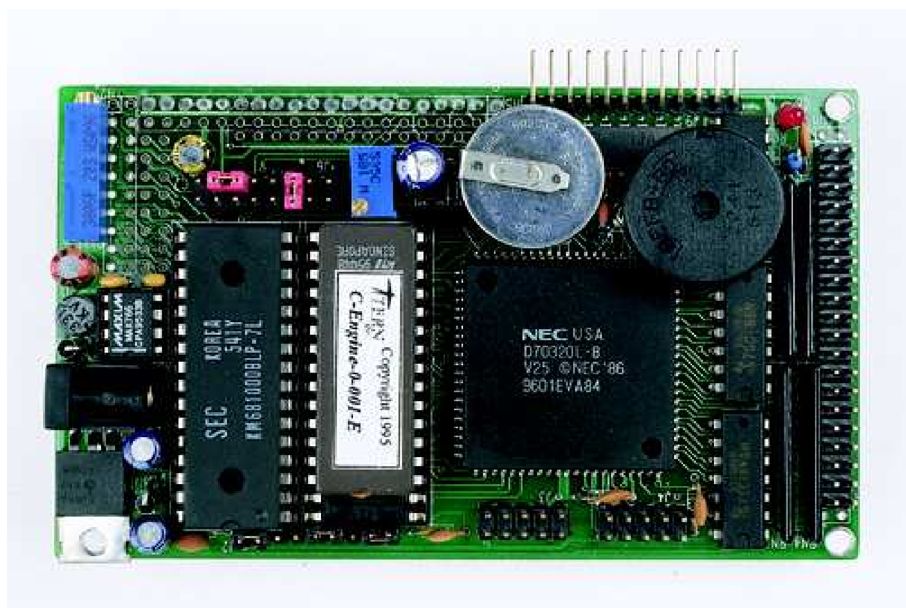
MiniDrive™

C/C++ Programmable LCD Controller
plus
40 I/O lines



Technical Manual

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

COPYRIGHT

MiniDrive™ is a trademark of TERN, Inc..
V25 is a Trademark of NEC Electronics Inc.
Turbo Debugger, Turbo C, and Borland C++ are trademarks of Borland International.
Microsoft, MS-DOS and Windows are trademarks of Microsoft Corporation.
IBM is a trademark of InSTEational Business Machines Corporation.
Paradigm LOCATE, DEBUG/RT-V25 and PDREM are trademarks of Paradigm Systems.

Version 2.00

October 29, 2010

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

© 1996-2010
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 MiniDrive™ is a low cost, C/C++ programmable, 16-bit industrial controller. It is designed for embedded applications that require compactness, low power consumption, and high reliability. The MiniDrive™ has a total of 40 I/O lines, including eight 4-bit ADC inputs, 7 high voltage outputs, 24 bi-directional I/Os. It supports various type character or graphic LCDs. It can be integrated into an OEM product, or as a stand-alone controller in an application system. By building your product with the MiniDrive™, you reduce the time from design to market introduction, cut development costs, minimize technical risks, and deliver a more reliable product.

Measuring 4.8 x 3.4 inches, the MiniDrive™ offers a complete C/C++ programmable standalone computer system with 16-bit CPU (NEC V25), upto 1MB memory, EPROM or Flash, battery-backed SRAM, EEPROM, real time clock, three serial channels, two 16-bit timers, 42 I/O lines, power failure detection, watchdog timer, LED, beeper, and interface to varies character type or graphics LCD modules. It is an ideal controller for a user interface design. A functional block diagram of the MiniDrive™ is shown in Fig. 1.1.

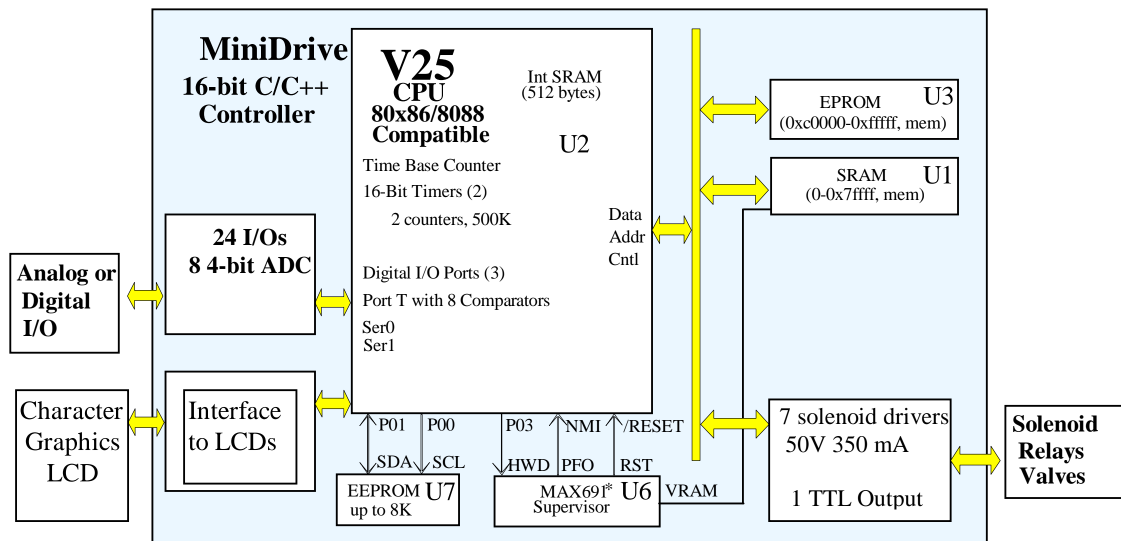
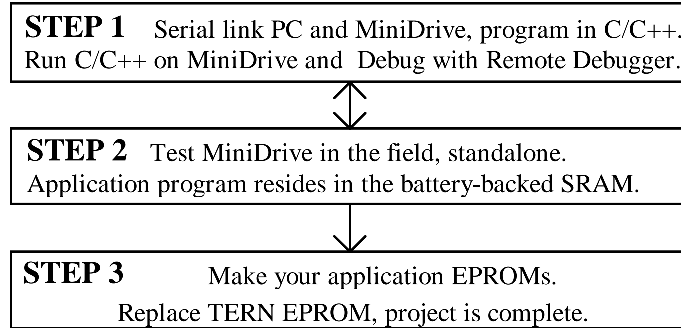


Fig. 1.1. Functional block diagram of the MiniDrive™.

How do you program the MiniDrive™ ? You can program the MiniDrive™ from your PC via serial link with a RS232 interface. You can use your favorite Borland or Microsoft C/C++ compilers. Your C/C++ program can be remotely debugged with remote debugger over the serial link at 115,000 baud rate. TERN provides I/O driver libraries, sample programs, remote DEBUG EPROM, batch files, and all the hardware necessary for you to quickly start developing your application software. After debugging a program, by setting a STEP2 jumper, you can test run the MiniDrive™ in the field standalone, away from the PC, with your application program in the battery-backed SRAM. When the field test is complete, an application EPROM may be burned to replace the DEBUG EPROM. The .HEX or .BIN file can be easily generated with the makefile provided. Development of application software consists of three easy steps. Setting a jumper on the MiniDrive™ J6 pin 1-2, (see Fig 3.5) will set the MiniDrive™ operating mode in STEP 2. With no jumper on J6 pin 1-2 while power on or reset, the MiniDrive™ will operate in the DEBUG mode(STEP1). A 128K/512K flash EEPROM can be used in the ROM socket, providing field programming or data storage.



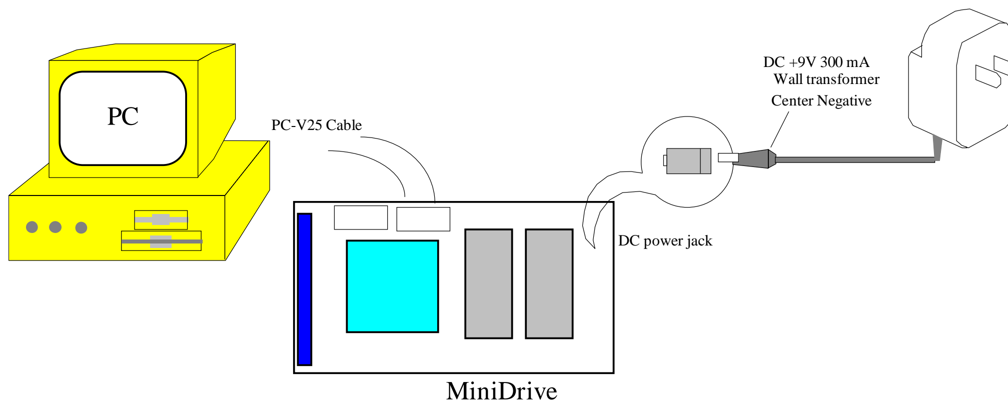


Fig. 1.2. Serial link a PC with the MiniDrive™, program on the PC, run C/C++ on the MiniDrive™

1.2 Features:

- 4.4 x 2.6 inches , 16-bit CPU (NEC V25), 16 Mhz crystal
- Power consumption: 120 mA normal
- Low power version: 75 mA full speed, 20 mA standby
- Upto 512K EPROM/Flash, 512K SRAM, 512 bytes EE
- Three 16-bit timers and two 16-bit counters(500KHz)
- 7 channels of high voltage/current drivers, 50V 350 mA per driver.
- 24 bi-directional I/O lines from V25
- 8 comparator inputs as 4-bit ADC and 3 interrupt inputs
- 2 high speed RS-232 serial ports(115,200 baud or higher)
- Watchdog timer, powerfail reset, lithium coin battery
- Interface to various graphic or character type LCDs.
- Beeper, keypads support.

1.3 Layout and Procedure

The physical layout of the MiniDrive™ is shown in Fig. 1.3.

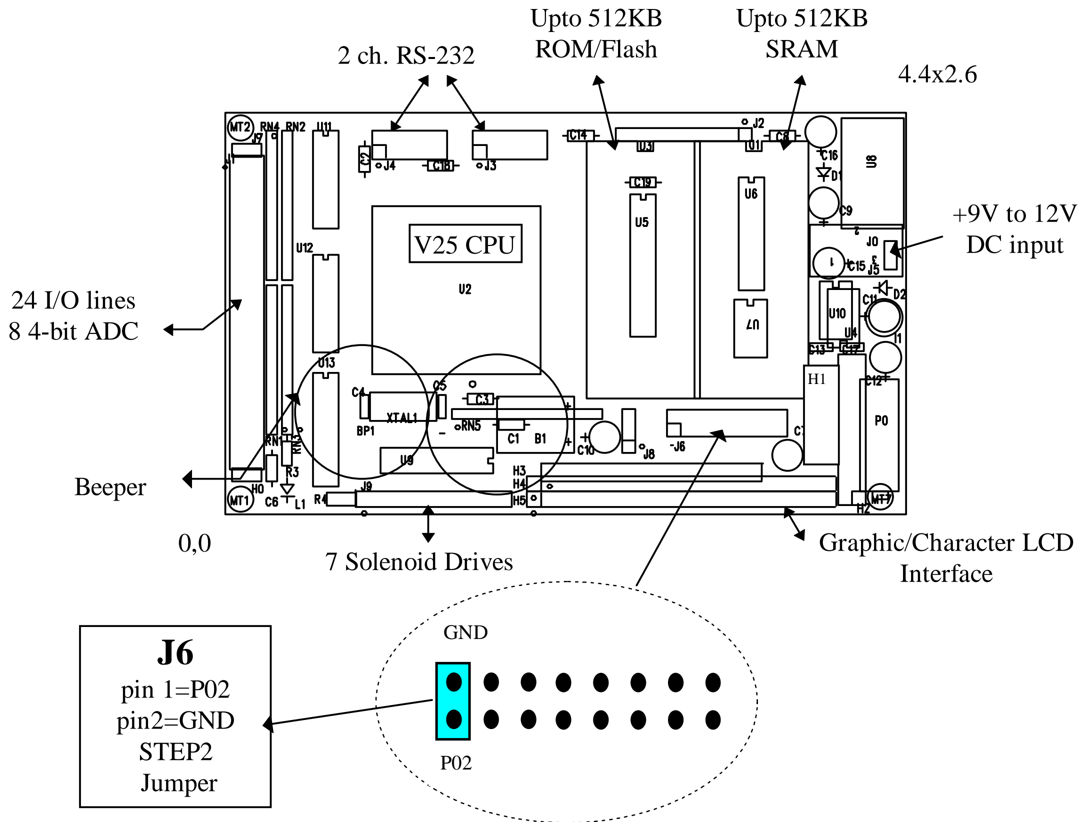


Fig. 1.3. Layout of the MiniDrive™ and the STEP2 jumper.

If a jumper is on J6 pin 1-2 (see Fig 1.3), while power on or reset, the MiniDrive™ will operate in STEP 2. If no jumper on the J6 pin 1-2 while reset or power on, it will operate in STEP 1. The status of J6 pin 2, which is V25 P02, is only checked at power on or at reset.
A simple functional flowchart of the DEBUG EPROM is shown in Fig. 1-4.

