U-Drive™

C/C++ Programmable User Interface with QVGA TFT and touchscreen.
Supports CAN, Host USB ports, Ethernet, RS232/485, CompactFlash, 24-bit ADCs,
16-bit DACs, Solenoid Drivers, and Relays

Technical Manual

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

Version 1.02

May 27, 2010

© 1993-2008 TERN INC.
1950 5th Street, Davis, CA 95616, USA
Tel: 530-758-0180 Fax: 530-758-0181
Internet Email: tern@netcom.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.

Chapter 1: Introduction

1.1 Functional Description

The ***U-Drive (UD)*** is a very low-cost industrial GUI controller, ideal for OEM applications requiring a user interface. The ultra-bright, wide viewing angle active TFT color display with touch screen is attractive, and easy to program. Other peripherals on the board make this board a powerful and comprehensive industrial user interface and control solution.

DISPLAY

The onboard LCD controller (S1D13075, EPSON) has internal 80KB image buffer, supporting QVGA color graphic LCDs with 320x240 pixels. The integrated TFT display is ultra-bright (rated at up to 750 nits), with brightness controlled by external DC power input (8-11V). An accurate touch screen controller (ADS7846) supports 4-wire resistive touch screen.

All components are installed on single PCB, mounted on the backside of the QVGA TFT for easy integration into user applications. Comprehensive, user-friendly software libraries and samples are provided. User can easily design their custom functions keys, text, logo, and graphics. Supported by the CompactFlash-based file system (up to 2 GB), 20 user screens can be displayed per second via DMA transfer.

USB

A Host USB controller can be installed to provide two Host USB ports. Port 1 can interface to USB keyboard/mouse, allowing a flexible mechanism for accepting user input in addition to touch screen. Port 2 supports a hot-removable USB flash disk, using a simple command set to manipulate a FAT filesystem format. No other USB specific firmware programming is required on the controller side.

COMMUNICATIONS

A Controller Area Network (CAN) controller (SJA1000, 20 MHz clock) is available. It supports network baud rates up to 1M-bit per second. Software drivers allow access to all CAN controller registers, as well as a buffering software layer.

A Fast Ethernet Module can be installed to provide 100M Base-T network connectivity. This Ethernet module has a hardware LSI TCP/IP stack, implementing TCP/IP, UDP, ICMP and ARP support in hardware. Socket-based software drivers allow the U-Drive to be used as a web or SMTP server.

Also available are 4x RS232 and 2x RS485 serial ports.

INDUSTRIAL I/O

There are 30+ TTL I/Os, 4 mechanical relays (200V, 0.5 Amp.), 4 opto-couplers and 14 solenoid drivers (sinking/sourcing 350 mA at 50V each). The solenoid drivers can be hardware configured to be high voltage (0-30V) inputs. Three 16-bit CPU internal timer/counters support timing and external counting.

There are up to a total of 31 ADC inputs and 12 DAC outputs:

4 ch.16-bit parallel ADC chip (AD7655, 1 MHz, 0-5V), 24-bit ADC (LTC2448□5KHz□0-1.25V) configurable for 8 ch. differential or 16 ch. single-ended input channels. A 12-bit ADC (TLC2543, 10KHz□0-5V) provides 11 ch. analog inputs. Eight ch. 16-bit DACs (LTC2600, 0-5V, 10 KHz), and four ch. 12-bit parallel DAC (DA7625, 200 KHz, 0-2.5V) are available.

Features:

* 6.5 x 4.8 inches, 1x86 CPU, program in C/C++
* Ultra-bright 5.7” QVGA Color TFT display

* 4-wire analog touch screen controller
* Host USB ports for user-input (keyboard/mouse)
* 256 KW SRAM, 256 KW Flash, 512 bytes EEPROM
* 30+ TTL I/O, 3 16-bit timer/counters, RTC, Battery
* 4 mechanical relays, 4 Opto-coupler inputs.
* 14+ solenoid drivers or high voltage inputs (0-30V DC)
* 11 ch. 12-bit ADCs, 4 ch. 16-bit ADCs, 16 ch. 24-bit ADCs
* Precision reference and on-board temperature sensor
* 8 ch. 16-bit DACs, 4 ch. 12-bit DAC
* Controller Area Network (CAN2.0B) port
* 4 RS232/and 2 RS485 serial ports
* 100 M Ethernet with hardware TCP/IP stack
* Switching regulator, 50 mA standby, 160 mA at 12V
* CompactFlash with Windows compatible file system support

1.2 Physical Description

The physical layout of the UD is shown below.

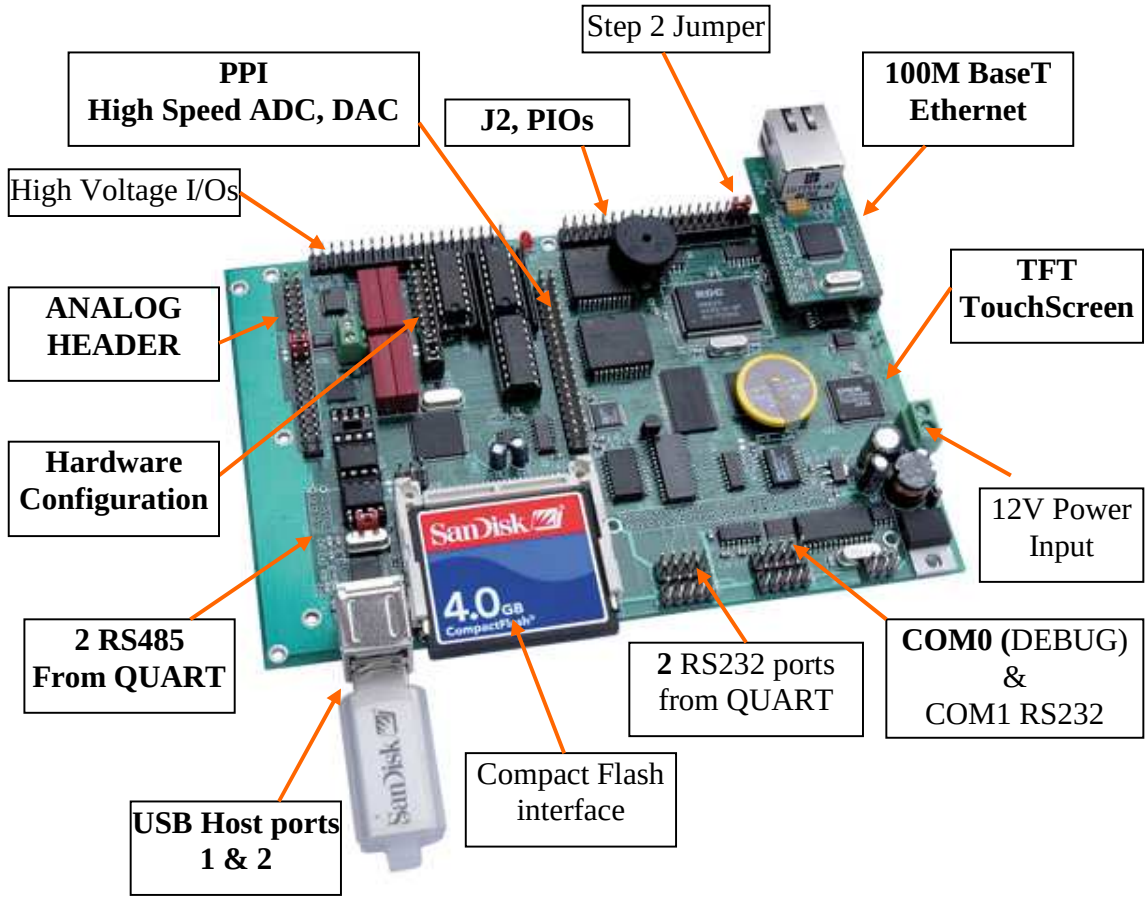


Figure 1.1 Physical layout of the U-Drive

Step 1 settings

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

1) EE40_115.HEX must be pre-loaded into Flash starting address 0xFA000.

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

For a 32K SRAM, the physical address is 0x00000-0x07fff
For a 128K SRAM, the physical address is 0x00000-0x01ffff
For a 512K SRAM, the physical address is 0x00000-0x07ffff

3) The on-board EEPROM must have a jump address for the EE40_115.HEX with starting address of 0xFA000.

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

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

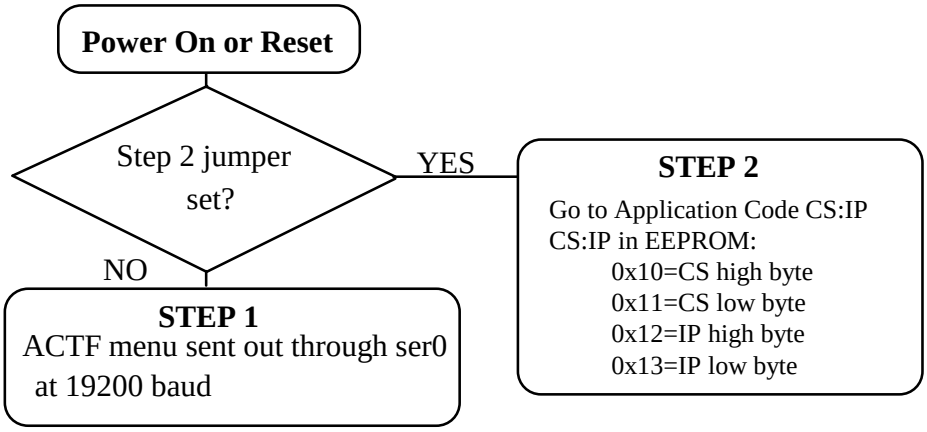


Figure 1.2 Flow chart for ACTF operation

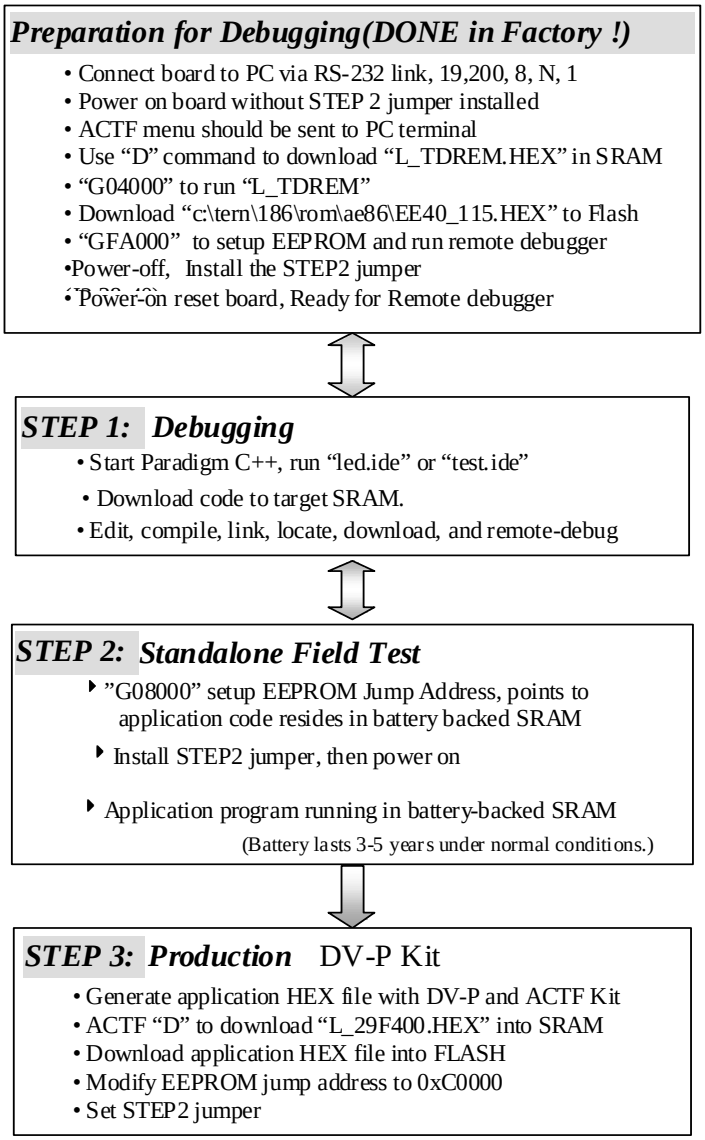
The “ACTF boot loader” resides in the top protected sector of the 512KB on-board Flash chip (29F400).

By default, in the factory, before shipping, the DEBUG kernel (EE40_115.hex) is pre-loaded in the Flash starting at 0xFA000, and the RED STEP2 jumper is installed, ready for Paradigm C++ debugger. User does not need to download a DEBUG kernel to start with.

At power-on or RESET, the “ACTF” will check the STEP 2 jumper. If STEP 2 jumper is not installed, the ACTF menu will be sent out from serial port0 at 19200 baud for a **UD**.
If the STEP 2 jumper is installed, the “jump address” located in the on-board serial EEPROM will be read out and then jump to that address. A DEBUG kernel “EE40_115.hex” for the **UD** can be downloaded, residing in “0xFA000” of the 512KB on-board flash chip.

1.3 U-Drive Programming Overview

Steps for product development:



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

Chapter 2: Installation

2.1 Software Installation

Please refer to the "tern_docs\Software_kit.pdf" Technical manual on TERN CD, for information on installing software.

2.2 Hardware Installation

Overview

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

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

2.2.1 Connecting the UD to the PC

The ***UD*** is linked to the PC via a serial cable (DB9-IDE) which is supplied with TERN EV-P / DV-P Kits.

The ***UD*** communicates through SER0 by default. Install the 5x2 IDE connector on the SER0 H4 pin header. Powering-on the UD.

By factory default setting:

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

Connect +9-12V DC to the DC power terminal. The screw terminal at the corner of the board is positive 12V input and the other terminal is GND (see figure for details). A power jack adapter (seen below) is included with the TERN EV-P/DV-P kit. It can be used to connect the output of the power jack adapter and the ***UD***. Note that the output of the power jack adapter is center negative.

