

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



Technical Manual

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

COPYRIGHT

A-Core, A-Engine, A-Engine-P, VE232, MiniDrive88, and ACTF are trademarks of TERN, Inc.
 Am188ES and Am186ES are trademarks of Advanced Micro Devices, Inc. Turbo C and Borland C++ are trademarks of Borland International.
 Microsoft, MS-DOS, Windows, Windows95, and Windows98 are trademarks of Microsoft Corporation.
 IBM is a trademark of International Business Machines Corporation.

Version 3.10

July 24, 2012

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

© 1997-2012

1950 5th Street, Davis, CA 95616, USA Tel: 530-758-0180 Fax: 530-758-0181

Email: <u>sales@tern.com</u>

http://www.tern.com

Important Notice

TERN is developing complex, high technology integration systems. These systems are integrated with software and hardware that are not 100% defect free. **TERN products are not designed, intended, authorized, or warranted to be suitable for use in life-support applications, devices, or systems, or in other critical applications. TERN** and the Buyer agree that **TERN** will not be liable for incidental or consequential damages arising from the use of **TERN** products. It is the Buyer's responsibility to protect life and property against incidental failure.

TERN reserves the right to make changes and improvements to its products without providing notice.

Temperature readings for controllers are based on the results of limited sample tests; they are provided for design reference use only.

Chapter 1: Introduction

1.1 Functional Description

The *MiniDrive88* (*MD88*) is a low-cost, high performance, C/C++ programmable, 16-bit microprocessor core module. It is designed for embedded applications that require compactness, low power consumption, and high reliability. The *MD88* is a modified version of the *A-Core*TM with 35 buffered high voltage I/Os. The *MD88* can be integrated into an OEM product as a processor core component.



Figure 1.1 Functional block diagram of the MD88

Measuring 3.1 by 2.6 by 0.3 inches, the *MD88* offers a C/C++ programmable computer system with a 16-bit high performance CPU (Am188ES, AMD) and operates at 40 MHz (or 20 MHz) system clock with zerowait-state. Optional features include up to 512K EPROM/Flash and up to 512K battery-backed SRAM. A 512-byte serial EEPROM is included on-board. An optional real-time clock provides information on the year, month, date, hour, minute, second, and 1/64 second, and an interrupt signal. The *MD88* is available with on-board 5-volt regulator and 2 channels of RS232 drivers.

Two DMA-driven serial ports from the Am188ES support high-speed, reliable serial communication at a rate of up to 115,200 baud while supporting 8-bit and 9-bit communication.

The *MD*88 offers three 16-bit programmable timers/counters and a watchdog timer. The timer can be used to count or time external events, at a rate of up to 10 MHz, or to generate non-repetitive or variable-duty-cycle waveforms as PWM outputs. Pulse Width Demodulation (PWD), a distinctive feature, can be used to measure the width of a signal in both its high and low phases. It can be used in many applications, such as bar-code reading.

There are 14 solenoid drivers outputs, 7 high voltage inputs, and 14 hardware configurable high voltage I/Os. High voltage I/O lines are routed to user programmable I/O pins, external interrupt inputs of the Am188ES and a 74HC259 data latch. A supervisor chip with power failure detection, a watchdog timer, an LED, and expansion ports are on-board.

Standard Features

- Dimensions: 3.1 x 2.6 x 0.3 inches
- 16-bit CPU (Am188ES), Intel 80x86 compatible. Easy to program in C/C++
- Power consumption: 160/120 mA at 5V for 40/20 MHz
- Power saving mode: 30/24 mA at 5V for 40/20 MHz

Chapter 1: Introduction

- Power input: +9V to +12 V unregulated DC
- Up to 512KB Flash/ROM
- 2 high-speed PWM outputs and Pulse Width Demodulation
- 32 I/O lines from Am188ES
- 512-byte serial EEPROM
- 8 external interrupt inputs, three 16-bit timer/counters
- 2 serial ports support 8-bit or 9-bit asynchronous communication
- Supervisor chip (691) for power failure, reset and watchdog

Optional Features (*):

- 32KB, 128KB, or 512KB SRAM*
- Real-time clock RTC72423*, lithium coin battery*

1.2 Physical Description



Figure 1.2 physical layout of the MD88

1.3 MD88 Programming Overview

Development of application software for the *MD88* consists of three easy steps, as shown in the block diagram below.



1-2

MiniDrive88

There are three possible steps in the development of a TERN controller. The *EV* Kit is capable of doing Step 1 and Step 2. The *EV* Kit cannot perform Step 3. Step 3 allows you to generate a HEX or BIN file and allows you to produce your own ROM/Flash chip. You will need the *DV* Kit and/or the *ACTF Flash* Kit to perform Step 3.