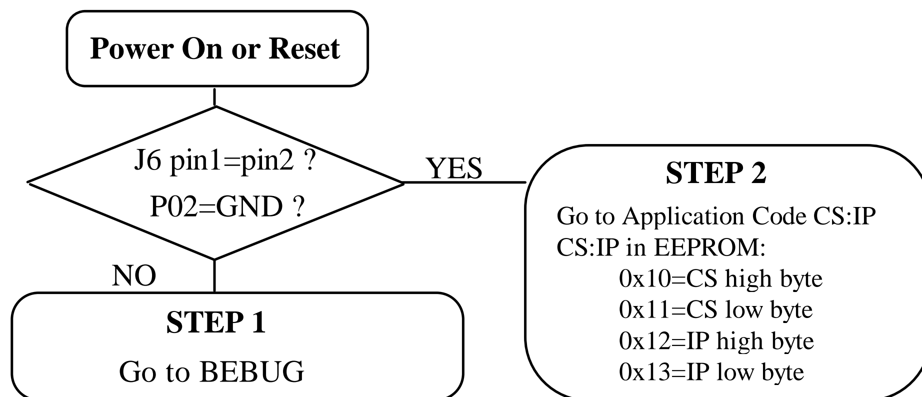


Fig. 1.4. Flowchart of the power-on procedure.

There are three steps in terms of the development of a C/C++ application program.

STEP 1: DEBUG.
You write your C/C++ application program in C/C++. You connect your controller to your PC via the PC-V25 serial link cable. Use the batch file (p.bat or e.bat) to compile, link, locate, download, and debug your C/C++ application program. See Chapter 2 and Chapter 3 for more details.
STEP 2: Standalone Field Test.
Set a jumper on J2 pin 2-4. While power on or reset, if J2 pin 2(P02), is low, the CPU will run the code in the battery backup SRAM. The start address must be pre-loaded in the on board EEPROM. You may use a sample program step2.c to write the CS:IP into EEPROM before the STEP 2 test.
STEP 3: Make your EPROM.
If you are happy with your STEP 2 test, you may go back to your PC to generate your application EPROM to replace the TERN EPROM. You need change the DEBUG=2 to DEBUG=0 in the makefile.
A sample program step2.c is in the TERN disk samples\ve directory. You may use the step2.c program to setup the STEP 2 address for your application code. If SRAM=0 in your makefile, the LOCATE will locate your code starting at 0400:0000. If SRAM=1 in your makefile, your code starts at 0800:0000. You must use DEBUG to download your code in the battery backup SRAM before you can run STEP 2.
A “step2.c” sample program is included in TERN disk. You may use it to initialize the EEPROM.

1.4 Power Consumption

With quarter-power PAL16V8Q in U5, 1488 in U12, 1489 in U11, the MiniDrive™ consumes approximate 180 mA in normal mode(16MHz, crystal), and 100 mA in standby mode. With zero-power PALCE16V8Z in U5, 75C188 in U12 and 75C189 in U11, as a low power version, the power consumption can be reduced to less than 75 mA in normal mode and 20 mA in standby mode.

1.5 Minimum Requirements for System Development

Hardware:
A PC or PC compatible computer;
A MiniDrive™ with DEBUG EPROM (C-Engine-∗-xxx-E).
where ∗ =0, using SER0; ∗ = 1, using SER1 for debug, xxx is the version number;
E, indicating use Paradigm DEBUG/RT-EV or DEBUG/RT-V25.
A PC-V25 serial cable(Fig. 1.2); and a wall transformer (+9V 500 mA, center negative).
Software:
Borland C/C++ 5.0/4.5/4.0/3.1, or Turbo C/C++ 3.0 and TASM, or
Microsoft visual C/C++ 1.0/1.5/1.52, and MASM6.11
Paradigm LOCATE, or LOCATE-EV and Paradigm DEBUG/RT-V25, or DEBUG/RT-EV.
TERN C libraries.

A C/C++ Evaluation Kit(EV-C), and a C/C++ Development Kit(DV) is available from TERN. The EV-C kit is a limited functionality version of DV. Using the EV-C kit, user can program and debug the MiniDrive™ in STEP1 and STEP2, but not in STEP3. In order to generate a user EPROM and complete the project, user should use the Development Kit(DV).

Chapter 2
Installation

The installation procedure presented here assumes that you have already established a Borland C/C++, or Microsoft C/C++ environment on your PC.

2.1. Software Installation

1. Install Paradigm LOCATE in C:/LOCATE.
2. Install Paradigm DEBUG/RT-V25 or DEBUG/RT-EV in C:\PD.
3. Specify parameters for the remote debugger communication between PC and the MiniDrive™ select "Serial" from "Remote Type" option, select "COM1/2" from "Remote Link Port" option, select "115000 baud" from "Link Speed" option.
4. Copy the TERN disk to your hard disk in C:\STExxxx

2.2 Hardware Installation

1. Serial link the MiniDrive™ and your PC with the PC-V25 cable.

If your are using SER0 EPROM(MiniDrive-0-xxx-E), install the 5x2 IDC connector to J3 header. Note that the red side of the cable must point to the pin 1 of the MiniDrive™ J3 header. A small circle is drawn for indication in the Fig. 2.1. The DB9 connector of the PC-V25 cable is connected to the PC's COM1/2.

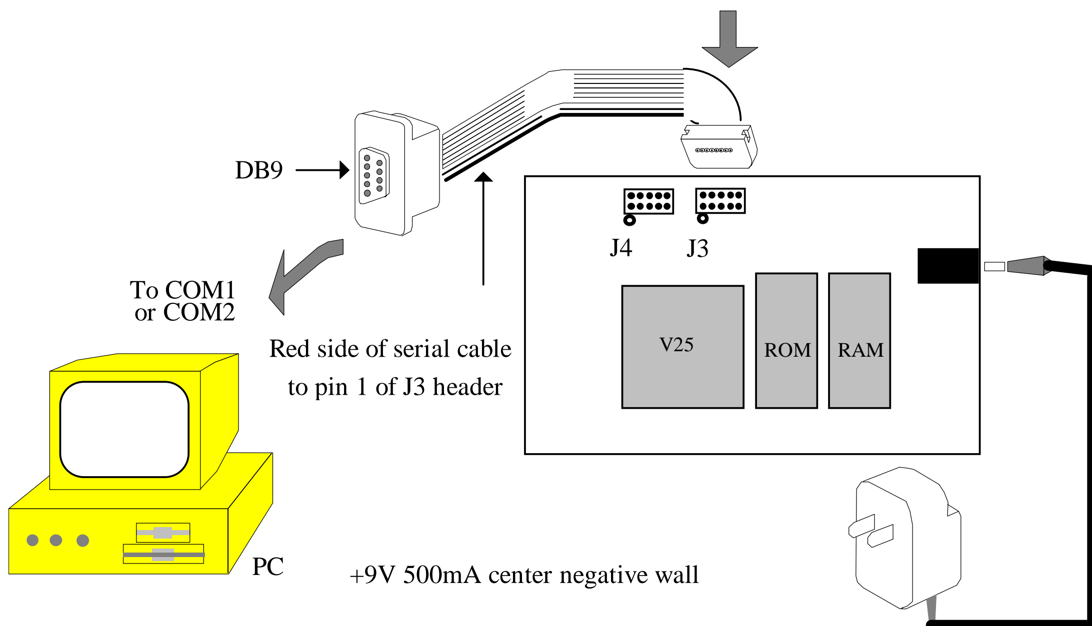


Fig. 2.1.Serial connection between the MiniDrive™ and the PC

2. Connect the wall transformer +9V DC plug to the MiniDrive™ DC power jack. LED should blink twice after power on or reset, indicating initialization success. If not, see Chapter 3.

Chapter 3. Tutorial

In this tutorial, we will run a sample program **led.c** to test both the software and hardware installation performed in chapter 2. We will illustrate the use of Paradigm DEBUG/RT-V25.

3.1 STEP1 Debug a Sample Program

A sample program, **led.c** is in the directory of TERN disk a:\samples\ve. It is also in your working directory c:\stebc31. We use BC31 for Borland C/C++ 3.1 as an example. A batch file **m.bat** is designed to test the C/C++ software environment. At DOS prompt c:\stebc31>

