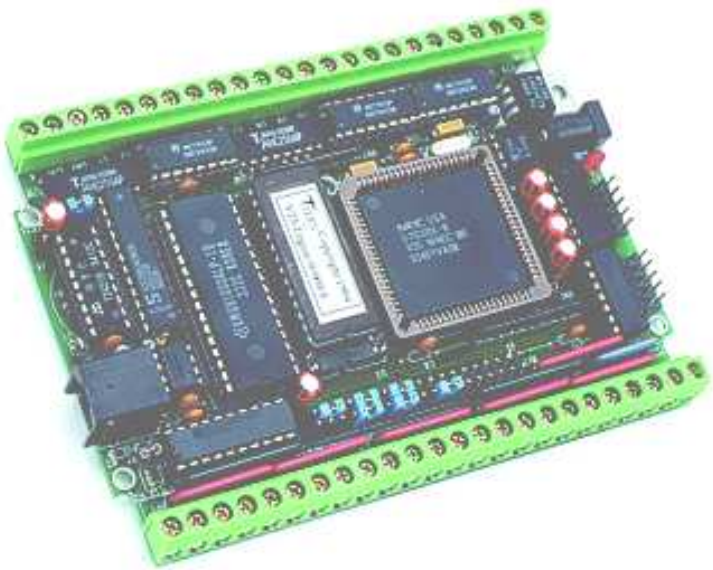
TinyDrive ™

C/C++ Programmable 16-bit Industrial Controller
Based on the NEC V25



Technical Manual

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

Version 3.00

October 29, 2010

© 1995-2010 TERN INC.
1950 5th Street, Davis, CA 95616, USA
Tel: 530-758-0180 Fax: 530-758-0181
Email: ***sales@tern.com*** ***http://www.tern.com***

Important Notice

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

Chapter 1: Introduction

1.1 Functional Description

The ***TinyDrive-A*** (***TD***) from TERN is a low cost, high-performance, C/C++ programmable, 16-bit industrial controller. It is designed for embedded applications that require compactness, low power consumption, and high reliability. The TinyDrive has a total of 42 I/O lines, including 12-bit ADC inputs, 4-bit ADC inputs, high voltage inputs, TTL inputs, and high voltage outputs. The TinyDrive can be integrated into an OEM product as a processor core component or as a stand-alone controller in an application system.

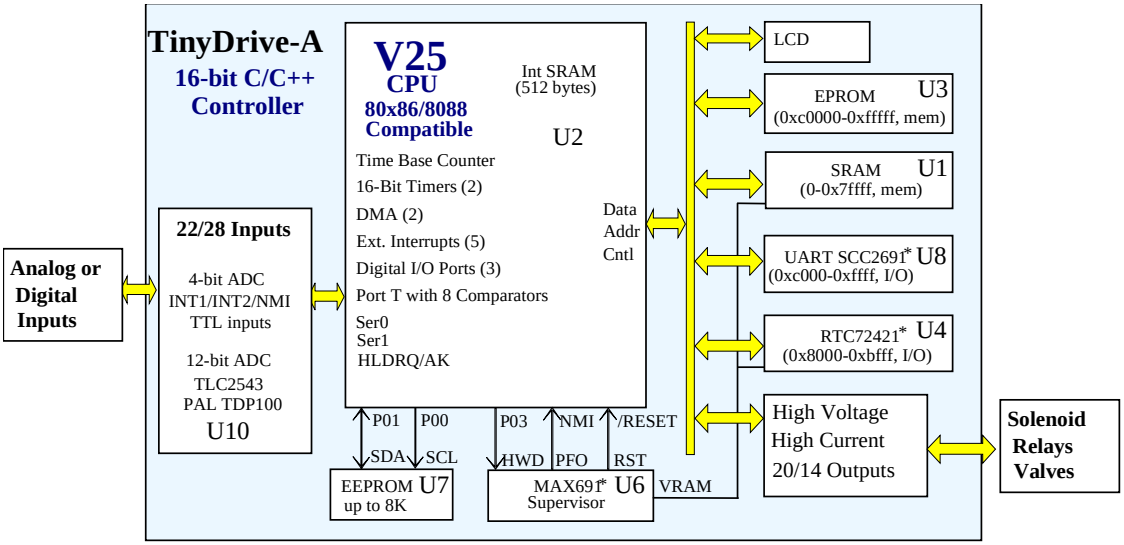


Figure 1.1 Functional block diagram of the TinyDrive

Measuring 4.8 by 3.4 inches, the TinyDrive offers a complete C/C++ programmable computer system with a 16-bit high performance CPU (NEC V25) and operates at 8 MHz with zero-wait-state. The TinyDrive offers two 16-bit timers, 42 I/O lines, power failure detection, watchdog timer, LED, and 48 positions of screw terminal blocks. 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.

At 24x2 screw terminals, the TD provides 22 resistor-protected inputs, including eight comparator inputs, 11 12-bit ADC, three interrupt inputs, and 20 solenoid driver outputs. Six outputs can be configured as inputs. The 20 solenoid drivers can sink up to 350 mA at 50V and can drive solenoid coils or relays.

The eight comparator ADC inputs can be used to measure either digital or 0 to 5V analog inputs in 16 levels. The buffered digital inputs take +24V, +5V, or ground inputs. The optional 11 12-bit ADC features reference inputs (2.5V or 5V), a sample rate of up to 2.5 kHz, and a 0-5V input voltage range. A PAL (TDP100) can provide eight digital inputs, replacing the ADC.

Two RS-232 serial ports from the V25 support up to 115,200 baud. The RS-485 port for the optional UART (SCC2691) supports normal 8-bit and 9-bit networking. Three 16-bit timers provide precise timing.

Two standby modes, HALT and STOP, can reduce power consumption. An optional LCD interface can be installed replacing the SCC2691. Up to four channels of 12-bit DAC (optional) can be installed in the two solenoid driver sockets.

TERN also offers custom hardware and software design, based on the TinyDrive or other TERN controllers.

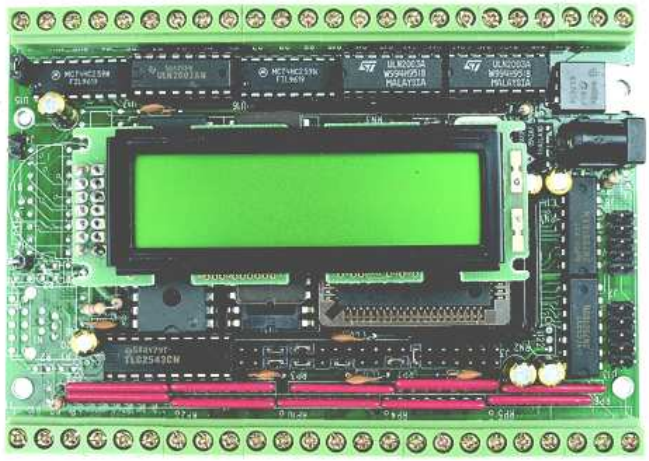


Figure 1.2 The TinyDrive with a 16x2 character LCD installed

1.2 Features

Standard Features

- Dimensions: 4.8 x 3.4 inches
- Power consumption: 180 mA at 9V
- Low power version: 75 mA full speed, 20 mA standby
- Power Input: +9V to +12V unregulated DC
- 16-bit CPU (NEC V25), 8 MHz, Intel 80x86 compatible, C/C++ programmable
- ROM and SRAM up to 1MB, 512-byte EEPROM (or up to 2KB) and 256 bytes built-in-CPU SRAM
- 5 external interrupts, 24 bi-directional digital I/O lines, 8 comparators (some are used by system)
- 20 high-voltage drivers, all routed to screw terminals (6 outputs may be configured as inputs)
- Interface for LCD, keypad
- Two 16-bit timers, one 16-bit time base counter
- Two 16-bit counters or DMA. The counter can count external signal rising edges up to 500 kHz.
- Two high speed serial ports from the V25 CPU
- Supervisor chip (691) for power failure, reset and watchdog

Optional Features:

- 32KB, 128KB, or 512KB SRAM
- 11 channels of 12-bit ADC, sample rate up to 10 kHz
- Up to 4 channels 12-bit DAC, 0-4.095V output
- SCC2691 UART (on-board) with RS-485 drivers supports 8-bit or 9-bit networking
- Real-time clock RTC72423, lithium coin battery
- Precision reference, 20 PPM/°C, 5V
- Low-power version

- LCD interface (PAL TDLCD)
- 8 additional digital inputs (PAL TDP100) in U10 socket (if no ADC installed)

1.3 Physical Description

The physical layout of the TinyDrive is shown in Figure 1.3.

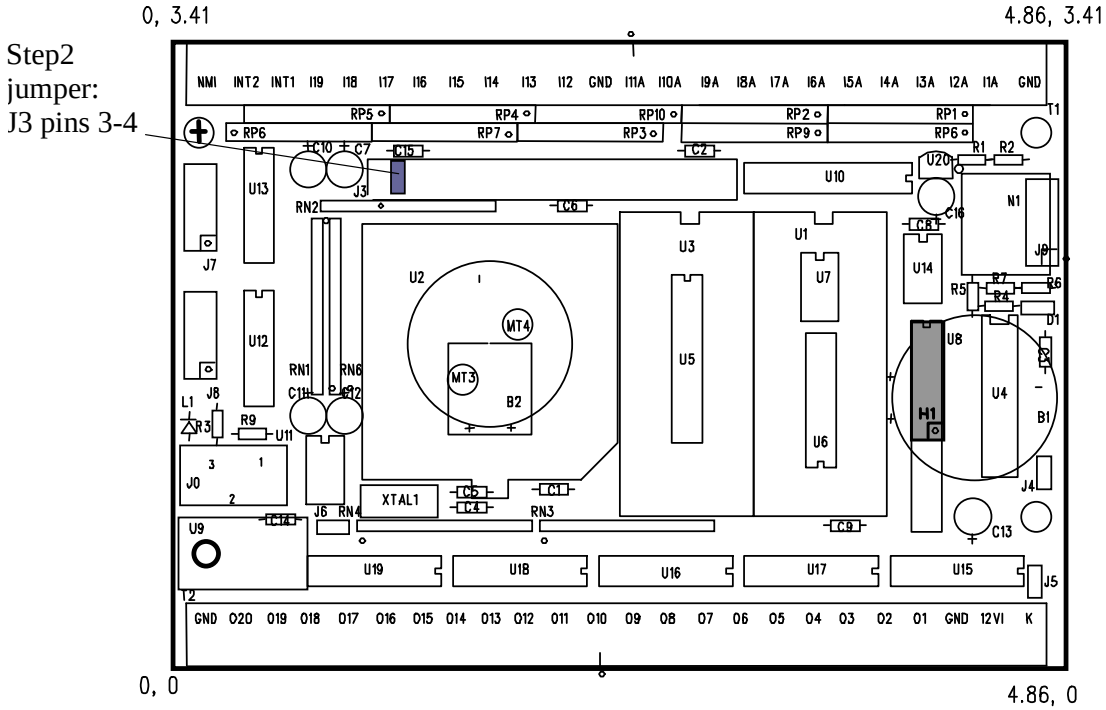
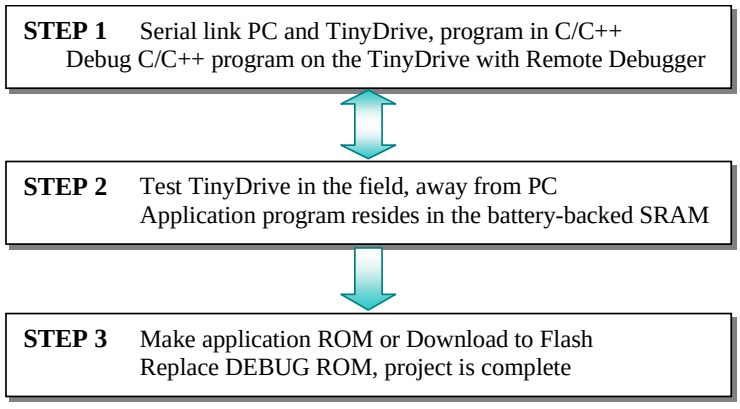


Figure 1.3 Physical layout of the TinyDrive

1.4 TinyDrive Programming Overview

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



You can program the TinyDrive from your PC via serial link with an RS232 interface. Your C/C++ program can be remotely debugged over the serial link at a rate of 115,000 baud. The C/C++ Evaluation Kit (EV) or Development Kit (DV) from TERN provides a Borland C/C++ compiler, TASM, LOC31, Turbo Remote Debugger, I/O driver libraries, sample programs, and batch files. These kits also include a DEBUG ROM (*TDREM_V25)* to communicate with Turbo Debugger, a DB9-IDE10 (PC-V25) serial cable to connect the controller to the PC, and a 9-volt wall transformer. *See your Evaluation/Development Kit Technical Manual for more information on these kits.*

After you debug your program, you can test run the TinyDrive in the field, away from the PC, by changing a single jumper, with the application program residing in the battery-backed SRAM. When the field test is complete, application ROMs can be produced to replace the DEBUG ROM. The .HEX or .BIN file can be easily generated with the makefile provided. You may also use the DV Kit or ACTF Kit to download your application code to on-board Flash.

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

1.4.1 Step 1

STEP 1: Debugging

- Write your C/C++ application program in C/C++.
- Connect your controller 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.

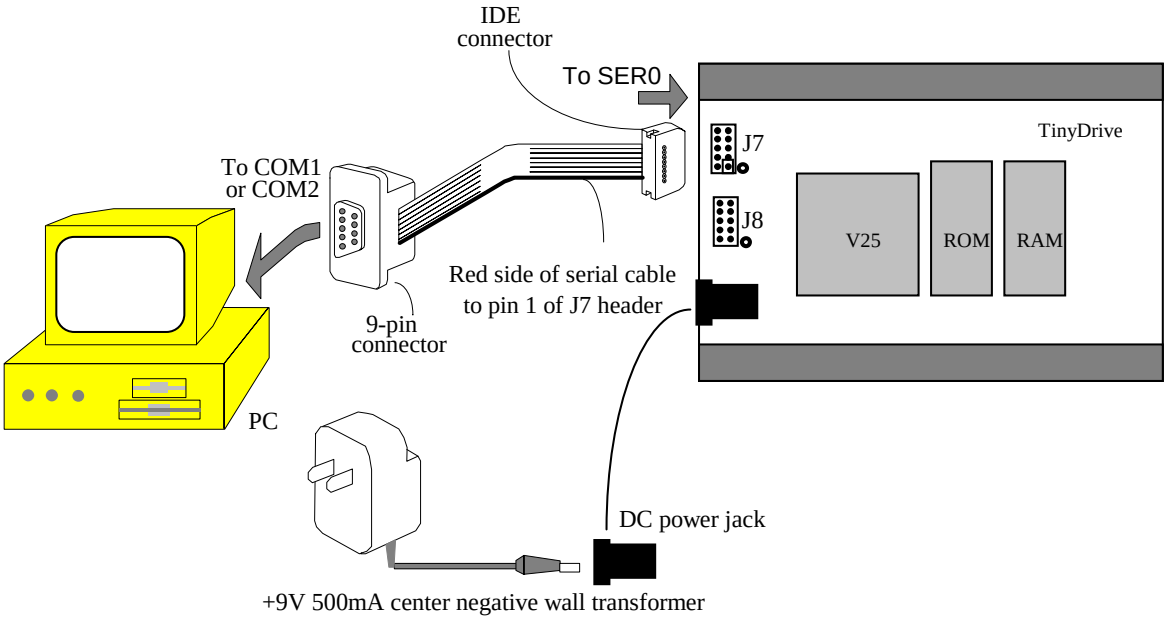


Figure 1.4 Step 1 connections for the TinyDrive

1.4.2 Step 2

STEP 2: Standalone Field Test.