Please refer to the Tutorial section in the Technical Manuals for the *EV/DV* Kit to obtain further details on programming the *MD*88.

1.3.1 Step1

STEP 1: Debugging

Write your C/C++ application program in C/C++. Connect the *MD88* (with a DEBUG ROM TDREM_AC installed) to your PC via the PC-V25 serial link cable.

Use the batch file **m.bat** to compile, link, and locate, or use **t.bat** to compile, link locate, download, and debug your C/C++ application program.



Figure 1.3 Connecting a serial link cable and wall transformer (with adapter).

1.3.2 Step 2

STEP 2: Standalone Field Test.

Set the jumper on J6 pins 1-2 (Figure 1.4). At power-on or reset, if J6 pin 1 (A19) is low, the CPU will run the code that resides in the battery-backed SRAM. The *MD88* will operate in STEP 2 if a jumper is placed on J6 pins 1-2 during power-on or reset. If the jumper is off J6 pins 1-2 during power-on or reset, the *MD88* will operate in STEP 1. The status of the J6 pin 1 (signal A19) for the Am188ES is only checked at power-on or at reset.



Figure 1.4 Location of Step 2 jumper on the MD88 (J6)

1.3.3 Step 3

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

If you are happy with your Step 2 test, you can go back to your PC to generate your application ROM to replace the DEBUG ROM (*TDREM_AC*). **Important:** You need to change **DEBUG=1** to **DEBUG=0** in the makefile.

You must have the DV Kit and/or ACTF Kit to perform Step 3.

Please refer to the Tutorial chapter in the EV/DV Kit manual, for further details on programming the MD88.

1.4 Minimum Requirements for MD88 System Development

1.4.1 Minimum Hardware Requirements

- PC or PC-compatible computer with serial COMx port that supports 115,200 baud
- *MD*88 controller with DEBUG ROM *TDREM_AC*
- PC-V25 serial cable (RS232; DB9 connector for PC COM port and IDC 2x5 connector for controller)
- Center negative wall transformer (+9V 500mA)

1.4.2 Minimum Software Requirements

- TERN EV/DV Kit installation diskettes
- PC software environment: DOS, Windows 3.1, Windows95, or Windows98

Chapter 2: Installation

2.1 Software Installation

Please refer to the Evaluation/Development (EV/DV) Kit Technical for information on installing software.

The README.TXT file, located on the TERN *EV/DV* disk, contains important information for installing and evaluating TERN controllers.

2.2 Hardware Installation

Overview

- Connect PC to the MD88 using the PC-V25 cable.
- Connect 9V wall transformer to screw terminal on MD88
- User is ready to begin development.

2.2.1 Powering-on the MD88

Connect a +9V-12V DC power to the screw terminal J1 pin 1 (12V) and pin2 (GND) on the MD88 (Figure 2.1). The wall transformer may be connected to the MD88 via a power jack adapter. Make sure the debug ROM is installed and STEP 2 jumper is off.



Figure 2.1 Power connection for *MD88* with LED blinking twice at power-on.

Chapter 3: Hardware

3.1 Am188ES - Introduction

The Am188ES is based on the industry-standard x86 architecture. Am188ES controllers are highperformance, more integrated versions of the 80C188 microprocessors. In addition, the Am188ES has new peripherals. The on-chip system interface logic minimizes total system cost. The Am188ES has two asynchronous serial ports, thirty-two PIOs, a watchdog timer, additional interrupt pins, pulse width demodulation option, DMA to and from serial ports, a 16-bit reset configuration register, and enhanced chip-select functionality.

3.2 Am188ES – Features

3.2.1 Clock

Due to its integrated clock generation circuitry, the Am188ES microcontroller allows the use of a times-one crystal frequency. The design achieves 40MHz CPU operation, while using a 40 MHz crystal.

3.2.2 External Interrupts

There are eight external interrupts:

INT0, CPU pin 56 and U14 pin 11. INT1, CPU pin 55 and U14 pin 12. INT2, CPU pin 54 and U14 pin 15. INT3, CPU pin 53 and U14 pin 13 INT4, CPU pin 52 and U14 pin 14 INT5=P12=DRQ0, CPU pin 77, used as output pin for LED/EE/HWD INT6=P13=DRQ1, CPU pin 76 and U14 pin 10 NMI, CPU pin47 and U14 pin 16.

All eight interrupts are edge-triggered and are pulled up with 10K resistors. A ULN2003 or a resistor pack can be installed in U14 buffering all interrupt inputs. They all require a raising edge (LOW-to-HIGH) to generate an interrupt.

The *MD*88 uses vector interrupt functions to respond to external interrupts. Refer to the Am188ES User's Manual for information regarding interrupt vectors.

