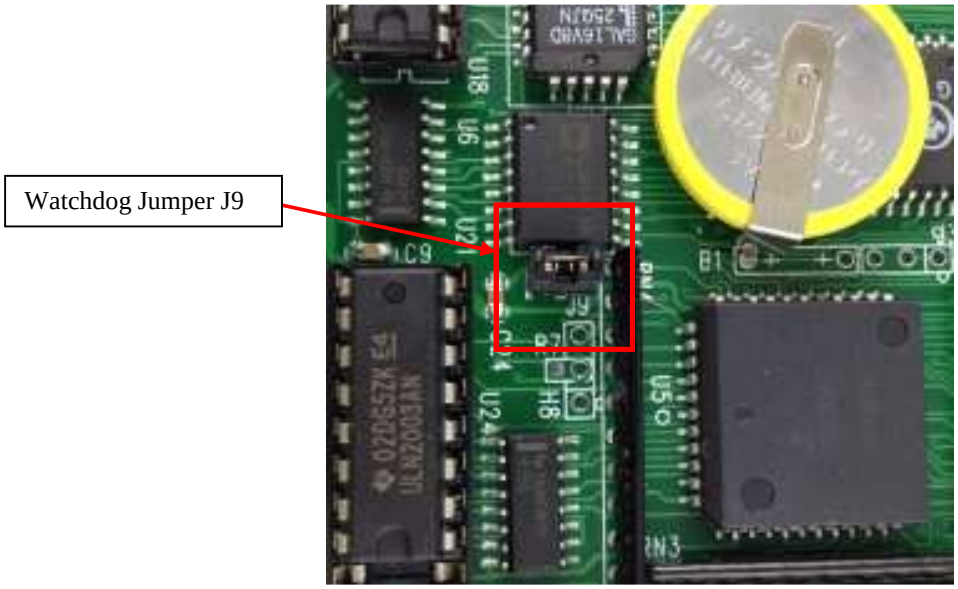


Figure 3.4 Location of watchdog timer enable jumper

Battery Backup Protection

The backup battery protection protects data stored in the SRAM and RTC. The battery-switch-over circuit compares VCC to VBAT (+3V 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), or 2K bytes (24C16) can be installed in U7. The TDU uses the P12=SCL (serial clock) and P11=SDA (serial data) to interface with the EEPROM. The EEPROM can be used to store important data such as a node address, calibration coefficients, and configuration codes. It typically has 1,000,000 erase/write cycles. The data retention is more than 40 years. EEPROM can be read and written by simply calling functions the **ee_rd()** and **ee_wr()**. A range of lower addresses in the EEPROM is reserved for TERN use.

3.6 Inputs and Outputs

The TDU offers more than 70 I/O lines, include TTL level I/O, high voltage I/O, and analog I/O. The below diagram gives a brief summary of available I/O.