- Set the jumper on J3 pins 3 and 4 on the TinyDrive (Figure 1.5).
- At power-on or reset, if J3 pin 3 (P02) is low, the CPU will run the code that resides in the battery-backed SRAM.
- If a jumper is on J3 pins 3-4 at power-on or reset, the TD will operate in Step Two mode. If the jumper is off J3 pins 3-4 at power-on or reset, the TD will operate in Step One mode. The status of J3 pin 3 (signal P02 of the NEC V25) is only checked at power-on or at reset.

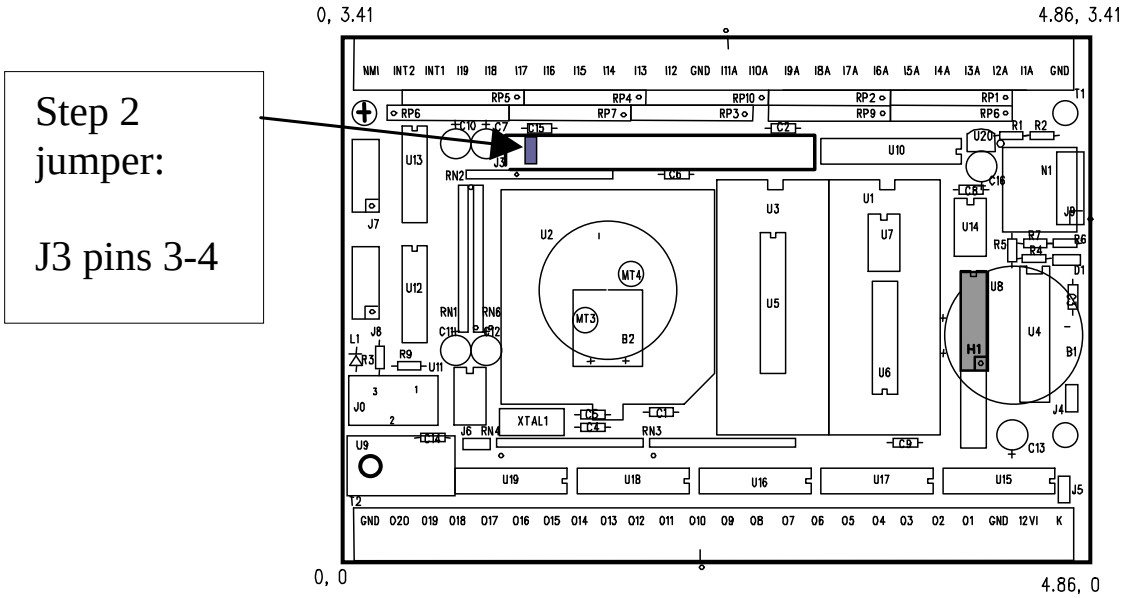


Figure 1.5 Location of Step 2 jumper on the TinyDrive

1.4.3 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 Two test, you can go back to your PC to generate your application ROM to replace the DEBUG ROM (*TDREM_V25*). You need to change ***DEBUG=1*** to ***DEBUG=0*** in the makefile.

You need to have the DV Kit to complete Step Three.

Please refer to the Tutorial of the Technical Manual of the EV/DV Kit for further details on programming the TinyDrive.

1.5 Minimum Requirements for TinyDrive System Development

1.5.1 Minimum Hardware Requirements

- PC or PC-compatible computer with serial COMx port that supports 115,200 baud
- TinyDrive controller with DEBUG ROM *TDREM_V25*
- DB9-IDE10 (PC-V25) serial cable (RS232; DB9 connector for PC COM port and IDC 2x5 connector for controller)
- center negative wall transformer (+9V 500 mA)

1.5.2 Minimum Software Requirements

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

The C/C++ Evaluation Kit (EV) and C/C++ Development Kit (DV) are available from TERN. The EV Kit is a limited-functionality version of the DV Kit. With the EV Kit, you can program and debug the TinyDrive in Step 1 and Step 2, but you cannot run Step 3. In order to generate an application ROM/Flash file, make production version ROMs, and complete the project, you will need the Development Kit (DV).

Chapter 2: Installation

2.1 Software Installation

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

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

2.2 Hardware Installation

Overview

- Connect PC-V25 cable:
  For debugging (Step One), place connector on SER0 with red edge of cable at pin 1
- Connect wall transformer:
  Connect 9V wall transformer to power and plug into power jack

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

2.2.1 Connecting the TinyDrive to the PC

The following diagram (Figure 2.1) illustrates the connection between the TinyDrive and the PC. The TinyDrive is linked to the PC via a serial cable (PC-V25).

The *TDREM_V25* DEBUG ROM communicates through SER0 by default. Install the 5x2 IDC connector on the SER0 header of the TinyDrive. ***IMPORTANT:*** *Note that the* ***red*** *side of the cable must point to pin 1 of the TinyDrive J7 header.* The DB9 connector should be connected to one of your PC's COM Ports (COM1 or COM2).

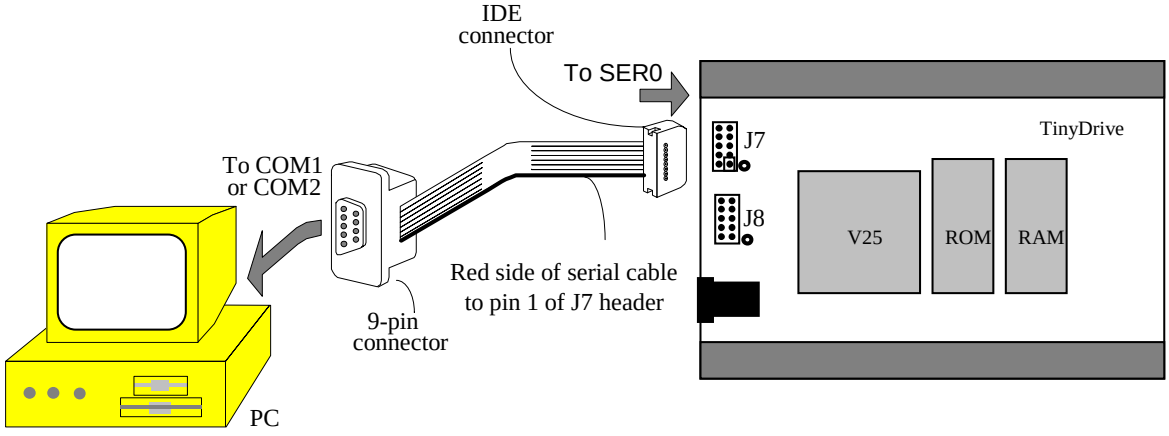


Figure 2.1 Connecting the TinyDrive to the PC

2.2.2 Powering-on the TinyDrive

Connect a wall transformer +9V DC output to the TinyDrive's DC power jack.

The on-board LED should blink twice and remain on after the TinyDrive is powered-on or reset (Figure 2.2).

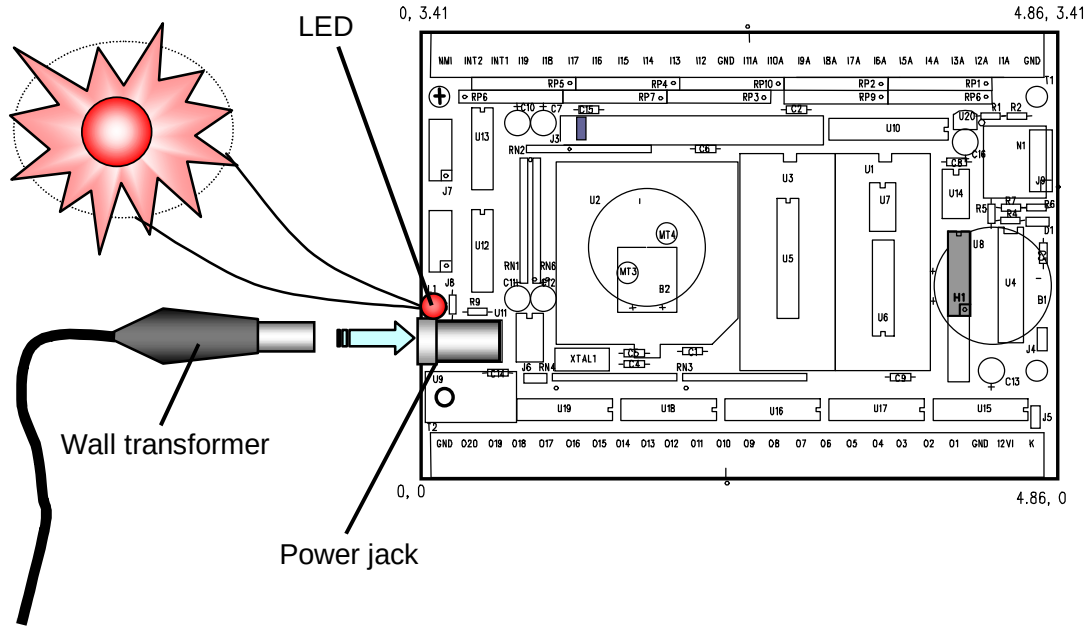


Figure 2.2 The LED blinks twice after the TinyDrive is powered-on or reset

Chapter 3: Hardware

3.1 NEC V25 - Introduction

The NEC V25 is based on industry-standard x86 architecture. The NEC V25 controllers are a higher-performance, more integrated versions of the 80C188 microprocessors. In addition, the NEC V25 has new peripherals including 256 bytes built-in RAM, high speed serial ports, parallel I/O ports, comparator ports, timers, and DMA. The on-chip system interface logic can minimize total system cost.

3.2 NEC V25 – Features

3.2.1 Clock

Due to its integrated clock generation circuitry, the NEC V25 microcontroller allows the use of a selectable system clock frequency of *fx/2*, *fx/4* and *fx/8*. The design achieves a maximum 8 MHz CPU operation, while using a 16 MHz crystal.

A built-in clock generator supplies various clocks to the CPU and peripheral hardware. The TinyDrive uses a 16 MHz crystal. The default system clock output after initialization is 8 MHz on the CLK. The clock cycle is 125 ns. The normal bus cycle requires two clock cycles, which is 250 ns. With built-in wait state generation, up to 2 wait states can be inserted. With the default initialization of 2 wait states, EPROMs of 120 ns to 150 ns can be used. A time base counter operates continuously since the TinyDrive is powered on. It provides clock signals for two 16-bit timers, baud rate generator, refresh timing, refresh address, and time base interrupt request flag. A time base interrupt may be generated at four different intervals: 128 us, 1.024 ms, 8.192 ms, and 131.072 ms, selectable by software.

3.2.2 External Interrupts

The V25 processor has a built-in, high-performance interrupt controller that can control multiple processing of 17 interrupt sources. /NMI, /INT1 and /INT2 are routed to the TinyDrive terminal block T2 via a buffer resistor pack, RP6.

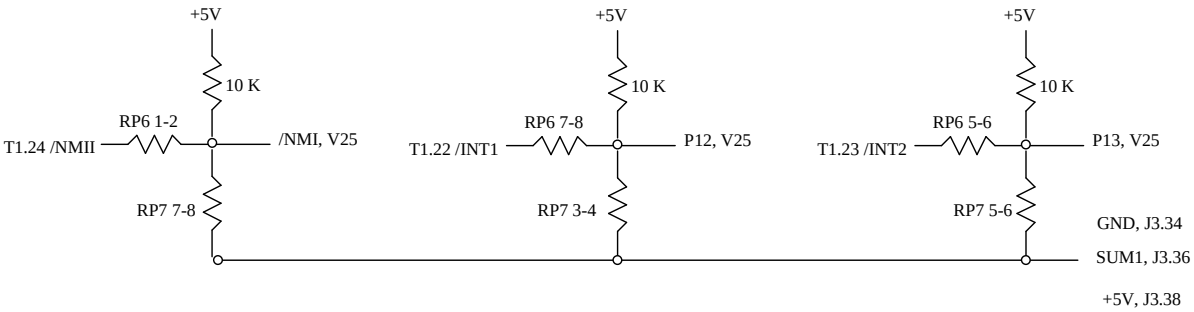


Figure 3.1 Protective resistors and landing resistors for interrupt inputs, /NMI, /INTP1, /INTP2

In order to support interrupt input voltage level 0 to 24V, protective resistor networks are built into the circuit, as shown in Figure 3.1. The locations of RP6 and RP7 are shown in Figure 3.2.

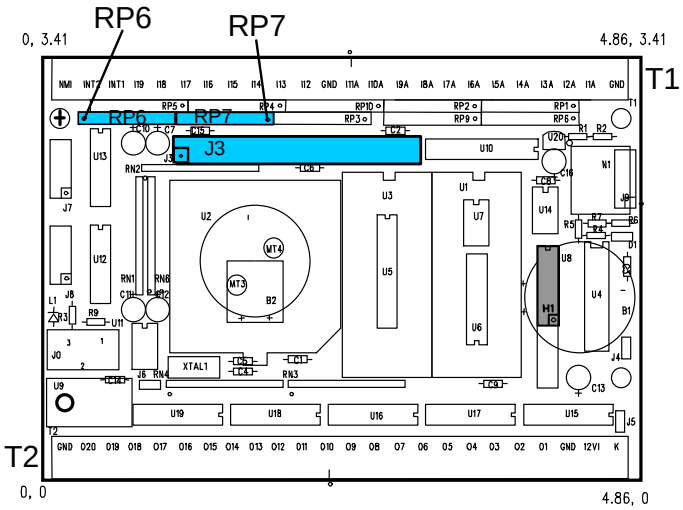


Figure 3.2 Position of the protection resistors and the landing resistors for interrupt inputs.

The interrupt inputs /NMI, /INT1, and /INT2 are falling edge active. Users should be aware that the maximum input voltage at the V25 chip is +5V. A valid input low voltage is less than 0.8 V and a valid input high voltage is higher than 4V and less than 5V.

If your input voltage has to be higher than 5V, the landing resistor RP7 is 10K, the protection resistor RP6 is 10K, and a jumper must be on J3 pins 34-36 (SUM1=GND).

If the inputs are normally at 5V maximum, do not install a jumper on J3 pins 34-36 or pins 36-38 (SUM1=open).