3.2.3 Asynchronous Serial Ports

The AM188ES CPU has two asynchronous serial channels: SER0 and SER1. Each asynchronous serial port supports:

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

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

3.2.4 Timer Control Unit

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

Timer0 and Timer1 are connected to four external pins:

Timer0 output = P10 = CPU pin 74=U4 pin 2 to O9 Timer0 input = P11 = CPU pin 75, used by EE Timer1 output = P1 = CPU pin 73 = U4 pin 1 to O8 Timer1 input = P0 = CPU pin 72 = U15 pin 5 from IO11

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 Timer0 and Timer1 or be used as a DMA request source. The maximum rate at which each timer can operate is 10 MHz, since each timer is serviced on every fourth clock cycle. Timer output takes up to six clock cycles to respond to clock or gate events. See sample programs *timer12.c* and *ae_cnt1.c* in \samples\ae.

3.2.5 PWM outputs and PWD

The Timer0 and Timer1 outputs can also be used to generate non-repetitive or variable-duty-cycle waveforms. The timer output takes up to 6 clock cycles to respond to the clock input. The minimum timer output cycle, therefore, is 25 ns x 6 = 150 ns (at 40 MHz).

Each timer has a maximum count register that defines the timer's maximum value. Both Timer0 and Timer1 have a secondary maximum count register for variable duty cycle output. Using both the primary and secondary maximum count registers lets the timer alternate between two maximum values.



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

3.2.6 Power-save Mode

The *MD88* is an ideal core module for low power consumption applications. The power-save mode of the Am188ES 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 Am188ES PIO lines

The Am188ES has thirty-two 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 opendrain output. A pin's behavior, either pull-up or pull-down, is pre-determined as shown in Table 3.1.

After power-on/reset, PIO pins default to various configurations. The initialization routine provided by TERN libraries also reconfigures some of these pins as needed for specific on-board usage. These

PIO	Function	Power-On/Reset status	MD88 Pin No.	MD88 Initial
P0	Timer1 in	Input with pull-up	J2 pin 22=IO11	
P1	Timer1 out	Input with pull-down	J2 pin 26=O8	CLK_1
P2	/PCS6/A2	Input with pull-up	J2 pin 17=IO2	RTC chip select
P3	/PCS5/A1	Input with pull-up	J2 pin 18=IO1	/PCS5
P4	DT/R	Normal	J2 pin 12=IO7	STEP2
P5	/DEN/DS	Normal	J2 pin 14=IO5	Input with pull-up
P6	SRDY	Normal	J2 pin 13=IO6	Input with pull-down
P7	A17	Normal		A17
P8	A18	Normal		A18
P9	A19	Input w. pull-up	J6 pin 1	Input with pull-up
P10	Timer0 out	Input with pull-down	J2 pin 27=O9	Input with pull-down
P11	Timer0 in	Input with pull-up	EE pin 5	Input with pull-up
P12	DRQ0/INT5	Input with pull-up	LED	Output for LED/EE/HWD
P13	DRQ1/INT6	Input with pull-up	J2 pin 11=I7	Input with pull-up
P14	/MCS0	Input with pull-up	J2 pin 15=IO4	Input with pull-up
P15	/MCS1	Input with pull-up	J2 pin 16=IO3	Input with pull-up
P16	/PCS0	Input with pull-up	74HC259 pin 14	74HC259 latch chip select
P17	/PCS1	Input with pull-up	J2 pin 23=IO10	Input with pull-up
P18	CTS1/PCS2	Input with pull-up	J2 pin 24=IO9	Input with pull-up
P19	RTS1/PCS3	Input with pull-up	J2 pin 25=IO8	Input with pull-up
P20	RTS0	Input with pull-up	J2 pin 20=IO13	Input with pull-up
P21	CTS0	Input with pull-up	J2 pin 21=IO12	Input with pull-up
P22	TxD0	Input with pull-up	232, U8 pin 10	TxD0
P23	RxD0	Input with pull-up	232, U8 pin 9	RxD0
P24	/MCS2	Input with pull-up	J2 pin 28=O10	Input with pull-up
P25	/MCS3	Input with pull-up	J2 pin 29=O11	Input with pull-up
P26	UZI	Input with pull-up	J2 pin 30=O12	
P27	TxD1	Input with pull-up	232, U8 pin 11	TxD1
P28	RxD1	Input with pull-up	232, U8 pin 12	RxD1
P29	/CLKDIV2	Input with pull-up	J2 pin 31=O13	Input with pull-up*
P30	INT4	Input with pull-up	J2 pin 7=I3	Input with pull-up
P31	INT2	Input with pull-up	J2 pin 6=I2	Input with pull-up

configurations, as well as the processor-internal peripheral usage configurations, are listed below in Table 3.1.