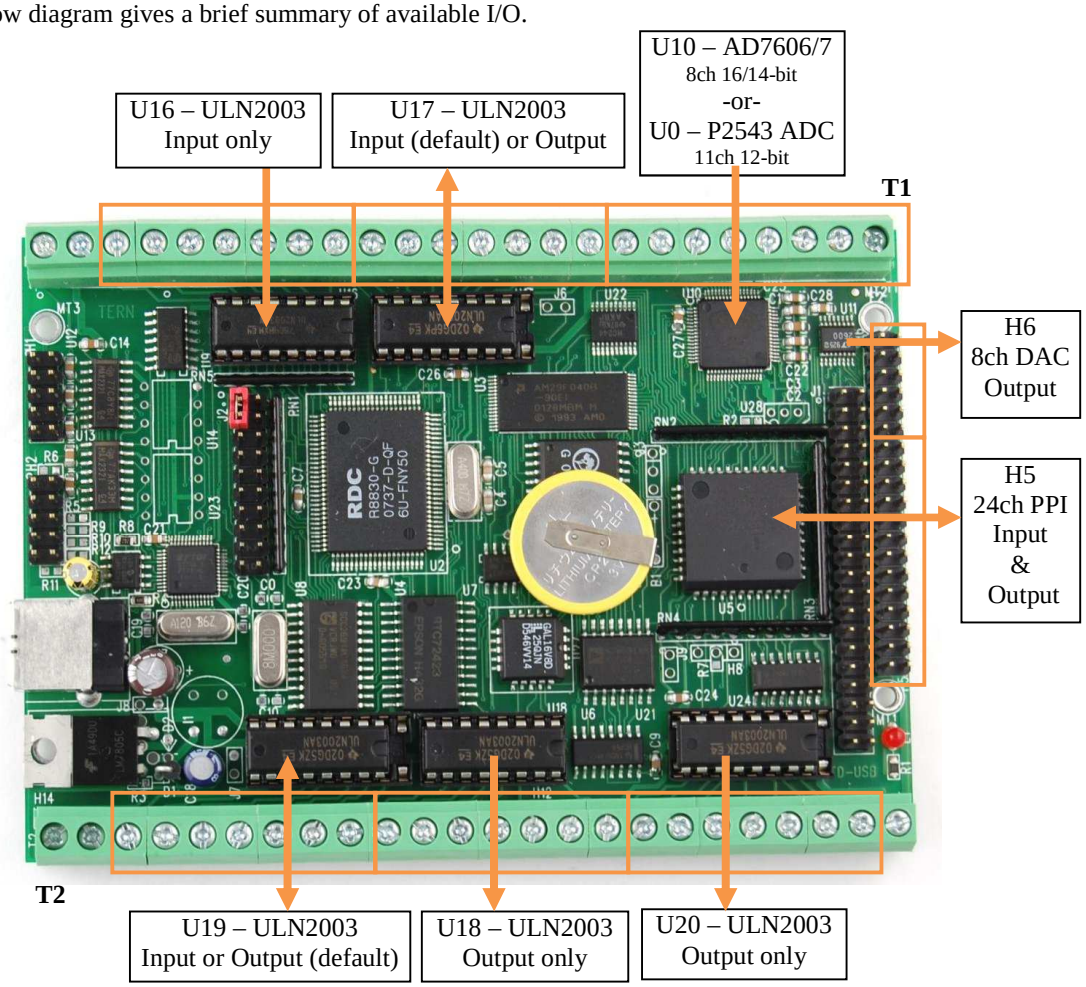


Figure 3.5 TDU Inputs and Outputs

3.6.2 AD7606 16-bit parallel and AD7607 14-bit high speed ADC

The TDU supports either the AD7606, 16-bit ADC or the AD7607, 14-bit ADC. Both ADC's can accept ±10V or ±5V true bipolar analog signals while sampling at throughput rates up to 200 kSPS for all 8 analog inputs. Each analog input contains second-order antialansing filter, sample-and-hold amplifier and clamp protection tolerant up to ±16.5V. With 1M ohm analog input impedance, a 7000V ESD rating, and sustaining up to ±10 mA input current, the analog inputs are designed to survive in a rough industrial environment. The TDU allows simultaneous sampling on all eight analog inputs. Using the 16-bit parallel

DMA interface, the TDU can transfer 8 channels of 16-bit (AD7606) or 14-bit (AD7607) data into the SRAM or CompactFlash with minimal software overhead.

The analog inputs AI1 to AD8 are available at T1 terminal 1 through 8. The AD7606 or Ad7607 ADC may only be installed if the P2543 is NOT installed.

See sample program \tern\186\samples\tdu\tdu_ad.c for details on reading the ADC. The sample program is also included in the pre-built sample project: \tern\186\samples\tdu\tdu.ide.

3.6.3 12-bit ADC (P2543)

The P2543 is a 12-bit, switched-capacitor, successive-approximation, 11 channels, serial interface, analog-to-digital converter. Three PPI I/O lines are used to handle the ADC, with /CS=I20; CLK=I22; and DIN=I21.

The ADC digital data output communicates with a host through a serial tri-state output (DOUT=P11). If I20=/CS is low, the P2543 will have output on P11. If I20=/CS is high, the P2543 is disabled and P11 is free. I20 and P11 are pulled high by 10K resistors on board. The P2543 has an on-chip 14-channel multiplexer that can select any one of 11 inputs or any one of three internal self-test voltages. The sample-and-hold function is automatic. At the end of conversion, the end-of-conversion (EOC) output is not connected, although it goes high to indicate that conversion is complete.

P2543 features differential high-impedance inputs that facilitate ratiometric conversion, scaling, and isolation of analog circuitry from logic and supply noise. A switched-capacitor design allows low-error conversion over the full operating temperature range. The analog input signal source impedance should be less than 50Ω and capable of slewing the analog input voltage into a 60 pF capacitor.

The CLK signal to the ADC is toggled through an I/O pin, and serial access allows a conversion rate of up to approximately 10 KHz.

The analog reference is provided by the REF1 signal. Normally REF1 is tied to VCC to provide a 5V reference. An optional precision reference can be install at U28. Note that REF1 is shared by the DAC2600 chip U11.

Eight analog inputs of P2543 are available at T1 terminal 1 through 8 (AI1 to AD8). The P2543 ADC may only be installed if AD7606 and AD7607 are NOT installed.

3.6.4 Eight channel 16-bit DAC (TLC2600)

The TLC2600 is an eight channel 16-bit digital-to-analog converter (DAC) in an SO-8 package. It is complete with a rail-to-rail voltage output amplifier capable of driving up to 15mA. It uses a 3-wire SPI compatable serial interface and has an output range of 0-REF volts, making 1 LSB equal to REF/65535 V. The analog reference is provided by the REF1 signal. Normally REF1 is tied to VCC to provide a 5V reference. An optional precision reference can be install at U28. Note that REF1 is shared by the P2543 ADC chip. The REF voltage must be greater than GND and less than VCC. The DAC outputs are routed to the H6 pin header, pins 1-8.

The DAC is installed on the TDU at location U11 and uses /RTS0 as the chip select. The synchronous serial interface is used to send data to the device. Refer to the sample code \tern\186\samples\tdu\tdu_da.c for an example on driving the DAC. The sample is also included in the pre-built sample project \tern\186\samples\tdu\tdu.pdl.