type **m led**

It will perform the operation defined in the makefile. It should pass the stage of compile, link, and locate.
A batch file **e.bat**(Evaluation Kit), or **p.bat**(Development Kit) is designed to test software and hardware installation. It will handle all file making and debugging activities. We use **p.bat** in this sample. Type **p led**. It compiles **led.c**, links and locates its output files, produces a **test.axe** file, and then down loads the **test.axe** to the MiniDrive™. If both software and hardware are successfully installed, the following Paradigm DEBUG/RT screen (Fig. 3.1) will show on the screen and "**ready**" will be shown at the upper right corner of the screen. This means that the executable file of **led.c** has been **downloaded** into the MiniDrive™

```
 ≡  File  Edit  View  Run  Breakpoint  Data  Options  Window  Help    Ready
[■]= Module: LED FILE:  C:\stebc31\LED.C ==================== 1 = [↑][↓]

/*************************************************************/
//                      led.c
//                      Test LED
//                      TURN LED ON AND OFF
//              Copyright (C) 1995 STE. All Rights reserved
/*************************************************************/
#include       <dos.h>
#include       "ve.h"
char ledd;
unsigned int i, k;
► void main(void)
{

Watches ---------------------------------------------------------- 2


F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Fig. 3.1. "Ready" on the screen indicating a successful installation

If a message, "Remote link time-out", is displayed, you may check:
1) Is the MiniDrive™ DC power on ?
2) Does the red led on the MiniDrive™ flash twice after the power on ?
3) Is the cable connecting to J3 of MiniDrive™ with a DEBUG EPROM(C-Engine-0-xxx-?) installed ?
4) Is the red line on the cable pointing to the pin 1 of the J3 header ?
5) Is the DB9 connector connecting the correct PC serial port COM1/2 ?
6) Does your PC have other devices occupying the same COM port you specified for the MiniDrive™ ?
7) Did you specify correct parameters (115,000 Baud, Remote, COM1/COM2) in DEBUG installation ?
8) Try Paradigm BBS at 607-786-0705. An utility RTTEST.ZIP may help.
If all efforts fail, call TERN tech-support at 916-758-0180.

The number circled in Fig. 3.2 is the line number of program led.c in the program window, which is the #1 window. Use **F8** to single-step, and the line number will increase accordingly. You may "step" over the program. **ALT-2** will activate the Watch window, which is window #2 as shown on the screen. Typing "i" in the Watch window allows you to watch the change of the variable "i" during the execution of the

debugger (Fig. 3.2). **F9-Run** runs the **led.c** program, and you can see the LED on MiniDrive™ blinking (Fig. 2.4) continuously and "Ready" is replaced by "Running" (Fig. 3.3). To stop running the program, use **Ctrl-Break**. Move cursor to the line you want, and set a breakpoint with **F2**, which will highlight that line. Toggling **F2** removes or sets a breakpoint. **F9** to run the program and the program should stop at the breakpoint

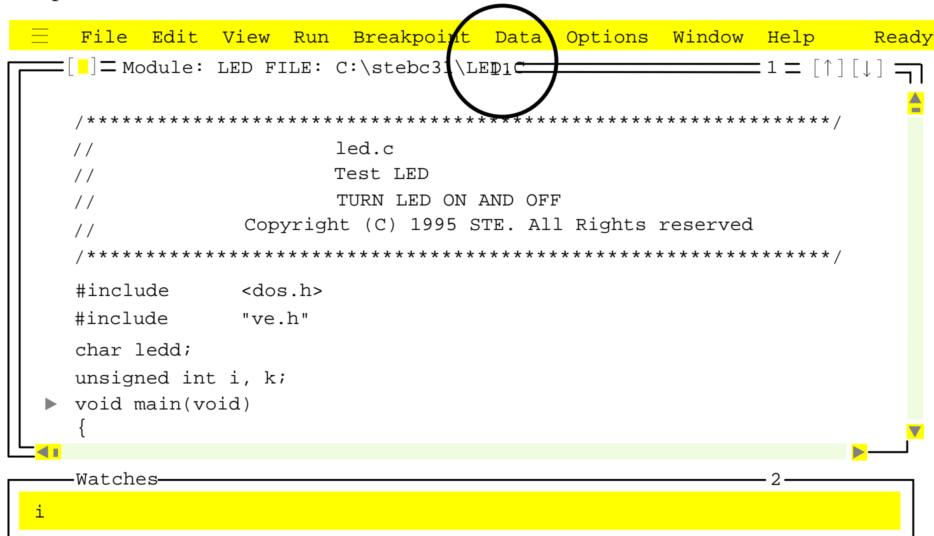


Fig. 3.2. Line number of the program and use of the Watch window

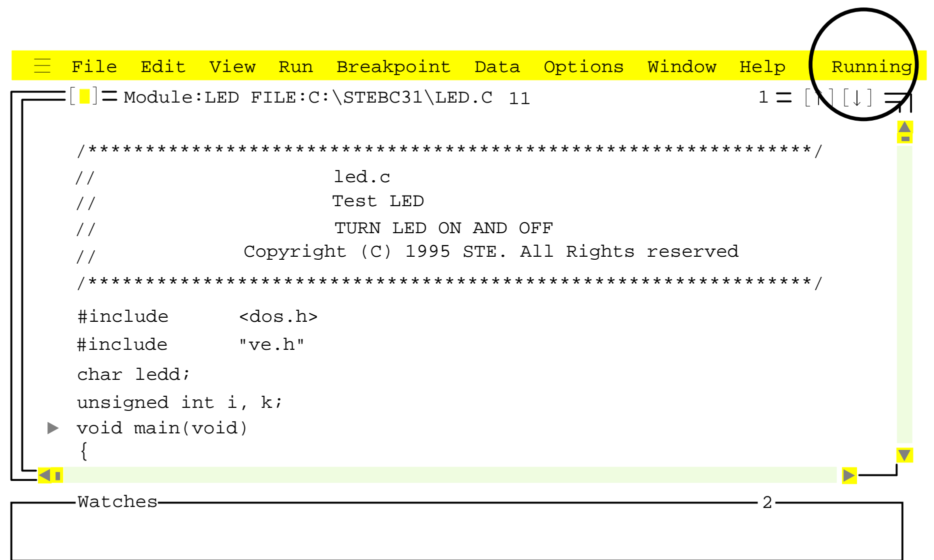


Fig. 3.3. Screen shows a program is "running" on the MiniDrive™

Select "view" from the main menu in Fig. 3.1, and pull down a sub menu, which allows more windows to be opened. You may open the CPU window to see the assembly equivalent of your C program and registers. With "F8" single step, you can examine details of the C program execution, including the interpreting of C by the assembly language, the changing contents in the registers, and the changing of the machine status. Use **Alt-X** to exit from the debugger. To see the LED blinking slower, modify the value of variable "i" in the **led.c** with the editor, for example, i = 30,000, and then repeat the above steps.

3.2 STEP2, Running the MiniDrive™ Standalone

If a battery is installed, program and data in the SRAM will not be lost while the main power is turned off. You can run the downloaded program on the MiniDrive™ away from the PC. You must pre-load the correct CS:IP value into EEPROM with step2.c program. See chapter 1 for the list of step2.c. If the SRAM=0 is in the makefile, the led.c code will be located and downloaded to the SRAM starts at 0400:000. If the SRAM=1 is in the makefile, the led.c code will be located and downloaded to the SRAM starts at 0800:000. You must run the step2.c program in the STEP 1 to load the CS:IP into the EEPROM. After you pre-load the CS:IP into EEPROM, you can download the sample program **led.c** in STEP 1. You may power off the controller and set a jumper on the J6 pin 1-2 (Fig. 1.3). After power on, the LED should blink again. The MiniDrive™ is running your program standalone, and you may disconnect the serial cable from your PC.
For a 128K SRAM, you must run the step2.c program to load 0800:000 as CS:IP into EEPROM. then you put a jumper on the pins 1 and 2 of the J6 on the MiniDrive™(Fig. 1.3). It will run the program starting at 0x0800:0000 after a power on or reset.

3.3 STEP3, Make an EPROM File

You can make an EPROM for your applications program to replace TERN's DEBUG EPROM. You need modify the makefile. Note: The C/C++ Evaluation Kit does not support generating EPROM files.

Modify the Makefile: You only need to change EPROM=0 in the makefile to make an EPROM file.
DEBUG = 0 # 0 - none, 2 - PDREMOTE

Generate an EPROM File: After modifying the **makefile** to set DEBUG=0, you can run the batch file **m.bat** for your application program **appl.c** by typing "**m appl**". This will produce an Intel extended hex file **appl.hex,** or a binary file **appl.bin,**

3.4 How to debug multiple interrupts

We have a sample program "int_demo.c" in TERN disk. We will discuss how to debug multiple interrupts on the MiniDrive™ with Paradigm DEBUG/RT-V25 dynamic mode. The sample program is on the TERN disk a:\samples\ve\int_demo.c.

1. Start with "**p int_demo**".

2. ***F10***, active menu.
   To set dynamic mode: main menu, view, Paradigm, Debug Controls, dynamic Enabled ...
   To save dynamic mode: main menu, view, Paradigm, Save setup

3. ***Alt 2***, active the watch window. Type the variables you want to watch
   intp0_count, intp1_count, intp2_count, nmi_count, timer0_count, or timbas_count

4. ***F10***, select the top line main menu.
   view, target, SFR

5. ***F10***, select the top line main menu.
   view, target, variables

6. ***F9***, run. You should see the timer0_count and timbas_count are changing in the watch window and the variable window. P0 should change every 10 second in the SFR window.

7. Any time you may activate the main program window and set a breakpoint with ***F2*** (toggle F2 will reset / set the breakpoint). If you set a breakpoint at * timer0_count=0; DEBUG will stop and show "ready" at that breakpoint. Toggle ***F2*** to remove the breakpoint and ***F9*** to run again. Since V25 has only one interrupt controller, it will not be released until a ***fint***(); instruction. The remote DEBUG is interrupt driven and

shares the same interrupt controller. If you set a breakpoint at ***fint***(); or before ***fint***(); in a interrupt service routing, the DEBUG will be lost and responsed with "remote time-out". You have to hardware reset the MiniDrive™ and start over again. A good practice is to modify your source code to temporally put the ***fint***(); at the very beginning of the interrupt service routing, as shown in the timer0 interrupt handler t0_isr. Then you can set a breakpoint in the t0_isr after the fint();.

8. You can force a INTP0(P11 J2 pin 8) or NMI(J10 pin2) by connecting the pins to ground. The intp0_count or nmi_count will change. Be very careful, study the pin locations, do not smoke your board.

9. ***F10***, select the top line main menu.
RUN, Halt
or Program reset, or Debugger reset
Then start over again.

10. If you want to modify the source code, you have to ***Ctrl-Break*** then ***Alt-x***. Exit from the DEBUG. Goto an editor and modify your C program.

Chapter 4. Hardware

4.1 V25 I/O Ports

V25 (µPD70320 NEC) CPU has 32 I/O lines which are basically organized as three bi-directional I/O ports(P0-2) and a comparator input port T. The 24 bi-directional I/O lines are multiplexed with different functions. One I/O line can be specified as an input, output, or a control line. There are three Special Function Registers (SFR) associated with each port: **Port Mode Control** Register (PMC0, PMC1, PMC2), **Port Mode** Register (PM0, PM1, PM2), and **Port Data** Register (P0, P1, P2). The SFRs are memory mapped. You can write or read these registers via: ***pokeb(0xfff0, 0x??, 0x!!);*** or ***peekb(0xfff0, 0x??);***

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

SFR addresses and Port operation tables are listed in the NEC V25 User's Manual.

For example, in order to use port 0 P05 as output, you need program port 0 in 3 steps:

1) program the PMC0 register and set PMC0 bit 5=0, which defines P05 as I/O function.

2) program PM0 register and set PM0 bit 5=0, which defines P05 as output.

3) Write a "1" to P0 data register bit 5, the P05 pin on the MiniDrive™ J2-5 should be high (5V).
   Write a "0" to P0 data register bit 5, the P05 pin on the MiniDrive™ J2-5 should be low (0V).

Some I/O lines are used by the MiniDrive™ system as listed below:

P00	I/O	EEPROM (U7 pin 6) clock SCL
P01	I/O	EEPROM (U7 pin 5) data SDA
P02	I/O	J6 pin 2. STEP2 jumper
P05*	I/O	J1 pin 30, on board LED control
P07	CLK	J1 pin 40, CLK, 8 Mhz system clock, U5.1.
P17	RDY	J1 pin 16, V25 ready signal, used for more wait states, U5.12.

P00, P01 are used by system EEPROM. While using the DEBUG EPROM, P02 is used to select STEP 1 (DEBUG mode) or STEP 2 (standalone mode) during the power on or reset, see Fig 1.3 for detail. P17 is assigned as RDY signal for inserting more wait states in order to interface with slow LCD modules. If you do not need LCD functions, you may assign P17 as a I/O function line and cutoff the pin on the U5 PAL pin 16. The P17=RDY is also routed to J1 pin 16. P05 is used for on board LED control, but P05 is also can be used as for application, if you do not need LED.

Due to SFR registers of PMC0-2, PM0-2 are write only, image registers are assigned to locations in the on board EEPROM at:

PM0	0x08	*ee_rd*(0x08); or *ee_wr*(0x08, pm0);
PMC0	0x09	*ee_rd*(0x09); or *ee_wr*(0x09, pmc0);
PM1	0x0a	*ee_rd*(0x0a); or *ee_wr*(0x0a, pm1);
PMC1	0x0b	*ee_rd*(0x0b); or *ee_wr*(0x0b, pmc1);
PM2	0x0c	*ee_rd*(0x0c); or *ee_wr*(0x0c, pm2);
PMC2	0x0d	*ee_rd*(0x0d); or *ee_wr*(0x0d, pmc2);

The *ee_rd*() and *ee_wr*() functions are very slow. The EEPROM is only modified by the ve_init();. Other functions may change the PMCx and PMx registers without modifying the EEPROM. If you need fast access the image registers, you may use SRAM variables instead.

After *ve_init*(void);, the initial register control bytes are written into EEPROM. You may use these image registers to determine the status of the port. You may also need to update these registers in your applications. The port0-2 are initialized by the *ve_init*(void) as listed 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 */
        pokeb(0xfff0,0x11,0xf7);   /* Set PM2 for input, P23=EN485 output */ }
```

The port **data** registers can be read and write. In order to modify only one bit, you need to read back the data byte from that data register first, then do OR/AND operation on that bit.

For example, you can manipulate P05 to low or high with these functions:

```
pokeb(0xfff0,0x00,(unsigned char) (peekb(0xfff0,0)&0xdf));   /* Set P05=low  */
pokeb(0xfff0,0x00,(unsigned char) (peekb(0xfff0,0)|0x20));   /* Set P05=high */
```

Name		Pin #	Pin #		Name
P20	----	1	2	----	P21
P22	----	3	4	----	P23
P24	----	5	6	----	P25
P26	----	7	8	----	P27
NMI/P10	----	9	10	----	P11
P12	----	11	12	----	P13
P14	----	13	14	----	P15
P16	----	15	16	----	P17
PT0	----	17	18	----	PT1
PT2	----	19	20	----	PT3
PT4	----	21	22	----	PT5
PT6	----	23	24	----	PT7
GND	----	25	26	----	P00
GND	----	27	28	----	P01
GND	----	29	30	----	P02
GND	----	31	32	----	P03
GND	----	33	34	----	P04
GND	----	35	36	----	P05
GND	----	37	38	----	P06
GND	----	39	40	----	CLK/P07

Table 4.1 J1, 20x2 pin I/O port

4.2 Jumpers and Headers

Name	Size	Function
J0	DJ-005	DC power jack, 9V-12V
J1	20x2	main I/O port
R4	2x1	VCC and TTL output signal HV8I
J2	9x1	Memory selection:
	SRAM Size:	pin 2-3 SRAM 256K-512K, pin 1-2 SRAM 32K-128K
	ROM Size:	pin 5-6 ROM size 32K-128K
		pin 4-5 ROM size 256K-512K
	EPROM/Flash read in U3	pin 7-8; Program Flash in U3 pin 8-9
J3	5x2	SER0 RS-232
J4	5x2	SER1 RS-232
J5	2x1	+12VI and GND
J6	8x2	Function Control: pin 1-2, STEP2 control; pin 3-4, reset; pin 5-6, switching power on/off ; pin 7-8 and pin 9-10, Graphics LCD function select pin 11=12, EEPROM write protect; pin 11-13, No write protect pin 15-16, watchdog enabled, software must exercise LCD2(I/O 0xA0)
J7	2x1	VCC and GND
J8	3x1	-15V, VLC, and GND. a pot for graphics LCD contrast adjustment
J9	10x1	Solenoid drivers
H0	2x1	Vcc and GND
H1	8x2	LCD interface, character 16x2
H2	10x2	LCD interface, 240x64 pixels
H3	14x1	LCD interface, character 20x4
H4	20x1	LCD interface, 320x240 pixels, Seiko G324E
H5	20x1	LCD interface, 240x128 pixels, HDM128GS24Y

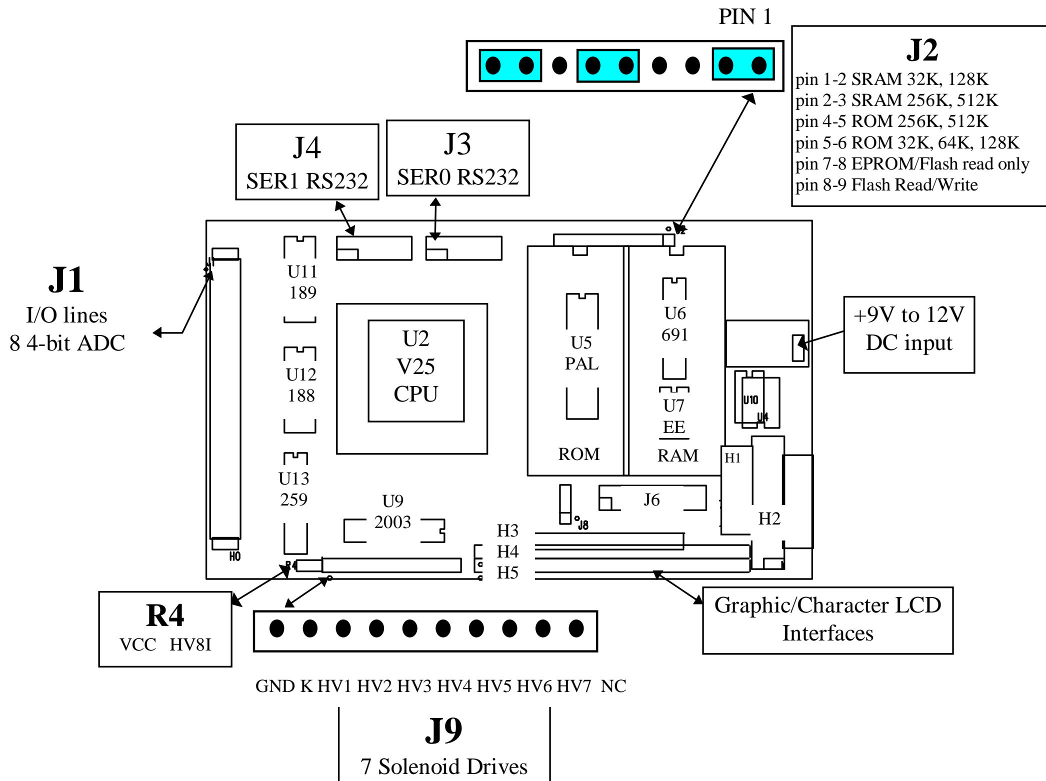


Fig. 4.1. Layout of Jumpers and Headers

All CPU-on-chip peripherals are memory mapped. They are controlled by a bank of 256-byte special function registers (SFRs). Most of the CPU-on-chip peripheral signals can be reached from J1.