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

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

Three external interrupt lines are not shared with PIO pins and only can be used to generate interrupts:

INT0 = U14 pin 11=J2 pin 10=I6 INT1 = U14 pin 12=J2 pin 9 =I5 INT3 = U14 pin 13=J2 pin 8 =I4

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

MODE	PIOMODE reg.	PIODIRECTION reg.	PIN FUNCTION
0	0	0	Normal operation

1	0	1	INPUT with pull-up/pull-down
2	1	0	OUTPUT
3	1	1	INPUT without pull-up/pull-down

MD88 initialization on PIO pins in ae_init() is listed below:

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

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

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

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

Example:

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

void pio_wr(char bit, char dat);

pio_wr(12,1); set P12 pin high, if P12 is in output mode pio_wr(12,0); set P12 pin low, if P12 is in output mode

unsigned int *pio_rd*(char port);

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

Some of the I/O lines are used by the *MD88* system for on-board components (Table 3.2). We suggest that you do not use these lines unless you are sure that you are not interfering with the operation of such components (i.e., if the component is not installed).

Signal	Pin	Function
P2	/PCS6	U4 RTC chip select. It can be used as IO2, if RTC not installed
P4	J6 pin 1	Step2 jumper
P11	Timer0 input	U7 24C04 EE data input
		The EE data output can be tri-state, while disabled
P12	DRQ0/INT5	Output for LED or U7 serial EE clock or Hit watchdog
P16	/PCS0	U11 74HC259 chip select at base I/O address 0x0000
P22	TxD0	Default SER0 debug
P23	RxD0	Default SER0 debug

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

3.4 I/O Mapped Devices

3.4.1 I/O Space

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

The default I/O access time is 15 wait states. You may use the function void *io_wait*(char wait) to define the I/O wait states from 0 through 15. The system clock is either 25 or 50 ns, giving a clock speed of either 40 or 20 MHz respectively. Details regarding this can be found in the Software chapter, and in the Am188ES User's Manual. Slower components, such as most LCD interfaces, might find the maximum programmable wait state of 15 cycles still insufficient. Due to the high bus speed of the system, some components need to be attached to I/O pins directly.

For details regarding the chip select unit, please see Chapter 5 of the Am188ES User's Manual.

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

I/O space	Select	Usage	Location
0x0000-0x00ff	/PCS0	74HC259	U11 pin 14=P16
0x0100-0x01ff	/PCS1	USER	P17
0x0200-0x02ff	/PCS2	USER	CTS1
0x0300-0x03ff	/PCS3	USER	RTS1
0x0400-0x04ff	/PCS4	Reserved	
0x0500-0x05ff	/PCS5	USER	J2 pin 16=P3
0x0600-0x06ff	/PCS6	RTC 72423	U5 pin 3=P2

3.4.2 74HC259

The 74HC259 8-bit decoder latch provides eight additional output lines for the *MD88*. The 74HC259 is mapped in the I/O address space 0x0000. You may access this device by using the following code:

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





3.4.3 Real-time Clock RTC72423

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

3.5 Other Devices

A number of other devices are also available on the *MD*88. Some of these are optional and might not be installed on the particular controller you are using. For a discussion regarding the software interface for these components, please see the Software chapter.

3.5.1 On-board Supervisor with Watchdog Timer

The MAX691/LTC691 (U6) is a supervisor chip. With it installed, the *MD*88 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

Setting a jumper on J9 of the MD88 (see Figure 3.2) activates the watchdog timer. The watchdog timer provides a means of verifying proper software execution. In the user's application program, calls to the function **hitwd**() (a routine that toggles the P12=HWD pin of the MAX691) should be arranged such that the HWD pin is accessed at least once every 1.6 seconds. If the J9 jumper is on and the HWD pin is *not* accessed within this time-out period, the watchdog timer pulls the WDO pin low, which asserts /RESET. This automatic assertion of /RESET may recover the application program if something is wrong. After the *MD88* is reset, WDO remains low until a transition occurs at the WDI pin of the MAX691. When controllers are shipped from the factory the J9 jumper is off, which disables the watchdog timer.

In addition, the Am188ES has an internal watchdog timer. This is disabled by default with ae_init().



Figure 3.2 Location of watchdog timer enable jumper

Battery Backup Protection

The backup battery protection protects data stored in the SRAM and RTC. The battery-switch-over circuit compares VCC to VBAT (+3V lithium battery positive pin), and connects the one that 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 to 5 years without external power being supplied. When the external power is on, the battery-switch-over circuit will select the VCC to connect to the VRAM.