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

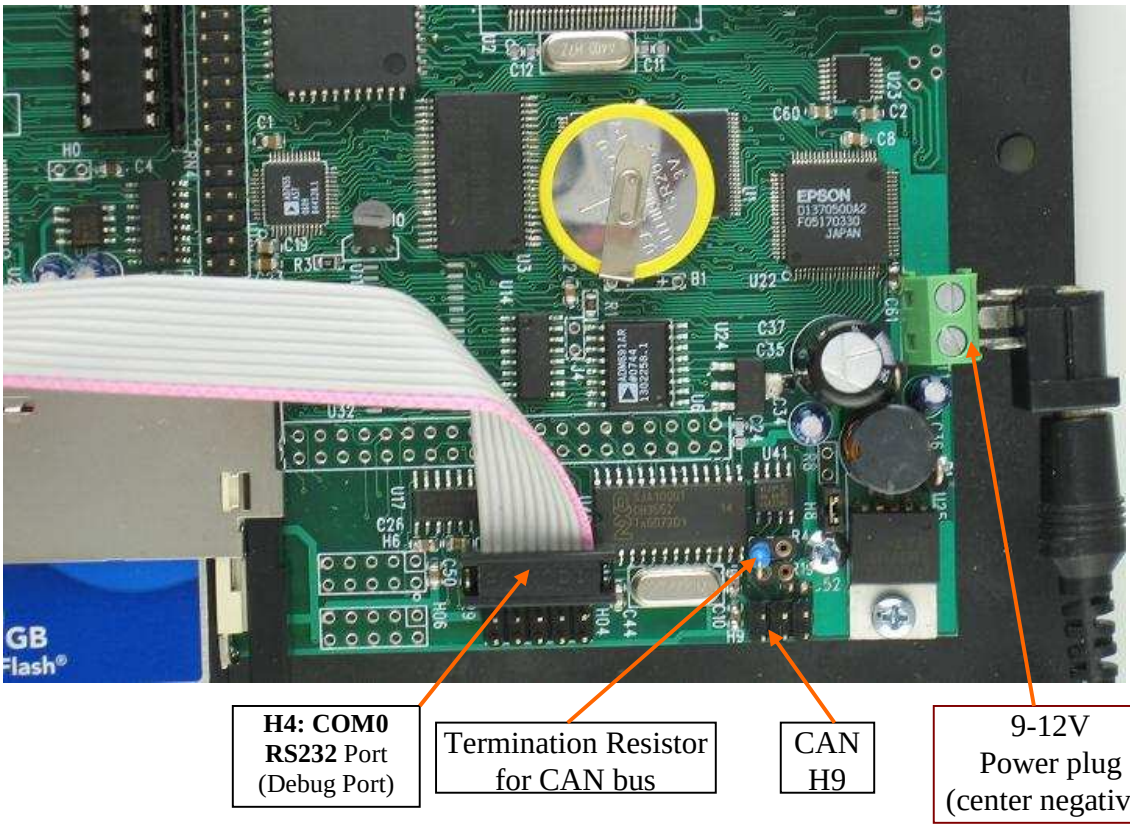
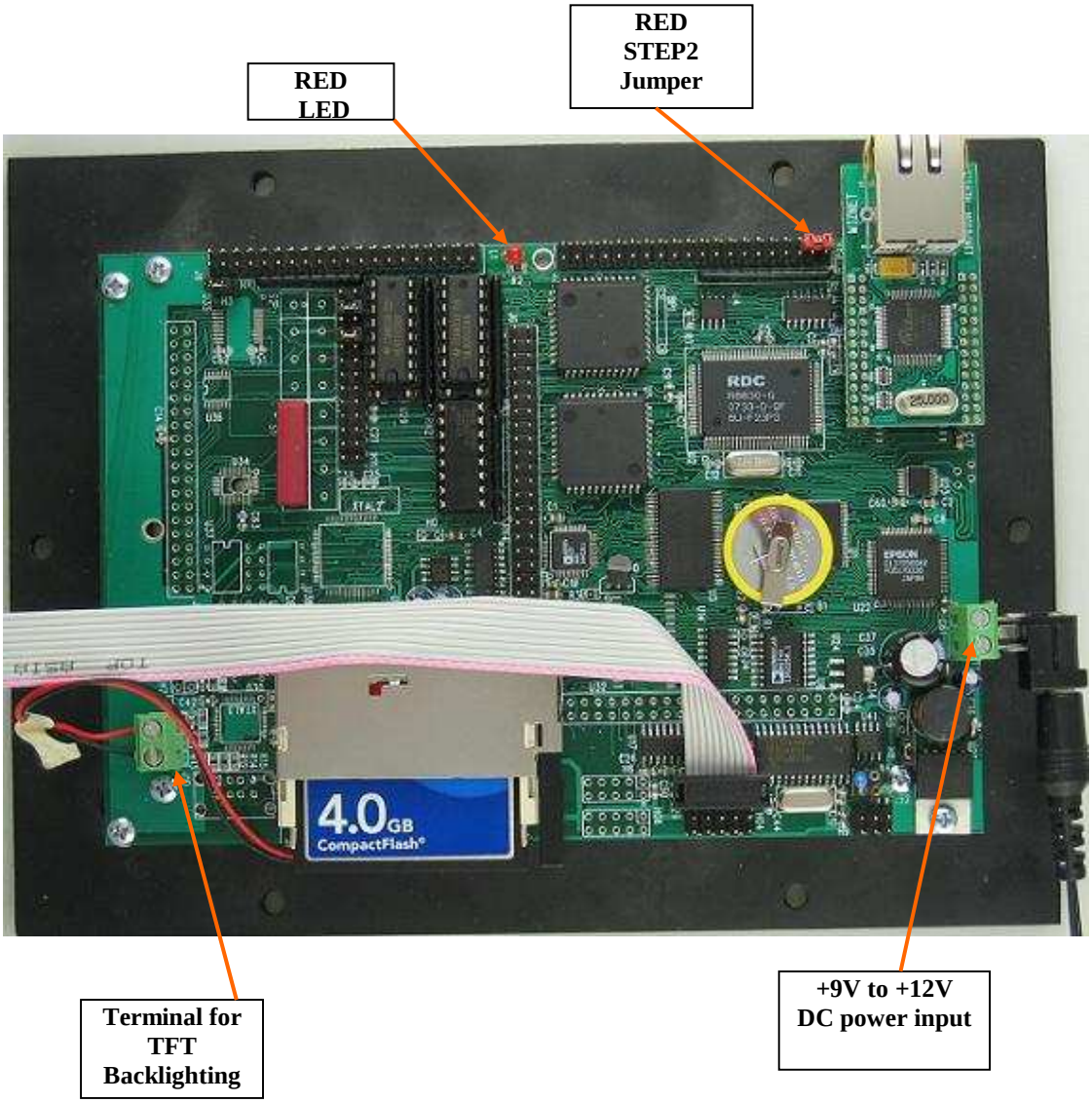


Figure 2.1 Location of Power input, DEBUG Cable, CAN port and Termination Resistor


RED
LED
RED
STEP2
Jumper
Terminal for
TFT
Backlighting
+9V to +12V
DC power input

Chapter 3: Hardware

3.1 Am186ES/R8820/IA186 - Introduction

The Am186ES controllers uses 16-bit external data bus, are higher-performance, more integrated versions of the 80C188 microprocessors which uses 8-bit external data bus. In addition, the Am186ES has new peripherals. The on-chip system interface logic can minimize total system cost. The Am186ES has two asynchronous serial ports, 32 PIOs, a watchdog timer, additional interrupt pins, a pulse width demodulation option, DMA to and from serial ports, a 16-bit reset configuration register, and enhanced chip-select functionality.

There are a total of three compatible CPU chips can be used:
R8820 from RDC is a drop-in replacement 5V, 40MHz chip for the AM186ES, AM186ES (AMD, 5V, 40 MHz), R8820 (RDC, 5V, 40 MHz), and IA186ES (INNOVASIC, 5V, 40 MHz). The multiple sources of the CPU can support longer life time. The technical specifications and discussions in this manual are based on AM186ES.

By default, the UD uses 5V 40 MHz R8820 and low power 55ns SRAM.
There are three pads on the PCB for battery. One pad is ground, and the other two pads allowing a 3V backup lithium battery is installed in two different positions:

3.2 Am186ES – Features

3.2.1 Clock and crystal

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

The system CLKOUTA signal is routed to J1 pin 4, default 40 MHz.

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

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

The R8820 uses a 40 MHz crystal.

Debug kernels for Paradigm C++ TERN Edition are available:

c:\tern\186\rom\ae86\EE40_115.hex

The EE40_115.hex will allow 40 MHz ***UD*** talk to Paradigm C++ TERN Edition at 115,200 baud.

By default, the EE40_115.hex is pre-programmed for the 40 MHz ***UD***.

User can use software to setup the CPU speed:

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

3.2.2 External Interrupts and Schmitt Trigger Input Buffer

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

INT0, J2 pin 8, used by CAN controller(SJA1000).
/INT1, J2 pin 6, free to use.
INT2, J2 pin 19, used by Touchscreen controller(ADS7846)
INT3, used by QUART INTA
/INT4, J2 pin 33, used by 100M BaseT Ethernet
INT5=P12=DRQ0, used by LED/EE/HWD/RTC
INT6=P13=DRQ1, J2 pin 11, used by QUART INTB
/NMI, J2 pin 7, used by MAX691 as PFO

Some of external interrupt inputs, /INT0, /INT1, /INT2, /INT4 and /NMI, are buffered by Schmitt-trigger inverters (U9, 74HC14), in order to increase noise immunity and transform slowly changing input signals to fast changing and jitter-free signals. As a result of this buffering, these pins are capable of only acting as input.

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

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

3.2.3 Asynchronous Serial Ports

The Am186ES CPU has two asynchronous serial channels: SER0 and SER1. Both asynchronous serial ports support the following:

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

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

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 external pins:

Timer0 output = P10 = J2 pin 12
Timer0 input = P11 = U7 EEPROM data
Timer1 output = P1 = J2 pin 29, used for on-board beeper
Timer1 input = P0 = J2 pin 20

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

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

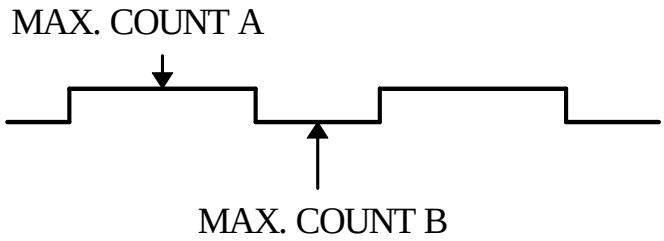
Timer2 is not connected to any external pin. It can be used as an internal timer for real-time coding or time-delay applications. It can also prescale timer 0 and timer 1 or be used as a DMA request source.

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

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 19, assuming the QUART is not installed.

3.2.6 Power-save Mode

The ***UD*** can be used for low power consumption applications. The power-save mode of the Am186ES reduces power consumption and heat dissipation, thereby extending battery life in portable systems. In power-save mode, operation of the CPU and internal peripherals continues at a slower clock frequency. When an interrupt occurs, it automatically returns to its normal operating frequency.

3.3 Am186ES PIO lines

The Am186ES has 32 pins available as user-programmable I/O lines. Each of these pins can be used as a user-programmable input or output signal, if the normal shared function is not needed. A PIO line can be configured to operate as an input or output with or without a weak pull-up or pull-down, or as an open-drain output. A pin's behavior, either pull-up or pull-down, is pre-determined and shown in the table below.

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

PIO	***Function***	***Power-On/Reset status***	***UD Pin No.***	***UD Initial***
P0	Timer1 in	Input with pull-up	J2 pin 20	Input with pull-up
P1	Timer1 out	Input with pull-down	J2 pin 29, Beeper	Input with pull-down
P2	/PCS6/A2	Input with pull-up	J2 pin 24	I/O chips select
P3	/PCS5/A1	Input with pull-up	J2 pin 15	Input with pull-up
P4	DT/R	Normal	J2 pin 38	Input with pull-up Step 2
P5	/DEN/DS	Normal	J2 pin 30	Input with pull-up
P6	SRDY	Normal	J2 pin 35	Input with pull-down
P7	A17	Normal	N/A	A17
P8	A18	Normal	N/A	A18
P9	A19	Normal	J2 pin 10	A19=/INT2, touchscreen
P10	Timer0 out	Input with pull-down	J2 pin 12	Input with pull-down
P11	Timer0 in	Input with pull-up	EEPROM	Input with pull-up
P12	DRQ0/INT5	Input with pull-up	N/A	Output for LED/EE/HWD
P13	DRQ1/INT6	Input with pull-up	J2 pin 11; QUART	Input with pull-up
P14	/MCS0	Input with pull-up	J2 pin 37, TFT	Input with pull-up
P15	/MCS1	Input with pull-up	J2 pin 23	Input with pull-up
P16	/PCS0	Input with pull-up	J1 pin 19	/PCS0
P17	/PCS1	Input with pull-up	HC138 U31.4,5	/PCS1
P18	CTS1/PCS2	Input with pull-up	J2 pin 22, Ethernet	Input with pull-up
P19	RTS1/PCS3	Input with pull-up	J2 pin 31	Input with pull-up
P20	RTS0	Input with pull-up	J2 pin 27	Input with pull-up
P21	CTS0	Input with pull-up	J2 pin 36	Input with pull-up
P22	TxD0	Input with pull-up	J2 pin 34	TxD0
P23	RxD0	Input with pull-up	J2 pin 32	RxD0
P24	/MCS2	Input with pull-up	J2 pin 17	Input with pull-up
P25	/MCS3	Input with pull-up	J2 pin 18	Input with pull-up
P26	UZI	Input with pull-up	J2 pin 4, USB RST	Input with pull-up*
P27	TxD1	Input with pull-up	J2 pin 28	TxD1
P28	RxD1	Input with pull-up	J2 pin 26	RxD1
P29	/CLKDIV2	Input with pull-up	J2 pin 3; USB AC4	Input with pull-up*
P30	INT4	Input with pull-up	J2 pin 33;JP1.2 (ET)	Input with pull-up
P31	INT2	Input with pull-up	J2 pin 19; TouchS	Input with pull-up

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

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

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

MODE	PIOMODE reg.	PIODIRECTION reg.	PIN FUNCTION
0	0	0	Normal operation
1	0	1	INPUT with pull-up/pull-down
2	1	0	OUTPUT
3	1	1	INPUT without pull-up/pull-down

UD 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 initialize PIO pins.

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

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

Example: ***pio_init***(12, 2); will set P12 as output

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

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

pio_wr(12,1); set P12 pin high, if P12 is in output mode

pio_wr(12,0); set P12 pin low, if P12 is in output mode

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

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

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

Some of the I/O lines are used by the ***UD*** 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).