4.3 Interrupts

V25 has a built-in high performance interrupt controller that can control multiple processing of 17 interrupt sources. Five of these interrupt sources, NMI, INTP0, INTP1, INTP2, and INT are external and accessible via memory mapped SFRs. The MAX691/LTC691 PFO (Power Failure Output) U6 pin 10 may be wire connected to NMI, J1.9. The user may connect the PFI (Power Failure Input) pin of MAX691 to an external voltage divider to monitor the power voltage level. The PFI pin has been pulled high to VCC with a 10K resistor on the MiniDrive™. When the external DC power drops and the voltage on the PFI (U6 pin 9 of MiniDrive™) is less than 1.3 V, the MAX691 will pull down PFO pin.
V25 has three different methods of responding to an interrupt: vector interrupt functions, register bank switching functions, and macro service functions. MiniDrive™ uses vector interrupt.

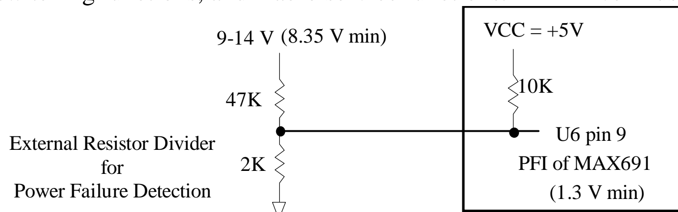


Fig. 4.2. Using PFI to monitor power voltage level

4.4 Comparator Input Port (PORTT)

Port T is an 8-bit comparator input port. The threshold voltage VTH can be fixed to VCC or connected to a variable voltage source. Software can set the reference voltage to one of 16 levels (1/16xVTH to 16/16xVTH). It provide users with an easy and inexpensive way to measure analog input signals.

4.5 External Event Counters/DMA

V25 has two DMA channels, DMA0 and DMA1. The DMA controllers can be used as 16-bit external event counters. After you set a 16-bit counter value into counter0 or counter1 with

counter0_init(unsigned int cnt0); or ***counter0_init***(unsigned int cnt0);

Every rising edge input signal on J1 pin 1(P20/DR0) will decrement the counter0. Every rising edge input signal on J1 pin 4(P23) will decrement the counter1.

An interrupt will occur, after counting to zero. You need an interrupt service route to serve the counter interrupt. For more detail, please see a sample program in TERN disk, a:\samples\ve\ve_count.c.

4.6 Clock and Timers

A built-in clock generator supplies various clocks to the CPU and peripheral hardware. The MiniDrive™ uses a 16 MHz crystal. Default system clock output after initialization is 8 MHz on CLK line (pin 4 of J1). One clock cycle is 125 ns. The normal bus cycle requires two clock cycle, which is 250 ns. With built-in wait state generation, up to 2 wait states can be inserted. Additional wait states can be inserted by using the RDY line. With the default initialization of 2 wait states, EPROMs of 120 ns to 150 ns can be used. More delays may be required to support slow I/O devices, such as LCD (Liquid Crystal Display).

The time base counter operates continuously since the MiniDrive™ 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 is generated at 4 different intervals, 128 us, 1.024 ms, 8.192 ms, and 131.072 ms, selectable by software.

Two 16-bit timer units, TM0 and TM1, can operate in interval timer mode or one-shot timer mode. The TOUT=P15 is available on pin 14 of J1 (Table 4.1).

4.7 Serial Channels

The MiniDrive™ has 2 serial channels: two internal UART, SER0, SER1. They can operate in full-duplex communication mode.

The internal serial channels can operate in asynchronous mode and I/O interface mode. In asynchronous mode, the start/stop bit transmit/receive method is employed so that bit synchronization and character synchronization are obtained by the start bit. In I/O interface mode, data is transferred in synchronization with the controlled serial clock. Each internal serial channel includes serial data input RxDn, serial data output TxDn, and Clear-to-Send signal input (CTSn). The CTS0 and CTS1 signals are connected to GND on the MiniDrive™. For SER0 and SER1, a built-in baud rate generator can be used to select standard baud rates from 110 to 1.25 M. One of these internal serial ports is used by the MiniDrive™ for programming with the PC. It uses 115,000 Baud rate for programming. It is possible to use both SER0 and SER1 in applications. The user can use SER0 to debug an application program for SER1, and then use SER1 to debug(EPROM available from TERN) application programs for SER0. The application programs can be combined and downloaded via either serial channel. Application program using both SER0 and SER1 can run at the same time, but not debug at the same time.

4.8 Halt and Stop Mode

The MiniDrive™ is an ideal core module for low power consumption applications, such as a battery operated instrument. V25 has two standby modes, which are set by halt(); and stop(); In the HALT mode, the CPU clock is stopped and program execution is halted, the registers are retained, and peripheral hardware continues to function. The total power consumption is approximately 20 mA. The HALT mode is released by interrupt input or reset input. In STOP mode, all clocks stop, but data in registers and RAM are retained. The total power consumption is less than 10 mA. The STOP mode only can be released by NMI input or reset input.

4.9 Supervisor chip

The MAX691/LTC691 (U6) is a supervisor chip. With it installed, the MiniDrive™ has functions of watchdog timer, battery backup, power-on-reset delay, power-supply monitoring, and power-failure warning. These will significantly improve the system reliability.

The watchdog timer will be activated by setting a jumper on J6 pin 15-16 of the MiniDrive™ (Fig. 4.2). The watchdog timer provides a means of verifying proper software execution. In user's application program, use C function inportb(0x00a0); to hit watchdaog. It must be accessed at least once every 1.6 seconds. If the J6 pin 15-16 jumper is on and the inportb(0x00a0); 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 MiniDrive™ being reset, the WDO

remains low until a transition occurs at the WDI pin of the MAX691. When shipping, NO jumper is on J6 pin 15-16, the watchdog timer is disabled.
You may solder a wire jumper from J1 pin 9=NMI to the U6 pin 10=PFO (Power Failure Output) pin of the MAX691. When the power failure is sensed by the PFI pin of the MAX691 (lower than 1.3 V), the PFO pulls NMI low, and an NMI interrupt occurs before the power-failure occurs. You may design a NMI service routine to take protect actions before the +5V drops and processor dies.
A battery-switch-over circuit compares VCC to VBAT (+3 V lithium battery positive pin), and connects whichever is higher to the VRAM (power for SRAM). The SRAM is protected with battery backed up. The lithium battery should last about 3-5 years in normal use. When the external power is on, the battery-switch-over circuit will select the VCC to connect to the VRAM.

4.10 EEPROM

A serial EEPROM of 128 bytes (24C01), 512 bytes (24C04), 2K bytes (24C16), or 8K bytes (24C65) can be installed in U7. The MiniDrive™ uses the P00 pin and the P01 pin to interface with the EEPROM by connecting them to the SCL (Serial Clock) and the SDA (Serial Data) signal on EEPROM. The EEPROM can be used to store important data, such as a node address, calibration coefficients, and configuration codes. It has typically 1,000,000 erase/write cycles. The data retention is more than 40 years.
J6 pin 11=WP, pin 12=GND, and pin 13=Vcc provide write protection for the EEPROM. When J6 pins 11-12 of the J7 are connected by putting on a jumper, write is prohibited in low pages. When J6 pins 11-13 are connected, no write protection is enforced. EEPROM can be read and written by simply calling functions ee_rd() and ee_wr().

4.11 High-voltage, High-current Drivers

ULN2003 has high voltage, high current Darlington transistor arrays, consisting of 7 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. The 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 degree C. The common substrate G is routed to J9 pin1 GND. All currents sinking in must be return from J9 pin 1 GND. A heavy gage(20) wire may be used to connect GND terminal to external power supply ground return. K is connecting to the protection diodes. K should be tied to highest voltage in the external load system. K is connected to J9 pin 2.

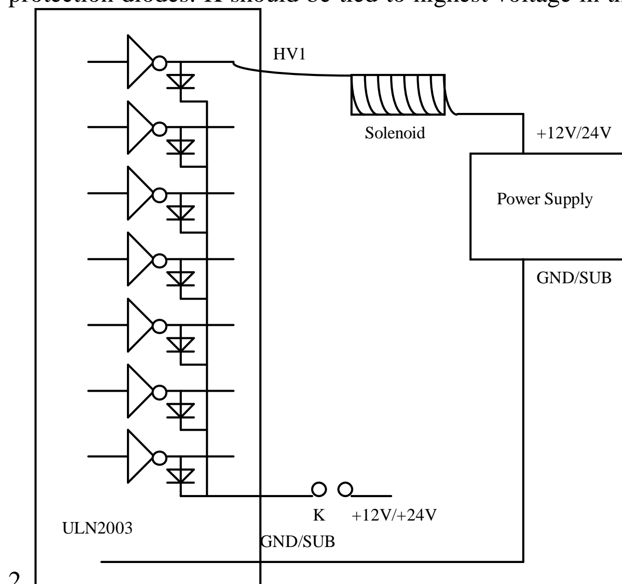


Fig. 4.3 Drive inductive load with high voltage/current drives.

ULN2003 is a sink driver not sourcing driver. A typical application wiring is shown in Fig. 4.3
The ULN2003 is driven by U13 74HC259. You can write the 74HC259 with ***outportb(0x80A0+hv, dat);*** where hv=0-6 for solenoid drivers HV1-7 on J9;

hv=7 for TTL output HV8I on R4
dat=0/1, off/on

4.12 I/O Space Mapped Devices

External I/O device 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.

I/O space	Usage
0x0080	LCD
0x00A0	LCD2
0x80A0	/259 for solenoid drivers

4.13 LCD support

MiniDrive™ can directly interface to many types of LCD modules. Many LCD modules have a build in controller support a microprocessor bus interface, consists of 8 data line, 1 address, 1 enable, read/write, Vcc, GND, and a contrast adjustment voltage. MiniDrive™ provides all these signals on 5 different pin out headers, H1-5. These headers are designed to match most Seiko(213-517-7770), Hantronix(408-252-1100) LCD modules.

MiniDrive™	LCD Manufacture	Size of LCD	Part Number	PAL in MiniDrive U5
H1	Seiko	16 ch. x 2 lines	M1632	mdp000.pds
H1	Seiko	40 ch. x 4 lines	M4024	mdp000.pds
H2	Toshiba/Hantr.	240 x 64 pixels Graphics	TLX711A/HDM64GS24Y	mdp010.pds
H3	Seiko	20 ch. x 4 lines	L2014	mdp000.pds
H3	Seiko	16 ch. x 1 lines	L1641	mdp000.pds
H3	Seiko	16 ch. x 2 lines	L1642	mdp000.pds
H3	Seiko	16 ch. x 1 lines	L1651	mdp000.pds
H3	Seiko	16 ch. x 2 lines	L1652	mdp000.pds
H3	Seiko	16 ch. x 4 lines	L1614	mdp000.pds
H3	Seiko	24 ch. x 1 lines	M24111	mdp000.pds
H4	Seiko.	320 x 240 pixels Graphics	G324E	mdp010.pds
H5	Hantronix.	240 x 128 pixels Graphics	HDM128GS24Y	mdp010.pds

You may directly solder attach the MiniDrive™ on the back side of the LCD module, without cable.
The PAL mdp000 provides active high LCD enable signal. The PAL mdp010 provides active low signal.
A MAX766 switching requlator can be imstalled in U10 in order to provide -15V to graphics LCDs.
You may also drive a keypads, and a beeper with I/O lines from the MiniDrive™

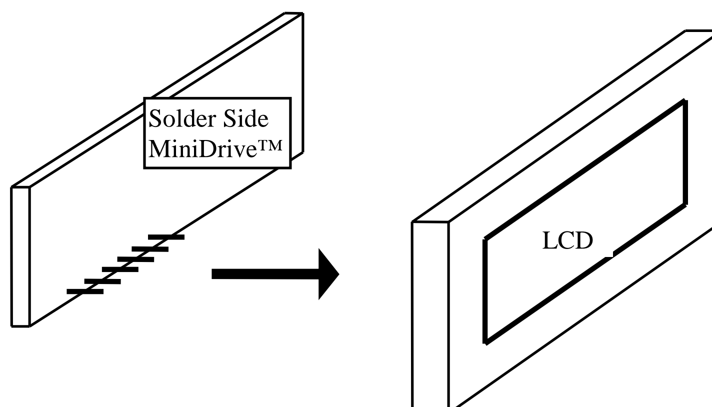


Fig. Install a MiniDrive™ on the back side of a LCD module.