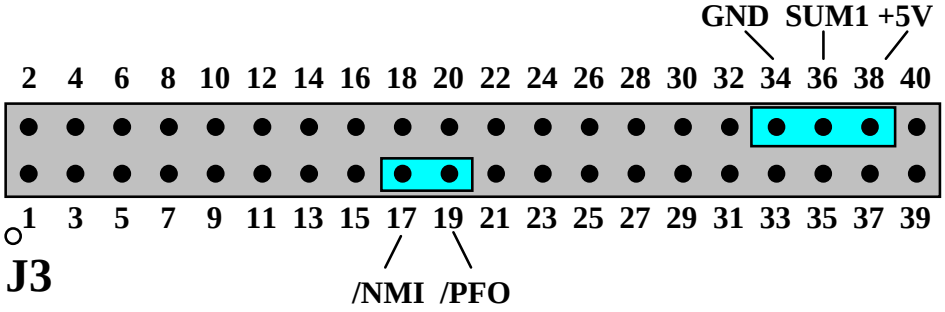


Figure 3.3 Use of SUM, SUM1, /PFO and /NMI

3.2.3 Asynchronous Serial Ports

The NEC V25 CPU has two asynchronous serial channels: SER0 and SER1. Both asynchronous serial ports support the following:

- Full-duplex operation
- 7-bit and 8-bit data transfers
- Odd, even, and no parity
- One or two stop bits

- Error detection
- Hardware flow control
- Transmit and receive interrupts for each port
- Independent baud rate generators

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

An optional external SCC2691 UART is located in U8. For more information about the external UART SCC2691, please refer to section 3.4.4 and Appendix B.

3.2.4 Timer Control Unit

The NEC V25 CPU has two 16-bit programmable timers: Timer 0 and Timer 1. Both programmable 16-bit timers are comprised of a 16-bit modulo register, a 16-bit timer register, and an 8-bit control register.

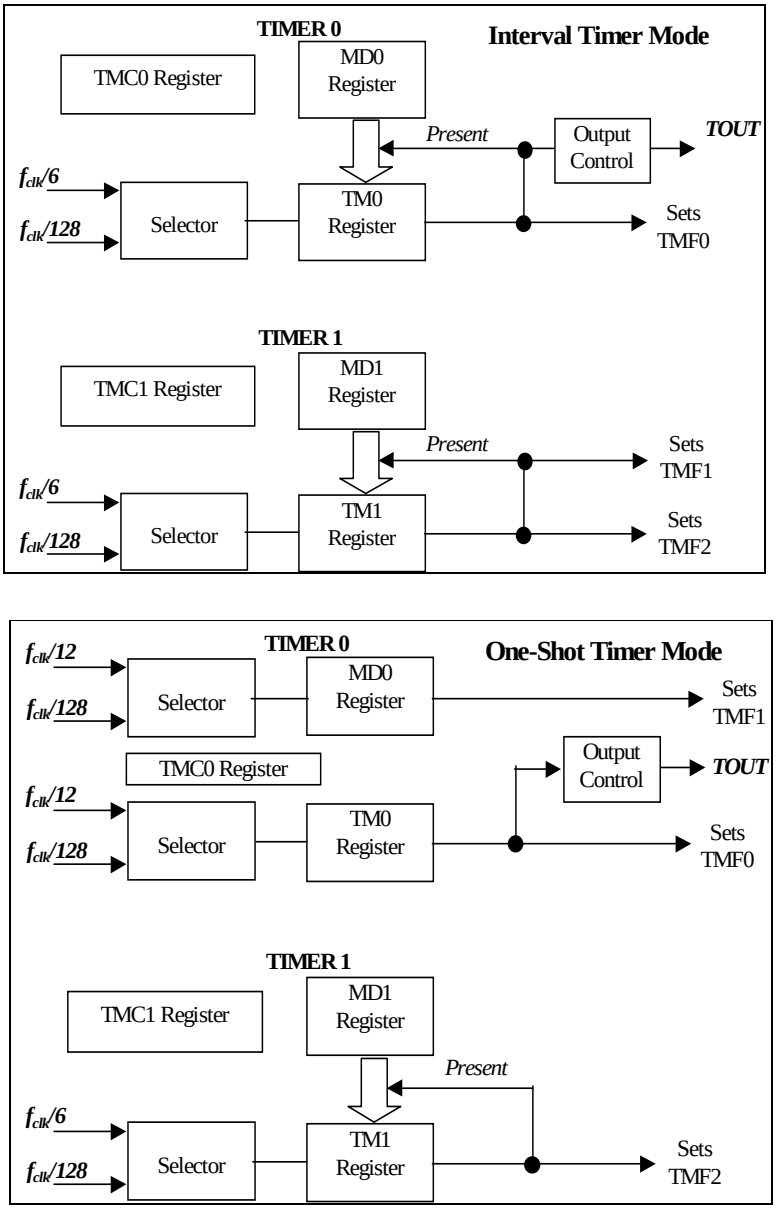


Figure 3.4 Interval Timer Mode and One-Shot Timer Mode Configuration

Timer0 can be programmed as an interval timer or as a one-shot timer. In interval timer mode, the MD0 register value is set to the TM0 register, and then the TM0 countdown begins (Figure 3.4). When TM0 underflows, the TMF0 output is set to 01 and the MD0 register is again set to TM0. The countdown clock, f_{clk}, is divided by 6 or 128, as defined in the TMC0 register. The square wave generated by Timer0 can be output to ***TOUT*** (T2 O17 = P15) on U19 pin 3.

As a one-shot timer, Timer 0 is configured as two independent timers that count down from the value set in MD0 and TM0 (Figure 3.4). The countdown frequency is divided by 12 or 128. If the counter is stopped by either reaching 0 in the count or by setting the TS0 bit to 0 (STOP = 0), a single pulse outputs at ***TOUT***.

Timer1 can only act as an interval timer and has no external output.

3.2.5 Standby Modes

The TinyDrive is an ideal core module for applications that require low power consumption. The V25 CPU has two standby modes, HALT and STOP mode, which reduce power consumption and heat dissipation, thereby extending battery life in portable systems.

In HALT or STOP mode, operation of the CPU clock is stopped and program execution is halted. All registers and RAM content are preserved. HALT mode can drop the power consumption to 50 mA. When an interrupt occurs, it automatically returns to normal operation.

The difference between HALT mode and STOP mode is that HALT mode allows peripheral hardware (such as serial ports, DMA controller, etc.) to function. STOP mode disables all devices. The following table shows which devices are active and which are inactive during HALT and STOP modes.

Item	HALT Mode	STOP Mode
Oscillator	Operates	Stops
Internal System Clock	Stops	Stops
16-bit timer	Operates	Stops
Time Base counter	Operates	Stops
HOLD circuit	Operates	Stops
Serial interface	Operates	Stops
Interrupt request controller	Operates	Stops
DMA controller	Operates	Stops
I/O lines	Data Retained	Data Retained

Table 3.1 Hardware Status During Standby Mode

To release stop mode, /NMI or /RESET must be triggered. A non-maskable interrupt request, DMA request, macro service request, or a reset will release HALT mode. Since the serial ports are functional during HALT mode, it is possible to send a break command to the serial Port To resume operations.

The VOFF pin (J3.15) can be connected to /NMI (J3.17) to wake up the V25 from STOP mode.

3.3 NEC V25 I/O Ports

3.3.1 Port 0, 1, and 2

The NEC V25 has three 8-bit user-programmable I/O ports available. The 24 bi-directional I/O ports (0-2) are multiplexed with different functions. Individual I/O lines can be specified as an input, output, or control line. Each port is controlled by a Port Mode Control Register (PMC), a Port Mode Register (PM), and a Port Data Register (P). You can write or read these registers via the following functions:

pokeb(0xfff0, 0x??, 0x!!)
or ***peekb(0xfff0, 0x??)***

where ***??*** is the register offset address and ***!!*** is the control/data byte.

The following is a list of the register addresses.

Register Symbol	Register Offset Address	R/W	Access Units (bits)
P0	0x00	R/W	8/1
PM0	0x01	W	8
PMC0	0x02	R/W	8/1
P1	0x08	R/W	8
PM1	0x09	W	8/1
PMC1	0x0A	R/W	8
P2	0x10	R/W	8/1
PM2	0x11	W	8
PMC2	0x12	R/W	8/1

After power-on/reset, I/O 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 3.2.

Port	PMC_n = 1	PMC_n = 0		Status after	TinyDrive
I/O	PM = X	PM_n = 1	PM_n = 0	td_init()	Location/Function
P00	-	Input	Output	Input	EEPROM (U7 pin 6) clock SCL
P01	-	Input	Output	Input	EEPROM (U7 pin 5) data SDA
P02	-	Input	Output	Input	J3 pin 3=4, Step 2 jumper If low, V25 runs code starting at 0800:0000, based on EE setting.
P03	-	Input	Output	Output	J3 pin 11. HWD (hit watchdog)
P04	-	Input	Output	Input	WDO (Read watchdog output), if low, watchdog time-out reset
P05	-	Input	Output	Output	LED control or U19 pin 1
P06	-	Input	Output	Input	J3 pin 5.
P07	CLKOUT	Input	Output	CLK	Internal use.
P10	-	/NMI	-	/NMI	J3 pin 17, and T1.24 via buffer resistor RP6.
P11	-	/INTP0	-	/INTP0	SCC2691 UART interrupt (if installed)
P12	-	/INTP1	-	/INTP1	T1.22
P13	/INTAK	/INTP2	-	/INTP2	T1.23.
P14	INT	/POLL	Output	Output	RTS1 for SER1 and U19.15=

Port I/O	PMC_n = 1 PM = X	PMC_n = 0 PM_n = 1	PM_n = 0	Status after td_init()	TinyDrive Location/Function
					HV O16
P15	TOUT	Input	Output	Output	T2.5 HV O17
P16	/SCK0	Input	Output	Output	RTS0 for SER0, T2.4 HV O18
P17	READY	Input	Output	READY	RP6.3, internal use.
P20	DMARQ0	Input	Output	Output	HV O19, T2.3
P21	/DMAAK0	Input	Output	Output	HV O20, T2.2
P22	/TC0	Input	Output	Input	J3 pin 1
P23	DMARQ1	Input	Output	Output	EN485 for SCC RS-485 driver. If low, receiving.
P24	/DMAAK1	Input	Output	Output	12-bit ADC CLK
P25	/TC1	Input	Output	Output	12-bit ADC DIN
P26	/HLDAK	Input	Output	Input	12-bit ADC DOUT
P27	HLDRQ	Input	Output	Output	12-bit ADC CS, pulled high

Table 3.2 I/O pin default configuration

TinyDrive I/O initialization in **ve_init()** is listed below:

```
pokeb(0xfff0,0x02,0x80);   /* Set PMC0 P07=CLK   */
pokeb(0xfff0,0x01,0xd7);   /* Set PM0 for input, P05=LED P03=HWD output */
pokeb(0xfff0,0x0a,0x80);   /* Set PMC1 P17 for READY */
pokeb(0xfff0,0x09,0xaf);   /* Set PM1 for input, P14=RTS1,P16=RTS0 OUTPUT */
pokeb(0xfff0,0x12,0x00);   /* Set P20-P27 for port mode */
pokeb(0xfff0,0x11,0xf7);   /* Set PM2 for input, P23=EN485 output */
```

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

void ***port_init***(char ***p***, unsigned char ***pmc***, unsigned char ***pm***);

Where ***p*** = port 0, 1 or 2.

pmc = define each pin as CONTROL or I/O (0 = I/O; 1 = CONTROL).
pm = define each I/O pin as input or output (0 = output; 1 = input).

3.3.2 Port T Comparator Inport

Port T is an 8-bit input port whose threshold voltage can be changed in 16 steps. Each Port T input is compared with the selected threshold voltage (Vth). PTn > Vth results in a value 1, PTn < Vth results in a value 0. All eight results from PT0 to PT7 are latched to the Port T input latches.

The threshold voltage VTH is fixed to +5V (J3 13-14) by default. The comparator of each input can set the reference voltage to one of 16 levels (1/16 x VTH to 16/16 x VTH). This provides users with an easy and inexpensive way to measure analog input signals, in 4-bit resolution.

The comparator 8-bit latch can be accessed by the function ***portt_rd***(void), which returns the 8-bit result. VTH can be changed by the function ***portt_wr***(char *vref*). The variable *vref* {0 .. 15} sets the reference voltage by the following equation: Reference = ***Vth * vref*/16**. ***vref*** = 0 sets Reference = VTH.

J3.36=J3.38 SUM1=VCC. PT0-7 pulled high

J3.36=J3.34 SUM1=GND. PTO-7 pulled low

Port I/O	**TinyDrive** Location/Function
PT0	I12, T1.14
PT1	I13, T1.15
PT2	I14, T1.16
PT3	I15, T1.17
PT4	I16, T1.18
PT5	I17, T1.19
PT6	I18, T1.20
PT7	I19, T1.21

Table 3.3 I/O pin default configuration

3.4 I/O Mapped Devices

3.4.1 I/O Space

External I/O devices use I/O mapping. You may access I/O with ***inportb***(port) or ***outportb***(port,dat). The external I/O space is 64K, ranging from 0x0000 to 0xffff. In the I/O space of 0x0000-0x7fff, the I/O access time is 500 ns. In the I/O space of 0x8000-0xffff, the I/O access time is 250 ns. Table 3.4 shows more information on I/O mapped devices.

I/O space	**Usage**
0x0000	HV1 high voltage driver for U15 and U17
0x4000	HV2 high voltage driver for U16 and U18
0x8000	RTC U4
0xc000	SCC U8

Table 3.4 I/O Mapped devices

3.4.2 74HC259

The 74HC259 8-bit decoder latch provides eight additional output lines for the TD. The U15 74HC259 is mapped in the I/O address space 0x0000 (selected by HV1). The U16 74HC259 is mapped in the I/O address space 0x4000 (selected by HV2). You may access this device by using the following code. The output of U15 drives the high voltage driver U17. The output of U16 drives the high voltage driver U18. *See the schematics for the TD-A (TinyDrive version A).*

```
outportb(0x0000 + i, val); // U16  i = output pin, val = 0/1 to set or reset latch.
outportb(0x4000 + i, val); // U15  i = output pin, val = 0/1 to set or reset latch.
```

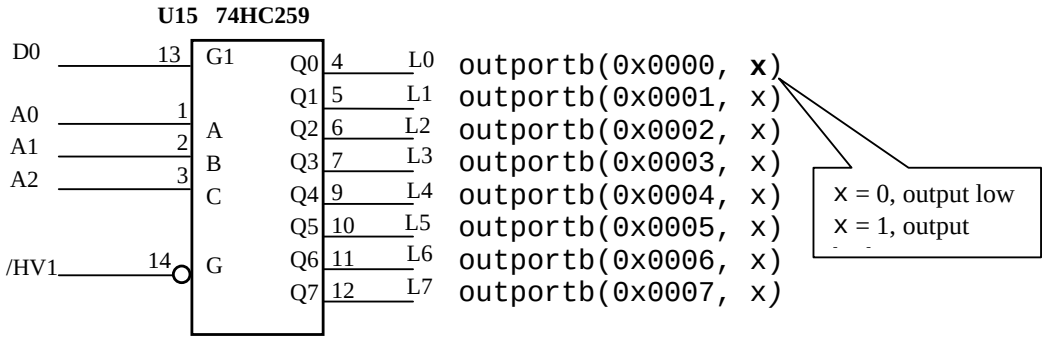


Figure 3.5 74HC259 diagram with corresponding outport addresses

3.4.3 Real-time Clock RTC72421

If installed, a real-time clock RTC72421 (EPSON, U4) is mapped in the I/O address space 0x8000-0xffff. 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).

It is also possible to configure the real-time clock to raise an output line attached to an external interrupt, at 1/64 second, 1 second, 1 minute, or 1 hour intervals. This can be used in a time-driven application, or the **VOFF** signal can be used to turn on/off the controller using an external switching power supply. An example of a program showing a similar application can be found in **`tern\v25\samples\ve\poweroff.c`**.