You should also note that the external interrupt PIO pins INT1 and 4 are not available for use as output because of the inverters attached. The input values of these PIO interrupt lines will also be inverted for the same reason. As a result, calling ***pio_rd*** to read the value of P31 (**INT2**) will return 1 when pin 19 on header J2 is pulled low, with the result reversed if the pin is pulled high.

Signal	Pin	Function
P2	/PCS6; J2.24	U14(74HC138) I/O chip select for RTC, PPI, CAN,…
P4	/DT; J2.38	STEP2 jumper
P11	Timer0 in	Shared with EEPROM data input
P13	/INT6; J2.11	QUART INTB
P14	/MCS0, J2.37	SED1375,TFT controller
P18	/CTS1; J2.22	100M BaseT Ethernet
P22	TxD0; J2.34	Default SER0 debug

Signal	Pin	Function
P23	RxD0; J2.32	Default SER0 debug
P26	UZI; J2.4	USB RST
P27	TxD1; J2.28	Serial Port 1 Transmit
P28	RxD1; J2.26	Serial Port 1 Receive
P29	/CLKDIV2; J2.3	USB AC4
P30	INT4; J2.33	Ethernet interrupt JP1.2
P31	INT2; J2.19	Touchscreen controller

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 Am186ES User's Manual.

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

I/O space	Select	Location	Usage
0x0000-0x00ff	/PCS0	J1 pin 19=P16	USER*
0x0100	/UR2	U31.15, U20.9	QUART UR2 select
0x0120	/UR3	U31.14, U20.13	QUART UR3 select
0x0140	/UR4	U31.13, U20.49	QUART UR4 select
0x0160	/UR5	U31.12, U20.53	QUART UR5 select
0x0180	/RSTC	U31.11	CAN Hardware Reset
0x01A0	/RDU	U31.10	Read USB
0x01C0	/WRU	U31.9	Write USB
0x01E0	/CF	U31.7	CompactFlash select
0x0600	/RTC	U14.15	Real Time Clock select
0x0620	/PPI	U14.14	U5 PPI chip select
0x0640	/PP	U14.13	U33 PPI chip select
0x0660	/CAN	U14.12	CAN SJA1000 chip select
0x0680	/RDK	U14.11	Read Status inputs on U8
0x06A0	/LA	U14.10	Write page register for ET
0x06C0	/DA	U14.9	Write to DAC7625
0x06E0	/AD	U14.7	Read AD7655
0x0200	/PCS2	JP1.5	Ethernet select

*PCS0 may be used for other TERN peripheral boards.

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

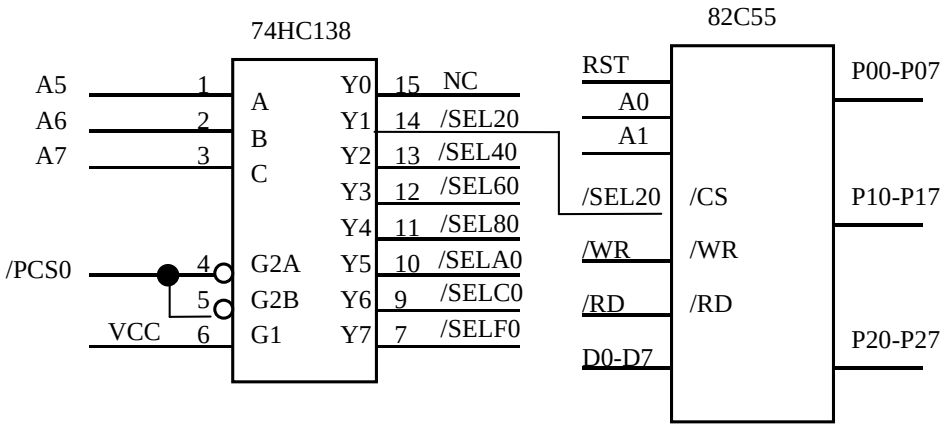


Figure 3.1 Interface to external I/O devices

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

3.4.2 SID13075

The SID13075 is a color/monochrome LCD graphics controller with an embedded 80K Byte SRAM display buffer. The LCD controller can achieve up to 20 frames per second and support 256 colors. A unique aspect of the ***ST*** is the CompactFlash storage working in concert with this LCD controller. With this design, the x186 can DMA images directly from the CompactFlash into the image buffer to achieve higher performance. Because of the 80KB image buffer, the function call *ud_init();* re-defines the upper half of the memory map to accommodate this buffer. After calling *ud_init();* the memory map will be as in the following diagram: **(diagram not to scale)**

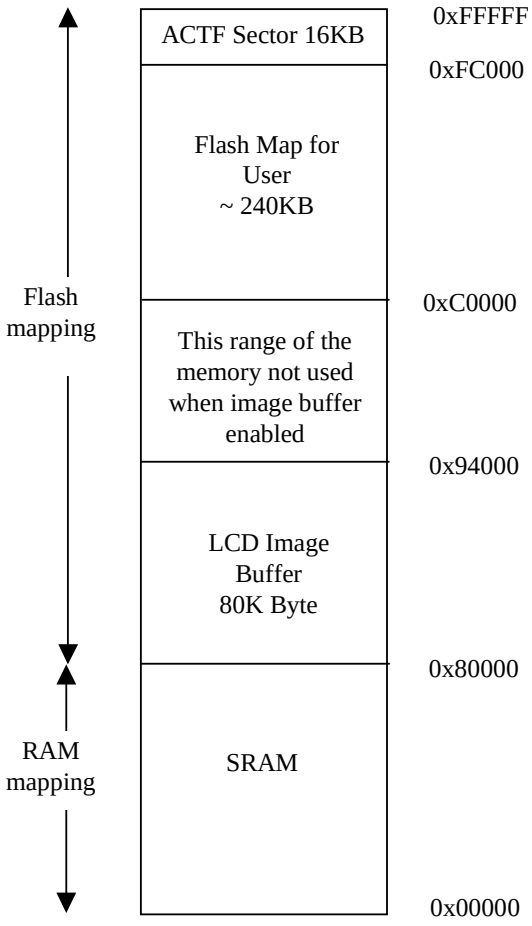


Figure 3.2 Memory Map after *ud_init();* to accommodate Image Buffer

Sample code has been provided in the **tern\186\samples\ud** directory to illustrate the process of transferring an image from the Compact Flash into the Image Buffer. It is mandatory that slc_init(); be called so as to re-initialize the memory map to accommodate the image buffer. Note that the entire ROM/Flash mapping is 512KB, but the re-mapping of the ROM/Flash only allows configuring to certain sizes. Thus the map available for the user then becomes 256KB, starting from 0xC0000 and going to 0xFFFFF, yet the ACTF utility occupies the top 16KB sector, again reducing the usable Flash to the range 0xC0000 to 0xFC000, or about 240KB. It is possible to map the Image Buffer into the SRAM, leaving all 512KB of Flash space for the user. This would require the user to re-define the /LMCS line used to select the SRAM. Refer to the AM186ES manual chapter 5 for details on how to re-map the SRAM. This re-mapping of the Image Buffer into the SRAM would also require the DMA process from the Compact Flash to the LCD controller to be altered. Refer to sample code in the **\tern\186\samples\ud** directory.

3.4.3 Touch Screen Controller

The ADS7846E is a 12-bit sampling ADC with a synchronous serial interface and low on-resistance switches for driving touch screens. The ADS7846E is routed to a 4-pin terminal at H2/H16 which connects to a flax cable to drive the touch screen. This controller allows the user to specify the touch screen resolution needed a particular application. See sample program that allows for calibration of touch screen in **\tern\186\samples\ud** directory. The sample program is “ud_grid.c”. This sample is already included in the pre-made project “ud.ide” in the **\tern\186\samples\ud** directory.

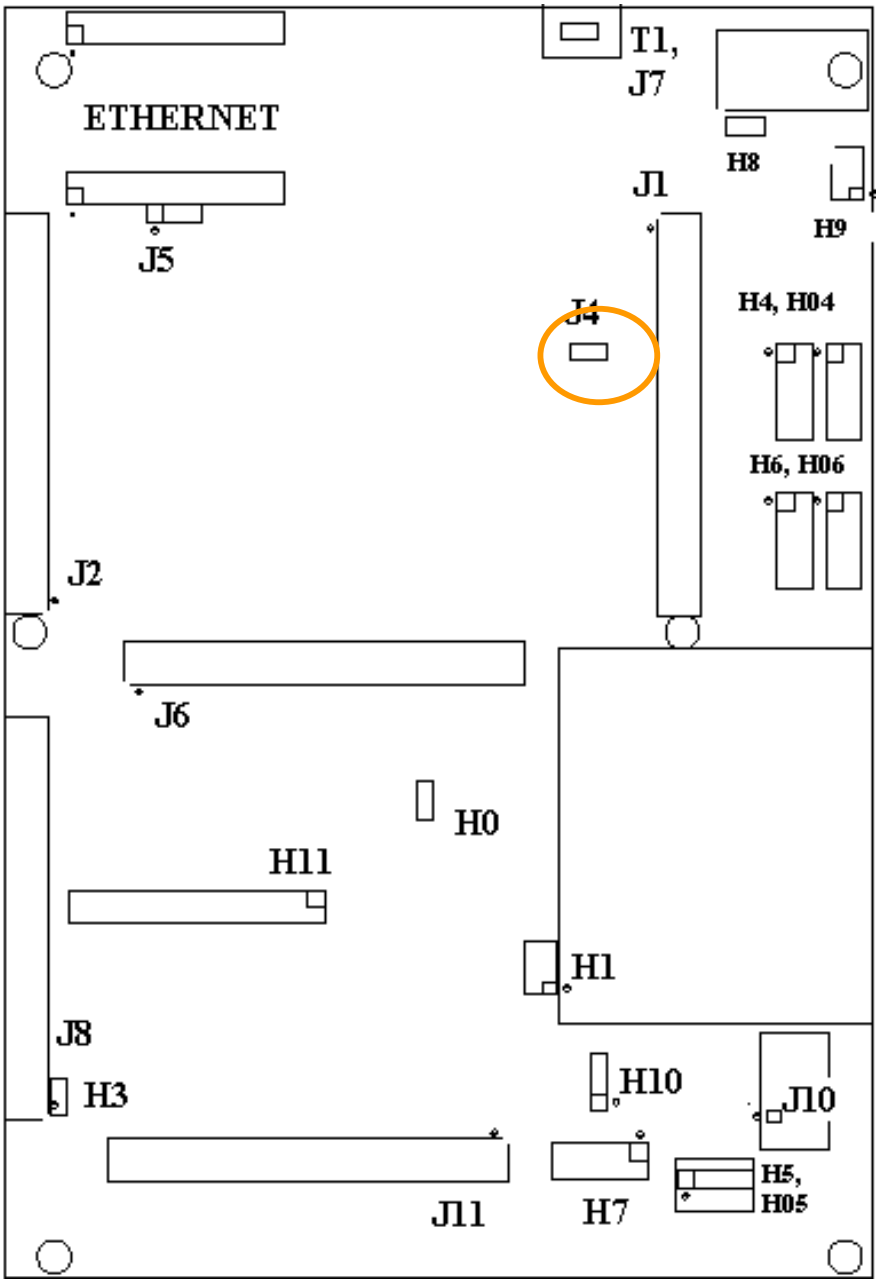
3.4.4 Supervisor with Watchdog Timer

The MAX691/LTC691 (U6) is a supervisor chip. With it installed, the ***UD*** 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 J4 of the ***UD***. 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 J4 jumper is on and the HWD pin is not accessed within this time-out period, the watchdog timer pulls the WDO pin low, which asserts /RESET. This automatic assertion of /RESET may recover the application program if something is wrong. After the ***UD*** 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 J4 jumper is off, which disables the watchdog timer.

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



J4 – Watchdog Header

Battery Backup Protection

The backup battery protection protects data stored in the SRAM and RTC. The battery-switch-over circuit compares VCC to VBAT (+3 V lithium battery positive pin), and connects whichever is higher to the VRAM (power for SRAM and RTC). Thus, the SRAM and the real-time clock RTC72423 are backed up. In normal use, the lithium battery should last about 3-5 years without external power being supplied. When the external power is on, the battery-switch-over circuit will select the VCC to connect to the VRAM.

3.4.5 EEPROM

A serial EEPROM of 512 bytes (24C04), or 2K bytes (24C16) can be installed in U4. The ***UD*** uses the P12=SCL (serial clock) and P11=SDA (serial data) to interface with the EEPROM. The EEPROM can be used to store important data such as a node address, calibration coefficients, and configuration codes. It typically has 1,000,000 erase/write cycles. The data retention is more than 40 years. EEPROM can be read and written by simply calling the functions **ee_rd()** and **ee_wr()**.

A range of lower addresses in the EEPROM is reserved for TERN use. Details regarding which addresses are reserved, and for what purpose, can be found in Appendix C of this manual.

3.4.6 Programmable Peripheral Interface (82C55A)

Two PPI chips(U5 and U33) are included on the UD.

The 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.

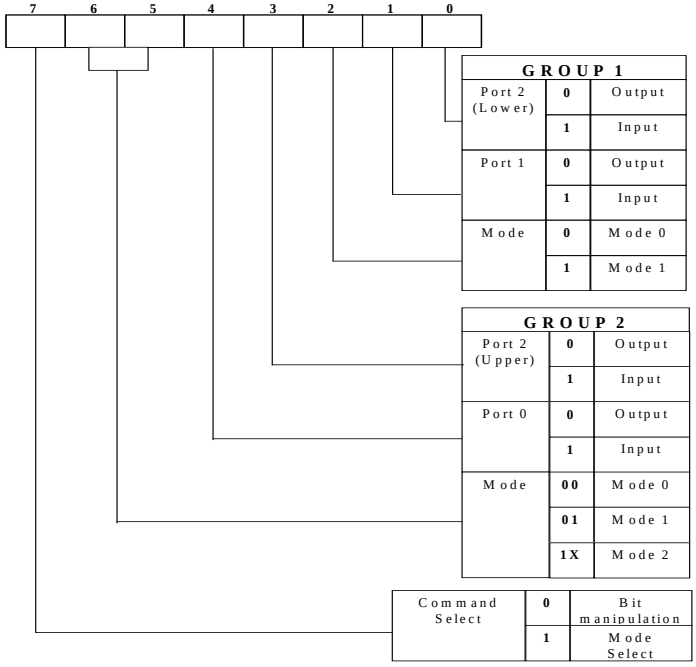


Figure 3.3 Mode Select Command Word

The UD maps U5 /PPI, at base I/O address of 0x0620, and U33 /PP at based address of 0x640.