Chapter 5 Software

Many powerful C language software development environments are available based on PC platform. The leading compilers are Borland or Microsoft C/C++. These compilers are excellent in their code generation, run-time libraries, and complete documentation. Many versions of the Borland and Microsoft C/C++ compilers, can be used to develop embedded software and runs on the C-Engine™. By not being tied to a particular editor or compiler, the C-Engine™ users are free to choose the tools they are most comfortable with. TERN provides C/C++ development kits for standalone embedded controllers. The Evaluation Kit and Development Kit integrate compiler, debugger, LOCATE, remote kernel, I/O libraries, and practical sample programs.

What are TERN's Development kits?

TERN has produced two different software kits that allow you, the user, to easily debug and download your code to your TERN controllers. Both of these kits contain versions of Paradigm Debugger and Locate, TERN C-Libraries, batch files, configuration files, makefiles, sample programs, technical manuals, schematics, wall transformers, RS232 cables, DEBUG EPROM for the controller, and lots of everything else that you need to start your project immediately. These two products are essential if you plan to create applications for your TERN controllers by programming with Borland/Microsoft C/C++ in 3 STEPS. STEP 1, debug. STEP 2, stand-alone with battery-backed SRAM. STEP 3, user EPROM.

What's in the Evaluation Kit?

The EV-C, or Evaluation Kit, is a good investment for the first-time buyer who isn't prepared to goto production or doesn't need the full capabilities of a Development Kit at a cost of only $299. This software kit includes Evaluation versions of Paradigm Debugger and Locate that allows the user to debug and download code into the controller. The user can also run the downloaded code stand-alone from the battery-backed SRAM away from the PC for prototyping purposes. This kit however is limited to downloading no more than 64K bytes of your .AXE file. The EV-C Kit also does not contain the feature of creating Intel extended HEX files that can be programmed into application EPROMs. In other words, the EV-C Kit supports STEP 1 and STEP 2, but not STEP 3.

What's different about the Development Kit?

The DV, or Development Kit, at a cost of $799 provides all the extra support that an engineer will need to complete his or her project. You can do everything the Evaluation Kit does with the Development Kit, as well as the ability to download .AXE files limited only by memory hardware, and the ability to burn EPROMs for OEM products.

What are the software requirements?

First, you will need a C/C++ compiler. TERN supports Microsoft Visual C/C++ 1.0, 1.5, 1.52, Borland Turbo C/C++ 3.0 for DOS, and Borland C/C++ 3.1, 4.0, 4.5. You will also need an Assembler. If you are using Microsoft C/C++, you will need MASM 6.11. If you are using a Borland C compiler, you will need to have TASM.

The software development for C-Engine involves three major steps:

STEP1.

Verify that the hardware related variables ***VE,*** and ***SRAM*** in the makefile is matching the hardware you are using.

```
VE     =   0   # 0-CE only
SRAM   =   1   # 0 - 32K, 1 - 128K/512K
EPROM  =   0   # 0 - 32K, 1 - 64K
DEBUG  =   2   # 0 - Make EPROM, 2 - PDREMOTE
```

If you are not using float point calculation, in order to reduce program size set

```
FLOAT  =   0   # 0 - none, 2 - emulator
```

If you have to use float point calculation, at least 128K SRAM must be installed and set

```
FLOAT  =   2   # 0 - none, 2 - emulator
```

At least 128K SRAM should be used in the STEP1. For a floating point application or a large C/C++ program debugging, 32K SRAM will not run. The LOCATE will complain about the overlap violation.
Use any text editor you are most comfortable with to write or edit your source program in Microsoft C or Borland C language. Use **m.bat** to test your software environment. Use **e.bat**(EV kit) or **p.bat**(DV kit) to invoke the makefile to process the source file, produce finish code and download it to the C-Engine™. via serial link. You may single step, set breakpoints, debug, or run the program. Repeat these steps (refer to Chapter 1.1) till you are happy with the performance of a functional program.
If you are going to STEP2, you must run the "step2.c" program during STEP1 to setup the CS:IP in the EEPROM, then download your application program with **p.bat** or **e.bat**.
STEP2.
You place a jumper on J2 pin 2-4 to set the C-Engine™ standalone mode(refer to Fig. 1.2a). Turn the DC power off and disconnect the C-Engine™ from the PC. Then power on (or reset) again with the mode setting jumper on, and the C-Engine™ will run the program which resides in the battery backed up SRAM. You may test your C-Engine™ based system in a remote field with the program resides in the battery backed-up SRAM. In normal condition, the battery should operate at least 5 years. With external DC input on, the internal battery is disconnected.
STEP3.
After the field test, it is the time to burn a EPROM. A romable .HEX file or Binary .BIN file can be generated by simply modifying the output parameter in the makefile(see Chapter 3.3) and run the **m.bat**.

```
DEBUG =     0     # 0 - Make EPROM, 2 - PDREMOTE
EPROM =     0     # 0 - 32K, 1 - 64K
```

You can program your EPROM with the output .HEX or .BIN file on a EPROM programmer.

By following three steps, you can develop an application program for the C-Engine™. However, the C-Engine™ is not a complete PC. We have to explain how to support the C programming in a non-PC environment. Paradigm LOCATE package provides different version of startup module for all Microsoft C/C++ or Borland C/C++ compilers. The modified startup module allows running the C-Engine™ without DOS control. All C/C++ standard library functions that call the BIOS or DOS are copied, modified, and renamed during the installation of LOCATE. We have to relocate the .EXE file to codes fits the C-Engine™ memory mapping . It is a complex process, especially in order to cover all major C/C++ compilers. But it is also very simple, because the user just simply type "**m**", "**e",** or "**p**". The whole process will be under control of the well designed makefile.

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 follow, otherwise it will result in a system crash, PC hang-up, and a lot of confusion. Always a concern is the debugging of interrupt handlers with the Remote Debugger. It is possible to debug an interrupt handler but it is not without hazard. Most problems occur in multi-interrupt driven situation. Since the remote kernel running on the C-Engine™ is interrupt driven, it demands interrupt services from the CPU. If an application program enables interrupt and occupying interrupt controller longer time than the remote debugger can live with, then the debugger will be timeout. Your PC may hang-up, and a power reset may be required to restart your PC. Always be aware, our system is remote kernel interrupt driven for debugging. In order to debug a interrupt service routing(ISR), a fint() function(V25 finish interrupt instruction) must be issued before the breakpoint, because it will release the interrupt controller to serve the debugger interrupt request.

VE.LIB
VE.LIB is a C library for basic V25 operations. It includes modules: VE.OBJ, SER0.OBJ, SER1.OBJ, SCC.OBJ, VEEE.OBJ, LCD.OBJ.
You need to include the VE.LIB in your applications and include the corresponding header files. Here is the list of the header files:

Include-file name	Description
VE.H	timers, time base counter, port t, internal RAM, Watchdog,

SER0.H	V25 internal serial port 0
SER1.H	V25 internal serial port 1
SCC.H	V25 external UART SCC2691
VEEE.H	C-Engine™ on-board EEPROM
LCD.H	LCD functions

Functions in the VE.OBJ
Initialization
void **ve_init**(void);
Initialize SFRs and set I/O pin functions After *ve_init*(void);, the initial register control bytes are written into EEPROM(see Appendix J). You may use these image registers to determine the status of the port. You may also need to update these registers in your applications. The port0-2 are initialized as 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 */
    pokeb(0xfff0,0x11,0xf7);   /* Set PM2 for input, P23=EN485 output */ }
```

Time Base Counter
void **time_base_init**(unsigned char interval);
Set interval for time base interrupt,
interval = 0 time base interrupts every 128 us,
interval = 1 time base interrupts every 1.024 ms,
interval = 2 time base interrupts every 8.192 ms,
interval = 3 time base interrupts every 131.072 ms.

void **time_base_interrupt** (unsigned char i, void interrupt far (*timbas_isr)());
Set time base interrupt service routine,
i = 0, time base interrupt disabled,
i = 1, time base interrupt enabled, and serviced by timbas_isr (user defined)

Timer 0
void **timer0_init**(unsigned char mode, unsigned int md0, unsigned int tm0);
Initialize timer 0 with mode, md0, and tm0.
where mode is written into TMC0(see V25 User's Manual, page 8-3).
Select start/stop, clock freq. one-shot or timer, TOUT.
Examples: mode=0xc0, 125x128=16µs per count
mode=0x80, 125x6=750 ns per count
md0 is written into MD0, md0 = 0 to 65535
tm0 is written into TM0, tm0 = 0 to 65535

void **timer0_interrupt**(unsigned char i, void interrupt far (*timer0_isr)());
Set interrupt service routine for timer 0,
i = 0, timer 0 interrupt disabled
i = 1, timer 0 interrupt enabled, and serviced by timer0_isr (user defined)

unsigned int **timer0_rd** (unsigned char i);
i = 0 read 16-bit data from MD0 i = 1 read 16-bit data from TM0

void **delay**(unsigned int t); Simple software loop, delay(t){ while(t){ t--; } }

void **delay_ms**(int m); where m=0-999 for delay approximate 0 to 999 ms

Timer 1
void **timer1_init**(unsigned char mode, unsigned int md1, unsigned int tm1);
Initialize timer 1 with mode, md1, and tm1,
where mode is written into TMC1(see V25 User's Manual, page 8-3),

select start/stop, clock freq. timer1 has no TOUT and one-shot mode
mode=0xc0, starts 16µs count down
mode=0x80, starts 750 ns count down
md1 is written into MD1, md1=0 to 65535
tm1 is written into TM1, tm1=0 to 65535

void **timer1_interrupt**(unsigned char i, void interrupt far (*timer1_isr)());
Set interrupt service routine for timer 1,
i = 0, timer 0 interrupt disabled,
i = 1, timer 0 interrupt enabled, and serviced by timer1_isr (user defined)

unsigned int **timer1_rd** (unsigned char i); i = 0 read 16-bit data from MD1, i = 1 read from TM1.

Timer2_interrupt

void **timer2_interrupt**(unsigned char i, void interrupt far (*tmf2_isr)());
This interrupt request is generated by underflow of TM1 register of timer 1. Only in one-shot mode, TMF2 bit of TMIC2 set, generates an interrupt,
i = 0, timer_flag2 interrupt disabled,
i = 1, timer_flag2 interrupt enabled and serviced by tmf2_isr (users defined).

I/O Ports

```
//                V25 port0,1,2 PMCx and PMC registers setting
//                See V25 User's Manual Table 7-5, Table 7-6 and Table 7-7
//                There are 3 registers associated with each port:
//                        PMC0, PM0, and P0 for PORT0
//                        PMC1, PM1, and P1 for PORT1
//                        PMC2, PM2, and P2 for PORT2
//        PMCx select CONTROL function or I/O function for every bit
//        PMx select INPUT or OUTPUT for every bit
//        Px is the DATA register for reading inputs or writing outputs
//
//                void port_init(char port, unsigned char pmc, unsigned char pm);
//    where port = 0, 1, or 2
//    pmc is a control byte to define each pin as CONTROL or I/O
//                if bitx=0, I/O;    bitx=1, CONTROL
//    pm is a control byte to define each pin as input or output
//                if bitx=0, output;  bitx = 1, input
//                note to check the pins used by the system
//
```

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

unsigned char **port_rd**(unsigned char p); Reads a byte from the data register of port p

void **port_wr**(unsigned char p, unsigned char dat); Write character "dat" to the data register of port p. if the pin is output, the dat bit will be output. if the pin is input, no change.

Comparator Port T

unsigned char **portt_rd**(void); Reads port T 8 comparator inputs
void **portt_wr**(unsigned char vref); Set reference voltage (0-15) for port T

External Interrupts

void **nmi_init**(void interrupt far (* nmi_isr)());
Set NMI interrupt service routine nmi_isr for NMI (user defined nmi_isr).

void **intp0_init**(unsigned char i, void interrupt far (* intp0_isr());

Set interrupt service routine intp0_isr for INPT0, i = 0 INPT0 interrupt disabled,
i = 1 INPT0 interrupt enabled and serviced by intp0_isr (user defined).

void **intp1_init**(unsigned char i,void interrupt far (* intp1_isr());
Set interrupt service routine intp1_isr for INPT1,
i = 0 INPT1 interrupt disabled,
i = 1 INPT1 interrupt enabled and serviced by intp1_isr (user defined).

void **intp2_init**(unsigned char i,void interrupt far (* intp2_isr());
Set interrupt service routine intp2_isr for INPT2,
i = 0 INPT2 interrupt disabled,
i = 1 INPT2 interrupt enabled and serviced by intp2_isr (user defined).

void **fint**(void); Finish interrupt, required by all interrupt service routine to release V25 interrupt controller, except NMI, INT, and software interrupts.

Watchdog Timer

void **hitwd**(void); Toggle J6 pin2 HWD once, hit watchdog, if J6 jumper is on.
void **wdo**(void); Read the Watchdog Out pin (P04 of V25).

LED

void **led** (int i); Turn LED on if i=1, turn LED off if I=0.

Counters / DMA

void **counter0_init**(unsigned int count0);
Load 16-bit count0 word into counter0. Every low to high edge at P20=DR0 will decreament the counter till zero. The counter must be reload by user. An interrupt can be generated at the end of count zero. Use V25 DMA0 16-bit address counter as external event counter. counter0_init() defines P20 as DMARQ0. It also uses eeprom 0x0d as PMC2 image register. The follow code are included.