3.4.4 UART SCC2691

The UART SCC2691 (Signetics, U8) is mapped in the I/O address space 0xc000-0xffff. The SCC2691 offers the following:

- a full-duplex asynchronous receiver/transmitter
- a quadruple buffered receiver data register
- an interrupt control mechanism
- programmable data format
- selectable baud rate for the receiver and transmitter
- a multi-functional and programmable 16-bit counter/timer
- an on-chip crystal oscillator
- a multi-purpose input/output, including RTS and CTS mechanism

For more information, refer to Appendix B. The SCC2691 on the TinyDrive may be used as a network 9th-bit UART. An RJ11-6 phone connector N1 provides network signals. Use N1 pin 3 (RS485+) and pin 4 (RS485-), or J9 to join the multi-drop RS485 twist pair network.

3.5 Other Devices

A number of other devices are also available on the TinyDrive. Some of these are optional, and might not be installed on the particular controller you are using. For a discussion regarding the software interfaces 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 TinyDrive has several functions that significantly improve system reliability:

- watchdog timer
- power-failure warning
- battery backup
- power-on-reset delay
- power-supply monitoring

Watchdog Timer

The watchdog timer is activated by setting a jumper on J3 pins 11-12 of the TinyDrive. 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 P03=HWD pin of the MAX691) should be arranged such that the HWD pin is accessed at least once every 1.6 seconds. If the J3 11=12 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 TinyDrive is reset, WDO remains low until a transition occurs at the WDI pin of the MAX691. When controllers are shipped from the factory the J3 11=12 jumper is off, which disables the watchdog timer.

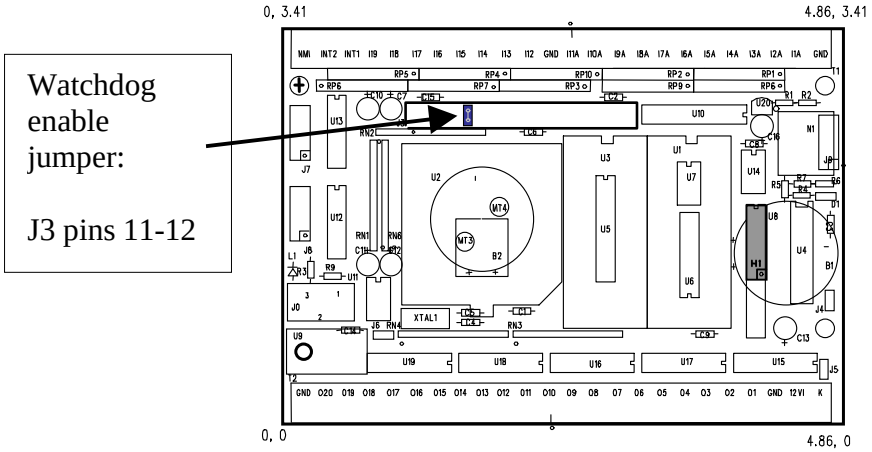


Figure 3.6 Location of watchdog timer enable jumper

Power-failure Warning

The MAX691/LTC691 PFO (Power Failure Output) pin may be connected to NMI via J3 pins 17-19. The user may connect the PFI (Power Failure Input) pin of MAX691 to an external voltage divider to monitor the power voltage level (Figure 3.7). The PFI pin has been pulled high to VCC with a 10K resistor on the TinyDrive. When the external DC power drops to 8.35 V, the voltage on the PFI is less than 1.3 V, the MAX691 will pull down PFO pin, and NMI will occur. You can write an NMI interrupt service routine to meet your requirements.

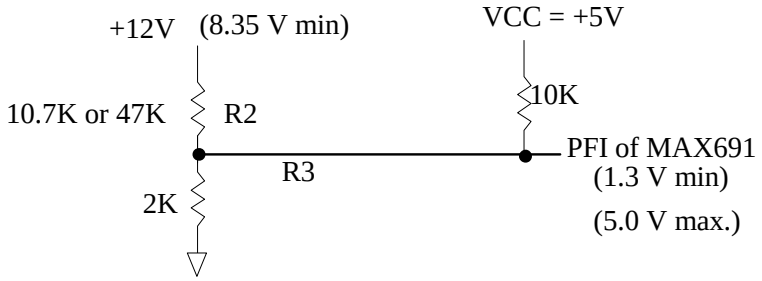


Figure 3.7 Power-fail detection with PFI

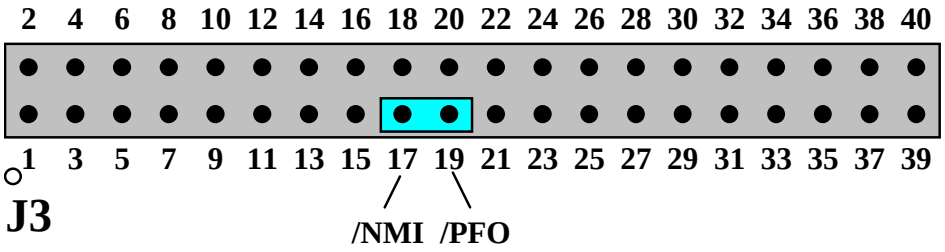


Figure 3.8 Use of /PFO and /NMI

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 VRAM (power for SRAM and RTC). Thus, the SRAM and the real-time clock RTC72421 are backed up. In normal use, the lithium battery should last approximately 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.

NOTE: **When there is no battery on the TinyDrive, the VBAT signal should be shorted to ground**.

3.5.2 EEPROM

A serial EEPROM of 128 bytes (24C01), 512 bytes (24C04), or 2K bytes (24C16) can be installed in U7. The TinyDrive uses the P00=SCL (serial clock) and P01=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, and the data retention is more than 40 years. EEPROM can be read and written to by simply calling the functions **ee_rd()** and **ee_wr()**.

In order to use the EEPROM correctly, J3 pins 18 and 20 must be connected (see Figure 3.9).

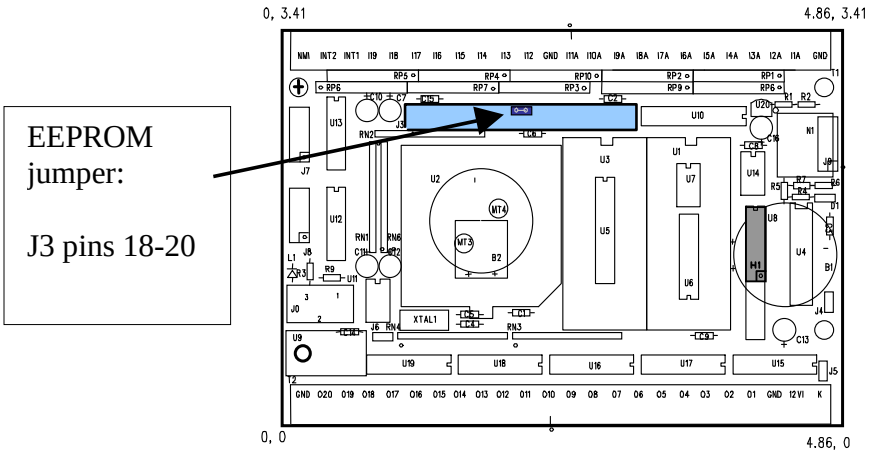


Figure 3.9 Location jumper for EEPROM

3.6 Inputs and Outputs

The TinyDrive offers 22 inputs and 20 outputs, as shown below.

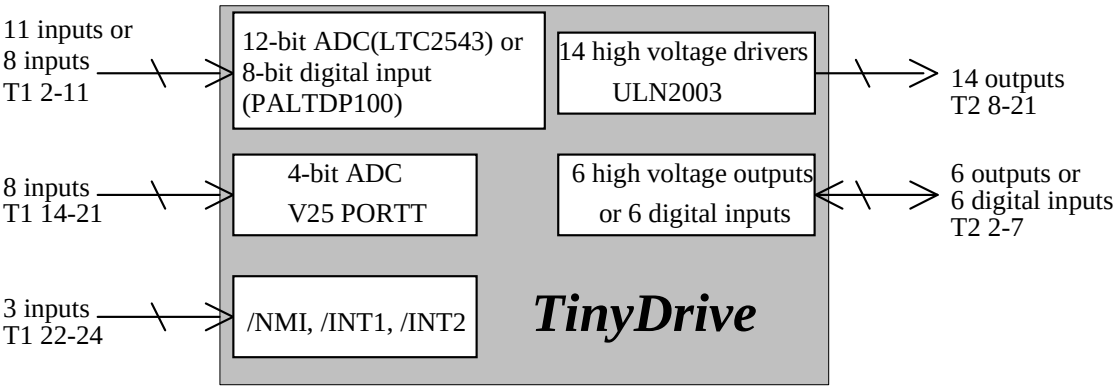


Figure 3.10 Functional block diagram of inputs and outputs for the TinyDrive

3.6.1 12-bit ADC (TLC2543)

The TLC2543 is a 12-bit, switched-capacitor, successive-approximation, 11 channels, serial interface, analog-to-digital converter. It has three control inputs (/CS=P27; CLK=P24; DIN=P25) and is designed for communication with a host through a serial tri-state output (DOUT=P26).

The ADC digital data output communicates with a host through a serial tri-state output (DOUT=P26). If P27=/CS is low, the TLC2543 will have output on P26. If P27=/CS is high, the TLC2543 is disabled and P26 is free. P27 and P11 are pulled high by 10K resistors on board. The TLC2543 has an on-chip 14-channel multiplexer that can select any one of 11 inputs or any one of three internal self-test voltages. The sample-and-hold function is automatic.

TLC2543 features differential high-impedance inputs that facilitate ratiometric conversion, scaling, and isolation of analog circuitry from logic and supply noise. A switched-capacitor design allows low-error conversion over the full operating temperature range. The analog input signal source impedance should be less than 50Ω and capable of slewing the analog input voltage into a 60 pF capacitor.

A reference voltage less than VCC (+5V) can be provided for the TLC2543 if additional precision is required. A voltage above 2.5V and less than +5V can be used for this purpose, and can be connected to the **REF+** pin.

The reference voltage REF+ can be tied to VCC for ratiometric application via a precision reference, such as LT1029 (5V), U20. By default, REF+ is pulled up to VCC by R8.

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

In order to operate the TLC2543, five V25 I/O lines are used, as listed below:

/CS	Chip select = P27, high to low transition enables DOUT, DIN and CLK. Low to high transition disables DOUT, DIN and CLK.
DIN	P25, serial data input
DOUT	P26, 3-state serial data output.
EOC	P13, End of Conversion, high indicates conversion complete and data is ready
CLK	I/O clock = P24
REF+	Upper reference voltage (normally VCC)
REF-	Lower reference voltage (normally GND)
VCC	Power supply, +5 V input
GND	Ground

The analog inputs AD0 to AD10 (I1A to I11A) are available at T1 terminal 1 through 12.

RP8, RP9, and RP10 are 10K landing resistors for ADC inputs AD0-AD10. Their locations are shown in Figure 3.11.

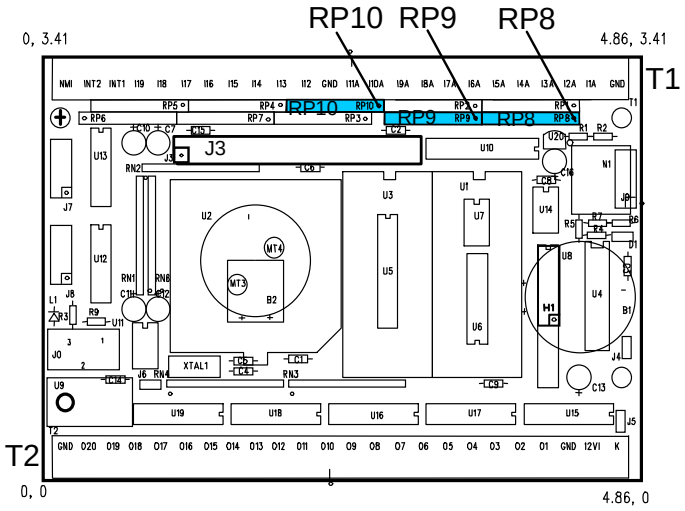


Figure 3.11 Position of the protection resistors and the landing resistors for ADC inputs.

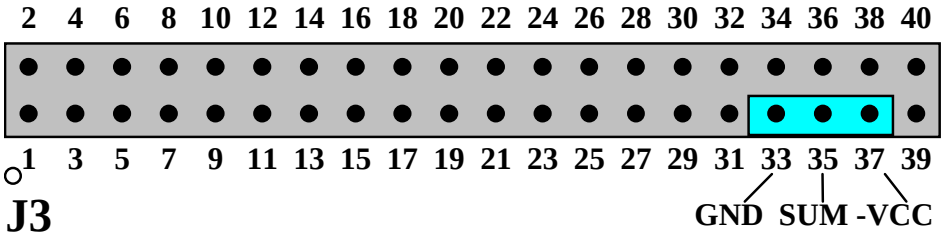


Figure 3.12 Use of SUM.

Note that if the 12-bit ADC is not needed, a ULN2003 can optionally be installed in the U10 socket. See section 3.6.5 for details.

3.6.2 Dual 12-bit DAC

If high-voltage drivers are not needed in U17, U18, or U19, the user may install DAC LT1446 in those sockets.

The LTC1446/LTC1446L is a dual 12-bit digital-to-analog converter (DAC) in a DIP-8 package. It is complete with a rail-to-rail voltage output amplifier, an internal reference and a 3-wire serial interface. The LTC1446 outputs a full-scale of 4.096V, making 1 LSB equal to 1 mV. The LTC1446L outputs a full-scale of 2.5 V, making 1 LSB equal to 0.61 mV.

The buffered outputs can source or sink 5 mA. The outputs swing to within a few millivolts of supply rail when unloaded. They have an equivalent output resistance of 40 Ω when driving a load to the rails. The buffer amplifiers can drive 1000 pf without going into oscillation.

The DAC chips may be installed in U17, U18, and U19 on the TinyDrive.

See sample program in **`c:\tern\v25\samples\td\td_da12.c`**.

3.6.3 Protective resistor networks for inputs

In order to support digital signal input voltage level 0 to 24V, protection resistor networks are built into the circuit. Beware that the maximum input voltage at the V25 or ADC chip is +5V. A valid input low voltage is less than 0.8V and the input high voltage is higher than 4V and less than 5V.

For ADC applications, if the analog inputs are 0-10V, a landing resistor must be installed to divide the input voltage to 5V maximum.

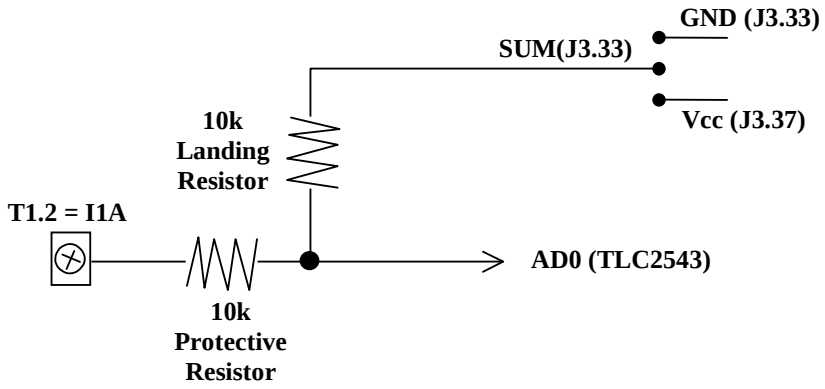


Figure 3.13 ADC/Digital Inputs resistor configuration.