3.5.2 EEPROM

A serial 512-byte EEPROM (24C04) can be installed in U7. The *MD88* uses the P12=SCL (serial clock) and P11=SDA (serial data) to interface with the EEPROM. The EEPROM can be used to store important data, such as a node address, calibration coefficients, and configuration codes. It has typically 1,000,000 erase/write cycles. The data retention lasts for more than 40 years. EEPROM can be read and written by simply calling functions **ee_rd()** and **ee_wr()**. See Appendix E for more information.

3.6 Headers and Connectors

3.6.11/O Headers

The *MD*88 has one 17x2 pin header (J2) and a 10x1-screw terminals (J1) for user high voltage I/Os. Signals are buffered by resistors or ULN2003, not directly routed to the Am188ES processor. The signals for both J1 and J2 are listed in Table 3.3.

J7 and J8 are 5x2 headers for SER0 and SER1 RS232 serial ports. The signal for J7 and J8 are listed in Table 3.4.

J1 Si	gnal			J	2 Sign	al
12V	1		12V	1	2	Κ
GND	2		GND	3	4	GND
VCC	3		I1	5	35	I2
07	4		I3	7	33	I4
06	5		I5	9	10	I6
05	6		I7	11	12	IO7
O4	7		IO6	13	14	IO5
03	8		IO4	15	16	IO3
O2	9		IO2	17	18	IO1
01	10		IO14	19	20	IO13
		-	IO12	21	22	IO11
			IO10	23	24	IO9
			IO8	25	26	08
			09	27	28	O10
		O11	29	30	O12	
			013	31	32	O14
			GND	33	34	GND

Table 3.3 Signals for J1 (10x1) and J2 (17x2) expansion ports

J7 or J8 Signal						
	1	2				
/TXD0 or 1	3	4				
/RXD0 or 1	5	6				
	7	8				
GND	9	10	GND			

Table 3.4 Signals for J7 (5x2) and J8 (5x2)

3.6.2 Connectors, Headers and Jumpers

Table 3.5 lists the jumpers and connectors on the *MD*88. Figure 3.3 indicates their locations. For a larger scale diagram, refer to Appendix B.

Name	Size	Function	Possible Configuration
J1	10x1	Screw terminals for O1-7	12V power input
J2	17x2	High voltage I/Os	
J3	3x1	SRAM selection	pin 2-3 SRAM 256K-512K
			pin 1-2 SRAM 32K-128K
J4	3x1	ROM size selection	pin 1-2: ROM size 32K-128K
			pin 2-3: ROM size 256K-512K
J5	3x1	EPROM/FLASH	pin 1-2: EPROM size 512K
		selection	pin 2-3 FLASH size 128K/512K, or
			EPROM 32K-256K
J6	2x1	A19=P9	Step Two Jumper with TDREM_AC ROM
J7	5x2	RS-232 SER0 connector	
J8	5x2	RS-232 SER1 connector	
J9	2x1	Watchdog timer	Enabled if Jumper is on
		-	Disabled if jumper is off





Figure 3.3 Locations of jumpers and connectors on the MD88

3.6.3 Protective high voltage inputs

In order to support high voltage digital signal input up to 30V, Darlington Transistor Arrays (ULN2003A) can be installed in U14, U5, and U15. The input pin has a resistance load of 12.7K towards 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. Please refer to Figure 3.4 for a schematic circuit diagram of a Darlington Transistor used as a protective high voltage input.





U5 and U15 can be configured as input or output. By factory default, U5 and U15 are sink output. They also can be sourcing output with optional UDS2982 installed.

The user can re-install the ULN2003 in U5 and U15 with different orientation. The input and output orientation for U5 and U15 is illustrated below in Figure 3.5. Follow these illustrations carefully to prevent damage to the chips. **Notice that U5 and U15 can be aligned differently.** In addition, the ULN2003 chips may be replaced with a resistor pack to provide digital inputs or outputs to the terminal blocks.



Figure 3.5 Locations of user configurable Darlington Transistor Arrays.

3.6.4 High-Voltage, High-Current Drivers

ULN2003 has high voltage, high current Darlington transistor arrays, consisting of seven silicon NPN Darlington pairs on a common monolithic substrate. All channels feature open-collector outputs for sinking 350 mA at 50V. They also contain integral protection diodes for driving inductive loads. Peak inrush currents of up to 600 mA sinking are allowed. U4 and U13 are dedicated high-voltage drivers while U5 and

U15 are configurable as high-voltage drivers. These outputs may be paralleled to achieve high-load capability, although each driver has a maximum continuous collector current rating of 350 mA at 50V. The maximum power dissipation allowed is 2.20 W per chip at 25 degrees Celsius (°C). The common substrate G is routed to T2 GND pins. All currents sinking in must return to the J2 or J1 GND pin. A heavy gauge (20) wire must be used to connect the J1, or J2 GND terminal to an external common ground return. The signal, K, connects to the protection diodes in the ULN2003 chips and should be tied to highest voltage in the external load system. K can be connected to an unregulated on board +12V via J2 pin 1-2. **ULN2003 is a sinking driver, not a sourcing driver.** An example of typical application wiring is shown in Figure 3.6.