```
pmc=0x01|ee_rd(0x0d);
ee_wr(0x0d,pmc);
pokeb(0xfff0,0x12,pmc); /* Set P20=DR0 */
```

See sample program samples\ve\ve_cnt0.c

unsigned int **counter0_rd**(void); returns 16-bit counts in the counter0.

void **counter0_interrupt**(unsigned char i, void interrupt far (* cnt0_isr)());
Setup an interrupt service routine cnt0_isr for counter0 counts to 0 interrupt,
i = 0 counter0 interrupt disabled,
i = 1 counter0 interrupt enabled and serviced by cnt0_isr (user defined).

void **counter1_init**(unsigned int count1);
Load a 16-bit word count1 into counter1. Every low to high edge at P23=DR1 will decreament the counter till zero. The counter must be reload by user. An interrupt can be generated at the end of count zero. Use V25 DMA1 16-bit address counter as external event counter. counter1_init() defines P23 as DMARQ1. It also uses eeprom 0x0d as PMC2 image register. The follow code are included.

```
pmc=0x08|ee_rd(0x0d);
ee_wr(0x0d,pmc);
pokeb(0xfff0,0x12,pmc); /* Set P23=DR1 */
```

See samples program samples\ve\ve_cnt1.c

unsigned int **counter1_rd**(void); returns 16-bit counts.

void **counter1_interrupt**(unsigned char i, void interrupt far (* cnt1_isr)());
Setup an interrupt service routine cnt1_isr for counter1 counts to zero interrupt,
i = 0 counter1 interrupt disabled,
i = 1 counter1 interrupt enabled and serviced by cnt0_isr (user defined).

CPU States

void **poll**(void); Poll and wait until the /POLL pin (P14) low.

void **halt**(void); Force CPU into HALT state until an INT, NMI, or RESET event occurs.
void **stop**(void); STOP, forcing CPU into STOP mode. Only wakeup by NMI or RESET

Functions in SER0.OBJ Module

void **s0_init**(char m, char b, char* ibuf, int isiz, char* obuf, int osiz, COM *c);

initialize SER0, where:
- m = mode,
- b = baud rate
- ibuf = pointer to input buffer
- isiz = size of input data buffer, use 512, 1024, 2048, 4096,... bytes.
- obuf = pointer to output buffer
- osiz = size of output data buffer
- c = serial data structure, COM

where b = baud rate expressed in number 0 to 11

baud = 1,	110 baud
baud = 2,	150 baud
baud = 3,	300 baud
baud = 4,	600 baud
baud = 5,	1200 baud
baud = 6,	2400 baud
baud = 7,	4800 baud
baud = 8,	9600 baud (default)
baud = 9,	19,200 baud
baud = 10,	38,400 baud
baud = 11,	57,600 baud
baud = 12,	76,800 baud
baud = 13,	115,200 baud
baud = 14,	230,400 baud
baud = 15,	460,800 baud
baud = 16,	1M baud

mode - 8 bits = SCM0

bit 7=TxE=0,	transmit disable
bit 7=TxE=1,	transmit enable
bit 6=RxE=0,	receive disable
bit 6=RxE=1,	recieve enable
bit 5,4 = 0,0	No parity
bit 5,4 = 0,1	Trans. Parity bit=0, Rec. ignore parity
bit 5,4 = 1,0	Odd parity
bit 5,4 = 1,1	Even parity
bit 3 = 0	7 data bits
bit 3 = 1	8 data bits
bit 2 = 0	1 stop bit
bit 2 = 1	2 stop bits
bit 1 = 0	
bit 0 = 1	Async mode

For example, mode=0xc9:
transmit enable, receive enable, no parity, 8 bits, 1 stop bit

unsigned char **getser0**(COM *c); get a character from input buffer of SER0

int **getsers0**(COM *c, int len, unsigned char *str);

get a string from SER0, where
- c = serial data structure, COM
- len = max length of input buffer of structure c
- str = pointer to a buffer for storing the string
- output = number of characters read
  = -1 if abort

int **putser0**(unsigned char outch, COM *c);
output a character to the SER0 MACRO service
return 1, if MACRO service valid and the character will be output.
return 0, if MACRO service is not available, NO operation.

int **putsers0**(unsigned char *str, COM *c);
output a string to SER0 MACRO service, appends a 0x0d=CR, 0x0a=LF, 0x00='\0'
where str = pointer to the string being output
c = serial data structure, COM
return = 1, if MACRO service valid, string will be output
return = 0, if MACRO service is not available, NO operation.

int **puts0**(unsigned char *str, unsigned char n, COM *c);
output str to SER0 MACRO service, where str = pointer to str being output
n = number of unsigned characters being output
c = serial data structure, COM
return = 1, if MACRO service valid, string will be output
return = 0, if MACRO service is not available, NO operation.

int **serhit0**(COM *c); check if SER0 has received any character. return 1, if any, return 0, if nothing

Functions in SER1.OBJ Module

void **s1_init**(char m, char b, char* ibuf, int isiz, char* obuf, int osiz, COM *c);
initialize SER1, where: m = mode, b = Baud rate
ibuf = pointer to input buffer
isiz = size of input data buffer
obuf = pointer to output buffer
osiz = size of output data buffer
c = serial data structure, COM
where b = baud rate expressed in number 0 to 11
baud = 1, 110 baud
baud = 2, 150 baud
baud = 3, 300 baud
baud = 4, 600 baud
baud = 5, 1200 baud
baud = 6, 2400 baud
baud = 7, 4800 baud
baud = 8, 9600 baud (default)
baud = 9, 19,200 baud
baud = 10, 38,400 baud
baud = 11, 57,600 baud
baud = 12, 76,800 baud
baud = 13, 115,200 baud
baud = 14, 230,400 baud
baud = 15, 460,800 baud
baud = 16, 1M baud
mode - 8 bits = SCM1
bit 7=TxE=0, transmit disable
bit 7=TxE=1, transmit enable
bit 6=RxE=0, receive disable
bit 6=RxE=1, recieve enable
bit 5,4 = 0,0 No parity
bit 5,4 = 0,1 Trans. Parity bit=0, Rec. ignore parity
bit 5,4 = 1,0 Odd parity
bit 5,4 = 1,1 Even parity
bit 3 = 0 7 data bits

bit 3 = 1 8 data bits
bit 2 = 0 1 stop bit
bit 2 = 1 2 stop bits
bit 1 = 0
bit 0 = 1 Async mode
For example, mode=0xc9:
transmit enable, receive enable, no parity, 8 bits, 1 stop bit

unsigned char **getser1**(COM *c); get a character from input buffer of SER1

int **getsers1**(COM *c, int len, unsigned char *str);
get a string from SER1, where: c = serial data structure, COM
len = max length of input buffer
str = pointer to a buffer for storing the string
return = number of characters read, or -1 if abort

int **putser1**(unsigned char outch, COM *c);
output a character to the SER1 MACRO service
return 1, if MACRO service valid and the character will be output.
return 0, if MACRO service is not available, NO operation.

int **putsers1**(unsigned char *str, COM *c);
output a string to SER1 MACRO service, appends a 0x0d=CR, 0x0a=LF, 0x00=‘\0’
where str = pointer to the string being output
c = serial data structure, COM
return = 1, if MACRO service valid, string will be output
return = 0, if MACRO service is not available, NO operation.

int **puts1**(unsigned char *str, unsigned char n, COM *c);
output str to SER1 MACRO service, where str = pointer to str being output
n = number of unsigned characters being output
c = serial data structure, COM
return = 1, if MACRO service valid, string will be output
return = 0, if MACRO service is not available, NO operation.

int **serhit1**(COM *c); check if SER1 has received character, return 1 if any, return 0, if nothing

Functions in VEEE.OBJ Module

int **ee_wr**(int addr,unsigned char dat); Write a byte "dat" to address "addr"
int **ee_rd**(int addr); Read from an address "addr"

Functions in LCD.OBJ Module

void **lcd_init**(void); Initialize the LCD1, for BB/SW/PD

void **lcdcmd**(int); Write the low byte of the int to the LCD1 command register
LCD command list:
0x01 Clear Screen
0x0c Cursor OFF (cursor invisible, blink off)
0x0e Cursor ON (solid cursor block)
0x0d Cursor BLINK (blinks continuously)
0x18 Shift Display Left
0x1c Shift Display Right
0x80 - 0x93 Position Cursor Line 0
0xC0 - 0xD3 Position Cursor Line 1

0x94 - 0xA7 Position Cursor Line 2
0xD4 - 0xE7 Position Cursor Line 3

void **lcddat**(int); Write the low byte of the int to LCD1 data register

void **lcd_wait**(void); Waiting for the LCD1 response

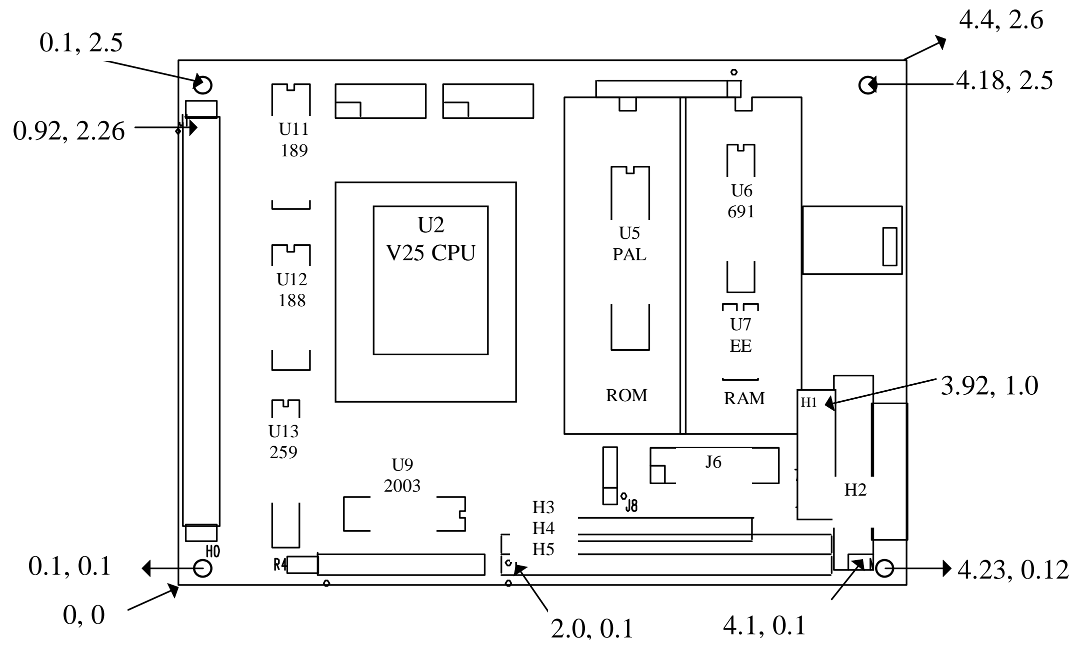
void **lputc**(unsigned char); Put a charater on the LCD1 at the cursor pointing position

void **lcd_clr_line**(unsigned char); Clear a line.
lcd_clr_line(0x80) clear line 0; lcd_clr_line(0xc0) clear line 1

printf() and int **_putch**(int); Microsoft / Borland C/C++ printf() function will use this putch() to print formated charaters to the LCD. At least 10K of codes will be linked in the program, if you use the **printf**(). In order to reduce your program size, you may consider use **lcdcmd**() and **lcddat**() directly for your LCD application.

// functions of LCD 2 are for the second LCD module, functions of LCD3-4 are for the V104™ LCD.
void **lcd2_init**(void);
void **lcd2cmd**(int);
void **lcd2dat**(int);
void **lcd2_wait**(void);
void **lputc2**(unsigned char);
void **lcd2_clr_line**(unsigned char);
void **lcd3_init**(void);
void **lcd3cmd**(int);
void **lcd3dat**(int);
void **lcd3_wait**(void);
void **lputc3**(unsigned char);
void **lcd3_clr_line**(unsigned char);
void **lcd4_init**(void);
void **lcd4cmd**(int);
void **lcd4dat**(int);
void **lcd4_wait**(void);
void **lputc4**(unsigned char);
void **lcd4_clr_line**(unsigned char);

A. Mounting Hole Locations



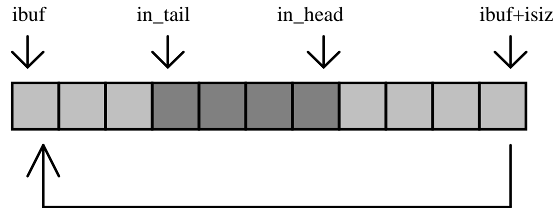
B. MiniDrive™ Part List

MiniDrive™ Part List Revised: April 7, 1996

Item	Quantity	Reference	Part
1	1	B1	BTH1
2	1	BP1	BEEP
3	8	C1,C2,C3,C6,C8,C14,C18,C19	0.01 UF, DIPCAP
4	2	C4,C5	10PF
5	7	C7,C9,C10,C11,C12,C15,C16	10uf35V Al. El. Cap.
6	2	C13,C17	0.1UF
7	2	D1,D2	1N5817
8	3	J5,J7,H0	HDRD2
9	2	H1,J6	HDRD16
10	1	H2	HDRD20
11	1	H3	HDRS14
12	2	H4,H5	HDRS20
13	1	I1	22UF, Inductor
14	1	J0	DJ-005
15	1	J1	HDRD40
16	1	J2	HDRS9
17	2	J3,J4	HDRD10
18	1	J8	HDRS3
19	1	J9	HDRS10
20	1	L1	LED
21	1	P0	POT
22	1	R3	1K
23	6	RN1,RN2,RN3,R4,RN4,RN5	10K
24	1	U1	RAM271024
25	1	U2	PD70320_V25
26	1	U3	PROM1024
27	1	U4	ICL7662
28	1	U5	PAL16V8
29	1	U6	MAX691
30	1	U7	24C04
31	1	U8	LM7805Z
32	1	U9	ULN2003
33	1	U10	MAX766
34	1	U11	1489/75C189
35	1	U12	1488/75C188
36	1	U13	74HC259
37	1	XTAL1	16MHZ

Appendix C. Serial Port SER0/SER1 Drivers

Most embedded applications require serial communications among controllers and PCs. A reliable, high speed, easy to use serial port driver can save you weeks of time, and is essential for all success OEM products. We know what frustrations are, if using a controller with an un-reliable serial port. It eventually will kill the whole project.
TERN controllers and serial port software drivers will provide you with the best serial communication performance in the embedded control industry. There are 2 serial ports in the V25 CPU, SER0 and SER1. By default, SER0 is used by DEBUG. You also can use SER1 for DEBUG with TERN EPROM(1). We will use SER1 as the example in the following discussion. After initialization(***s1_init()***;), SER1 is configured as a full-duplex serial port and is ready to transmit/receive serial data at one of the 16 baud rates from 110 to 1M. The supported baud rates are: 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 76800, 115200, 230400, 460800 and 1 M.
An input buffer(***ser1_in_buf***), size specified by user, will automatically store the input serial data TERNam. There is no software overhead or interrupt latency for user application program even at the highest 1M baud rate, except checking the buffer status(***serhit1()***;) and taking out the data from buffer(***getser1()***;), if any. The input buffer is used as a circular ring buffer, as shown in Fig E-1.



Due to the nature of high speed baud rate and unknown external environment, serial input data will automatically fill in the buffer circularly regardless without stop. If user did not take out the data(***getser1()***;) from the ring buffer before the ring buffer is full, the new data will overwrite the old data without warning or control. The input buffer(ibuf), buffer size(isiz), baud rate(baud), and mode(7/8 data bits, 1/2 stop bits, parity) can be specified by user with ***s1_init()***;. If you defined a 4K(0x1000) input buffer, at 9600 baud, at least in 4 seconds time you do not have to deal with the serial input, although it is always good to take out data from the input buffer before ring buffer round over. You should design higher baud rate for transmiting data out and slower baud rate for the receiving. It will give yourself more time to do other things without overrun the input buffer. You use ***serhit1***(); to check the status of the input buffer and return the offset of the in_head pointer from the in_tail pointer, if return 0, in_head=in_tail, nothing is available. You use ***getser1()***; to get the serial input data byte from FIFO. The in_tail pointer will automatically increment after every ***getser1()***;. There are no need to suspend external device from sending in serial data with /RTS. Only hardware reset or ***s1_close()***; can stop this receiving operation.
For transmission, you can use ***putser1()***, ***putsers1()***, or ***puts1()***; to send out a character, a string, or a block of memory. Before you transmit, your should check the availability of transmit buffer by using ***serout1()***;. If ***serout1()*** returns 0, you can take over the transmission, otherwise, it is busy, you have to wait. After call transmit functions, you are free to do other task with no software overhead on the transmitting operation. It will automatically send out all the data you specified. After all data is out, it will clear the busy flag, ready for the next transmission. A communication structure COM used by SER0, SER1, and SCC, is defined in VE.H. SER0 functions are defined in SER0.H. SER1 functions are defined in SER1.H. A sample program ser1_0.c, demonstrates how a protocol translator works. It will receive an input HEX file from SER1, and translates every ':' to '?', then transmit the HEX file out of SER0. It also works with 57,600 baud input and 115,200 baud output.

void ***s0_init***(unsigned char m, unsigned char b, unsigned char*ibuf, int isiz, unsigned char*obuf, int osiz, COM *c);
SER0 initialization.
void ***s1_init***(unsigned char m, unsigned char b, unsigned char*ibuf, int isiz, unsigned char*obuf, int osiz, COM *c);
SER1 initialization.