Input	Landing	Protective
ADC: AD0-10	RP8, RP9, RP10	RP1, RP2, RP3
Comparator: PT0-7	RP6	RP3, RP4, RP5
Interrupts: NMI, INTP1, INTP2	RN1, RN2	RP7

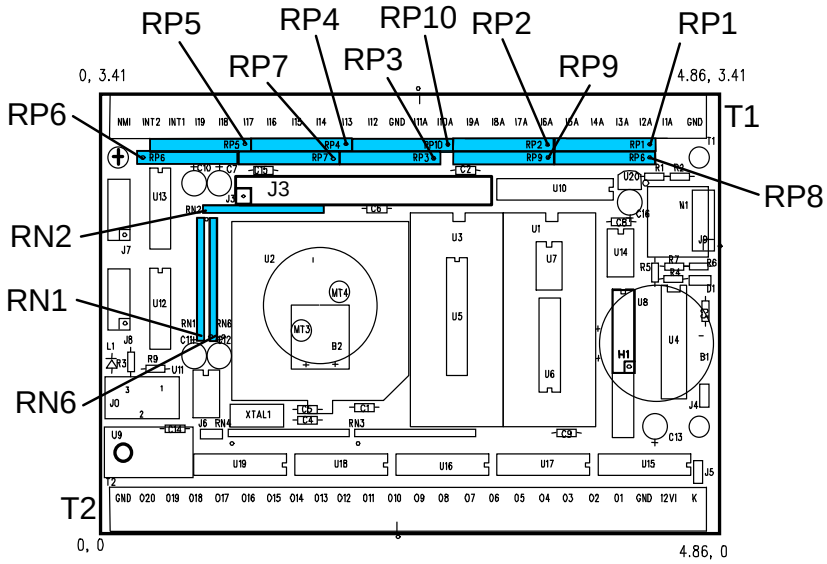


Figure 3.14 Locations of user configurable protective resistors and landing resistors.

If the analog inputs are less than 5V, do not install the landing resistors RP8/RP9/RP10. RP8/RP9/RP10 may be pulled high or low with a jumper connecting SUM to VCC or GND at J3 pins 35-37 or pins 33-35.

For 0-24V digital input applications, if the inputs are normally 24V, and grounding the input as a trigger signal, the landing resistors are 1K and the protection resistors are 10K, SUM=GND, and a jumper is on J3 pins 33-35.

If the inputs are normally at 5V maximum, and grounding as the trigger signal, the protection resistor is 1K and no landing resistors are installed.

For additional information on landing and protection resistors for interrupt inputs, see section 3.2.2.

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, and integral protection diodes for driving inductive loads. Peak inrush currents of up to 600 mA sinking are allowed.

Up to four ULN2003 can be installed on the TinyDrive. Three ULN2003 may be installed in U17, U18, and U19. One may also be installed in U10, replacing the TLC2543 12-bit ADC (for details, refer to section 3.6.5). These outputs may be paralleled to achieve high-load capability, although each driver has a maximum continuous collector current rating of 350 mA at 50V. The maximum power dissipation allowed is 2.20 W per chip at 25 degrees C (°C). The common substrate G is routed to T2 GND pins. All currents sinking in must return to the T2 GND pin. A heavy gauge (20) wire must be used to connect the T2 GND terminal to an external common ground return. K connects to the protection diodes in the ULN2003 chips and should be tied to highest voltage in the external load system. K can be connected to an unregulated on board +12V via J5. **ULN2003 is a *sinking* driver, not a sourcing driver.** An example of typical application wiring is shown below.

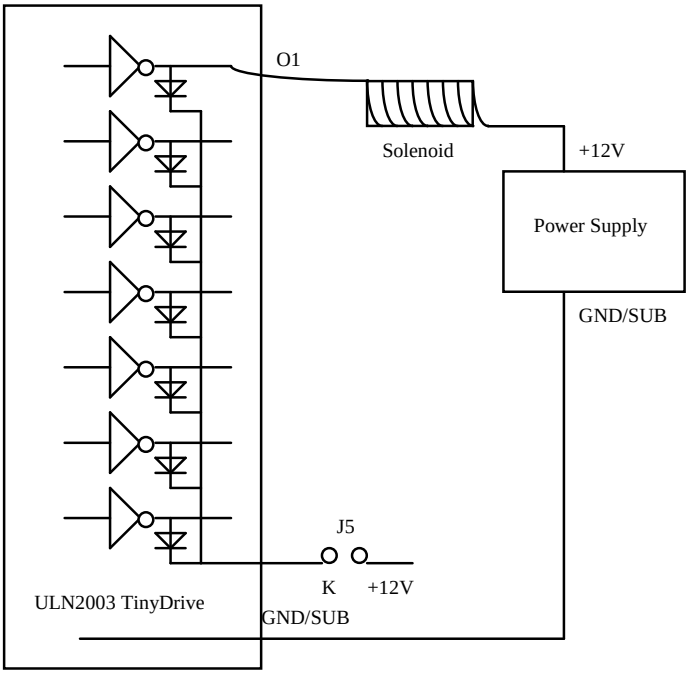


Figure 3.15 Drive inductive load with high voltage/current drivers.

3.6.5 High-voltage Drivers in U10

U10 is a 20-pin socket for 12-bit ADC TLC2543. If users require more high voltage drivers than those already provided on board (U17, U18, U19) instead of 12-bit ADC, a ULN2003 can be installed in the U10 socket.

The digital output signals from the V25 processor to the ULN2003 are P23=REF+, P24, P25, P26, and P27.

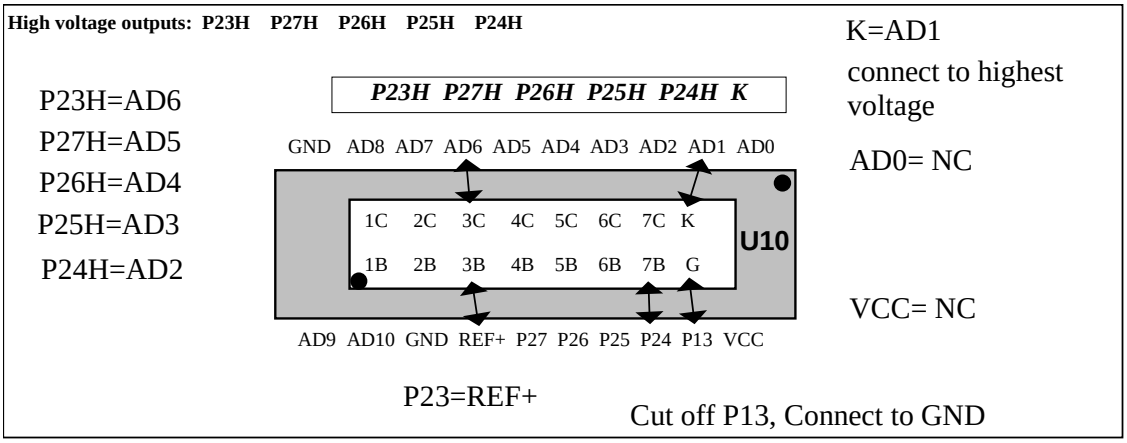


Figure 3.16 Installing high-voltage drivers in U10

To use U10 for high-voltage drivers, cut off the P13 connection to U10 pin 19 and add the following connections:

P23 = REF+ = U10 pin 14
GND = U10 pin 19

High voltage outputs on T1 are as follows:

T1 pin 2 = AD0 = NC = Not connected
T1 pin 3 = AD1 = K = connect to highest voltage in the system (<50V DC), may be connected to T2 pin 24=K
T1 pin 4 = AD2 = P24H
T1 pin 5 = AD3 = P25H
T1 pin 6 = AD4 = P26H
T1 pin 7 = AD5 = P27H
T1 pin 8 = AD6 = P13H

3.7 Headers and Connectors

3.7.1 Jumpers and Headers

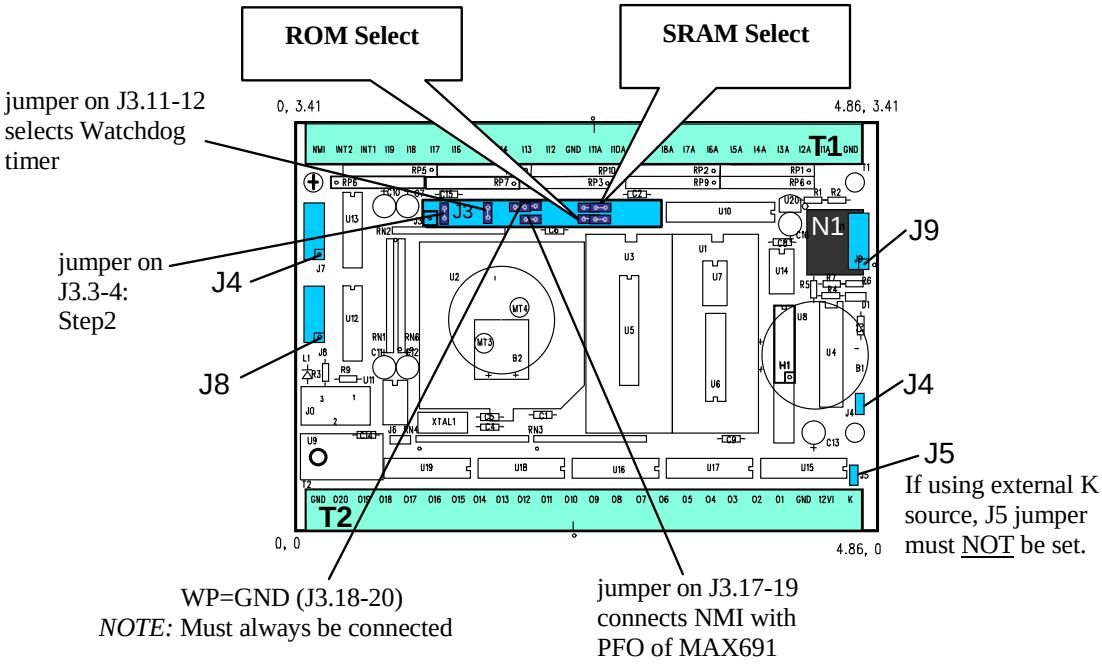


Figure 3.17 Jumpers and headers on the TinyDrive

The following table lists the jumpers and connectors on the TinyDrive:

Name	Size	Function	Possible Configuration
J3	20x2	Main configuration header	Pins 28-30: SRAM 256KB-512KB Pins 30-32: SRAM 32KB-128KB ROM/Flash size selection: Pins 29-31: ROM or Flash size 32KB-128KB Pins 27-29: ROM or Flash size 256KB-512KB ROM 512KB/Flash selection: Pins 24-26: ROM size 512KB Pins 22-24: Flash (all sizes), or ROM < 512 KB Pins 11-12= Watchdog Timer enabled. Else disabled. Pins 17-19: /PFO=/NMI. If jumper is on, when a power failure is sensed /PFO will generate interrupt on /NMI.
J4	2x1	Reset, GND	
J5	2x1	High voltage driver protection diode K pin to +12V	If using an external K source, J5 jumper must NOT be set.
J7	5x2	RS-232 SER0	Default debug port
J8	5x2	RS-232 SER1	for application
J9	5x2	UART SCC2691 RS-485 networking	
N1	RJ11-6	UART SCC2691 RS-485 networking	
T1	24x1	Terminal block	
T2	24x1	Terminal block	

3.7.2 J3 20x2 Header

J3 pin names and functions are as follows:

J3 Signals					
	P22	1	2	GND	
Step2: Jumper on P02=GND (J3.3-4)	P02	3	4	GND	
	P06	5	6	GND	
/RTS0, or U19.2 I/O pin	P14	7	8	GND	
Low active reset, may be connected to /RT for remote reset via N1	/RST	9	10	/RT	remote reset via J9/N1
hit watchdog, toggle by hitwd();	HWD	11	12	WDI	watchdog timer active input, if WDI=HWD
reference voltage for PORTT comparator	VTH	13	14	VCC	
U4 RTC72421 alarm output	VOFF	15	16	VCC	
	/NMI	17	18	WP	Always WP=GND
MAX691 power fail output	/PFO	19	20	GND	**(*NOTE:* this must always be connected)**
	-5V	21	22	R/W	
Landing resistor network RN5, RN6 summing point for inputs	SUM	23	24	AW	For ROM size 512K, AW=A18. For Flash (all sizes) or ROM <512K, AW=R/W.
NC	*(NC)*	25	26	A18	
	A17	27	28	A17	
For 32K-128K ROM/Flash, A17P=VCC. For greater than 128K, A17P=A17.	A17P	29	30	CE2	For 32K-128K SRAM, CE2=VRAM. For greater than 128K, CE2=A17
	VCC	31	32	VRAM	Battery back-up switching power source for SRAM and RTC
	GND	33	34	GND	
Landing resistor network RN5, RN6 summing point for inputs	SUM	35	36	SUM1	Resistor network RP7 summing point for NMI, INTP1, INTP2
	VCC	37	38	VCC	
	GND	39	40	SUM2	NC

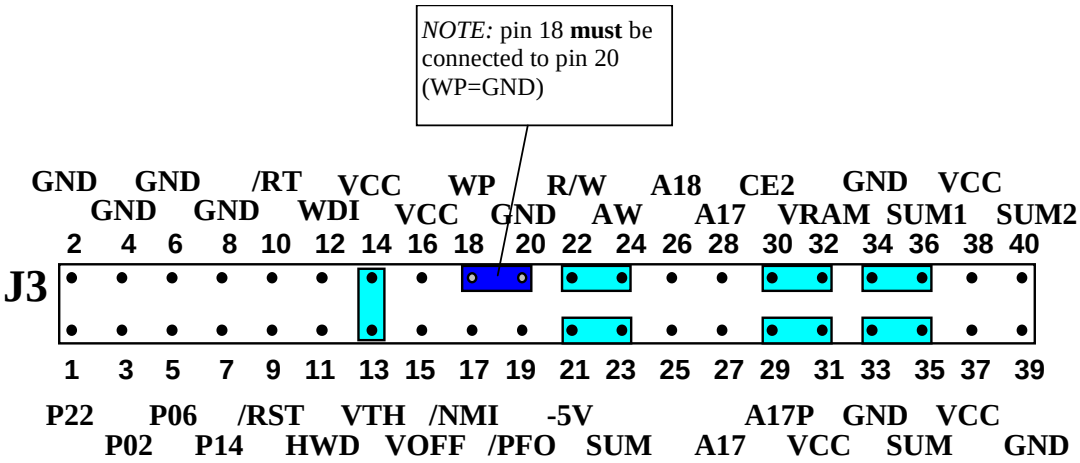


Figure 3.18 Default jumper settings for the main configuration header J3

3.7.3 Terminal Blocks

The TinyDrive has a total of 24x2 positions of terminal blocks. The signals are shown below. By default, T1 is for inputs, and T2 is for outputs.