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

3.6.5 Protective high voltage inputs

In order to support high voltage digital signal input up to 30V, Darlington Transistor Arrays (ULN2003A) are installed in U16, U19, U17. The maximum input voltage is 30V. The input pin has 12.7K resistance load to the GND. You have to provide a pulled high signal input. A valid input low voltage is less than 0.8V, and a valid input high voltage is higher than 3V and less than 30V.

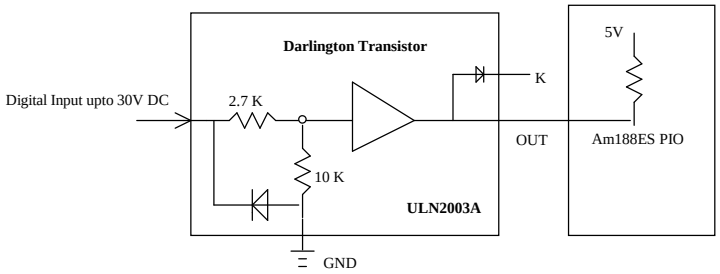


Figure 3.6 Darlington Transistors used as Protective High Voltage Inputs.

U17 and U19 may be set as input or output. By factory default, U19 is output and U17 is input. The input and output orientation for U17 and U19 is illustrated below. Follow these illustrations carefully to prevent damage to the chips. **Notice that U17 and U19 are aligned differently.** In addition, the ULN2003 chips may be replaced with a resistor pack to provide digital inputs or outputs to the terminal blocks.

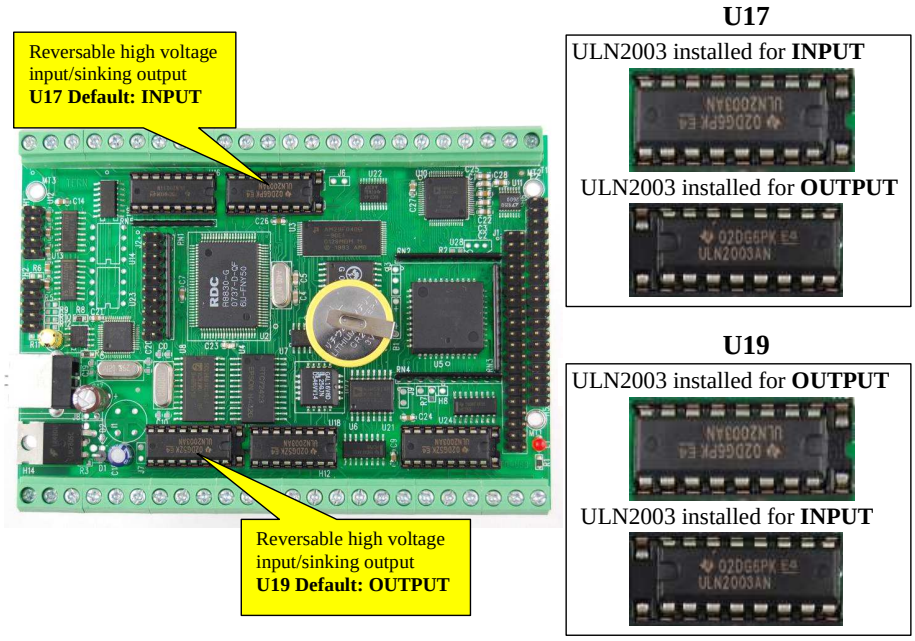


Figure 3.7 Locations of user configurable Darlington Transistor Arrays.

3.6.6 High-Voltage, High-Current Drivers

ULN2003 has high voltage, high current Darlington transistor arrays, consisting of seven silicon NPN Darlington pairs on a common monolithic substrate. All channels feature open-collector outputs for sinking 350 mA at 50V, and integral protection diodes for driving inductive loads. Peak inrush currents of up to 600 mA sinking are allowed. U18 and U20 are dedicated high-voltage drivers and U17 and U19 are configurable as high-voltage drivers. These outputs may be paralleled to achieve high-load capability, although each driver has a maximum continuous collector current rating of 350 mA at 50V. The maximum power dissipation allowed is 2.20 W per chip at 25 degrees C (°C). The common substrate G is routed to T2 GND pins. All currents sinking in must return to the T2 GND pin. A heavy gauge (20) wire must be used to connect the T2 GND terminal to an external common ground return. K connects to the protection diodes in the ULN2003 chips and should be tied to highest voltage in the external load system. K can be connected to an unregulated on board +12V via J6 and J7. **ULN2003 is a *sinking* driver, not a sourcing driver.** An example of typical application wiring is shown below.