Use U5 PPI as programming example,

the Command Register = 0x0626; Port 0 = 0x0620; Port 1 = 0x0622; and Port 2 = 0x0624.

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

```
outportb(0x0626,0x80); /* Mode 0 all output selection. */
outportb(0x0620,0x55); /* Sets port 0 to alternating high/low I/O pins. */
outportb(0x0622,0x55); /* Sets port 1 to alternating high/low I/O pins. */
outportb(0x0624,0x55); /* Sets port 2 to alternating high/low I/O pins. */
```

To set all ports to input mode:

```
outportb(0x0626,0x9f);     /* Mode 0 all input selection. */
```

You may read the ports with:

```
inportb(0x620); /* Port 0 */
inportb(0x622); /* Port 1 */
inportb(0x624); /* Port 2 */
```

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

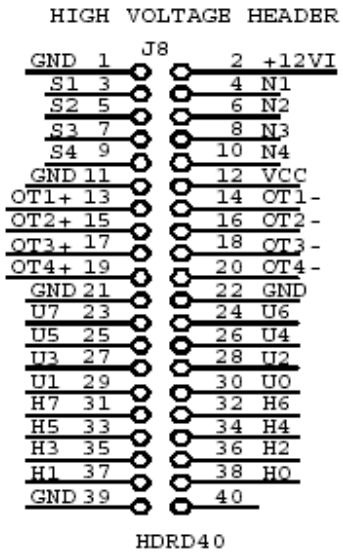
3.4.7 Real-time Clock (RTC72423)

If installed, the real-time clock RTC72423 (EPSON, U10) 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()*** (see Appendix D and the Software chapter for details).

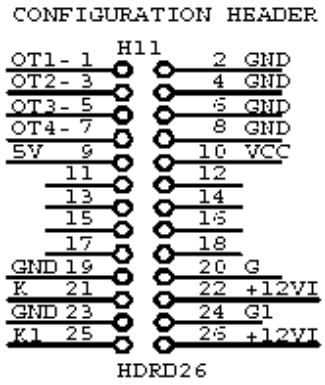
It is also possible to configure the real-time clock to raise an output line attached to an external interrupt, at 1/64 second, 1 second, 1 minute, or 1 hour intervals. This can be used in a time-driven application, or the **VOFF** signal can be used to turn on/off the controller using an external switching power supply, LM2575. An example of a program showing a similar application can be found in ***tern\ae\samples\ve\poweroff.c***.

3.4.8 Optocouplers

4 opto-couplers can be installed in U27on the UD, providing high voltage opto-isolation capability. They can be used for digitals inputs, relay contact monitoring, or power line monitoring. Typical ON time is 3µs, while typical OFF time is 5µs. All opto-coupler inputs are routed to the high voltage header J8. Both OTx+ and OTx- are available. User can provide both positive and negative signals to each opto-coupler.



User can also tie all negative inputs(OTx-) to GND, via the hardware configuration header H11.



The TTL status signals output from opto-couplers are routed to PPI(U33) L20-L23. So the PPI(U33) port2 lower nibble must be programmed as inputs in order to read opto-coupler status.

The opto-couplers (PS2701-4) are arranged in 4 opto-isolators per package. The data sheet for the opto-coupler package can be found in **\tern_docs\parts\ps2701.pdf**, on TERN CD-ROM.

3.4.9 Reed Relays

There are 4 Reed Relays can be installed on the ***UD***. The Reed Relay offers high speed switching compared to electromechanical relays, a specification of 200 V, maximum 1 Amp carry current, 0.5 Amp switching, and 100 million times operation. The relays are driven by U33 PPI L24-L27. We need to program the U33 PPI port2(P27-P24) as outputs. All relay contacts(S1-4 and N1-4) are routed to J8. See **tern\186\samples\UD\ud_relay.c** and **\tern_docs\parts\relay9007.pdf** for details.

One relay(S1) can be used to control the TFT backlighting. Install a jumper on H3 will route the +12VI power to relay contact, N1. While the relay is on, the N1=S1, so the TFT/LCD backlighting connecting at H5(or H05) will be powered on.

3.4.10 Hardware configurable Protective high voltage inputs or outputs

In order to support high voltage digital signal input up to 30V, Darlington Transistor Arrays (ULN2003A) can be installed in U28, and U29. 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.

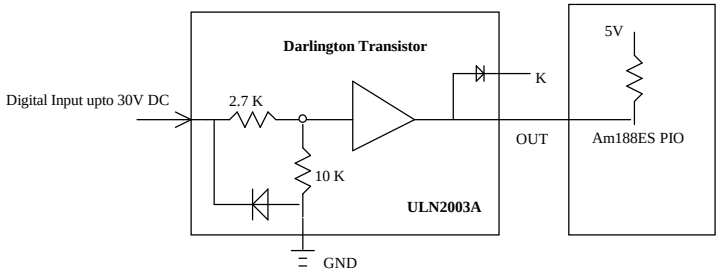


Figure 3.4 Darlington Transistors used as Protective High Voltage Inputs.

U28 and U29 may be set as input or output. By factory default, both are output. The input and output orientation for U28 and U29 is illustrated below. Follow these illustrations carefully to prevent damage to the chips. In addition, the ULN2003 chips may be replaced with a resistor pack to provide digital inputs or outputs to the terminal blocks.

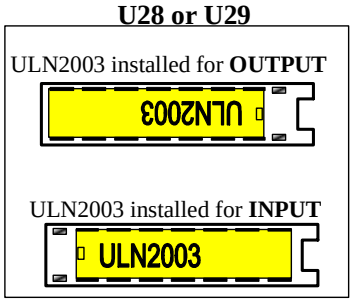
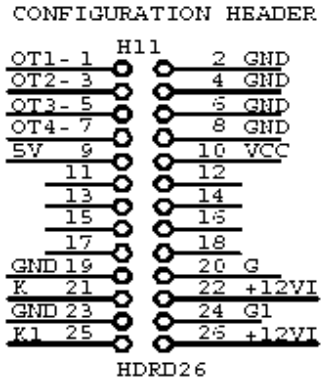


Figure 3.5 Locations of user configurable Darlington Transistor Arrays.

ULN2003A 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. By default, ULN2003 are installed in U28 and U29 to provide high-voltage sinking outputs. Optional high-voltage sourcing output chips (UDS2982) can be installed.

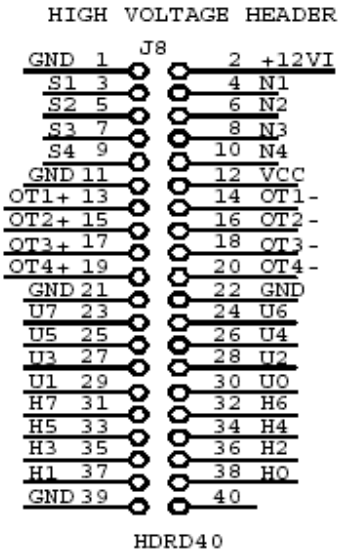
H11 is designed to be a hardware configuration header:



Jumper short PIN19=PIN20 and PIN23=PIN24 to force G=GND and G1=GND to support ULN2003 as sink drivers.
In order to use UDS2982 sourcing drivers in U28 and U29:
Short PIN20=PIN22 and PIN24=PIN26 to force G=+12VI and G1=+12VI
Short PIN19=PIN21 and PIN23=PIN25 to force K=K1=GND.

User may provide external sourcing voltage at G and G1, and not short G, G1 to +12VI, if do not want to use +12VI as sourcing power.

High voltage I/Os are routed to J8



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).
ULN2003A is a *sinking* driver. An example of typical application wiring is shown below.

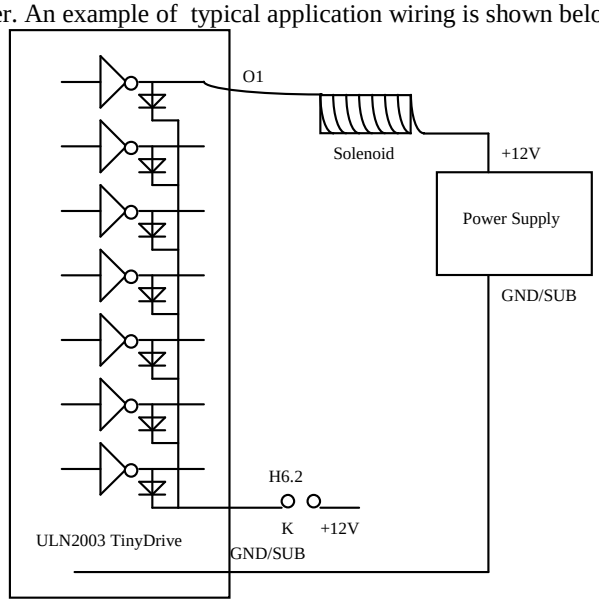


Figure 3.6 Drive inductive load with high voltage/current drivers.

3.4.11 USB

The ***UD*** integrates an Embedded USB Host controller(Vinculum VNC1L, FTDI). Not only is it able to habdle the USB Host Interface, and data transfer functions but owing to the inbuilt MCU and embedded Flash memory, Vinculum can encapsulate the USB device classes as well. When interfacing to mass storage devices, such as USB Flash drives, Vinculum also transparently handles the FAT File structure communicating via parallel FIFO interface with a simple command set. The onboard hardware fully handles USB stack processing, and provides for high-speed bi-directional 8-bit parallel communication. The hardware interface includes 384 bytes of FIFO transmit buffer, and 128 bytes of FIFO for the receiving buffer, making this an ideal low-overhead solution for all embedded

applications. No USB specific firmware programming is required on the controller side. The USB interface is seen as a transparent *Parallel FIFO* buffer tasked with transferring data back and forth with the remote host. The only control signals needed are "ready to transmit" and "data received" signals.

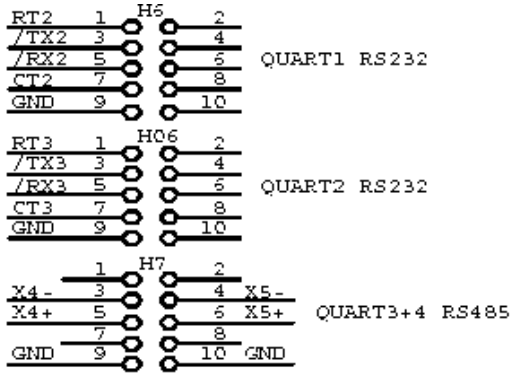
Two Host USB ports are provided on the ***UD.*** Port 1 (upper) can interface to a USB keyboard/mouse. Port 2 (lower) supports a USB Flash Disk. Simple commands can handle FAT file system applications. No USB specific firmware programming is required on the controller side. This is already taken care of in factory. Signal AC6 is jumpered to GND (H10.2=H10.3), while AC5 is pulled up to support *Parallel FIFO* operation. H1 is designed to be a VNC1L USB chip programming port, while H10.2=H10.1.
For more detailed information regarding this pre-loaded firmware, see **DS_VNC1L_FW_VDAP.pdf** in the **Tern_docs\parts\USB** directory (on any TERN CD).

3.4.12 100 MHz BaseT Ethernet

An WizNet™ Fast Ethernet Module can be installed to provide 100M Base-T network connectivity. This Ethernet module has a hardware LSI TCP/IP stack. It implements TCP/IP, UDP, ICMP and ARP in hardware, supporting internet protocol DLC and MAC. It has 16KB internal transmit and receiving buffer which is mapped into host processor's direct memory. The host can access the buffer via high speed DMA transfers. The hardware Ethernet module releases internet connectivity and protocol processing from the host processor. It supports 4 independent stack connections simultaneously at a 4Mbps protocol processing speed. An RJ45 8-pin connector is on-board for connecting to 10/100 Base-T Ethernet network. A software library is available for Ethernet connectivity.
Figure 3.4 (above) shows the location of the Ethernet module. See samples **httpd_fs.c** and **tcp_echo.c** in **\tern\186\samples\i2chip** directory for software details. These samples are also prebuilt into the **i2chip.ide** project, available in this same directory. Use the TERN_EE definition in Local Options>Defines for software compatibility.

3.4.13 QUAD UART

A QUAD UART (TL16C754B, U20) with 3.6864 MHz crystal can be installed to provide additional 4 serial ports. Two ports are supported with RS232 drivers for 232-level communication on header H6 and H06. Other two ports use RS485 drivers. Both RS485 ports are on header H7.



See sample **ud_echo.c** in the **\tern\186\samples\UD** directory for RS232 echo and RS485 transmit code.

3.4.14 CompactFlash Interface

By utilizing the compact flash interface on the ***UD***, users can easily add widely used 50-pin CF standard mass data storage cards to their embedded application. TERN software supports Linear Block Address mode, and 16-bit FAT flash file system. Users can write files to the CompactFlash card or read files from the CompactFlash card. Users can also transfer files to a PC via the CF reader port.

CF cards can also be used as a means to store images and data to be displayed onto the TFT/LCD. This allows users to have access to unlimited images to be used in an application in conjunction with the LCD. As discussed above, the AM186ES supports DMA to allow images/data to be transferred directly to the image buffer for increased speed.

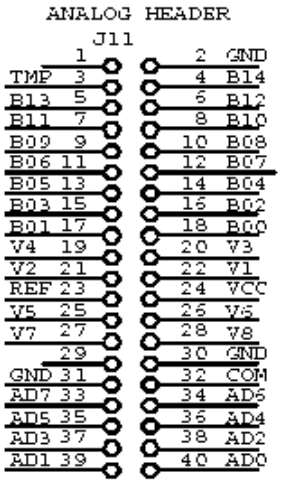
Sample code and function prototypes are available to assist in creating applications which use the file system to access the CF. Refer to the target **\tern\186\samples\UD\fs_cmds1.axe**. This sample uses the source code **\tern\186\samples\flashcore\fs_cmds1.c**. Also, for a complete listing of file system function prototypes and data types, refer to the header files "fileio.h" and "filegeo.h" found the **\tern\186\include** directory.

3.4.15 ADS8344 16-bit ADC

The ADS8344(U35) is an 8 channel, 16-bit sampling analog-to-digital converter with a synchronous serial interface. Input voltage range goes from 0V to 5V. The precision reference (LT1019 or REF02) installed at location U37 provides the 5V reference voltage ADS8344. Three control lines drive the ADS8344; /CS = AD12, CLK = CK, and DIN = DI.