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

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 Appendix G, "Software Glossary."

Guidelines, awareness, and problems in an interrupt driven environment

Although the C/C++ Development Kit provides a simple, low cost solution for 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, however, to debug an interrupt handler. *You do risk 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. Any memory-space mappings in place for this particular range of memory is 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 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 *MD88* operations. It includes the following modules: AE.OBJ, SER0.OBJ, SER1.OBJ, SCC.OBJ, and AEEE.OBJ. You need to link AE.LIB in your applications and include the corresponding header files. The following is a list of the header files:

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

4.2 Functions in AE.OBJ

4.2.1 MD88 Initialization

ae_init

This function should be called up at the beginning of every program running on *MD88* core controllers. It provides default initialization and configuration of the various I/O pins, interrupt vectors, sets up expanded DOS I/O, and provides other processor-specific updates needed at the beginning of every program.

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

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

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

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

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

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

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

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

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

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

<pre>outport(0xff78,0xe73c);</pre>	//	PDIR1	, TxD0,	RxD0,	TxD1,	RxD1,
	//	P16=PCS	0, P17=PCS	S1=PPI		
<pre>outport(0xff76,0x0000);</pre>	//	PIOM1				
<pre>outport(0xff72,0xec7b);</pre>	//	PDIR0,	P12,A19,A1	8,A17,P	2=PCS6=R	ГС
<pre>outport(0xff70,0x1000);</pre>	//	PIOM0,	P12=LED			

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

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

The chip select lines are by default set to 15 wait state. This makes it possible to interface with many slower external peripheral components. If you require faster I/O access, you can decrease this number 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_wa	ait							
Argument	s: char	wait						
Return va	lue: nor	ne.						
This functi	on sets	the current	wait st	ate dej	pending on	the ar	gument wa	pait.
wait=0,	wait	states	= 0,	I/0 I/0	enable	for	100 ns	ng

```
wait=1, wait states
                          \bot, I/O enable for
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 MD88, consisting of seven maskable interrupt pins (INT6-INT0) and one non-maskable interrupt (NMI). There is also an additional eight internal interrupt sources not connected to the external pins, consisting of three timers, two DMA channels, both asynchronous serial ports, and the NMI from the watchdog timer. For a detailed discussion involving the ICUs, the user should refer to Chapter 7 of the AMD Am188ES Microcontroller User's Manual.

TERN provides functions to enable/disable all of the 8 external interrupts. The user can call any of the interrupt init functions listed below for this purpose. The first argument indicates whether the particular interrupt should be enabled, and the second is a function pointer that is directed to the 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(* intx_isr) ()) **Return value: none**

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

4.2.3 I/O Initialization

There are two ports of 16 I/O pins available on the *MD*88. Hardware details regarding these PIO lines can be found in the Hardware chapter.

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

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

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

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

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

void pio_initArguments:char bit, char modeReturn 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 portReturn 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 MD88 can be used for a variety of applications. All three timers run at ¹/₄ of the processor clock rate, 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 AM188ES User's Manual.

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

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

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

It is also possible to use the output of **Timer2** to pre-scale one of the other timers, since 16-bit resolution at the maximum clock rate specified gives you only 150 Hz. Only by using **Timer2** can you slow this down even further. Sample files demonstrating this are *timer02.c* and *timer12.c* in the *MD88* sample file directory.

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

<pre>void t0_init void t1_init Arguments: int tm, int ta, int tb, void interrupt far(*t_isr)() Return values: none</pre>
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.
<pre>void t2_init Arguments: int tm, int ta, void interrupt far(*t_isr)() Return values: none.</pre>

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

4.2.5 Other library functions

On-board supervisor MAX691 or LTC691

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

void hitwd Arguments: none Return value: none

Resets the supervisor timer for another 1.6 seconds.

void led
Arguments: int ledd
Return value: none

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

Real-Time Clock

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