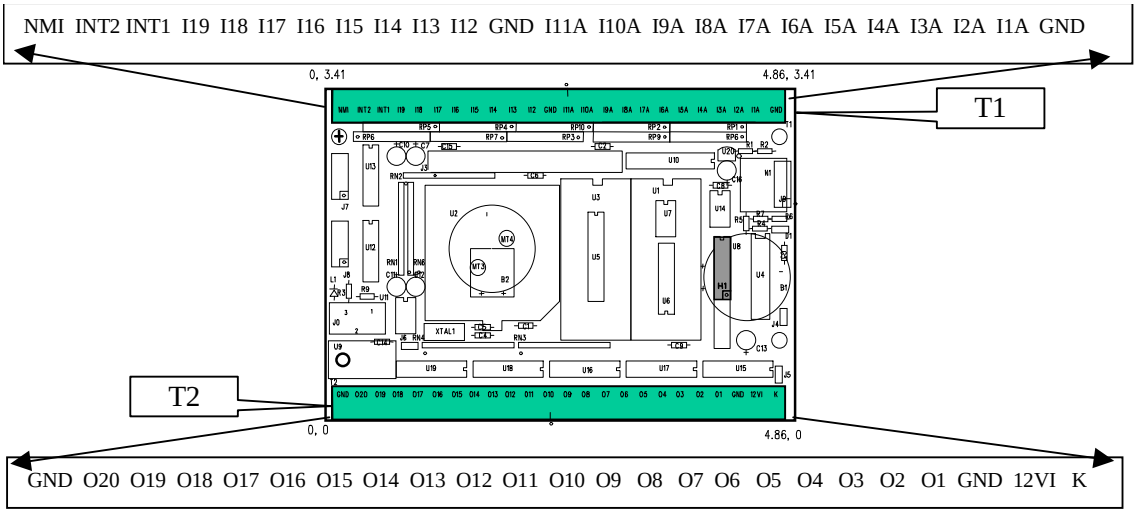


Figure 3.19 Terminal block diagram

The tables below provide a summary of the signals on terminals T1 and T2.

T1 signal	screw	Description
GND	1	Ground
I{1...11}A	1 … 12	Analog inputs
GND	13	Ground
I{12...19}	14 … 21	Port T inputs
INT{1…2}	22 … 22	Interrupt inputs
NMI	24	Interrupt inputs

Table 3.5 Signal definitions on terminal 1 (T1)

T2 signal	screw	Description
GND	1	Ground
O{20 … 1}	2 … 21	High-voltage driver
GND	22	Ground
+12VI	23	Input voltage from power supply
K	24	Protection diode

Table 3.6 Signal definitions on terminal 2 (T2)

Signal definitions for Terminal Block T1:

Pin	Signal	Description
1	GND	
2	I1A	Input to 12-bit ADC AD0/Digital input with PAL(TDP100) installed in U10
3	I2A	Input to 12-bit ADC AD1/Digital input with PAL(TDP100) installed in U10
4	I3A	Input to 12-bit ADC AD2/Digital input with PAL(TDP100) installed in U10
5	I4A	Input to 12-bit ADC AD3/Digital input with PAL(TDP100) installed in U10
6	I5A	Input to 12-bit ADC AD4/Digital input with PAL(TDP100) installed in U10
7	I6A	Input to 12-bit ADC AD5/Digital input with PAL(TDP100) installed in U10
8	I7A	Input to 12-bit ADC AD6/Digital input with PAL(TDP100) installed in U10
9	I8A	Input to 12-bit ADC AD7/Digital input with PAL(TDP100) installed in U10
10	I9A	Input to 12-bit ADC AD8
11	I10A	Input to 12-bit ADC AD9
12	I11A	Input to 12-bit ADC AD10
13	GND	
14	I12	Input to comparator/4-bit ADC PT0
15	I13	Input to comparator/4-bit ADC PT1
16	I14	Input to comparator/4-bit ADC PT2
17	I15	Input to comparator/4-bit ADC PT3
18	I16	Input to comparator/4-bit ADC PT4
19	I17	Input to comparator/4-bit ADC PT5
20	I18	Input to comparator/4-bit ADC PT6
21	I19	Input to comparator/4-bit ADC PT7
22	/INT1	External interrupt input1
23	/INT2	External interrupt input 2
24	/NMI	Non Mask Interrupt Input, can be used to wake up CPU from STOP mode

Signal definitions for Terminal Block T2:

Pin	Signal	Description
1	GND	
2	O20	U19 HV pin 11, as output, or P05 as input, do not install U19, connect 1-16
3	O19	U19 HV pin 12, as output, or P14 as input do not install U19, connect 2-15
4	O18	U19 HV pin 13, as output, or P15 as input do not install U19, connect 3-14
5	O17	U19 HV pin 14, as output, or P16 as input do not install U19, connect 4-13
6	O16	U19 HV pin 15, as output, or P20 as input do not install U19, connect 5-12
7	O15	U19 HV pin 16, as output, or P21 as input do not install U19, connect 6-11
8	O14	U18 HV pin 10
9	O13	U18 HV pin 11
10	O12	U18 HV pin 12
11	O11	U18 HV pin 13
12	O10	U18 HV pin 14
13	O9	U18 HV pin 15
14	O8	U18 HV pin 16
15	O7	U17 HV pin 10
16	O6	U17 HV pin 11
17	O5	U17 HV pin 12
18	O4	U17 HV pin 13
19	O3	U17 HV pin 14
20	O2	U17 HV pin 15
21	O1	U17 HV pin 16
22	GND	
23	+12VI	power supply input, +8V to +24V, polarity protected by a diode.
24	K	highest voltage in the system, protect diode, Use J5 connecting to +12V

A row of 0.1-inch spacing pads is located next to the screw terminal pads for both T1 and T2. These pads are designed for using 0.1-inch spacing pin headers. The even-numbered pins (2, 4, 6, etc.) are no contact. These pads are intended for applications in which the TinyDrive must be plugged into a customer's motherboard.

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 F.

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 value or a 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 VE.LIB

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

Include-file name	Description
VE.H	timer/counter, RTC, Watchdog
SER0.H	Internal serial port 0
SER1.H	Internal serial port 1
SCC.H	External UART SCC2691
VEEE.H	on-board EEPROM
TD.H	ADC, DI, HV
LCD.H	16 character by 2 line (16x2) LCD

4.2 Basic Functions

4.2.1 TinyDrive Initialization

ve_init

This function should be called at the beginning of every program running on TinyDrive 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.

ve_init will initialize the I/O pin functions and store the initial register control bytes into the EEPROM (see Appendix E). You may use these image registers to determine the status of the port but you must update these registers in your applications. The port0-2 are initialized as shown below:

```
void ve_init(void){
    pokeb(0xfff0,0x02,0x80);   /* Set PMC0 P07=CLK   */
    pokeb(0xfff0,0x01,0xd7);   /* Set PM0 for input, P05=LED P03=HWD output */
    pokeb(0xfff0,0x0a,0x80);   /* Set PMC1 P17 for READY */
    pokeb(0xfff0,0x09,0xaf);   /* Set PM1 for input, P14=RTS1,P16=RTS0 OUTPUT */
    pokeb(0xfff0,0x12,0x00);   /* Set P20-P27 for port mode */
```

td_init

This function is called immediately after ***ve_init***.

```
void td_init(char mode);
//
//   where m= hardware configuration
//
//   m=0:                    O15-O20 outputs, (P05,P14,P15,P16,P20,P21 output low)
//            RTS0=P16, RTS1=P14 !
//
//   m=1:                    O15-O20 inputs, (P05,P14,P15,P16,P20,P21 input)
//            You will lose RTS0=P16, RTS1=P14
//            Do not install U19, and
//            connect U19 socket pin 1-16,2-15,3-14,4-13,5-12,6-11 with 1K resistors
//            T2 pin 2-7, O15-O20 6 inputs(P05,P14,P15,P16,P20,P21 inputs)
//            Use td_di(char ch) to read inputs.
//                    where ch=20 to 25 for T2 pin 2-7 O15-O20 inputs
```

4.2.2 External Interrupt Initialization

There are up to five external interrupt sources on the TinyDrive, consisting of four maskable interrupt pins (**INTP2-INTP0, INT**) and one non-maskable interrupt (**NMI**). There are also additional internal interrupt sources not connected to the external pins, consisting of two timers, a time base counter, two DMA channels, both asynchronous serial ports, and the **NMI** from the watchdog timer. For a detailed discussion involving the interrupts, the user should refer to chapter 4 of the NEC V25 CPU User's Manual.

TERN provides functions to enable/disable all of the 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 user should run a finish interrupt routine. This can be done using the ***fint()*** function.

void intp*x*_init

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

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

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 nmi_init(void);
void intp0_init( unsigned char i, void interrupt far(* int0_isr)() );
void intp1_init( unsigned char i, void interrupt far(* int1_isr)() );
void intp2_init( unsigned char i, void interrupt far(* int2_isr)() );
void timer0_init(unsigned char i, void interrupt far(* timer0_isr)());
void timer1_init(unsigned char i, void interrupt far(* timer1_isr)());
void timer2_init(unsigned char i, void interrupt far(* timer2_isr)());
void time_base_init(char i, void interrupt far(*time_base_isr)());
```

4.2.3 I/O Initialization

There are two ports of 16 I/O pins available on the TinyDrive. 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 **ve_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 of the I/O ports, please refer to chapter 7 of the NEC V25 User's Manual.

Please see the sample program **portx.c** in **`tern\v25\samples\ve`**. 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 functions **port_wr** and **port_rd** can be quite slow when accessing the Port I/O pins. The maximum efficiency you can get from the Port I/O pins occurs if you modify the Port registers directly with an **outport** instruction instead of using **port_wr**/**port_rd**.

See the Hardware chapter for Port register addresses.

void port_init
Arguments: char p, unsigned char pmc, unsigned char pm
Return value: none

p refers to Port 0, Port 1 or Port2.

pmc refers to the 8-bit *port mode control* register value for port ***p***.

- A '0' bit sets the pin to I/O mode
- A '1' bit sets the pin to CONTROL mode

pm refers to the 8-bit *port mode* register value for port ***p***. This register is valid for pins only I/O mode pins.

- A '0' bit sets the pin to output
- A '1' bit sets the pin to input

char port_rd
Arguments: char p
Return value: byte indicating Port I/O status for port p.

Each bit of the returned 8-bit value indicates the current I/O value for the Port I/O pins in port ***p***.

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

Writes an 8-bit value to port ***p***. Only changes status of I/O mode output pins.

Example 4.1 Set port 0 as I/O, bits 0 – 3 as input, 4 – 7 as output.

port_init(0, 0x00, 0xf0);
p = Port 0,
pmc = 0 (I/O),
pm 0-3 = 1, pm 4-7 = 0.

Example 4.2 Set pins 20 and 23 as DMA Request. All other port 2 pins as output.

port_init(2, 0x09, 0x00);
p = Port 2,
pmc = bit 0 and 3 = 1 (Control), all others = 0 (I/O)
pm = all 0s. Since pins 20 and 23 are control functions, the ***pm*** field is not relevant.

In most cases it is only necessary to change the value of one or two pins in the port data register Since the port data register is a read/write register, it is possible to mask the pins that do not need to change. In this case, the ***port_init*** function cannot be used. Instead, the port data register can be directly accessed using the ***poke*** and ***peek*** functions.

Example 4.3 Using bitwise OR to set a single bit high, set pin 23 high without modifying the other pins. Assume all port 2 is output and that all pins are low.

pokeb(0xfff0, 0x10, (unsigned char) (***peekb***(0xfff0, 0x10) | *0x08*));

Assuming that all of port 2 is outputting low, the ***peekb*** function will return a value of *0x00*. A bitwise 'OR' with the value *0x00* and the mask *0x08* equals *0x08*. Port 2 now outputs *0x08*.

Example 4.4 Using bitwise AND to reset a single bit low, set pin 23 low without modifying the other pins. Assume settings are the same after executing Example 4.3.

pokeb(0xfff0, 0x10, (unsigned char) (***peekb***(0xfff0, 0x10) **&** *0xF7*));

Assuming the settings from Example 4.3 are still present, the ***peekb*** function should return a value of *0x08*. A bitwise 'AND' with the value *0x08* and the mask *0xF7* equals *0x00*. Port is again set to *0x00* (all pins low).

While the port data registers are read/write registers, the port control registers ***pmc*** and ***pm*** are not. Modifying only certain pins in these registers requires the use of global variables to store the values of these registers. This means that any changes to the ***pmc*** or ***pm*** registers must be accounted for in the global variable. As in the previous example, the bitwise OR and AND expressions can be used to mask the register bits.

Example 4.5 Set port 2 pins 0 through 3 as output.

```
// The following global variable defines the pm2 register

unsigned char pm2;

/* assume pm2 has been correctly maintained. The following code will modify the pm2
        register such that bits 0 through 3 are low. Use AND to set bits low */

pokeb(0xfff0, 0x11, ( pm2 = ( pm2 & 0xF0 ) ) ); //pm2 must be set to a new register
        value
```

Example 4.6 Set port 2 pins 0 through 3 as input.

```
// The following global variable defines the pm2 register

unsigned char pm2;

/* assume pm2 has been correctly maintained. The following code will modify the pm2
        register such that bits 0 through 3 are high. Use OR to set bits high */

pokeb(0xfff0, 0x11, ( pm2 = ( pm2 | 0x0F ) ) ); //pm2 must be set to new register value
```

4.2.4 Port T

Port T is an 8-bit input port whose threshold voltage can be changed in 16 steps. Comparator operation is performed through this port. Each Port T input is compared with the selected threshold voltage (Vth). PTn

> Vth results in a value 1, PTn < Vth results in a value 0. All eight results from PT0 to PT7 are latched to the port T input latches.

The resulting 8-bit latch can be accessed by the function ***portt_rd***(void) which returns the 8-bit result. Vth can be changed by the function ***portt_wr***(char *vref*). The variable *vref* {0 .. 15} sets the reference voltage by the following equation: Reference = ***Vth * vref*/16**. ***vref*** = 0 sets Reference = Vth. Vth is connected to a 10 K pullup resistor network and Vth ≈ 3.57V. PT0 – PT6 are on J2. PT0 – PT2 are pulled up by 10k resistors.

void portt_wr(char vref)

where vref is a number to select VREF

vref = 0	VTHx 1
vref = 1	VTHx 1/16
vref = 2	VTHx 2/16
vref = 3	VTHx 3/16
vref = 4	VTHx 4/16
vref = 5	VTHx 5/16
vref = 6	VTHx 6/16
vref = 7	VTHx 7/16
vref = 8	VTHx 8/16
vref = 9	VTHx 9/16
vref = 10	VTHx 10/16
vref = 11	VTHx 11/16
vref = 12	VTHx 12/16
vref = 13	VTHx 13/16
vref = 14	VTHx 14/16
vref = 15	VTHx 15/16

char portt_rd(void)

returns an 8-bit character representing the comparator output if the voltage at PT0 < Vref, bit 0=0 else 1.

4.2.5 Timer Units

The two timers present on the TinyDrive can be used for a variety of applications. The timers run at a maximum of 1/6 of the processor clock rate, which determines the maximum resolution that can be obtained.

These timers are controlled and configured through a mode register that is specified using the software interfaces. The mode register is described in detail in chapter 9 of the NEC V25 User's Manual.

The timers can be used to time execution of your user-defined code by reading the timer values before and after execution of any piece of code. For a sample file demonstrating this application, see the sample file ***timer.c*** in the directory ***tern\v25\samples\ve***.

The specific behavior that you might want to implement is described in detail in chapter 9 of the NEC V25 User's Manual.

void timer0_init
void timer1_init
Arguments: unsigned char mode, unsigned int md0, unsigned int tm0
Return values: none

The argument **mode** is the value that you wish placed into the **TMC0/TMC1** mode registers for configuring the two timers.

The argument **md0** is the modulo timer count and **t0** is the timer count.

void timer0_interrupt
void timer1_interrupt
Arguments: unsigned char i, void interrupt far (* timer0_isr)()
Return values: none

The argument **i** enables the interrupt and (*timer0_isr)() or (*timer1_isr)() points to the interrupt service routine. The interrupt service routine is called whenever count 0 is reached, with other behavior possible depending on the value specified for the control register.

4.2.6 High-voltage drivers: td_hv

For using the high-voltage drivers, use **td_hv**.