Refer to **\tern\186\samples\ud\ud_ad16.c** for sample code. The effective maximum sampling rate is 10KHz.

The ADS8344 can support 8 single-ended inputs or 4 differential inputs. By default TERN software drivers use 8 single ended inputs. This mode can be changed via the control byte.



All 16-bit analog inputs are routed to the J11 Analog Header.

3.4.16 16-bit, 8-channel DAC(LTC2600)

The LTC2600(U36) 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 compatible serial interface and has an output range of 0-REF volts, making 1 LSB equal to REF/65535 V. The 5V reference voltage is provided by U37 LT1019/REF02.

Eight DAC outputs are routed to the Analog Header(J11).

Refer to the sample code, **\tern\186\samples\UD\UD_da16.c** for an example on driving the DAC. Refer to the DAC data sheet for additional specifications; **\tern_docs\parts\ltc2600.pdf.**

3.4.17 24-bit, 16-channel ADC(LTC2448)

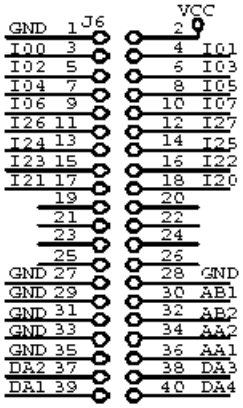
A 24-bit LTC2448 sigma-delta ADC can be installed. The LTC2448 chip offers 8 ch. differential or 16 ch. single-ended input channels. Variable speed/resolution settings can be configured. A peak single-channel output rate of 5 KHz can be achieved.

The LTC2448 switches the analog input to a 2 pf capacitor at 1.8MHz with an equivalent input resistance of 110K ohm. The ADC works well directly with strain gages, current shunts, RTDs, resistive sensors, and 4-20mA current loop sensors. The ADC can also work well directly with thermocouples in the differential mode. A precision reference with an internal temperature sensor (LT1019/REF02, 5V) can provide local temperature measurement for thermocouple applications. This 5V reference will grant a 0-2.5V analog input range per channel.

Inputs are routed directly to the Analog Header(J11). It should be noted that J11 pin 3=TMP corresponds to LTC2448 analog input B15, which is tied to the temperature pin on REF02 chip. This input cannot be used regularly if the temperature pin is still connected. The TMP pin can be lift out the U37 socket pin 6, if user wants to use all 16 ADC inputs. The software source sample code is at **c:\tern\186\samples\UD\UD_ad24.c**.

3.4.18 High Speed 16-bit ADC(AD7655)

The unique 16-bit parallel ADC (AD7655, 0-5V) supports ultra high-speed (1 MHz conversion rate) analog signal acquisition. The AD7655 (U12) contains two low noise, high bandwidth track-and-hold amplifiers that allow *simultaneous* sampling on two channels. Each track-and hold amplifier has a multiplexer in front to provide a total of 4 channels analog inputs. The parallel ADC achieves very high throughput by requiring only two CPU I/O operations (one start, one read) to complete a 16-bit ADC reading.
With a precision external 2.5V reference(U0, LT1009), the AD7655 accepts 0-5V analog inputs at 16-bit resolution, yielding 65536 counts/5000mV = 13 LSB/ mV.
The 4 high speed analog inputs(AA1, AA2, AB1, AB2) are available on J6.



See sample program **\tern\186\samples\ud\ud_ad.c** for details on reading the ADC. Refer to the data sheet for additional specifications; **\tern_docs\parts\ad7655.pdf**, from the root of the TERN installation CD-ROM.

3.4.19 DAC7625, 300KHz 12-bit DAC

The DAC7625(U11) is a parallel 12-bit D/A converter. This device includes 4 voltage output channels with an output range of 0-2.5V. It accepts 12-bit parallel input data and has double-buffered DAC input logic.

The DAC7625 has an average settling time of 5 µs, with a maximum settling time of 10µs. Refer to the sample program **ud_da.c** in the **\tern\186\samples\ud** directory for an example.

The 4 high speed analog outputs(DA1, DA2, DA3, DA4) are available on J6.

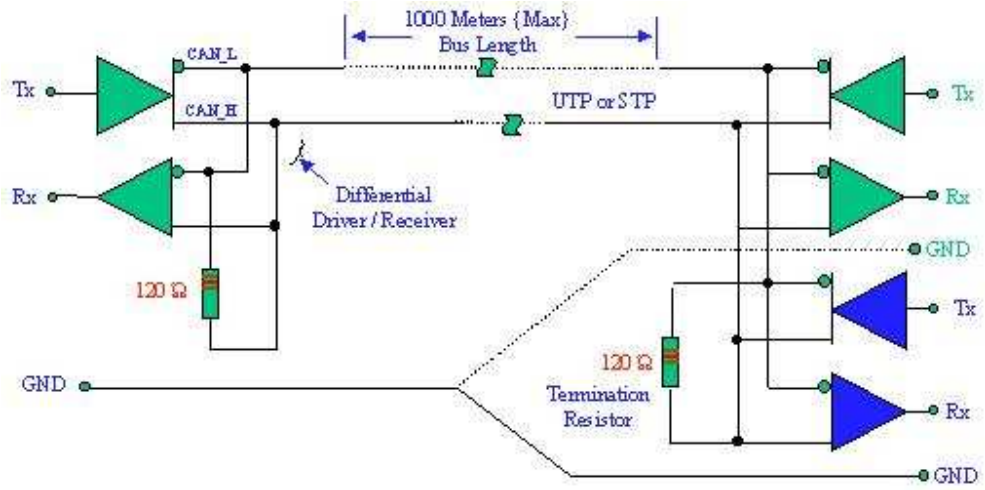
3.4.20 TFT Support

A Color QVGA TFT (320x240 pixels, 5.7") display can be installed. Aluminum Bezel is available. This TFT is installed on the back side of the ***UD***.

3.4.21 CAN(Controller Area Network)

The U-Drive can support an on-board Controller Area Network (CAN) controller(SJA1000, Philips). It supports network baud rates up to 1M-bit per second. Software drivers allow access to all CAN controller registers, as well as a buffering software layer.

The CAN bus is a balanced (differential) 2-wire interface running over either a Shielded Twisted Pair (STP), Un-shielded Twisted Pair (UTP), or Ribbon cable.

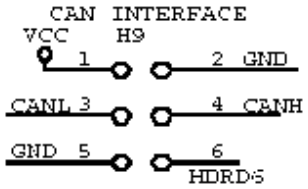


CAN Bus Electrical Interface Circuit

A number of different data rates are defined, with 1Mbps (Bits per second) being the top end, and 10kbps the minimum rate. Cable length depends on the data rate used. Normally all the devices in a system transfer uniform and fixed bit-rates. The maximum line length is 1Km, 40 meters at 1Mbps. Termination resistors are used at each end of the cable. The worst-case transmission time of an 8-byte frame with an 11-bit identifier is 134 bit times (that's 134 microseconds at the maximum baud rate of 1Mbits/sec).

The CAN Bus interface uses an asynchronous transmission scheme controlled by start and stop bits at the beginning and end of each character. This interface is used, employing serial binary interchange. Information is passed from transmitters to receivers in a data frame. The data frame is composed of an Arbitration field, Control field, Data field, CRC field, ACK field. The frame begins with a 'Start of frame' [SOF], and ends with an 'End of frame' [EOF] space. The data field may be from 0 to 8 bits.

The CAN bus pinout on H9 is shown below. It is a compact 0.1" spacing 3x2 pin header. User can use a IDE10-DB9 flat cable from TERN to connect the U-Drive CAN signals to an external standard DB9 CAN connector in the field.



The Application for CAN bus includes automotive and industrial field bus.
A low speed CAN bus may be employed to operate window and seat controls in a vehicle. A high speed CAN bus may be employed for engine management, brake control, Engine Sensors, and Anti-Skid Systems.

3.4.22 Power Supplies

The ***UD*** board without TFT panel can be powered by 2 ways:

1) Regulated external 5V DC power via J2.39=VCC and J2.40=GND, or J1.1=VCC and J1.2=GND..

2) Unregulated 9V to 12V DC power via two pin screw terminals(T1) while a 5V linear regulator(LM7805, U00) or 9V to 30V unregulated DC with an optional switching regulator(LM2575) is installed. There is a polarity protection diode installed for the screw terminal input DC power. The LM7805 is rated for 1A current, and can take as high as 35V. However, due to the linear regulation, all the input voltage has to drop to 5V, if the voltage drop with the current (300+ mA) is generating a lot of heat. Using the switching regulator, much less heat is generated.

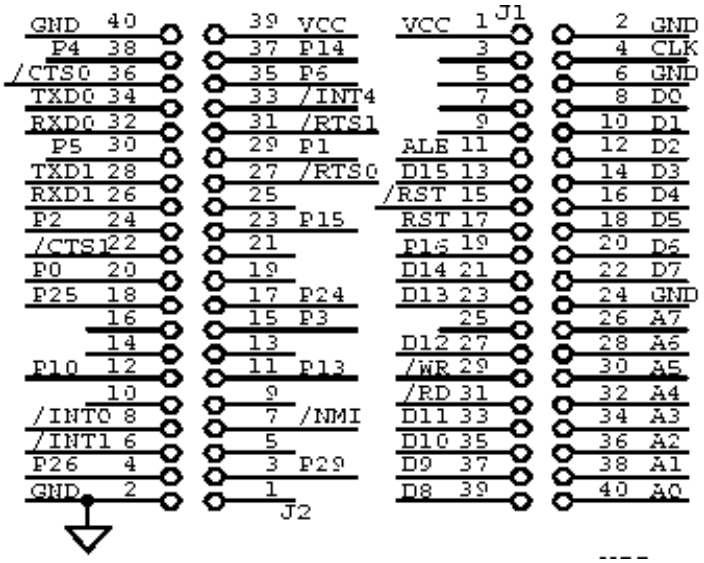
If using the Reed Relay(See chapter on Reed Relays) to control the TFT backlighting, the DC power input to both the UD and TFT(with LED backlighting) must in 9-11V range. The TFT backlighting can take 9V to 11V. The higher the DC power voltage, the brighter the LED backlighting.

The ***UD*** also requires regulated 3.3V DC power for the Ethernet, which is already taken care of on the 3.3V (U14) regulator.

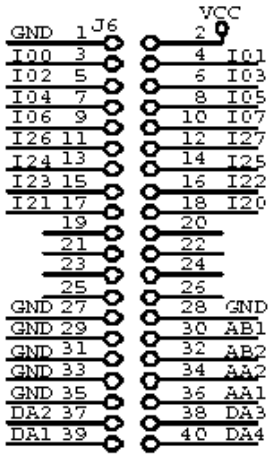
3.5 Headers and Connectors

3.5.1 Expansion Headers J1 and J2

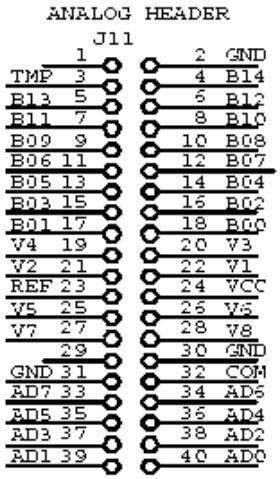
There are two 20x2 0.1 spacing headers for expansion. Most signals are directly routed to the Am186ES processor. These signals are 5V only, and any out-of-range voltages will most likely damage the board.



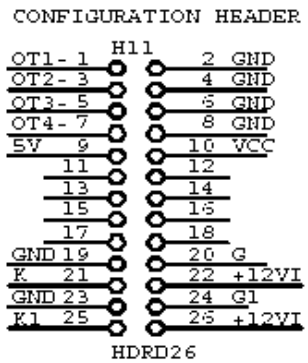
3.5.2 Connector J6, AD7655, DAC7625, and PPI



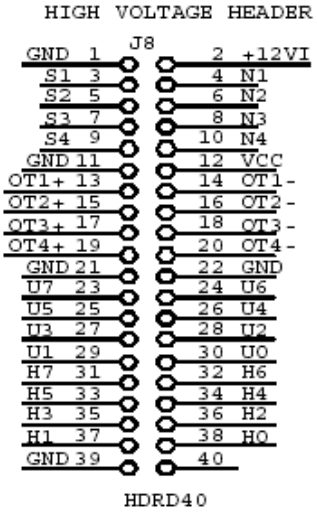
3.5.3 Analog Header J11 for ADS8344, LT2448, and LT2600 DAC outputs



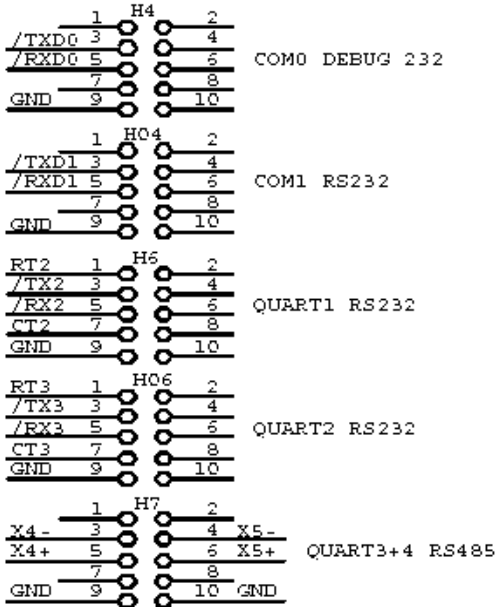
3.5.4 Hardware Configuration Header H11



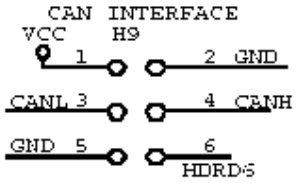
3.5.5 High Voltage Header J8



3.5.6 UART RS232 and RS485 Headers



3.5.7 CAN Interface H9



3.5.8 U-Drive Jumper/Header Summary

Connector	Default Setting	Usage
J1(20x2)		Address, Data, and Control bus signals for expansion
J2(20x2)	PIN38=PIN40	CPU PIO lines, STEP2 jumper on PIO4
J4(2x1)		Watchdog Enable
J5(3x1)	PIN1=PIN2	5V or 3.3V CPU select
J6(20x2)		PPI(U5), AD7655, DA7625
J7(2x1)		DC power input
J8(20x1)		High voltage I/O, Reed relays, Opto-couplers
J10(4x2)		Host USB Port1 and Port2
J11(20x2)		Analog I/O, LT2448, ADS7644, DAC2600
H1(3x2)		Host USB VNC1L firmware programming port
H2(4x1)		Touch Screen Flex cable connector
H3(2x1)		Reed Relay N1=+12VI
H4(5x2)		COM0 for RS232 DEBUG
H04(5x2)		COM1 RS232
H5(T2)		Screw terminal for TFT Backlighting
H05(4x1)		Header for TFT backlighting
H6(5x2)		QUART (Tx2, Rx2) RS232
H06(5x2)		QUART (Tx3, Rx3) RS232
H7(5x2)		QUART (X4, X5) RS485
H8(2x1)		VOFF=0 will shut off the Switching Regulator
H9(3x2)		CAN Interface
H10(3x1)		USB VNC1L Program(PIN1=PIN2) or RUN(PIN2=PIN3)
H11(13x2)	PIN 19=20, 23=24	Hardware configuration

Chapter 4: Software

Please refer to the Technical Manual of the "C/C++ Development Kit for TERN 16-bit Embedded Microcontrollers" for details on debugging and programming tools.