```
There is a common data structure used to access and use both interfaces.
typedef struct{
  unsigned char sec1; One second digit.
  unsigned char sec10; Ten second digit.
  unsigned char min1; One minute digit.
  unsigned char min10; Ten minute digit.
  unsigned char hourl; One hour digit.
  unsigned char hour10; Ten hour digit.
  unsigned char day1; One day digit.
  unsigned char day10; Ten day digit.
  unsigned char mon1; One month digit.
  unsigned char mon10; Ten month digit.
  unsigned char year1; One year digit.
  unsigned char year10; Ten year digit.
  unsigned char wk; Day of the week.
} TIM;
```

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

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

Void rtc_init Arguments: char* t Return value: none

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

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

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

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

Delay

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

void delay0 Arguments: unsigned int t Return value: none

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

While(t) { t--; }

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

void delay_ms
Arguments: unsigned int
Return value: none

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

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

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

void ae_reset
Arguments: none
Return value: none

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

4.3 Functions in SER0.OBJ/SER1.OBJ

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

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

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

By default, SER0 is used by the DEBUG ROM for application download/debugging in STEP 1 and in STEP 2. We will use SER1 as the example in the following discussion; any of the interface functions that are specific to SER1 can be easily changed into function calls for SER0. While selecting a serial port for use, please realize that some pins might be shared with other peripheral functions. This means that in certain limited cases, it might not be possible to use a certain serial port with other on-board controller functions. For details, you should see both chapter 10 of the Am188ES Microprocessor User's Manual and the schematic of the *MD88* provided at the end of this manual.

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

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

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, if there is any present, take out the data from the buffer with **getser1()**. The input buffer is used as a circular ring buffer, as shown in Figure 4.1. The transmit operation, however, 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 **sl_init(**) with a default mode of 8-bit, 1 stop bit, no parity. After **sl_init(**) you can set up a new mode with different numbers for data-bit, stop bit, or parity by directly accessing the Serial Port 0/1 Control Register (SP0CT/SP1CT) if necessary, as described in chapter 10 of the Am188ES manual for asynchronous serial ports.

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

It is always important, however, 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 **serhitl()** 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 **sl_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
                           *in_buf;
                                          /* Input buffer */
  unsigned char
                       /* Input buffer TAIL ptr */
  int in tail;
                      /* Input buffer HEAD ptr */
  int in_head;
                      /* Input buffer size */
  int in_size;
  int
      in crcnt;
                      /* Input <CR> count */
  unsigned char
                  in_mt;
                                  /* Input buffer FLAG */
                 in_full;
                                  /* input buffer full */
  unsigned char
                                 /* Output buffer */
  unsigned char
                  *out_buf;
                      /* Output buffer TAIL ptr */
  int out_tail;
                      /* Output buffer HEAD ptr */
  int out_head;
                      /* Output buffer size */
  int out_size;
 unsigned char out_full;
unsigned char out_mt;
                                 /* Output buffer FLAG */
                                /* 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;
                              /* input buffer segment */
  unsigned int in_segm;
  unsigned int in_offs;
                              /* input buffer offset */
                              /* output buffer segment */
  unsigned int out_segm;
  unsigned int out_offs;
                              /* output buffer offset */
                             /* V25 macro service byte delay */
  unsigned char byte_delay;
} COM;
```

sn_init

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

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

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

There are a couple different functions used for transmission of data. You can actually 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.

putsersn

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.

getsern

Arguments: COM *c Return value: unsigned char value

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

getsersn

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

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

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

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

Miscellaneous Serial Communication Functions

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

be aware that the peripheral pin function you are using might not be selected as needed. For details, please refer to the Am188ES User's Manual.

char sn_cts(void)
Retrieves value of CTS pin.

void sn_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.

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

The asynchronous serial I/O ports available on the Am188ES Processor have many other features that might be useful for your application. If you are truly interested in having more control, please read Chapter 10 of the User's manual for a detailed discussion of other features available to you.

4.4 Functions in AEEE.OBJ

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

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

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

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

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

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

ee_rd Arguments: int addr Return value: int data

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

Appendix A: Mechanical dimensions of MiniDrive88



MiniDrive88

Appendix B: Layout



A-1

Appendix C: RTC72421 / 72423

Function Table

	Address					Data				
A ₃	A ₂	A ₁	A ₀	Register	D ₃	D ₂	D ₁	D ₀	Count Value	Remarks
0	0	0	0	S ₁	s ₈	s ₄	s ₂	s ₁	0~9	1-second digit register
0	0	0	1	S ₁₀		s ₄₀	s ₂₀	s ₁₀	0~5	10-second digit register
0	0	1	0	MI ₁	mi ₈	mi ₄	mi ₂	mi ₁	0~9	1-minute digit register
0	0	1	1	MI ₁₀		mi ₄₀	mi ₂₀	mi ₁₀	0~5	10-minute digit register
0	1	0	0	H ₁	h ₈	h ₄	h ₂	h ₁	0~9	1-hour digit register
0	1	0	1	H ₁₀		PM/AM	h ₂₀	h ₁₀	0~2	PM/AM, 10-hour digit
									or	register
									0~1	
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	У 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	IRQ	Busy	Hold		Control register D
					Adj	Flag				
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");

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

5) Test bit should be "0".