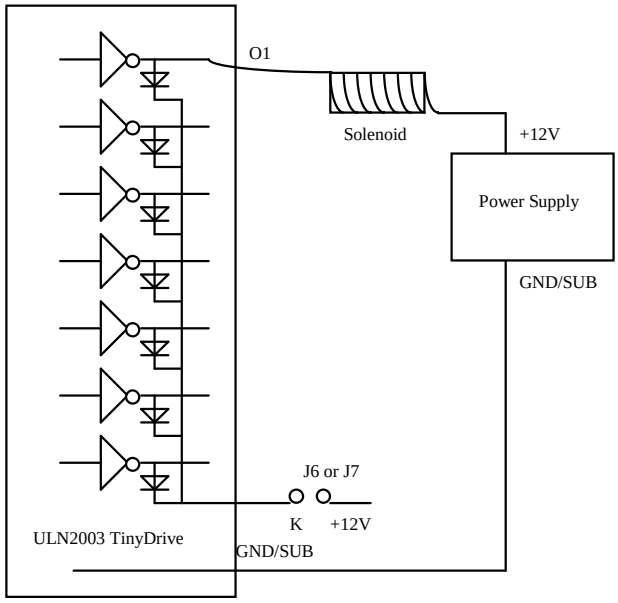


Figure 3.8 Drive inductive load with high voltage/current drivers.

3.1.1 High-Speed USB 1.1/2.0 Slave FT232H

FTDI's TF232H chip provids a USB 1.1/2.0 slave USB-B port. The FT232H handles USB stack processing and no USB specific firmware programming is required. The FT232H is configured to interface with the TDU's CPU using the FT245 style asynchronous FIFO interface. When configured in this mode, the pins on the FT232H connect directly to the databus of the CPU and is selected using an I/O mapped chip select.
The FT232H uses two types of USB software drivers: VCP or D2xx. When the FT232H is configured to use the VCP driver, the USB connection creates a virtual COM port on the PC. This allows the UB to communicate to a terminal program as if it were connected via RS232. The sample program \tern\186\samples\tdu\tdu_usb.c shows how to use the USB port as a virtual COM port.
By default, the FT232H is configured to use the D2xx driver. The data transfer rate can go as high as 8 MB/sec with the D2xx driver. The D2xx driver provides a dymanic linked library that the user can use in developing a Windows application interface (see FTDI.com for information on using the D2xx driver). See the appendix at the end of this manual for installing and configuring the FTDI drivers.

3.7 Headers and Connectors

The following are the jumpers and connectors on the TDU

Name	Size	Fuunction
H1	5x2	SER1 RS232
H2	5x2	SER0 and SCC RS232
H5	13x2	24 I/O from PPI 82C55
H6	6x2	8 DAC output
H8	3x1	POT for LCD contrast adjustment
J1	20x2	Address/Databus expansion port
J2	10x2	Programmable processor I/O expansion port
J3	3x1	SRAM selection: Pin 1=2: 128K, Pin 2=3: 512K
J6	2x1	HV proctection diode jumper for ULN2003 U17
J7	2x1	HV proctection diode jumper for ULN2003 U18, U19, U20
J8	2x1	VOFF for switching regulator
J9	2x1	Watchdog enable jumper
T1	24x1	Screw terminal for inputs
T2	24x2	Screw terminal for outputs

3.7.1 Header J1 and J2

Header J2			
GND	1	2	VCC
P4	3	4	P14
/CTS0	5	6	
TXD0	7	8	/INT4
RXD0	9	10	/RTS1
	11	12	P1
TXD1	13	14	
RXD1	15	16	GND
/NMI	17	18	
/CTS	19	20	/INT2

Header J1			
VCC	1	2	GND
	3	4	CLK
	5	6	GND
	7	8	D0
	9	10	D1
	11	12	D2
	13	14	D3
/RST	15	16	D4
RST	17	18	D5
P16	19	20	D6
	21	22	D7
	23	24	GND
	25	26	A7
	27	28	A6
/WR	29	30	A5
/RD	31	32	A4
	33	34	A3
	35	36	A2
	37	38	A1
	39	40	A0

3.7.2 Header H5 and H6

The 0.1 inch spacing dual row pin headers H6 and H5 are lined up as a 19x2 pin header next to the PCB right side edge of TDU. The headers have the following pin layout:

Header H6			
DA1	1	2	DA2
DA3	3	4	DA4
DA5	5	6	DA6
DA7	7	8	DA8
	9	10	
GND	11	12	
Header H5			
I10	26	25	I11
I12	24	23	I13
I14	22	21	I15
I16	20	19	I17
I20	18	17	I21
I22	16	15	I23
I24	14	13	I25
I26	12	11	I27
I00	10	9	I01
I02	8	7	I03
I04	6	5	I05
VLC	4	3	VCC
I06	2	1	I07

3.7.3 Terminal Blocks

The TDU has a total of 24x2 positions of terminal blocks. The signals are listed as below. As default, T1 is for inputs, and T2 is for outputs.