For details regarding software function prototypes and sample files demonstrating their use, please refer to the Software Glossary in Appendix C.

Guidelines, awareness, and problems in an interrupt driven environment
Although the C/C++ Development Kit provides a simple, low cost solution to application engineers, some guidelines must be followed. If they are not followed, you may experience system crashes, PC hang-ups, and other problems.

The debugging of interrupt handlers with the Remote Debugger can be a challenge. It is possible to debug an interrupt handler, but there is a risk of experiencing problems. Most problems occur in multi-interrupt-driven situations. Because the remote kernel running on the controller is interrupt-driven, it demands interrupt services from the CPU. If an application program enables interrupt and occupies the interrupt controller for longer than the remote debugger can accept, the debugger will time-out. As a result, your PC may hang-up. In extreme cases, a power reset may be required to restart your PC.

For your reference, be aware that our system is remote kernel interrupt-driven for debugging.

The run-time environment on TERN controllers consists of an I/O address space and a memory address space. I/O address space ranges from **0x0000** to **0xffff**, or 64 KB. Memory address space ranges from **0x00000** to **0xfffff** in real-mode, or 1 MB. These are accessed differently, and not all addresses can be translated and handled correctly by hardware. I/O and memory mappings are done in software to define how translations are implemented by the hardware. Implicit accesses to I/O and memory address space occur throughout your program from TERN libraries as well as simple memory accesses to either code or global and stack data. You can, however, explicitly access any address in I/O or memory space, and you will probably need to do so in order to access processor registers and on-board peripheral components (which often reside in I/O space) or non-mapped memory.

This is done with four different sets of similar functions, described below.

poke/pokeb
Arguments: unsigned int segment, unsigned int offset, unsigned int/unsigned char data
Return value: none

These standard C functions are used to place specified data at any memory space location. The **segment** argument is left shifted by four and added to the **offset** argument to indicate the 20-bit address within memory space. **poke** is used for writing 16 bits at a time, and **pokeb** is used for writing 8 bits.

The process of placing data into memory space means that the appropriate address and data are placed on the address and data-bus, and any memory-space mappings in place for this particular range of memory will be used to activate appropriate chip-select lines and the corresponding hardware component responsible for handling this data.

peek/peekb
Arguments: unsigned int segment, unsigned int offset
Return value: unsigned int/unsigned char data

These functions retrieve the data for a specified address in memory space. Once again, the **segment** address is shifted left by four bits and added to the **offset** to find the 20-bit address. This address is then output over the address bus, and the hardware component mapped to that address should return either an 8-bit or 16-bit value over the data bus. If there is no component mapped to that address, this function will return random garbage values every time you try to peek into that address.

outport/outportb
Arguments: unsigned int address, unsigned int/unsigned char data
Return value: none

This function is used to place the **data** into the appropriate **address** in I/O space. It is used most often when working with processor registers that are mapped into I/O space and must be accessed using either one of these functions. This is also the function used in most cases when dealing with user-configured peripheral components.

When dealing with processor registers, be sure to use the correct function. Use **outport** if you are dealing with a 16-bit register.

inport/inportb
Arguments: unsigned int address
Return value: unsigned int/unsigned char data

This function can be used to retrieve data from components in I/O space. You will find that most hardware options added to TERN controllers are mapped into I/O space, since memory space is valuable and is reserved for uses related to the code and data. Using I/O mappings, the address is output over the address bus, and the returned 16 or 8-bit value is the return value.

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

4.1 AE.LIB

AE.LIB is a C library for basic ***UD*** 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	timer/counter, Watchdog
SER0.H	Internal serial port 0
SER1.H	Internal serial port 1
AEEE.H	on-board EEPROM

4.2 Functions in AE.OBJ

4.2.1 U-Drive Initialization

ae_init

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

There are certain default pin modes and interrupt settings you might wish to change. With that in mind, the basic effects of **ae_init** are described below. For details regarding register use, you will want to refer to the AMD Am186ES Microcontroller User's manual.

Initialize the upper chip select to support the default ROM. The CPU registers are configured such that:

Address space for the ROM is from 0x80000-0xfffff (to map MemCard I/O window)
512K ROM Block size operation.
Three wait state operation (allowing it to support up to 120 ns ROMs). With 70 ns ROMs, this can actually be set to zero wait state if you require increased performance (at a risk of stability in noisy environments). For details, see the UMCS (Upper Memory Chip Select Register) reference in the processor User's manual.

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

Initialize LCS (*Lower Chip Select*) for use with the SRAM. It is configured so that:

Address space starts 0x00000, with a maximum of 512K RAM.
Three wait state operation. Reducing this value can improve performance.
Disables PSRAM, and disables need for external ready.

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

Initialize MMCS and MPCS so that **MCS0** and **PCS0**-**PCS6** (except for PCS4) are configured so:

MCS0 is mapped also to a 256K window at 0x80000. If used with MemCard, this chip select line is used for the I/O window.
Sets up **PCS5-6** lines as chip-select lines, with three wait state operation.

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

Initialize PACS so that **PCS0**-**PCS3** are configured so that:

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

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

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

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

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

void io_wait
Arguments: char wait
Return value: none.

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

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

4.2.2 External Interrupt Initialization

There are up to eight external interrupt sources on the ***UD***, consisting of seven maskable interrupt pins (**INT6-INT0**) and one non-maskable interrupt (**NMI**). There are also an additional eight internal interrupt sources not connected to the external pins, consisting of three timers, two DMA channels, both asynchronous serial ports, and the **NMI** from the watchdog timer. For a detailed discussion involving the ICUs, the user should refer to Chapter 7 of the AMD Am186ES Microcontroller User's Manual.

TERN provides functions to enable/disable all of the eight external interrupts. The user can call any of the interrupt init functions listed below for this purpose. The first argument indicates whether the particular interrupt should be enabled, and the second is a function pointer to an appropriate interrupt service routine that should be used to handle the interrupt. The TERN libraries will set up the interrupt vectors correctly for the specified external interrupt line.

At the end of interrupt handlers, the appropriate in-service bit for the IR signal currently being handled must be cleared. This can be done using the **Nonspecific EOI command**. At initialization time, interrupt priority was placed in **Fully Nested** mode. This means the current highest priority interrupt will be handled first, and a higher priority interrupt will interrupt any current interrupt handlers. So, if the user chooses to clear the in-service bit for the interrupt currently being handled, the interrupt service routine just needs to issue the nonspecific EOI command to clear the current highest priority IR.

To send the nonspecific EOI command, you need to write the **EOI** register word with 0x8000.

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

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

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

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

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

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

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

4.2.3 I/O Initialization

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

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

You should also confirm the PIO usage that is described above within **ae_init()**. During initialization, several lines are reserved for TERN usage and you should understand that these are not available for your application. There are several PIO lines that are used for other on-board purposes. These are all described in some detail in the Hardware chapter of this technical manual. For a detailed discussion toward the I/O ports, please refer to Chapter 11 of the AMD Am186ES User's Manual.

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

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

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

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

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

mode refers to one of four modes of operation.

- 0, High-impedance Input operation
- 1, Open-drain output operation
- 2, output
- 3, peripheral mode

unsigned int pio_rd:
Arguments: char port
Return value: byte indicating PIO status

Each bit of the returned 16-bit value indicates the current I/O value for the PIO pins in the selected port.

void pio_wr:
Arguments: char bit, char dat
Return value: none

Writes the passed in dat value (either 1/0) to the selected PIO.

4.2.4 Timer Units

The three timers present on the ***UD*** can be used for a variety of applications. All three timers run at 1/4 of the processor clock rate (10MHz based on 40MHz system clock, or one timer clock per 100ns), which determines the maximum resolution that can be obtained. Be aware that if you enter power save mode, that means the timers will operate at a reduced speed as well.

These timers are controlled and configured through a mode register which is specified using the software interfaces. The mode register is described in detail in chapter 8 of the AMD Am186ES User's Manual.

Pulse width demodulation is done by setting the PWD bit in the **SYSCON** register. Before doing this, you will want to specify your interrupt service routines, which are used whenever the incoming digital signal switches from high to low, and low to high. It is important to note the the interrupt latency generated by the ISRs that handle a signal transition will define the time resolution the user will be able to achieve.

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

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

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

The specific behavior that you might want to implement is described in detail in chapter 8 of the AMD Am186ES User's Manual.

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

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

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

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

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

4.2.5 Other library functions

On-board supervisor MAX691 or LTC691

The watchdog timer offered by the MAX691 or LTC691 offers an excellent way to monitor improper program execution. If the watchdog timer (**J5**) jumper is set, the function **hitwd()** must be called every 1.6 seconds of program execution. If this is not executed because of a run-time error, such as an infinite loop or stalled interrupt service routine, a hardware reset will occur.

void hitwd
Arguments: none
Return value: none

Resets the supervisor timer for another 1.6 seconds.

void led
Arguments: int ledd
Return value: none

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

Real-Time Clock

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

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

See sample ***ud_rtc.axe*** in *c:\tern\186\samples\ud\ud.pdl*

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 rtc16_rd
Arguments: TIM *r
Return value: int error_code

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

int rtc16_rds
Arguments: char* time_string
Return value: int error_code

This function places the current value of the real time clock as a printable ASCII string into the character array **time_string**. The array should have a size of 15 bytes. This function returns 0 on success and returns 1 in case of error, such as the clock failing to respond.

Void rtc16_init
Arguments: char* t
Return value: none

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

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

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

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

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

void delay0
Arguments: unsigned int t
Return value: none

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

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

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

void delay_ms
Arguments: unsigned int
Return value: none

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

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

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

void ae_reset

Arguments: none
Return value: none

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

4.3 Functions in SER0.OBJ/SER1.OBJ

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

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

Two asynchronous serial ports are integrated in the Am186ES CPU: SER0 and SER1. Both ports have baud rates based on the 40 MHz clock, and can operate at a maximum of 1/16 of that clock rate.

By default, SER0 is used by the DEBUG ROM for application download/debugging in Step One and Step Two. We will use SER1 as the example in the following discussion; any of the interface functions which are specific to SER1 can be easily changed into function calls for SER0. While selecting a serial port for use, please realize that some pins might be shared with other peripheral functions. This means that in certain limited cases, it might not be possible to use a certain serial port with other on-board controller functions. For details, you should see both chapter 10 of the Am186ES Microprocessor User's Manual and the schematic of the ***UD*** provided on the CD in the **tern_docs\schs** directory.

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

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

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

Table 4.1 Baud rate values

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

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

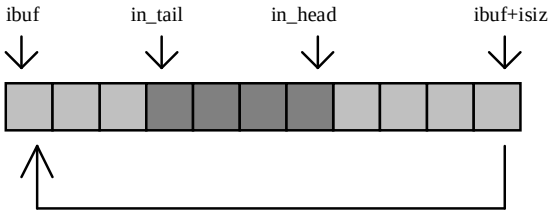


Figure 4.1 Circular ring input buffer

The input buffer (**ibuf**), buffer size (**isiz**), and baud rate (**baud**) are specified by the user with **`s1_init()`** with a default mode of 8-bit, 1 stop bit, no parity. After **`s1_init()`** you can set up a new mode with different numbers for data-bit, stop bit, or parity by directly accessing the Serial Port 0/1 Control Register (SP0CT/SP1CT) if necessary, as described in chapter 10 of the Am186ES manual for asynchronous serial ports.

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

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

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

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

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

Software Interface

Before using the serial ports, they must be initialized.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Miscellaneous Serial Communication Functions

One thing to be aware of in both transmission and receiving of data through the serial port is that TERN drivers only use the basic serial-port communication lines for transmitting and receiving data. Hardware flow control in the form of **CTS** (Clear-To-Send) and **RTS** (Ready-To-Send) is not implemented. There are, however, functions available that allow you to check and set the value of these I/O pins appropriate for whatever form of flow control you wish to implement. Before using these functions, you should once again be aware that the peripheral pin function you are using might not be selected as needed. For details, please refer to the Am186ES User's Manual.

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

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

Completing Serial Communications

After completing your serial communications, there are a few functions that can be used to reset default system resources.

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

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

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

The asynchronous serial I/O ports available on the Am186ES Processor have many other features that might be useful for your application. If you are truly interested in having more control, please read Chapter 10 of the AM186ES manual for a detailed discussion of other features available to you.

4.4 Functions in AEEE.OBJ

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

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

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

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

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

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

ee_rd
Arguments: int addr
Return value: int data

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

4.5 Analog-to-Digital Conversion

Two ADC chip can be installed on the ***UD***.

4.5.1 ADS8344 (16-bit, 8 channels)

The ADC unit provides 8 channels of analog inputs(0-5V) based on the 5V reference voltage supplied by LT1019(U37). For details regarding the hardware configuration, see the Hardware chapter.

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

unsigned int ud_ad8344 (c)
Arguments: unsigned char c
Return values: unsigned 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:

```
ud_ad8344(0); // Read from channel 0
chn_0_data = ud_ad8344(0); // Start the next conversion, retrieve value.
```

4.5.2 LT2448 (24-bit 16 channels)

Delta-Sigma ADC LTC2448

The LTC2448 ADC (U11) provides 16 channels of 0-2.5V analog single-ended (24 differential) inputs. The following functions will drive the 24-bit ADC. The order of functions given here should be followed in actual implementation.