Appendix D: Serial EEPROM Map

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

0x00	Node Address,	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, u	sed by A	CTR TM		
0x11	CS low byte, us	sed by A	CTR TM		
0x12	IP high byte, us	ed by A	CTR TM		
0x13	IP low byte, use	ed by AC	CTR TM		
0x18	MM page regis	ter 0			
0x19	MM page regis	ter 1			
0x1a	MM page regis	ter 2			
0x1b	MM page regis	ter 3			
	1000				

Appendix E: Software Glossary

The following is a glossary of library functions for the MD88.

void ae_init(void)

ae.h

Initializes the AM188ES processor. The following is the source code for *ae_init()* outport(0xffa0,0xc0bf); // UMCS, 256K ROM, 3 wait states, disable AD15-0 outport(0xffa2,0x7fbc); // 512K RAM, 0 wait states outport(0xffa8,0xa0bf); // 256K block, 64K MCS0, PCS I/O outport(0xffa6,0x81ff); // MMCS, base 0x80000 outport(0xffa4,0x007f); // PACS, base 0, 15 wait outport(0xff78,0xe73c); // PDIR1, TxD0, RxD0, TxD1, RxD1, P16=PCS0, P17=PCS1=PPI outport(0xff76,0x0000); // PIOM1 outport(0xff72,0xec7b); // PDIR0, P12,A19,A18,A17,P2=PCS6=RTC outport(0xff70,0x1000); // PIOM0, P12=LED outportb(0x0103,0x9a); // all pins are input, I20-23 output outportb(0x0100,0); outportb(0x0101,0); outportb(0x0102,0x01); // I20=ADCS high clka_en(0); $clkb_en(0);$ enable();

Reference: led.c

void ae_reset(void)

Resets AM188 processor.

void delay_ms(int m)

Approximate microsecond delay. Does not use timer.

Var: m - Delay in approximate ms

Reference: led.c

void led(int i)

Toggles P12. Used for the LED.

Var: i - Led on or off

Reference: led.c

ae.h

ae.h

ae.h

void delay0(unsigned int t)

Software loop delay: While (t--);

Var: m - Delay in number of iterations

Reference:

void pwr_save_en(int i)

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)
void clkb_en(int i)

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

Var: i - turns clock on/off.

Reference:

void hitwd(void)

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

ae.h

ae.h

ae.h

ae.h

E-2

void pio_wr(char bit, char dat)

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)

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)

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)

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)

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

Var: portid - I/O address

Reference: ae_ppi.c

ae.h

ae.h

dos.h

dos.h

dos.h

int inportb(int portid)

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)

Writes to the serial EEPROM.

Var: addr - EEPROM data address dat - data

Reference: ae_ee.c

int ee_rd(int addr)

Reads from the serial EEPROM. Returns 8-bit data

Var: addr - EEPROM data address

Reference: ae_ee.c

void io_wait(char wait)

Set up I/O wait states for I/O instructions.

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

ae.h

dos.h

aeee.h

aeee.h

void rtc_init(unsigned char * time)

ae.h

Sets real-time clock date and time.

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

Reference: rtc_init.c

int rtc_rd(TIM *r)

Reads from the real-time clock.

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

Reference: rtc.c

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

ae.h

Timer 0, 1, 2 initialization.

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

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

void interrupt far t2_isr(void); void interrupt far t1_isr(void); ae.h

ae.h

void interrupt far t0_isr(void);

Timer 0, 1, 2 interrupt service routine.

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

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

void nmi_init(void interrupt far (* nmi_isr)()); 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 (*int5_isr)());

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

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

Reference: intx.c

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

Interrupt service routine for int0 - int6 and NMI.

Var: None.

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

Reference: intx.c

ae.h

void s0_init(unsigned char b, unsigned char* ibuf, int isiz,	ser0.h
unsigned char* obuf, int osiz, COM *c) (void);	
void s1_init(unsigned char b, unsigned char* ibuf, int isiz,	ser1.h
unsigned char* obuf, int osiz, COM *c) (void);	

Serial port 0, 1 initialization.

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

b	baud (40MHz)	baud (20MHz)
1	110	55
2	150	110
3	300	150
4	600	300
5	1200	600
6	2400	1200
7	4800	2400
8	9600	4800
9	19200	9600
10	38400	19200
11	57600	38400
12	115200	57600
13	23400	115200
14	460800	23400
15	921600	460800

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

int putser0(unsigned char ch, COM *c);
int putser1(unsigned char ch, COM *c);

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);
int putsers1(unsigned char *str, COM *c);

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 ser0.h

ser1.h

ser0.h

ser1.h

Reference: ser1_sin.c

<pre>int serhit0(COM *c);</pre>	ser0.h
<pre>int serhit1(COM *c);</pre>	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.