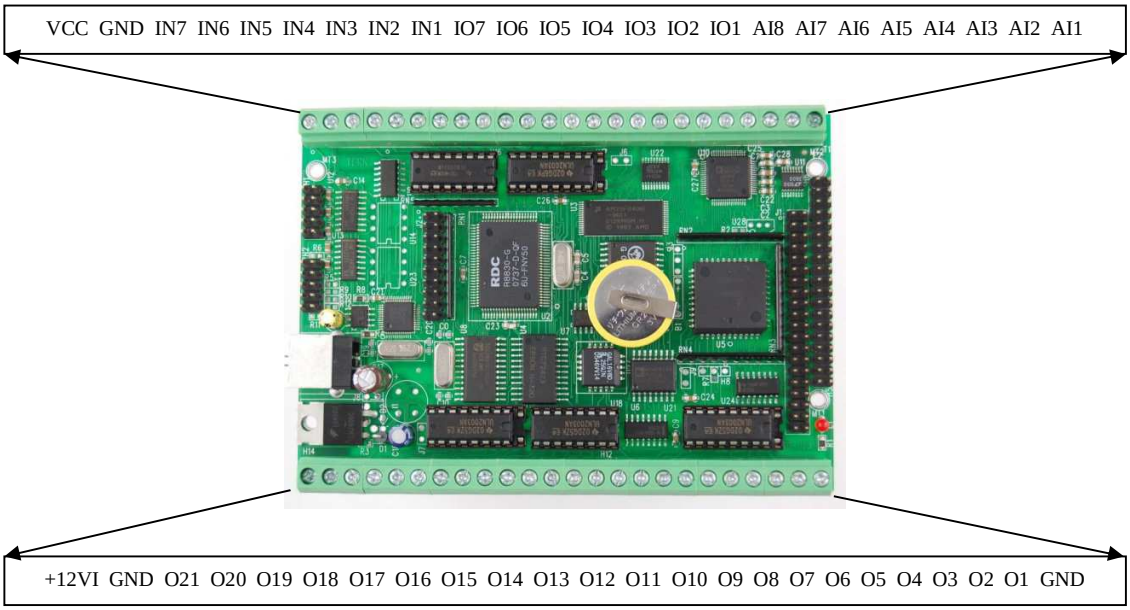


Figure 3.9 Terminal block diagram

		T1
A{1…8}	1 … 8	Analog input
IO{1...7}	9 … 15	Configurable high-voltage input/output
IN{1...7}	16 … 22	Dedicated high-voltage input
GND	23	Ground
VCC	24	+5V power supply, <200 mA

Table 3.3 Terminal 1 (T1)

		T2
+12VI	1	Unregulated input voltage from power supply
GND	2	Ground
O{21 … 15}	3 … 9	Configurable high-voltage input/output
O{14...1}	10 … 23	Dedicated high-voltage drivers
GND	24	Ground

Table 3.4 Terminal 2 (T2)

Chapter 4: Software

Please refer to the Technical Manual of the "C/C++ Development Kit Professsional 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 AE.LIB

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

Include-file name	Description
AE.H	PPI, timer/counter, ADC, DAC, RTC, Watchdog,
SER0.H	Internal serial port 0
SER1.H	Internal serial port 1
SCC.H	External UART SCC2691
AEEE.H	on-board EEPROM

4.2 Functions in AE.OBJ

4.2.1 TDU Initialization

ae_init

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

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

- Initialize the upper chip select to support the default ROM. The CPU registers are configured such that:
  - Address space for the ROM is from 0x80000-0xfffff (to map MemCard I/O window)

- 512K ROM Block size operation.
- Three wait state operation (allowing it to support up to 120 ns ROMs). With 70 ns ROMs, this can actually be set to zero wait state if you require increased performance (at a risk of stability in noisy environments). For details, see the UMCS (Upper Memory Chip Select Register) reference in the processor User's manual.

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

- Initialize LCS (*Lower Chip Select*) for use with the SRAM. It is configured so that:
  - Address space starts 0x00000, with a maximum of 512K RAM.
  - Three wait state operation. Reducing this value can improve performance.
  - Disables PSRAM, and disables need for external ready.

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

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

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

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

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

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

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

- Configure the PPI 82C55 to all inputs, except for lines I20-23 which are used as output for the ADC. You can reset these to inputs if not being used for that function.

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

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

void io_wait
Arguments: char wait
Return value: none.

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

```
wait=0, wait states = 0, I/O enable for 100 ns
wait=1, wait states = 1, I/O enable for 100+25 ns
```

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

4.2.2 External Interrupt Initialization

There are up to eight external interrupt sources on the TDU, 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, you need to write the **EOI** register word with 0x8000.

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

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

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

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

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

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

4.2.3 I/O Initialization

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

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

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

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

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

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

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

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

mode refers to one of four modes of operation.

- 0, 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 TDU 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 8 of the AMD AM188ES User's Manual.

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

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

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

It is also possible to use the output of **Timer2** to pre-scale one of the other timers, since 16-bit resolution at the maximum clock rate specified gives you only 150 Hz. Only by using **Timer2** can you slow this down even further. The sample files ***timer02.c*** and ***timer12.c***, located in ***tern\186\samples\ae***, demonstrate this.

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

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

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

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

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

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

4.2.5 Analog-to-Digital Conversion

The ADC unit provides 11 channels of analog inputs based on the reference voltage supplied to **REF+**. For details regarding the hardware configuration, see the Hardware chapter.