```
void td_hv(char hv, char k);
//
//      high voltage drivers
//      where
//              hv = 1-20 to select O1 to O20 on T2
//              k = 0, Ox is off
//              k=1,   Ox is on, sinks 350 mA to low
//
```

4.2.7 Analog-to-Digital Conversion

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

For a sample file demonstrating the use of the ADC, please see **ce_ad12.c** in **`tern\v25\samples\ce`**.

int ce_ad12
Arguments: char c
Return values: int ad_value

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

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

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

```
ce_ad12(0); // Read from channel 0
chn_0_data = ce_ad12(0); // Start the next conversion, retrieve value.
```

4.2.8 Digital-to-Analog Conversion

LTC1446 chips may be installed in U18 and U19 sockets if high-voltage drivers are not needed. Each chip offers two channels, A and B, for digital-to-analog conversion. Details regarding hardware, such as pin-outs and performance specifications, can be found in the Hardware chapter.

Sample programs demonstrating the U18 DAC (**td_da12a.c**) and the U19 DAC (**td_da12.c**) are located in the directory **`tern\v25\samples\td`**.

void td_da12
Arguments: int dat1, int dat2
Return value: none

Argument **dat1** is the current value to drive to channel A of the chip, while argument **dat2** is the value to drive channel B of the chip.

These argument values should range from 0-4095, with units of millivolts. This makes it possible to drive a maximum of 4.906 volts to each channel.

4.2.9 Other library functions

On-board supervisor MAX691 or LTC691

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

void hitwd
Arguments: none
Return value: none

Resets the supervisor timer for another 1.6 seconds.

void led
Arguments: int ledd
Return value: none

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

Real-Time Clock

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

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 rollover in digits in the year 2000. One solution might be to store an offset value in non-volatile storage such as the EEPROM.

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

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

```
   unsigned char wk; Day of the week.
} TIM;
```

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

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

Void rtc_init
Arguments: char* t
Return value: none

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

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

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

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

Delay

In many applications it becomes useful to pause before executing any further code. There are functions provided to make this process easy. For applications that require precision timing, you should use the 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 ve_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 files **ser0.h** and **ser1.h** in the `tern\v25\include` directory.

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 NEC V25 CPU: SER0 and SER1. Both ports have baud rates based on the 8 MHz clock.

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 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 chapter 11 of the NEC V25 User's 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 the processor frequency.

The following table shows the function arguments that express each baud rate, to be used in TERN functions.

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	76,800
13	115,000
14	230,000
15	460,800
16	1 Meg

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 16 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 macro service operation. In terms of receiving, there is no software overhead or interrupt latency for user application programs even at the highest baud rate. Macro service 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.

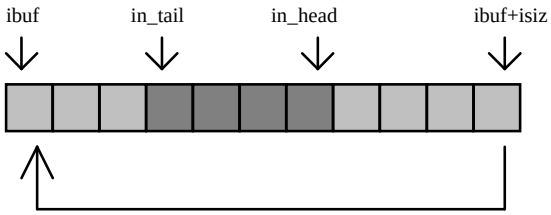


Figure 4.1 Circular ring input buffer

The input buffer (**ibuf**), buffer size (**isiz**), mode (**mode**), and baud rate (**baud**) are specified by the user with **`s1_init()`**.The mode is the setting value for the serial port control register. A value of ***0xC9*** will set the serial port in the following manner:

transmit enable, receive enable, no parity, 8 data bits, 1 stop bit

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

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

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

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

The sample program **`ser1_0.c`** demonstrates how a protocol translator works. It would receive an input HEX file from SER1 and translate every ':' character to '?'. The translated HEX file is then transmitted out of SER0. This sample program can be found in **`tern\v25\samples\ve`**.

Software Interface

Before you can use 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 **ve.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. **mode** is the serial control register value. **b** is the baud rate value shown in Table 4.1. The arguments **ibuf** and **isiz** specify the input-data buffer, and **obuf** and **osiz** specify the location and size of the transmit ring buffer.

If mode = 0xc9, 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.

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 appropriately 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 NEC V25 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 NEC V25 processor have many other features that might be useful for your application. If you are truly interested in having more control, please read Chapter 11 of the User's Manual for a detailed discussion of other features available to you.

4.4 Functions in SCC.OBJ

The functions found in this object file are prototyped in **scc.h** in the **`tern\v25\include`** directory.

The SCC is a component that is used to provide a third asynchronous port. It uses the 8 MHz system clock for driving serial communications. The divisors and function arguments for setting up the baud rate for this third port are different than for SER0 and SER1.

Function Argument	Baud Rate
1	110
2	150
3	300
4	600
5	1200
6	2400
7	4800
8	9600 (default)
9	19,200
10	31,250
11	62,500

Function Argument	Baud Rate
12	125,000
13	250,000

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

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

Initialization occurs in a manner otherwise similar to SER0 and SER1. A **COM** structure is once again used to hold state information for the serial port. The in-bound and out-bound buffers operate as before, and must be provided upon initialization.

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

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

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

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

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

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

When using RS232 in full-duplex mode, *transmit* and *receive* functions should both be enabled. Once this is done, you can transmit and receive data as needed. If you do need to do limited flow control, the MPO pin on the J1 header can be used for RTS. For a sample file showing RS232 full duplex communications, please see **ve_scc.c** in the directory **`tern\v25\samples\ve`**.

RS485 is slightly more complex to use than RS232. RS485 operation is half-duplex only, which means transmission does not occur concurrently with reception. The RS485 driver will echo back bytes sent to the SCC. As a result, assuming you are using the RS485 driver installed on another TERN peripheral board, you will need to disable *receive* while transmitting. While transmitting, you will also need to place the RS485 driver in transmission mode as well. This is done by using **scc_rts(1)**. This uses pin MP0 (multi-purpose output), found on the J1 header. While you are receiving data, the RS485 driver will need to be placed in receive mode using **scc_rts(0)**.

en485
Arguments: int i
Return value: none

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

scc_send_e/scc_rec_e
Arguments: none
Return value: none

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

scc_send_reset/scc_rec_reset
Arguments: none
Return value: none

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

Transmission and reception of data using the SCC is in most ways identical to SER0 and SER1. The functions used to transmit and receive data are similar. For details regarding these functions, please refer to the previous section.

putser_scc
See: putser*n*

putsers_scc
See: putsers*n*

getser_scc
See: getser*n*

getsers_scc
See: getsers*n*

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

scc_cts
See: s*n*_cts

scc_rts
See: s*n*_rts

Other SCC functions are similar to those for SER0 and SER1.

scc_close
See: s*n*_close

serhit_scc
`See: sn_hit`

clean_ser_scc
`See: clean_sn`

Occasionally, it might also be necessary to check the state of the SCC for information regarding errors that might have occurred. By calling **scc_err**, you can check for framing errors, parity errors (if parity is enabled), and overrun errors.

scc_err
Arguments: none
Return value: unsigned char val

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

4.5 Functions in VEEE.OBJ

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

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

Addresses **0x00** to **0x1f** on the EEPROM is reserved for system use, including configuration information about the controller itself, the 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.

4.6 Samples

Sample programs for the TinyDrive may be found in the following directories:

`C:\tern\v25\samples\ve`

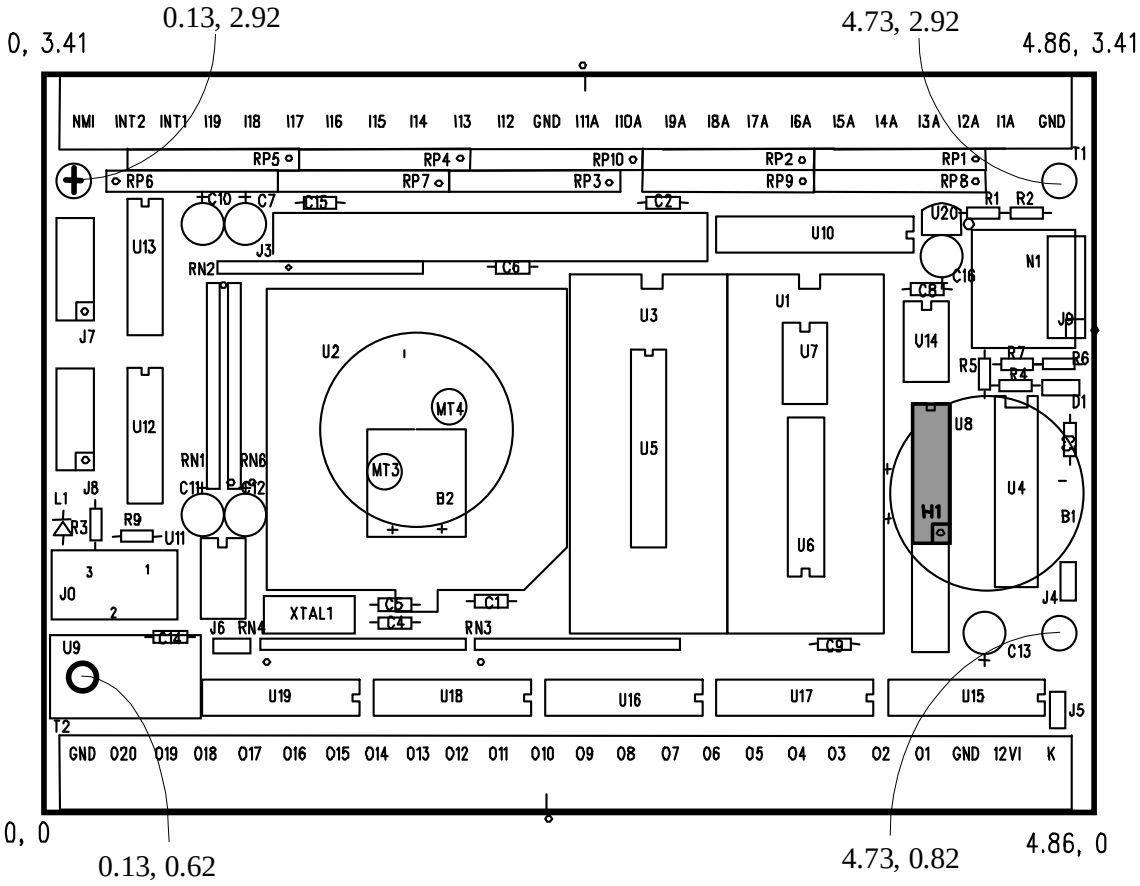
`C:\tern\v25\samples\td`

`C:\tern\v25\samples\ce`

Appendix A: TinyDrive version A Layout

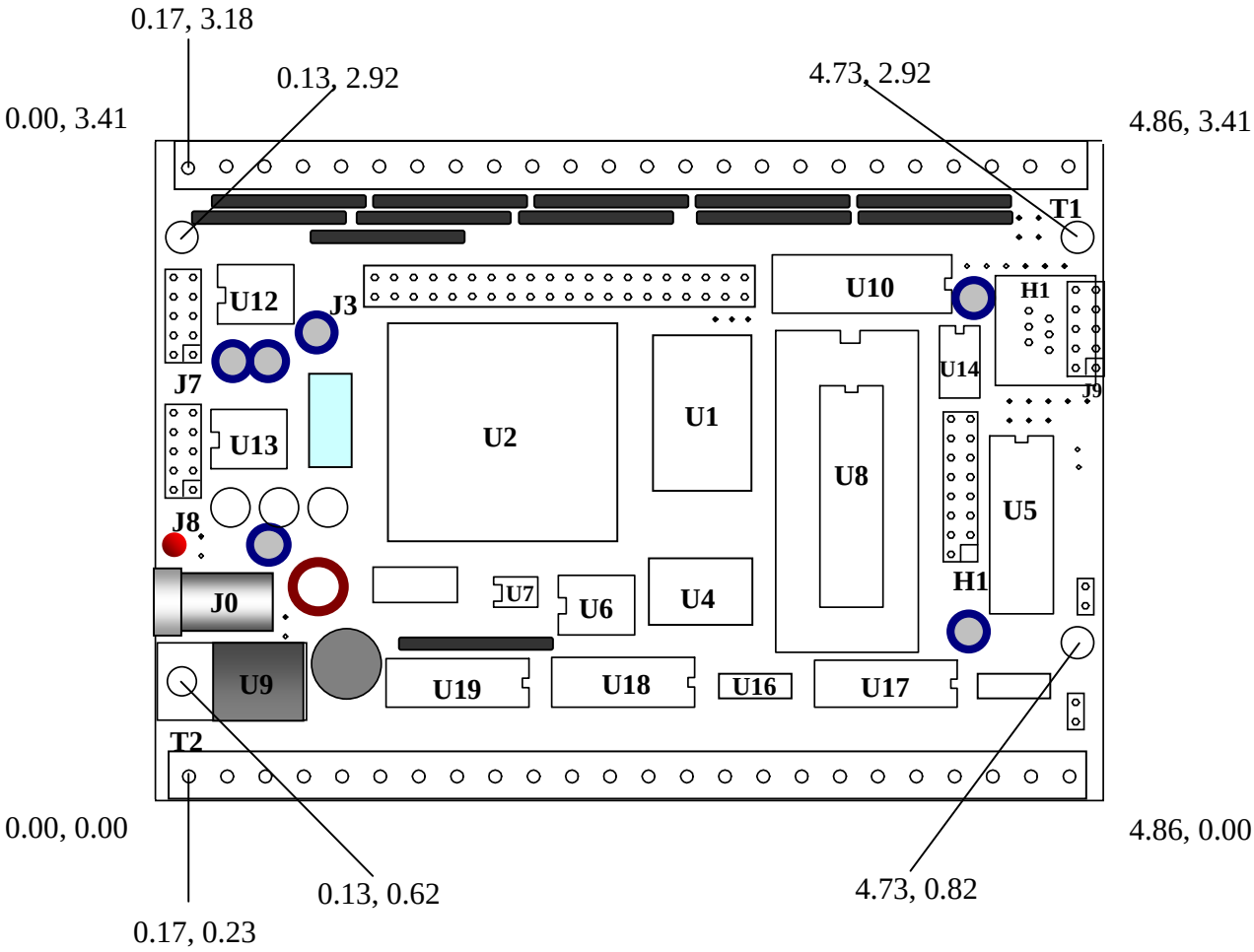
All dimensions are in inches.

NMI INT2 INT1	**i19 i18 i17**	**i16 i15 i14**	**i13 i12 GND**	**i11 i10 i9**	**i8 i7 i6**	**i5 i4 i3**	**i2 i1 GND**



GND o20 o19	**o18 o17 o16**	**o15 o14 o13**	**o12 o11 o10**	**o9 o8 o7**	**o6 o5 o4**	**o3 o2 o1**	**GND +12V K**

TinyDrive version B Layout



Appendix B: UART SCC2691

1. Pin Description

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

2. Register Addressing

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

Note:

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

3. Register Bit Formats

MR1 (Mode Register 1):

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

MR2 (Mode Register 2):

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

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

CSR (Clock Select Register):

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

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

CR (Command Register):

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

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

SR (Channel Status Register):

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

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

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

ACR (Auxiliary Control Register):

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

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

ISR (Interrupt Status Register):

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

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

IMR (Interrupt Mask Register):

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

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

CTUR (Counter/Timer Upper Register):

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

C/T [15]	C/T [14]	C/T [13]	C/T [12]	C/T [11]	C/T [10]	C/T [9]	C/T [8]

CTLR (Counter/Timer Lower Register):

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

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

Appendix C: RTC72421 / 72423

Function Table

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

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

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

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

4)

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

5) Test bit should be "0".

Appendix D: 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		

Appendix E: Software Glossary

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

---

void ve_init(void) ve.h

Initializes the V25 processor. The following is the source code for ***ve_init()***