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

void ad24 _rd(unsigned char* raw);

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

In code, the control byte is calculated this way:

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

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

ch_sel	On U34 chip	Header J11 pin
0	B00	18
1	B02	16
2	B04	14
3	B06	12
4	B08	10
5	B10	8
6	B12	6
7	B14	4
8	B01	17
9	B03	15
10	B05	13
11	B07	11
12	B09	9
13	B11	7
14	B13	5
15	B15(TMP)	3

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

For a sample file demonstrating the use of the ADC, please see **ud_ad24.c** in **`tern\186\samples\ud`**.

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

4.6 Digital-to-Analog Conversion

The LT2600 provides 8 channels of 16-bit digital to analog conversion. For software purposes, we must initialize the clock (SCK), data-in (SDI), and chip select (/DA):

void da_16 ()
Arguments: unsigned char mod, unsigned int dac
Return values: none

The argument **mod** selects the channel to write to. A value of 0x3[Y], where [Y]=0-7, corresponds to channels V1-V8 respectively. Alternatively, 0x3f corresponds to all channels.
The argument **dac** is ranged from 0 to 0xffff, on a 5-volt scale.

```
da_16(0, 0); // Output zero voltage at V1
da_16(0, 0xffff); // Output 5V voltage at V1
```

For sample DAC code, see **ud_da16.c** in the **\tern\186\samples\ud** directory.

4.7 QUART TL16C754B

The four UART ports from the QUART TL16C754B require separate driver code outside of *ae.lib*. The file **quart.c** in the **\tern\186\samples\ud** directory provides this code. A prebuilt project **ud.ide** in the same directory includes **ud_echo.c**, which is a sample of QUART software usage.
There is no library as of yet to enclose these drivers, so this file must be included in any project node using the QUART chip.

4.8 USB

The VNC1L chip using the Vinculum VDAP firmware is configured to provide 2 host USB ports. Port 1 is used to interfacing to a USB Keyboard or mouse, while Port 2 is used for USB Flash Disk interface.

The document concerning all details of this firmware is in the **\tern_docs\parts\USB** directory on the TERN CD, under the title **DS_VNC1L_FW_VDAP.pdf**.

Pages 10-13 of this document list out the entire firmware command set when communicating with the VNC1L, as well as the responses expected from the chip. In the **\tern\186\samples\ud** directory, **hd_ser1.c** can be used to talk to the VNC1L through a hyperterminal over serial port 1.

There are key bits to examine when transmiting/receiving data from the VNC1L.

Other samples include **keyboard.c** (USB Keyboard sample) and **usb_disk.c** (USB Flash disk sample). Both of which are located in **\tern\186\samples\UD**.

4.9 Controller-Area-Network (CAN) Interface

The U-Drive optionally provides the Philips SJA1000 stand-alone CAN controller. This controller allows the U-Drive to communicate over a Controller Area Network, a popular protocol and bus standard for microcontroller communication.

BACKGROUND
Controllers communicate over a CAN network using frames, at a specified baud rate. Controllers can send and receive equally on the CAN network, with the underlying chipset handling collision detection and basic buffering.

In simplified form, each transmitted frame consists primarily of:

- Recipient address (11-bits);
- Data bytes (0-8 bytes);
- Protocol information (CRC consistency, and other bits indicating frame properties).

A controller initializes the CAN chipset by defining the class of messages it wants to receive. This is done by defining an 8-bit address value as well as an 8-bit mask. The masked address value is used to compare to the highest 8-bits of all incoming frames; qualifying frames are received and inserted into a buffer for the application to handle. Unlike many other networking schemes, frames travelling on a CAN bus do not identify who the sender is, and does not necessarily indicate a specific recipient.

Transmitted and received packets are buffered both in the hardware chipset (up to 64 bytes), as well as the interrupt-driven TERN firmware drivers (buffer size defined by application).

TERN firmware drivers are configured to use the SJA1000 in BasicCAN mode only. More advanced features may be available by directly accessing the SJA1000's control registers. The datasheet for the SJA1000 may be found on the TERN development CD in the directory **\tern_docs\parts.**

SOFTWARE INTERFACE
The CAN driver software interface is shown in the header file: **\tern\186\include\can.h.**

The library file for the U-Drive implementation is at location: **\tern\186\lib\can.lib**, and **\tern\186\lib\large\can_l.lib**.

CAN messages are defined using this CanMsg structure (similar to SJA1000 hardware representation):

```
typedef struct _can_msg {
   UCHAR8 descriptor[2];
   UCHAR8 data[8];
} CanMsg;
```

The two-byte descriptor field consists of message ID (11 bits), Remote-Transmission-Request/RTR flag (1 bit), and Data Length Code/DLC value (4 bits). These fields can be accessed on a message using these macros defined in **can.h**:

```
SET_CAN_MSG_ID(msg, val)
READ_CAN_MSG_ID(msg)
SET_CAN_MSG_RTR(msg, val)
READ_CAN_MSG_RTR(msg)
SET_CAN_MSG_DLC(msg, val)
READ_CAN_MSG_DLC(msg)
```

TERN firmware drivers use a ring-buffer to store messages for transmit and receipt. The overall mechanism is similar to standard serial port implementation (see section 4.1). The best sample demonstrating these functions is: \tern\186\samples\cane\can_echo.c

int can_set_hw
Arguments: *unsigned char* ***board_type***
Return values: none

This function configures the CAN port according to the architecture of your board. This function should be called first, before any other CAN function is accessed. Available **board_type** values are defined in **can.h**. If this call is not accurate, the CAN port can not be accessed.

For the U-Drive, this call should read:

```
can_set_hw(BOARD_UD);
```

int can_init
Arguments: *unsigned char* ***baud,*** *CanMsg** ***inputBuf****, int* ***iSize****, CanMsg** ***outputBuf****, int* ***oSize****, unsigned char* ***address****, unsigned char* ***mask***
Return values: 0 for success, non-zero error code.

This function is used to initialize message buffer support for the CAN port.

Baud - specifies the baud rate to be used for communication; supported values are defined in **can.h**. These include: 1MHz, 500KHz, 250KHz, 125KHz, 100KHz, 50KHz, 20KHz, and 10KHz. *Note: at higher baud rates, termination resistors may be required on your TERN board for clean transmit and receive.*

`inputBuf, iSize` – these variables represent the ring-buffer allocated for receiving messages. **inputBuf** should be a CanMsg array, while **iSize** indicates the size of the array. TERN drivers will inject messages into this array on an interrupt-driven basis.

`outputBuf, oSize` – similar to above; these variables represent the ring-buffer allocated for buffering messages to be transmitted.

`address, mask` – these two byte values are used to determining which messages transmitted on the CAN network should be "received". For all messages, the **mask** value and the first 8-bits of the message address are AND'ed together, and then compared to the **address** value.

`void can_transceiver_enable`
`Arguments:` *`unsigned char`* ***`enable`***
`Return values:` none

After software drivers have been enabled, the CAN transceiver must still be enabled using a digital output pin. Once the transceiver is enabled, the port will be connected to the CAN bus, and able to transmit/receive messages.

On the U-Drive, the transceiver is controlled using PPI pin I17. If you're also using the PPI port in your own application, you will want to avoid calling this function, and instead access the pin directly:

```
outportb(PPI3,0x99);    // ppi mode 0, P1 outputs, other all input
outportb(PPI1,0x7F);    // P17/I17=RS low for CAN, tranceiver ON
```

`void can_hit`
`Arguments:` none
`Return value:` **non-zero** if packet received**, 0** if receive buffer is empty**.**

Use this function to determine whether a packet has been received and buffered. Call this function before calling can_get() to retrieve actual message.

`void can_get`
`Arguments:` *`CanMsg*`* ***`message`***
`Return value:` none

This function is used to retrieve a CAN message, after can_hit() has already been called. The argument should be a pointer to a separately allocated **CanMsg** variable. The next message in the receive buffer will be copied into this variable. *Note: Make sure'**message**' points to an allocated area of memory!*

`void can_put`
`Arguments:` *`CanMsg*`* ***`message`***
`Return values:` none.

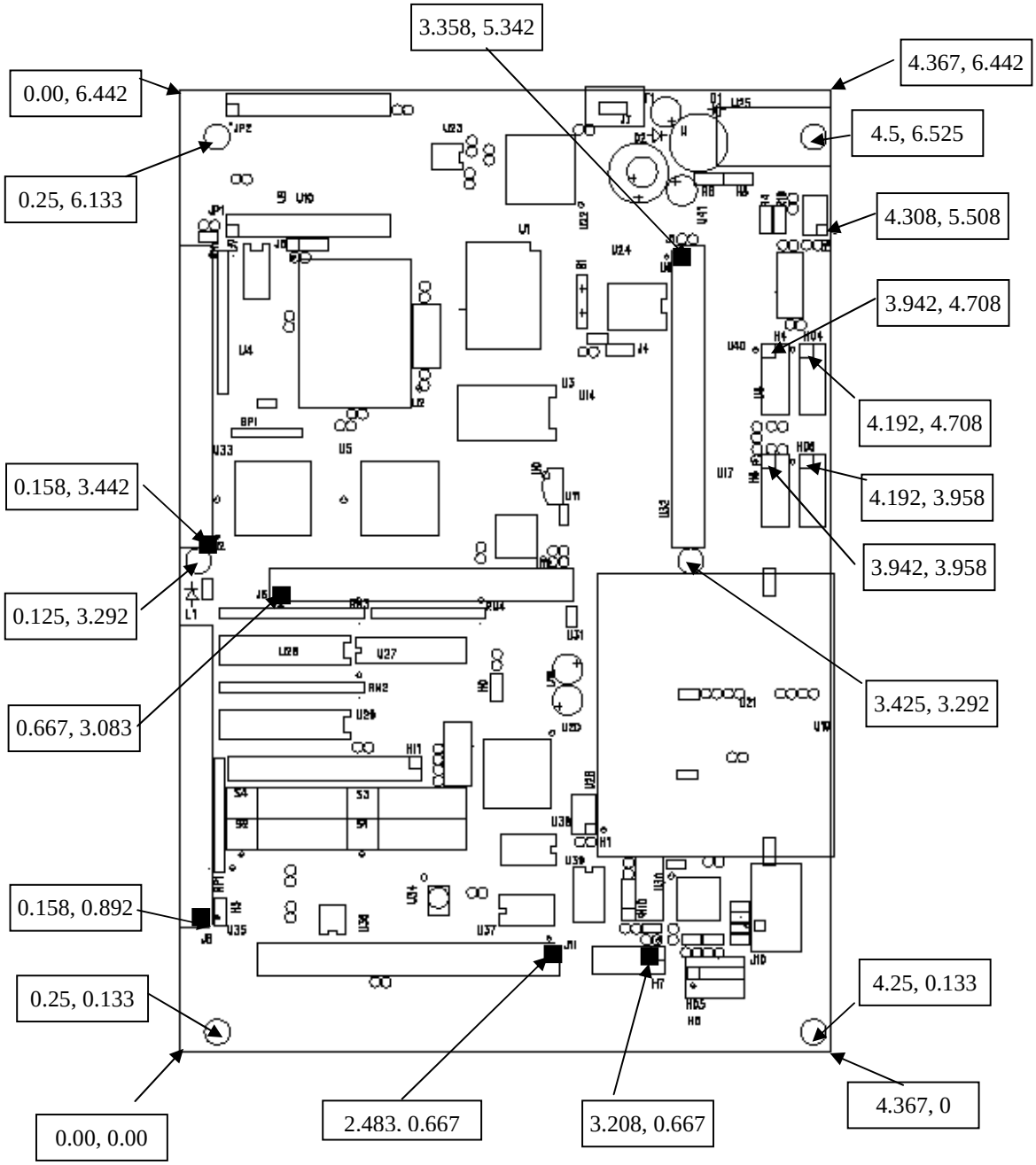
This function adds ***`message`*** to the transmit buffer. Messages in the transmit buffer are sent on a FIFO basis.

void can_flush
Arguments: none
Return values: none.

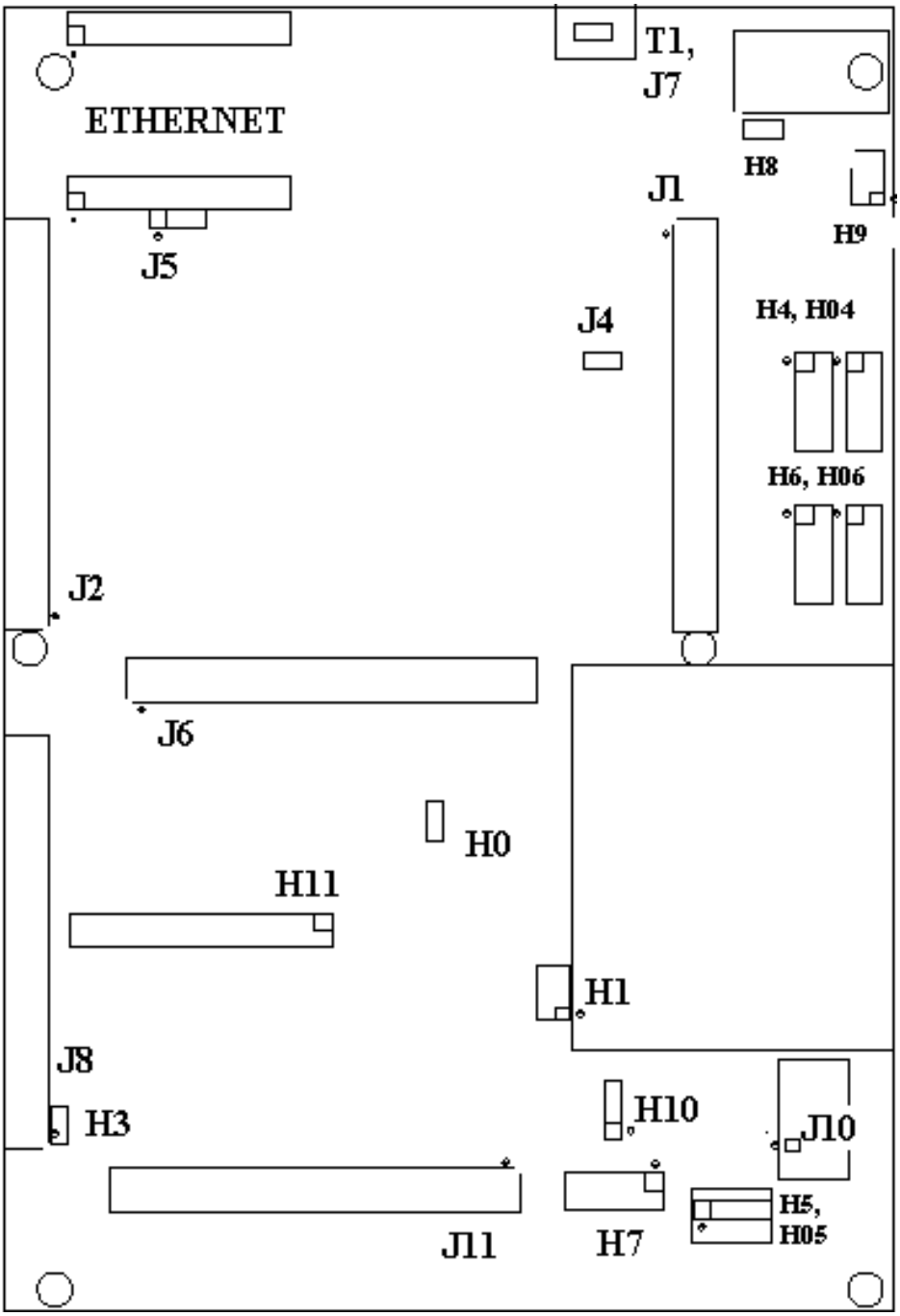
This function can be used to make sure the transmit buffer does not over-flow. It will not return until all currently buffered messages are fully transmitted, and the transmit buffer is completely empty.