In order to operate the ADC, lines I20, I21,I22 from the PPI must be configured as output. P11 must also be configured to be input. This line is also shared with the RTC and EEPROM, and left high at power-on/reset. You should be sure not to re-program these pins for your own use. Be careful when using the EEPROM concurrently with the ADC. If the ADC is enabled, the line P11 will be reserved for its use and any attempt to access the EEPROM will time-out after some time.

For a sample file demonstrating the use of the ADC, please see **ae_ad12.c** in **`tern\186\samples\ae`**.

int ae_ad12
Arguments: char c
Return values: int ad_value

The argument **c** selects the channel from which to do the next Analog to Digital conversion. A value of 0 corresponds to channel **AD0**, 1 corresponds to channel **AD1**, and so on.

The return value **ad_value** is the latched-in conversion value from the previous call to this function. This means each call to this function actually returns the value latched-in from the previous analog-to-digital conversion.

For example, this means the first analog-to-digital conversion done in an application will be similar to the following:

```
ae_ad12(0); // Read from channel 0
chn_0_data = ae_ad12(0); // Start the next conversion, retrieve value.
```

4.2.6 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.

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

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

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

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

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

Void rtc_init
Arguments: char* t
Return value: none

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

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

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

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

Delay

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

void delay0
Arguments: unsigned int t
Return value: none

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

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

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

void delay_ms
Arguments: unsigned int
Return value: none

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

unsigned int crc16
Arguments: unsigned char *wptr, unsigned int count
Return value: unsigned int value

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

void ae_reset
Arguments: none

Return value: none

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

4.3 Functions in SER0.OBJ/SER1.OBJ

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

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

Two asynchronous serial ports are integrated in the 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 that are specific to SER1 can be easily changed into function calls for SER0. While selecting a serial port for use, please realize that some pins might be shared with other peripheral functions. This means that in certain limited cases, it might not be possible to use a certain serial port with other on-board controller functions. For details, you should see both chapter 10 of the Am188ES Microprocessor User's Manual and the schematic of the TDU 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.

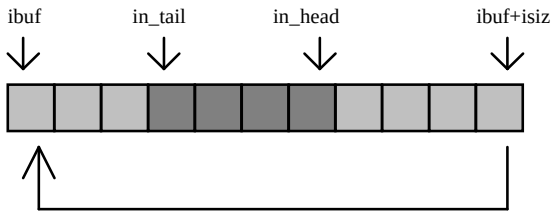


Figure 4.1 Circular ring input buffer

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()** you can set up a new mode with different numbers for data-bit, stop bit, or parity by directly accessing the Serial Port 0/1 Control Register (SP0CT/SP1CT) if necessary, as described in chapter 10 of the Am188ES manual for asynchronous serial ports.

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

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

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

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

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

Software Interface

Before using the serial ports, they must be initialized.

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

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

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

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

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

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

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

putser*n*
Arguments: unsigned char outch, COM *c
Return value: int return_value

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

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

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

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

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

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

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

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

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

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

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

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

Miscellaneous Serial Communication Functions

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

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

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

clean_ser*n*
Arguments: COM *c
Return value: none

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

The asynchronous serial I/O ports available on the Am188ES Processor have many other features that might be useful for your application. If you are truly interested in having more control, please read Chapter 10 of the 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 **scc.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 third port are different than for SER0 and SER1.

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 **ae_scc1.c**, found in the **tern\186\samples\ae** directory.

Function Argument	Baud Rate
1	110
2	150
3	300
4	600
5	1200
6	2400
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 Appendix B of this manual. 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 and SER1. 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 and SER1. 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 and SER1.

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.

scc_err
Arguments: none

Return value: unsigned char val
The returned value **val** will be in the form of 0ABC0000 in binary. Bit A is 1 to indicate a framing error. Bit B is 1 to indicate a parity error, and bit C indicates an over-run error.

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 your application use.

The EEPROM shares line P11 with the ADC. If the ADC is enabled, it can interfere with the EEPROM. The ADC is enabled if I20 is low. In the init function, it is brought high so that you can access the EEPROM. Be aware that if you modify the PPI control register by calling outportb(0x0103, xx); then all of the output lines on the PPI are brought low, including I20, which enables the ADC and disables the EEPROM. If you need to use the EEPROM, be sure to bring I20 high again to disable the ADC (refer to section 3.5.2).