```
pokeb(0xfff0,0x02,0x80);   /* Set PMC0 P07=CLK   */
pokeb(0xfff0,0x01,0xd7);   /* Set PM0 for input, P05=LED P03=HWD output */
pokeb(0xfff0,0x0a,0x80);   /* Set PMC1 P17 for READY */
pokeb(0xfff0,0x09,0xaf);   /* Set PM1 for input, P14=RTS1,P16=RTS0 OUTPUT */
pokeb(0xfff0,0x12,0x00);   /* Set P20-P27 for port mode */
pokeb(0xfff0,0x11,0xf7);// Set PM2 for input, P23=EN485 output
```

Reference: led.c

---

void ve_reset(void) ve.h

Resets the V25 processor.

---

void delay_ms(int m) ve.h

Approximate microsecond delay. Does not use timer.

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

Reference: led.c

---

void led(int i) ve.h

Toggles P05 used for led.

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

Reference: led.c

---

void delay0(unsigned int t) ve.h

Approximate loop delay. Does not use timer.

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

Reference:

void halt(void) ve.h

Enables HALT standby mode, which halts the system clock to reduce power consumption. Peripheral CPU devices (serial ports, timers, DMA ....) will not be effected. System clock restored by interrupt.

Reference: ve_halt.c

void hitwd(void) ve.h

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

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

void port_init(char p, char pmc, char pm) ve.h

Initializes I/O port mode control and port mode.

```
Var:  p = port 0, 1 or 2.
```
The *PMC* and *PM* variables define each pin of the 8-bit port selected.
For example: ***PM = 0xf0*** would set bits 0 – 3 as low and bits 4 – 7 as high.

```
pmc = CONTROL or I/O mode (0 = I/O; 1 = CONTROL).
pm = I/O pin as input or output(0 = output; 1 = input).
```

Reference: portx.c

void port_wr(char p, char dat) ve.h

Writes a bit to a Port I/O line. Port I/O line must be in an output mode

```
Var:  p – Port 0, 1, or 2
      dat – 8-bit data for port p
```

Reference: portx.c

unsigned int port_rd(char p) ve.h

Reads an 8-bit I/O port.

```
Var:  port – 0: Port 0
             1: Port 1
             2: Port 2
```

Reference: portx.c

void portt_wr(char vref) ve.h

Selects reference voltage for the comparator input port.

```
Var:  vref – {0 ... 15} defines reference as follows
            reference = Vth * vref/16.
            For vref – 0: reference = Vth.
      Vth : Threshold voltage ≈ 3.57V
```

Reference: portt.c

---

char portt_rd(void) ve.h

Reads from the 8-bit comparator input port. Returns 8-bit value.
bit = 0, PT_n < Vref
bit = 1, PT_n > Vref
where PT_n is the input voltage and Vref is the selected threshold voltage.

Reference: portt.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
```

---

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
```

---

int inport(int portid) dos.h

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

```
Var:  portid – I/O address
```

---

int inportb(int portid) dos.h

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

```
Var:  portid – I/O address
```

---

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

Writes to the serial EEPROM.

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

Reference: ve_ee.c

---

int ee_rd(int addr) veee.h

Reads from the serial EEPROM. Returns 8-bit data

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

Reference: ve_ee.c

---

void rtc_init(unsigned char * time) ve.h

Sets real time clock date, year and time.

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

Reference: rtc_init.c

---

int rtc_rd(TIM *r) ve.h

Reads from the real-time clock.

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

Reference: rtc.c

void timer0_init(unsigned char mode, int md0, int tm0); ve.h

void timer1_init(unsigned char mode, int md0, int tm0);

Timer 0, 1 initialization.

```
Var:  mode – TMC Timer mode. See ch. 9 for the TMC register
      tm – Count time for the count down timer.
      md – Count time for the modulo timer.
```

Reference: timer.c, timer0.c, timer1.c

---

void timer0_interrupt(char i, void interrupt far (*timer0_isr)()); ve.h
void timer1_interrupt (char i, void interrupt far (*timer1_isr)());

Initialization for timer interrupts.

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

Reference: timer0.c, timer1.c

---

void nmi_init(void interrupt far (* nmi_isr)()); ve.h
void intp0_init(unsigned char i, void interrupt far (*intp0_isr)());
void intp1_init(unsigned char i, void interrupt far (*intp1_isr)());
void intp2_init(unsigned char i, void interrupt far (*intp2_isr)());

Initialization for interrupts 0 through 2 and NMI (Non-Maskable Interrupt).

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

Reference: intpx.c

---

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

Serial port 0, 1 initialization.

```
Var:  m – mode for serial control register.
      b – baud rate.
      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 VE.H for COM
      structure.
```

b	Baud
1	110
2	150
3	300
4	600
5	1200

b	Baud
6	2400
7	4800
8	9600
9	19,200 (default)
10	38,400
11	57,600
12	76,800
13	115,000
14	230,000
15	460,800
16	1 Meg

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

---

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

Serial port 0, 1 initialization.

```
Var:  m1 = SCC691 MR1
      m2 = SCC691 MR2
      b – baud rate.
      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 VE.H for COM
      structure.
```

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

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

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

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

---

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

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

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

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

---

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

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

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

Reference: ser1_sin.c

---

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

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

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

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

---

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

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

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

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

---

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

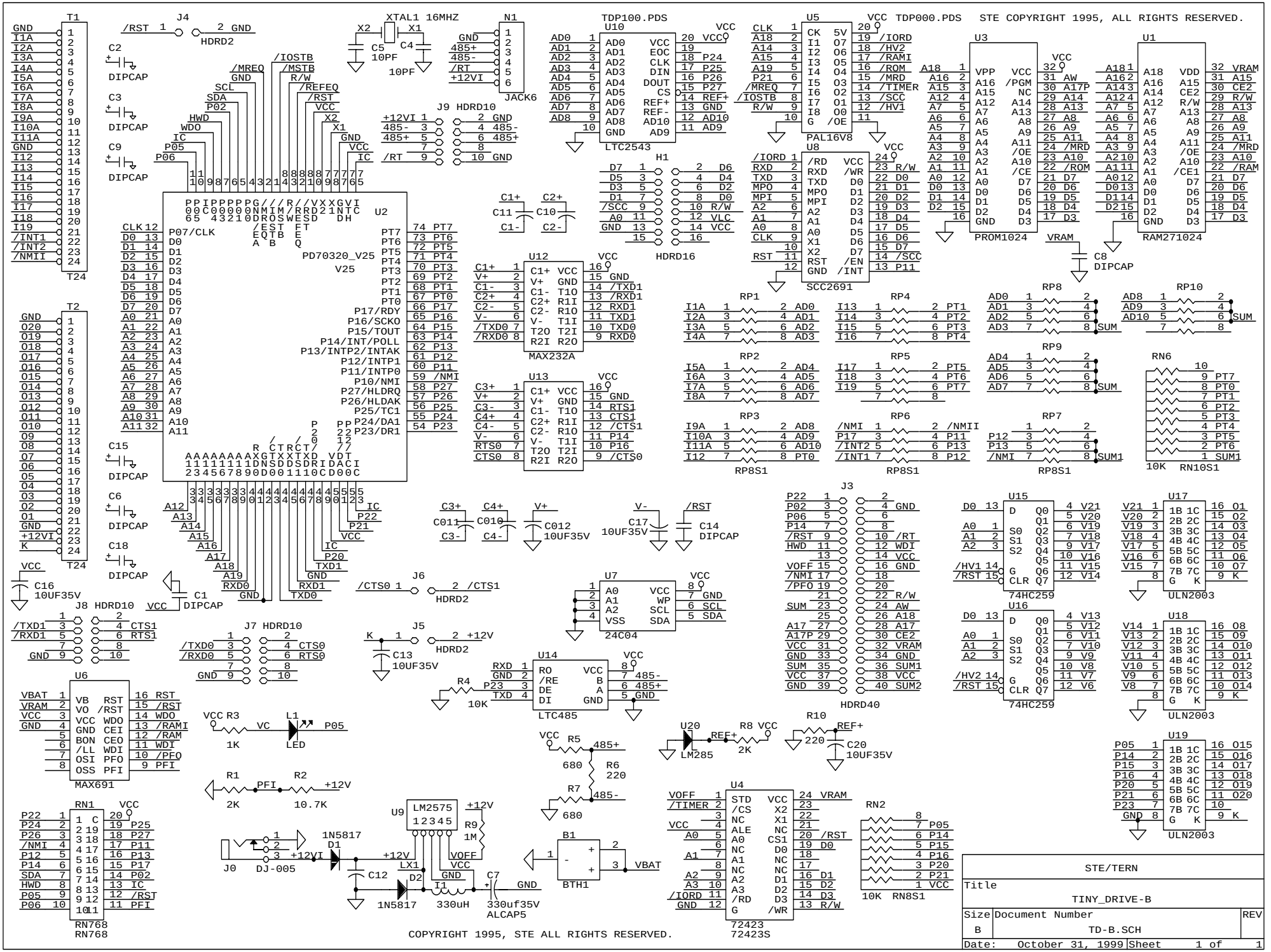
Retrieves a fixed-length character string from the input buffer. If the buffer contains less characters than the length requested, *str* will contain only the remaining characters from the buffer. Appends a '\0' character to the end of *str*. Returns the retrieved string length.

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

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

Appendix F: TinyDrive Part List

Item	Quantity	Reference	Part Description
1	1	B1	Coin Lithium Battery, PCB mounting, 3V
2	7	C1,C2,C3,C6,C8,C9,C15,C14	Ceramic Capacitor 0.01 µf 16V
3	2	C4,C5	Ceramic Capacitor ,15 PF 16V
4	6	C7,C10,C11,C12,C13,C16	Aluminum El. Capacitor, Radial, 10UF35V
5	1	D1	Diode, 1N5817
6	1	J0	DC power jack, ϕ 2mm, DJ-005
7	1	J3	20x2, 0.025" sqr header, Configuration jumpers
8	3	J4,J5,J6	2x1, 0.025" sqr header, Reset
9	2	J7,J8	5x2, 0.025" sqr header, RS232 ser0/1
10	1	L1	Red LED
11	1	N1	RJ11 phone jack, 6 contacts, JACK6
12	1	R1	1/4 w metal film resistor, 2K
13	1	R2	1/4 w metal film resistor, 10.7K
14	1	R3	1/4 w metal film resistor, 1K
15	1	R4	1/4 w metal film resistor, 10K
16	5	RN1,RN2,RN3,RN5,RN6	resistor network, 10 pin 9 resistors, 10K
17	2	R5,R7	resistor, 1/4w, 680Ω
18	2	R6,R8	resistor, 1/4w, 220Ω
19	1	R9	resistor, 1/4w, 300K
20	7	RP1,RP2,RP3,RP4,RP5,RP6,RP7	resistor pack, isolated, 8 pin, 4 resistors, 1K
21	2	T1,T2	Terminal block, 24 position each
22	1	U1	SRAM, 32K/128K/256K/512K, 100 ns, low power
23	1	U2	CPU, NEC, PD70320_V25, 84 pin PLCC
24	1	U3	EPROM, 32K/64K/256K, 120-150 ns
25	1	U4	Real-time clock, 18pin dip, EPSON RTC72421-B
26	1	U5	AMD PAL16V8Q
27	1	U6	Supervisor chip, 16 pin dip, MAX691/LTC691
28	1	U7	EEPROM, 128 -64K bits, 8 pin dip, 24C01
29	1	U8	UART SCC2691, Segnetic UART, 24 pin dip, 0.3w
30	1	U9	TO220, +5V requlator, LM7805
31	1	U10	12-bit ADC, TI LTC2543, 20pin dip, 11 channels
32	1	U11	Negative voltage convertor, ICL7662
33	1	U12	RS232 transmitter, 75C188/1488
34	1	U13	RS232 receiver, 75C189/1489
35	1	U14	RS485/422 transceiver, 75176/LTC485
36	2	U15,U16	8-bit latch, 74HC259
37	3	U17,U18,U19	Darlington transistor array, ULN2003
38	1	U20	reference zener, 2.5V/5V, LM285
39	1	U21	Electronic switch, TK112XX
40	1	XTAL1	low profile crystal, 16MHZ


T1
T24
T2
J4
HDRD2
XTAL1 16MHZ
C5 10PF
C4 10PF
N1
JACK6
J9 HDRD10
TDP100.PDS
U10
LTC2543
U5
PAL16V8
TDP000.PDS
STE COPYRIGHT 1995, ALL RIGHTS RESERVED.
U3
PROM1024
U1
RAM271024
U8
SCC2691
H1
HDRD16
U2
PD70320_V25
U12
MAX232A
U13
RP1
RP2
RP3
RP4
RP5
RP6
RP7
RP8
RP9
RP10
RP8S1
RN6
RN10S1
J3
HDRD40
U15
74HC259
U16
74HC259
U17
ULN2003
U18
ULN2003
U19
ULN2003
J8 HDRD10
J7 HDRD10
J6
HDRD2
J5
HDRD2
U7
24C04
U14
LTC485
U6
MAX691
U20
LM285
U4
72423
72423S
U9
LM2575
12345
B1
BTH1
J0
DJ-005
RN1
RN768
RN2
RN8S1
C1 DIPCAP
C2 DIPCAP
C3 DIPCAP
C6 DIPCAP
C8 DIPCAP
C9 DIPCAP
C15 DIPCAP
C16 10UF35V
C18 DIPCAP
C10 10UF35V
C11
C12 10UF35V
C13 10UF35V
C14 DIPCAP
C17 10UF35V
C20 10UF35V
R1 2K
R2 10.7K
R3 1K
R4 10K
R5 680
R6 220
R7 680
R8 2K
R9 1M
R10 220
L1 LED
D1 1N5817
D2 1N5817
I1 330uH
C7 330uf35V ALCAP5
COPYRIGHT 1995, STE ALL RIGHTS RESERVED.
STE/TERN
Title
TINY_DRIVE-B
Size B
Document Number TD-B.SCH
REV
Date: October 31, 1999
Sheet 1 of 1