Appendix A: U-Drive Mechanical Dimensions

All dimensions are in inches.



Appendix B: U-Drive Heaters



Appendix C: Software Glossary

The following is a glossary of library functions for ***UD***.

---

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 AM188 processor.

---

void delay_ms(int m) ae.h

Approximate microsecond delay. Does not use timer.

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

Reference: led.c

---

void led(int i) ae.h

Toggles P12 used for led.

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

Reference: led.c

---

void delay0(unsigned int t) ae.h

Approximate loop delay. Does not use timer.

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

Reference:

---

void pwr_save_en(int i) ae.h

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

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

Reference: ae_pwr.c

---

void clka_en(int i) ae.h

Enables signal CLK respectively for external peripheral use.

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

Reference:

---

void hitwd(void) ae.h

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

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

---

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

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

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

Reference: ae_pio.c

---

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

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

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

Reference: ae_pio.c

---

unsigned int pio_rd(char port) ae.h

Reads a 16 bit PIO port.

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

Reference: ae_pio.c

---

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

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

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

Reference: ae_ppi.c

---

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

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

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

Reference: ae_ppi.c

---

int inport(int portid) dos.h

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

```
Var:  portid – I/O address
```

Reference: ae_ppi.c

---

int inportb(int portid) dos.h

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

```
Var:  portid – I/O address
```

Reference: ae_ppi.c

---

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

Writes to the serial EEPROM.

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

Reference: ae_ee.c

---

int ee_rd(int addr) aeee.h

Reads from the serial EEPROM. Returns 8-bit data

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

Reference: ae_ee.c

---

void io_wait(char wait) ae.h

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

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

Reference:

---

void rtc16_init(unsigned char * time) c:\tern\186\samples\slc\slc_rtc.c

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: slc_rtc.c

---

int rtc16_rd(TIM *r) c:\tern\186\samples\slc\slc_rtc.c

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: slc_rtc.c

---

int rtc16_rds(char * time_string) c:\tern\186\samples\slc\slc_rtc.c

Reads from the real time clock in ASCII string form.

```
Var:  *time_string – 15 byte character array
String sequence is the following:
            time[0] = weekday
            time[1] = year1000
            time[2] = year100
            time[3] = year10
            time[4] = year1
            time[5] = mon10
            time[6] = mon1
            time[7] = day10
            time[8] = day1
            time[9] = hour10
            time[10] = hour1
            time[11] = min10
            time[12] = min1
            time[13] = sec10
            time[14] = sec1
            time[15] = '\0'
```

Reference: slc_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, unsigned char* obuf, int osiz, COM *c) (void); ser0.h

void s1_init(unsigned char b, unsigned char* ibuf, int isiz, unsigned char* obuf, int osiz, COM *c) (void); ser1.h

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

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

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

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

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

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

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

---

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

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

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

Reference: ser1_sin.c

---

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

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

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

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

---

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

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

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

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

---

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

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

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

Reference: ser1.h, ser0.h for source code.

Appendix D: RTC72421 / 72423

Function Table

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

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

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

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

4)

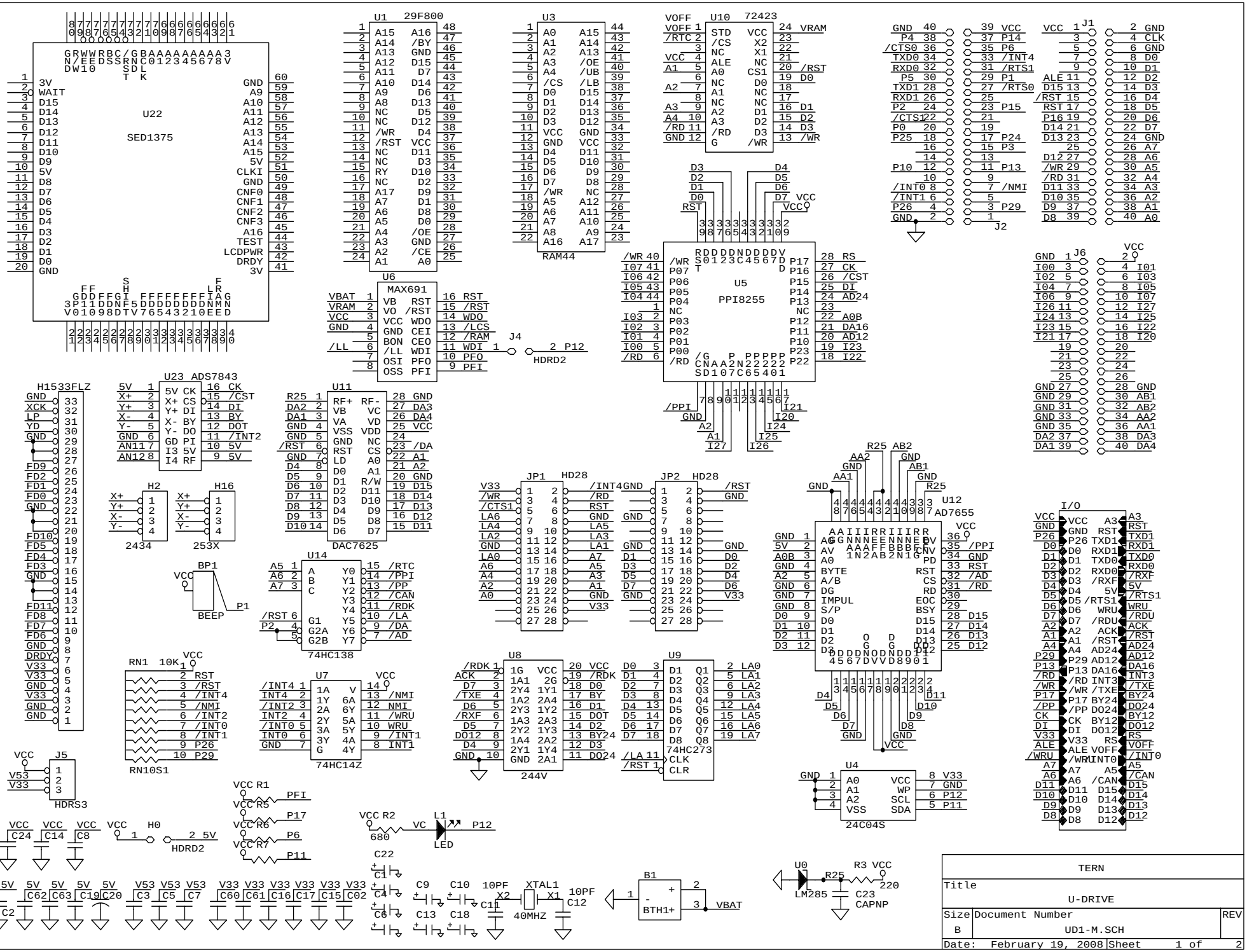
Data bit	PM/AM	INT/STD	24/12
1	PM	INT	24
0	AM	STD	12

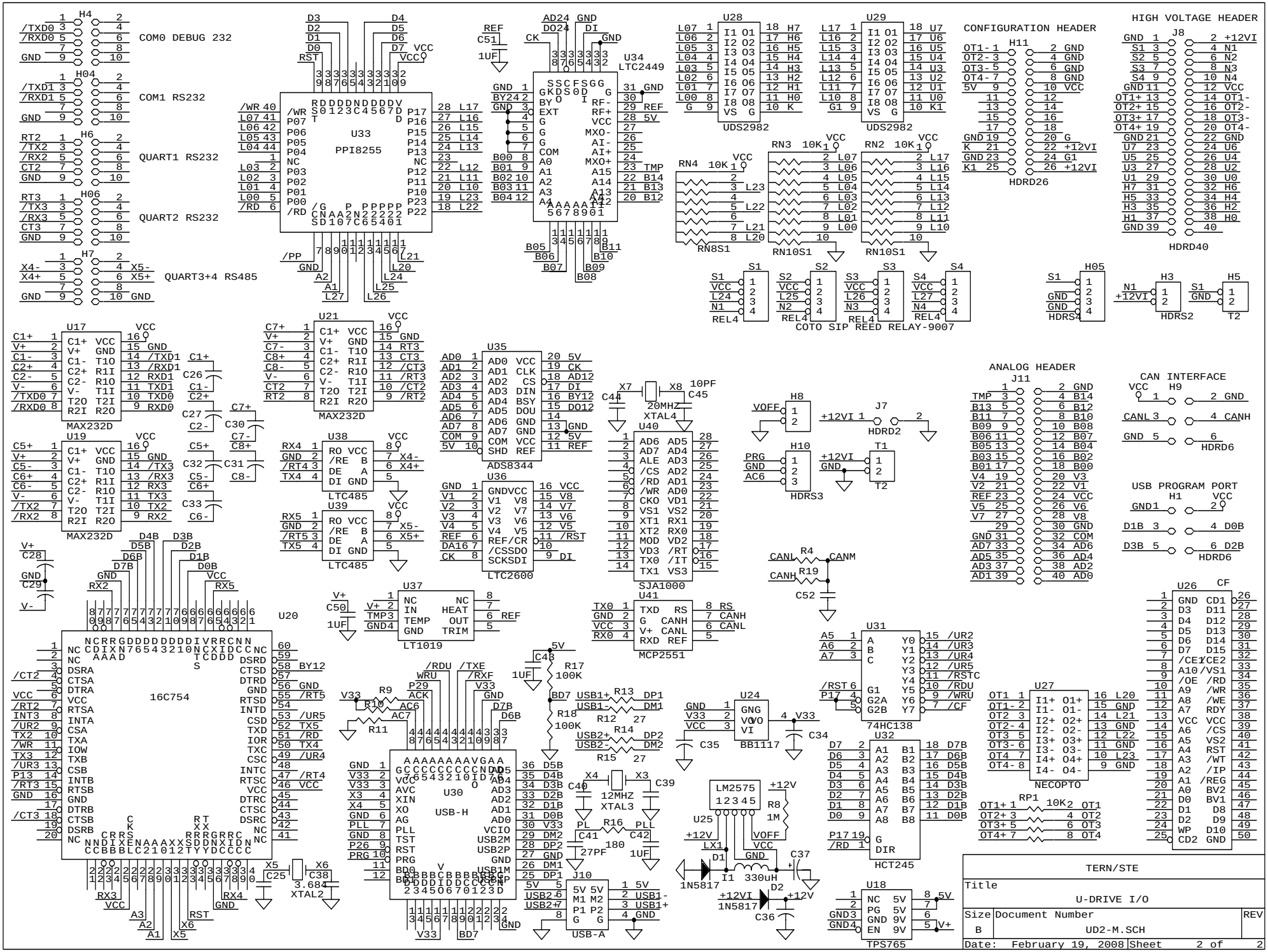
5) Test bit should be "0".

Serial EEPROM Map

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

0x00	Node Address, for networking		
0x01	Board Type	00	VE
		10	CE
		01	BB
		02	PD
		03	SW
		04	TD
		05	MC
0x02			
0x03			
0x04	SER0_receive, used by ser0.c		
0x05	SER0_transmit, used by ser0.c		
0x06	SER1_receive, used by ser1.c		
0x07	SER1_transmit, used by ser1.c		
0x10	CS high byte, used by ACTR™		
0x11	CS low byte, used by ACTR™		
0x12	IP high byte, used by ACTR™		
0x13	IP low byte, used by ACTR™		
0x18	MM page register 0		
0x19	MM page register 1		
0x1a	MM page register 2		
0x1b	MM page register 3		


U22
SED1375
U1 29F800
RAM44
U3
U10 72423
J1
J2
U6
MAX691
J4
HDRD2
U5
PPI8255
J6
U23 ADS7843
H2
2434
H16
253X
U11
DAC7625
JP1 HD28
JP2 HD28
U12
AD7655
I/O
U14
74HC138
BP1
BEEP
RN1 10K
RN10S1
U7
74HC14Z
U8
244V
U9
74HC273
U4
24C04S
J5
HDRS3
H0
HDRD2
L1
LED
B1
BTH1+
40MHZ
U0
LM285
R3 VCC
220
C23
CAPNP
TERN
U-DRIVE
UD1-M.SCH
February 19, 2008


COM0 DEBUG 232
COM1 RS232
QUART1 RS232
QUART2 RS232
QUART3+4 RS485
HIGH VOLTAGE HEADER
CONFIGURATION HEADER
HDRD26
HDRD40
UDS2982
UDS2982
LTC2449
PPI8255
RN8S1
RN10S1
RN10S1
COTO SIP REED RELAY-9007
HDRS4
HDRS2
T2
MAX232D
MAX232D
MAX232D
MAX232D
LTC485
LTC485
LTC485
ADS8344
LTC2600
SJA1000
MCP2551
LT1019
ANALOG HEADER
CAN INTERFACE
HDRD6
USB PROGRAM PORT
HDRD6
HDRD2
HDRS3
T2
16C754
USB-H
USB-A
BB1117
74HC138
NECOPTO
HCT245
LM2575
TPS765
1N5817
1N5817
330uH
XTAL4
XTAL3
XTAL2
CF
TERN/STE
U-DRIVE I/O
UD2-M.SCH
February 19, 2008