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

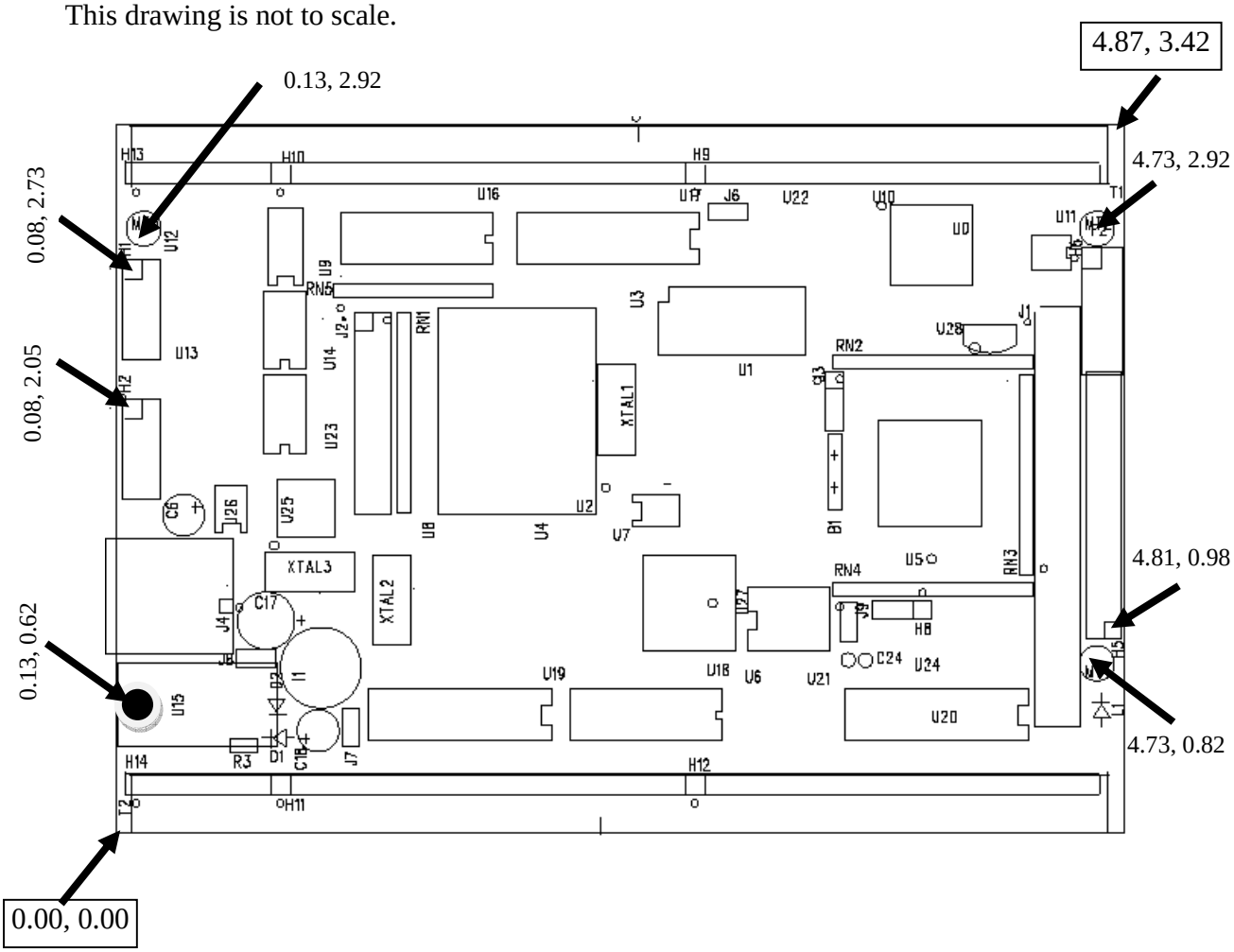
This function is used to write the passed in **dat** to the specified **addr**. The return value is 0 in success.

ee_rd
Arguments: int addr
Return value: int data

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

Appendix A: TDU Layout

The TDU measures 4.87 by 3.42 inches. All dimensions shown below are in inches. This drawing is not to scale.



Appendix B: Software Glossary

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

---

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);
enable();
```

Reference: led.c

---

void ae_reset(void) ae.h

Resets Am188ES 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

---

int ae_ad12(unsigned char c) ae.h

Reads from the 11-channel 12-bit ADC. Returns 12 bit AD data of the previous channel.
In order to operate ADC, I20,I21,I22 must be output and P11 must be input.
P11 is shared by RTC, EE. It must left high at power-on/reset.
Unipolar:
Vref- = 0x000
Vref+ = 0xfff
Use 1 wait state for Memory and I/O without RDY, < 300 us execution time
Use 0 wait state for Memory and I/O with VEP010, < 270 us execution time

```
Var:  c – ADC channel.
                  c = {0 … a}, input ch = 0 – 10
                  c = b,       input ch = (vref+ - vref-) /2
                  c = c,       input ch = vref-
                  c = d,       input ch = vref+
                  c = e,       software power down
```

Reference: ae_ad12.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);
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

---

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

Serial port 0, 1 initialization.

Var: m1 = SCC691 MR1
m2 = SCC691 MR2
b – baud rate. T*able 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: 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
int putser_scc(unsigned char ch, COM *c); scc.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
int putsers_scc(unsigned char ch, COM *c); scc.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
int serhit_scc(COM *c); scc.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
unsigned char getser_scc(COM *c); scc.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

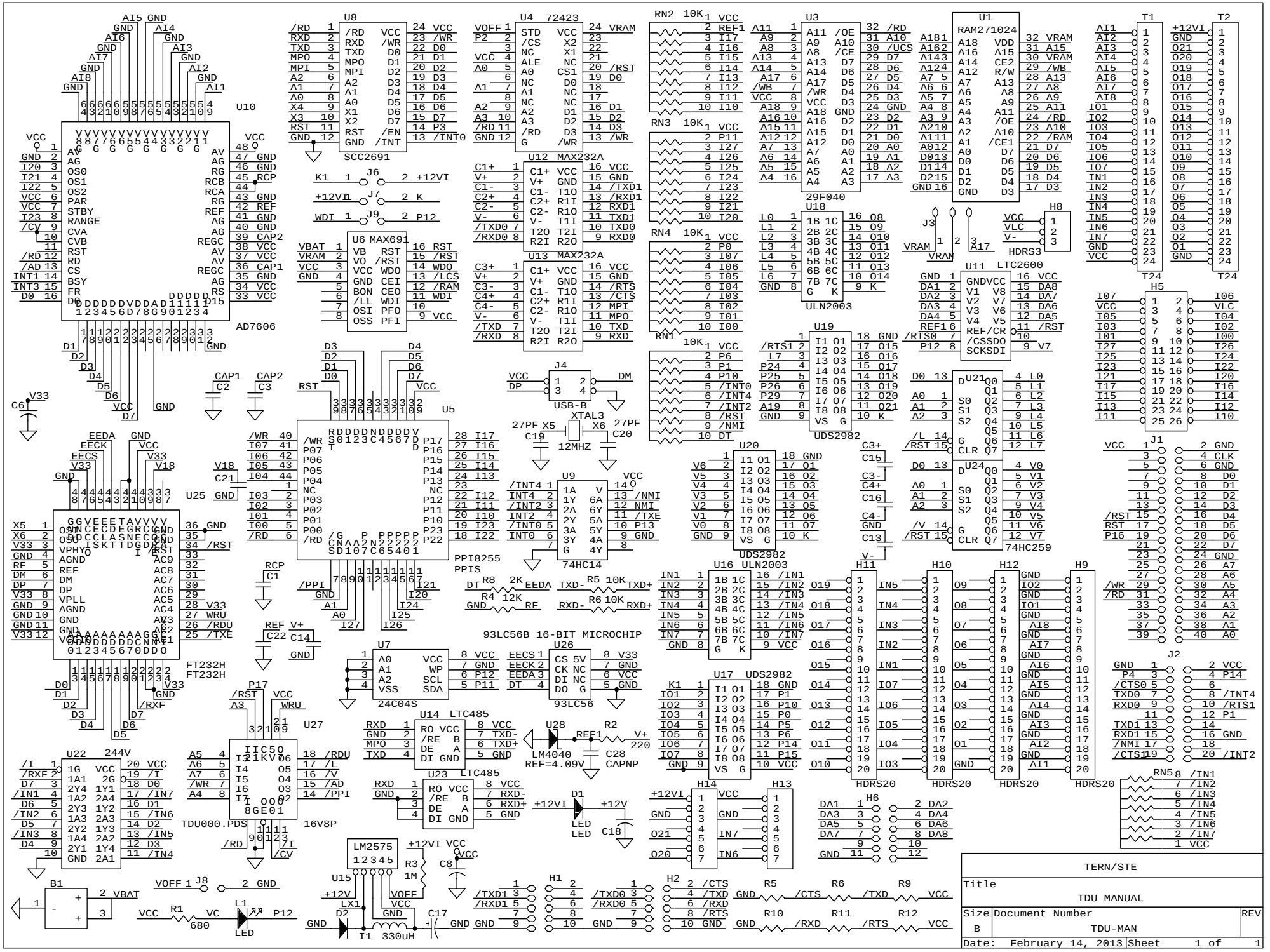
---

int getsers0(COM *c, int len, unsigned char *str); ser0.h
int getsers1(COM *c, int len, unsigned char *str); ser1.h
int getsers_scc(COM *c, int len, unsigned char *str); scc.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.


U10
AD7606
U8
SCC2691
U4 72423
U12 MAX232A
U13 MAX232A
U6 MAX691
J6
J7
J9
K1
+12VI
WDI
RN2 10K
RN3 10K
RN4 10K
RN1 10K
U3
29F040
U18
ULN2003
U1
RAM271024
J3
VRAM
A17
H8
HDRS3
U11 LTC2600
U19
UDS2982
U21
U24
74HC259
T1
T2
T24
H5
J1
J2
U5
PPI8255
PPIS
XTAL3
12MHZ
J4
USB-B
U9
74HC14
U20
U16 ULN2003
U26
93LC56B 16-BIT MICROCHIP
93LC56
U7
24C04S
U27
TDU000.PDS
16V8P
U14 LTC485
U23 LTC485
U28
LM4040
REF=4.09V
U22 244V
FT232H
U15
LM2575
12345
D1
LED
B1
VBAT
J8
VOFF
LED
H1
H2
H6
H11
H10
H12
H9
H13
H14
HDRS20
RN5
TERN/STE
TDU MANUAL
Size B
Document Number TDU-MAN
REV
Date: February 14, 2013
Sheet 1 of 1