```
where    m = mode- 8 bits
//          bit 7=TxE=0,        transmit disable
//          bit 7=TxE=1,        transmit enable
//          bit 6=RxE=0,        receive disable
//          bit 6=RxE=1,        receive enable
//          bit 5,4 = 0,0     No parity
//          bit 5,4 = 0,1     Transmit Parity bit=0, Receive ignore parity
//          bit 5,4 = 1,0     Odd parity
//          bit 5,4 = 1,1     Even parity
//          bit 3 = 0             7 data bits
//          bit 3 = 1             8 data bits
//          bit 2 = 0             1 stop bit
//          bit 2 = 1             2 stop bits
//          bit 1 = 0
//          bit 0 = 1             Async mode
//       For example, mode=0xc9
//       transmit enable, receive enable, no parity, 8 bits, 1 stop bit
//    where    baud - baud rate expressed in number 1 to 15
//                baud = 1,           110 baud
```

```
//          baud = 2,     150 baud
//          baud = 3,     300 baud
//          baud = 4,     600 baud
//          baud = 5,     1200 baud
//          baud = 6,     2400 baud
//          baud = 7,     4800 baud
//          baud = 8,     9600 baud (default)
//          baud = 9,     19,200 baud
//          baud = 10,    38,400 baud
//          baud = 11,    57,600 baud
//          baud = 12,    76,800 baud
//          baud = 13,    115,200 baud
//          baud = 14,    230,400 baud
//          baud = 15,    460,800 baud
//          baud = 15,    1 M baud
```

ibuf = input buffer pointer;   isiz = input buffer size;   obuf = output buffer pointer;   osiz = output buffer size

void ***cntl_rts0***(int flag, COM *c)
  rts0 pin control. where flag =0, rts0 low, flag=1, rts0 high active

void ***cntl_rts1***(int flag, COM *c)
  rts1 pin control. where flag =0, rts1 low, flag=1, rts1 high active

unsigned char ***getser0***(COM *c);
  returns the data in the ser0_in_buf(FIFO). increment the in_tail.

unsigned char ***getser1***(COM *c);
  returns the data in the ser1_in_buf(FIFO). increment the in_tail.

unsigned char ***serhit0***(COM *c);
unsigned char ***serhit1***(COM *c);
  if return 0, nothing received, in_head=in_tail.
  if return >0, the offset, unsigned char pointer in_head leading in_tail.
  if return <0, the offset, unsigned char pointer in_head behind of in_tail.

unsigned char ***serout0***(COM *c);
unsigned char ***serout1***(COM *c);
  returns the number of bytes remaining in the output buffer.
  Only if serout0/1()==0;, a new transmit task can be started.

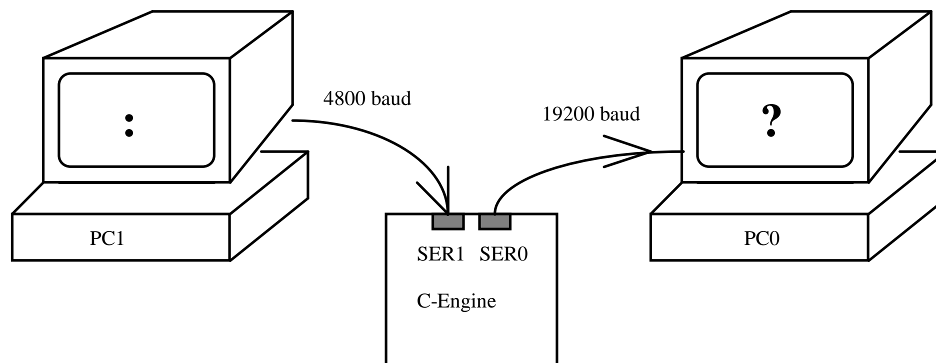
int ***putser0***(unsigned char outch, COM *c);
int ***putser1***(unsigned char outch, COM *c);
  return 0, if transmitter is busy. or
  return 1, after put the outch into out_buf.   It returns before outch being transmited out.

int ***putsers0***(unsigned char *str, COM *c);
int ***putsers1***(unsigned char *str, COM *c);
  return 0, if transmitter is busy. or
  return 1, after put a string *str, into out_buf. It returns before str being transmit out.

int ***puts0***(unsigned char *str, unsigned char n, COM *c);
int ***puts1***(unsigned char *str, unsigned char n, COM *c);
  return 0, if transmitter is busy. or
  return 1, after puting a block of data, pointed by *str, with a length of n bytes into out_buf. It   returns before the string being transmited out.

void ***s0_close***(COM *c);
void ***s1_close***(COM *c);
  close down the ser0/1 transmit and receive.

Sample program ser1_0.c

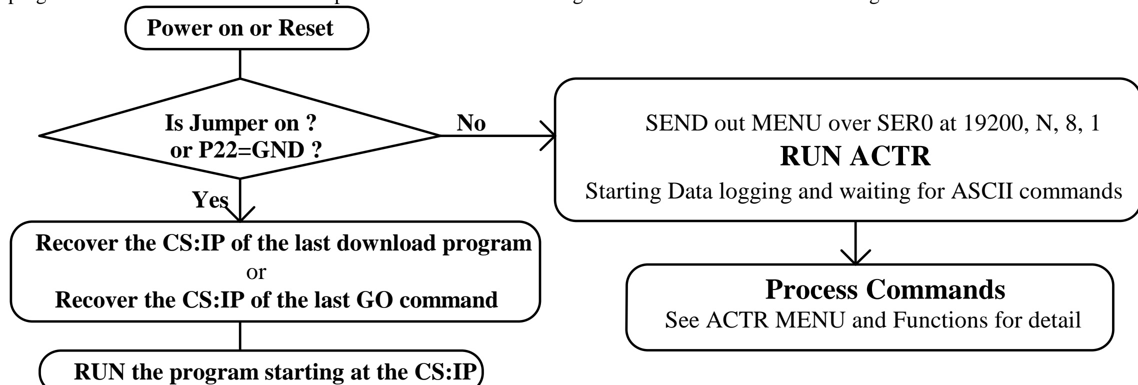


```
/***************************************************************************
        SER0 and SER1 exchange data DEMO 09-14-1994
        ser1_0.c

;   DESCRIPTION:
;       This program shows a possibility using both SER0 and SER1 in application
;       Needs two PCs or terminals to test this sample program.
;       We can not debug this sample program, but we can test it.
;       Load this program with TD/PD, then run from battery back SRAM
;       See Technical Manual for detail about how to run from SRAM
;   This sample program use SER0 9600 baud talk to PC0, window, terminal; SER1 19200 baud talk to PC1 window terminal;
;   Every character typed from PC0-SER0 will be sent to PC1-SER1
;   Every character typed from PC1-SER1 will be sent to PC0-SER0
;   Set PC1 send any HEX file, will shows on PC0, with every ':' translated to '?'
//  Copyright (C) 1995 TERN, INC.  All rights reserved.
***************************************************************************/
#include  <stdio.h>
#include  <dos.h>
#include  <string.h>
#include  "ve.h"        /* V25 initializations */
#include "ser1.h"
#include "ser0.h"

#define MAXISIZE 0x1000
#define MAXOSIZE 0x1000
unsigned char ser1_in_buf[MAXISIZE];
unsigned char ser1_out_buf[MAXOSIZE];
unsigned char ser0_in_buf[MAXISIZE];
unsigned char ser0_out_buf[MAXOSIZE];
int isize,osize;
int i,j;
unsigned char mode,baud;

extern COM ser1_com;
extern COM ser0_com;

void main(void)
{
COM * c0;
COM * c1;
        ve_init();
        c0 = &ser0_com;
        c1 = &ser1_com;
/* transmit enable, receive enable, no parity, 8 bits, 1 stop bit */
        mode = 0xc9;
        baud1 = 7;      /*      4800 baud for SER0      */
        baud0 = 9;      /*      19200 baud for SER1     */
        isize=MAXISIZE;
        osize=MAXOSIZE;
        s1_init(mode,baud,ser1_in_buf,isize,ser1_out_buf,osize,c1);
        s0_init(mode,baud,ser0_in_buf,isize,ser0_out_buf,osize,c0);
while(1){
        if( serhit0(c0) ){                      /* hit by PC0   */
        i = getser0(c0);                                // get the character
        if(i==':') i='?';
        while(!puts1(&i,1,c1));         }                       /* sent to PC1 */
        if( serhit1(c1) ){                      /* hit by PC1 */
        j = getser1(c1);                                // get the character
        if(j==':') j='?';
        while(!puts0(&j,1,c0));         }                       /* sent to PC0 */
        }
}
```

Appendix D. *What is ACTR™ ?*

Operating TERN's 16-bit controllers with ACTR™

What is the first thing you want to do with a new embedded controller ? Read the manual ? Look over the board ? Study the onboard components ? Study the schematics ? Install the software package ? Find a sample program ? Try to learn how to program it in C/C++ ? There are many things that you can do. Maybe the most desirable thing to do is just to power the board, read the ADC, display a message on the LCD, turn on/off LED, or relays... ACTR™ from TERN can do all these for you !
ACTR™ is an unique firmware in EPROM/EEPROM on TERN controllers. With a terminal, such as PC windows terminal, setup to 19200, 8, N, 1, via serial link, and by typing text commands, you can operate the controller and exercise all C functions immediately. For a project requiring embedded controllers, a lot of work must be done before you can have a hardware syTERNm ready for software development. You need to connect sensors to the signal conditioning circuit and the ADC. You may need to acquire several input signals, including analog and digital signal simultaneously. You need to turn relays or solenoids on and off. you need to have a user interface to know what is going on. You may display a message on a LCD, sound a beeper, or flash a LED. You may need to find out what is the practical sample rate for your sensors and syTERNm. You need a easy way to test the power consumption in standby mode. You need to read/write memory, regiTERNr and I/Os. In order to have all this try-outs, you may have to spend days or weeks. The ACTR™ from TERN is a instant actor or actuator for you to operate TERN controllers. ACTR™ provides an interactive menu and on-line help for you, so you do not have to dig into manuals. ACTR™ not only provides you an easy access to all C functions, but also allow you to download a debugger for your C/C++ program development. After you debug your program, you can download your program to the controller and run. An operation functional block diagram of the ACTR™ is shown in Fig. .



ACTR™ may be used in you final product. As long as the jumper is on, every time power on or reset, the controller always run your program. You may download several application programs in different memory location, take off the jumper, use ACTR™ Gxxxxx commands to setup a new CS:IP, then put the jumper back on, and it will run the program you want every time when power on or reset. You also may download a new program in the field with the ACTR™. If you have enough memory space, it is wise to have the ACTR™ onboard. It is also useful in the future for field testing or troubleshooting. The ACTR™ are available in EPROM or Flash EEPROM. Your application program may reside in battery backed SRAM or Flash EEPROM.

**** How to setup the controller ?***

Connect PC COM1/2 and the controller with a serial cable(PC-V25, TERN). Setup PC Windows terminal to 19200, N, 8, 1. Power on the controller with ACTR™ EPROM installed.
A menu should be displayed on the terminal after reset:

```
>C        C FUNCTIONS, SEE HELP C
>D        Download an Intel Extend Hex file
>G        Go and Run the Hex file
>H        HELP, HC1 for C FUNCTIONS
>L        Upload Data Log Record
>M        MENU
>N        Nodes on the network
>S        Set Data Log Sample Rate in the unit of 16 us
>U        Upload a block of Binary data
```

ACTR™ only takes commands of upper case keys. Every command must terminate with a "Enter"/CR.
If an error occur, it will show a message as: "Upper Case Keys only ! Start with ~ for remote, M for menu"

**** How to get help ?***

"H" will show a help menu:
"HC1 ---poke,peek,inport, outport"
"HC2 ---p0,p1,p2,pt,led,hitwd,rtc,ce_ad12"
"HC3 ---SER1 s1_init, putsers1, getser1, serhit1"
"HC4 ---lcd, lcd2, and printf"
"HC5 ---BirdBox PDC, bb_led, bb_beep"
"HC6 ---SensorWatch ad, da, led, beep, hv, relay, ptc"
"HC7 ---PowerDrive ad, da, led, beep, hv, relay, hp"

"HC8 ---TinyDrive ad, hv"
"HD ----Download a HEX FILE"
"HU ----Upload 256 bytes data(HEX) starts at xxxxx"
"HL ----Upload Data Log Record in the ring_buf"
"HS ----Set a Sample Rate for Data Log Operation"

"C" is the main command to active C functions. Similar to the way DOS handles BIOS function calls, ACTR™ uses a code to associate with each C function. In order to know the codes, you may use "HCx" for helping on C.
For example, If you want to know more on poke, peek, inport and outport, "**HC1**" will show:

"C11 SSSS OOOO BB-----pokeb(SSSS,OOOO,BB);";
"C12 SSSS OOOO BBBB---poke(SSSS,OOOO,BBBB);";
"C13 SSSS OOOO--------peekb(SSSS,OOOO);";
"C14 SSSS OOOO--------peek(SSSS,OOOO);";
"C15 PPPP-------------inportb(PPPP);";
"C16 PPPP BB----------outportb(PPPP,BB);";
"C17 -----------------halt()";
"C18 -----------------stop()";

C13 1234 0005 will show one byte memory content at address of 1234:0005.
If you want to know more on V25 p0,p1,p2,pt,led,hitwd,rtc,ce_ad12, "**HC2**" will show:

"C21----------------ve_init();"
"C22----------------hitwd();"
"C23 B--------------led(B); B=0/1"
"C24 P BB----------port_init(P,BB); P=0,1,2"
"C25 P BB----------port_wr(P,BB); P=0,1,2"
"C26 P--------------port_rd(P); P=0,1,2"
"C27 R--------------portt_wr(R); R=0-F, VREF"
"C28-----------------portt_rd(); read comparators"
"C29 AA BB-------iram_wr(AA,BB); write BB to AA"
"C2A AA-----------iram_rd(AA); read IRAM"
"C2B C--------------ce_ad12(C); read CE/TD 12-bit ADC channel C=0-A"
"C2C-----------------rtc_rd(); read RTC"
"C2D WYYMMDDHHMMSS--rtc_init();"

"C2C" will show you the real time
If you want to know how to use UART SER1 functions, s1_init, putsers1, getser1,serhit1,"**HC3**" will show:

"C30 MM BB----s1_init(mode,baud);"
"C31----------s1_close();"
"C33 AAAA ----putsers1; write string AAAA to SER1"
"C35----------getser1; read from SER1"
"C36----------serhit1; return 1, if any char. in the buffer"
"C38 AAAA DD--ee_wr(AAAA,DD); write EEPROM"
"C39 AAAA-----ee_rd(AAAA); read EEPROM"

"C30 C9 09" will initialize the SER1 to 9600 baud, 8, N, 1. Please see C-Engine™ manual for SER1 details. "C33 hello" will send a string of "hello" out of SER1 TXD1 pin.
If you want to know how to use LCD, printf(); functions, "**HC4**" will show:

"C40--------lcd_init();"
"C41 CC-----lcdcmd(CC);, CC=80, 1st line, CC=C0, 2nd line"
"C42 DD-----lcddat(DD);"
"C44 CC-----lputc(CC); put CC at cursor"
"C45 CC-----lcd_clr_line(CC);"
"C46--------lcd2_init();"
"C47 CC-----lcd2cmd(CC);, CC=80, 1st line, CC=C0, 2nd line"
"C48 DD-----lcd2dat(DD);"
"C4A CC-----lputc(CC); put char CC at cursor"
"C4B CC-----lcd2_clr_line(CC);"
"C4C xxxx---printf(xxxx)"

You must use "C40" to initial LCD first, before use LCD functions, otherwise it may lock up the syTERNm.
"**HG**" helps on how to goto a new program

"Gxxxxx ---Goto xxxxx and modify EEPROM start addr";

"**HU**" helps on how to upload a block of memory

"U12345 ---READ 256 bytes of MEMORY starts at 1234:0005"

"**HD**" helps on Intel Hex file download
"**HS**" helps on how to set the sample rate for ring buffer data logging

"Sxxxx ---Sample Rate, min. xxxx=0271 equ 625x16 us, FFFF=1.048s"

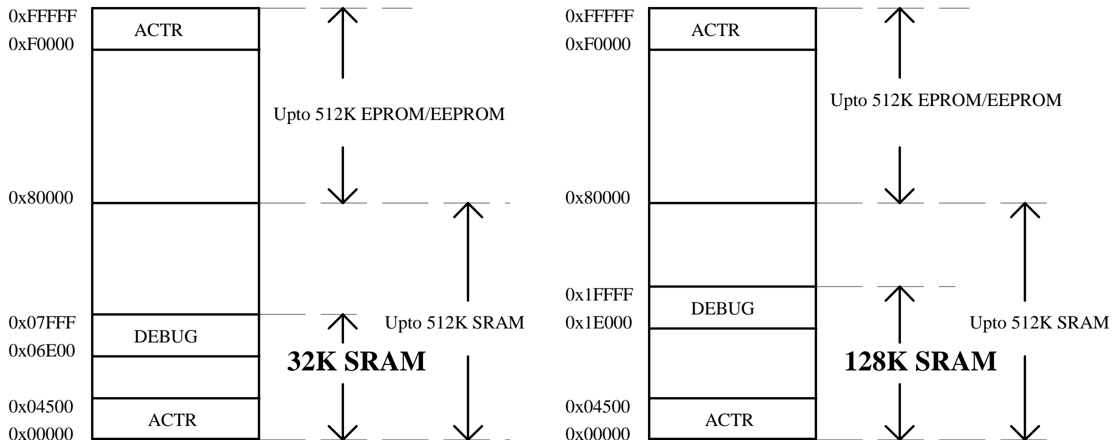
**** What is the data log ring buffer ?***
If no command is issued, ACTR™ will continuely sample ADC, P0, P1, P2, PT, RTC and flashing LED.
The sample rate can be setup with "S" command. The data collected will be recorded in a ring buffer in a Log data record format:

" 11 channels 12-bit ADC reading:"
" V25 P0, P1, P2, PT reading:"

" Real time YYMMDD HHMMSS:"

The memory map

You may use 32K SRAM to use ACTR™. At least 128K SRAM must be installed in order to program and debug in C/C++ with large program or floating point calculation.
ACTR™ is in EPROM/EEPROM in the 1 M byte memory, starting at 0xf0000 to 0xfffff.
ACTR™ uses SRAM starting at 0x01000 to 0x04500 for data log ring buffer, UART buffer and variables.
You may load DEBUG with the HEX file pdrem32.hex to 32K SRAM in 0x06e00 to 0x07fff.
You may load DEBUG with the HEX file pdrem128.hex to 128K SRAM in 0x1e000 to 0x1ffff.
You may load DEBUG with the HEX file pdrem512.hex to 512K SRAM in 0x7e000 to 0x7ffff.



How to make an Intel Extended Hex file suitable for ACTR™ downloading

You need the TERN C/C++ Development Kit(DV) and TERN C library disk.
The C/C++ program must have been well debugged.
Set the makefile DEBUG=0,
Change the TEST32 to load
Use load.rm as the configuration file
Set the correct segment map as you want, for example:
segment map

```
map     0x00000 to 0x00fff as reserved   // Interrupt vector table
map     0x01000 to 0x17fff as rdwr       // 64KB RAM address space
map     0x18000 to 0x1ffff as rdonly     // CODE
map     0x20000 to 0xfffff as reserved   // No access
dup             DATA ROMDATA             // Make a copy of initialized data
class           CODE = 0x1800            // Assume loading at address 18000H
class           DATA = 0x0100            // Data at address 01000H
```

How to program in C/C++ ?

1. You need the DEBUG(PDREM1E.HEX) for 128K or (PDREM7E.HEX) for 512K SRAM.
2. "D" download the PDREM1E.HEX to the controller with Windows, Terminal, Send Text File.
3. Set a jumper for P22=GND, Power off/on or reset. The controller is ready for debug.
4. Use p led for led.c, refer to C-Engine™ Technical Manual for details.

How to down load an Intel Extended Hex file ?

1. Use "D" command. It will response with
   "Ready to receive Intel Extend HEX file at 19200 baud"
2. Select windows terminal , File transfer, "Send Text File", select your Intel Extended Hex file, OK to download.
3. If the file transfer is successful, ACTR™ will show
   "END of File Record" "CHKSUM = 0" "CS=xxxx IP=yyyy"
   If error occur during file transfer, ACTR™ will show "Transfer Error ! CHKSUM=???"
   You have to reset and transfer again.

How do I set up the sample rate ?

"S0271" ---Sample Rate, min. xxxx=0271 equ 625x16 us, FFFF=1.048 second

How can I read ADC ?

for a C-Engine™, "C2B C"--------------ce_ad12(C); read CE/TD 12-bit ADC channel C=0-A

How to read the real time clock ?

"C2C"-----------------rtc_rd(); read RTC

How to turn a relay on ?

"C64 D"---------sw_relay(D); relay on, if D=1

How to show "Hello !" on the first line of the 16x2 LCD ?

"C40"--------lcd_init();

"C45 80"-----lcd_clr_line(80);
"C4C Hello !"---printf(Hello !)

**** How to turn a LED on/off ?***

"C23 1"--------------led(1); led on
"C23 0"--------------led(0); led off

**** How to upload a block of memory ?***

"U12345" ---READ 256 bytes of MEMORY starts at 1234:0005

**** How to read the data log ring buffer ?***

"L" will upload the whole ring buffer in the format of
" 11 channels 12-bit ADC reading:"
" V25 P0, P1, P2, PT reading:"
" Real time YYMMDD HHMMSS:"

**** How to transmit or receive a character with SER1 ?***

"C30 C9 09" will initialize the SER1 to 9600 baud, 8, N, 1
"C33 hello" will send a string of "hello" out of SER1 TXD1 pin.
"C36"----------serhit1; return 1, if any char. in the buffer. return 0, if nothing
"C35"----------getser1; read a character from SER1

**** How to actuate slaves over RS-485 network ?***

**** What software drivers are available ?*** we list part of the software drivers here. They are ready to be called, and linked in your applications:

```
void pokeb(unsigned int segm, unsigned int offs, unsigned char b);
void poke(unsigned int segm, unsigned int offs, unsigned int b);
unsigned char peekb(unsigned int segm, unsigned int offs);
unsigned int peek(unsigned int segm, unsigned int offs);
unsigned char inportb(unsigned char port);        void outportb(unsigned char port, unsigned char p);
void halt(void);        void stop(void);
void ve_init(void);        void hitwd(void);        void led (int);
void port_init(unsigned char p, unsigned char mode);        void port_wr(unsigned char p, unsigned char dat);
unsigned char port_rd(unsigned char p);        void portt_wr(unsigned char vref);
unsigned char portt_rd(void);
void iram_wr(int addr, unsigned char dat);        unsigned char iram_rd(int addr);
int ce_ad12(unsigned char ch);        int rtc_rd(TIM *r);
void rtc_init(unsigned char*);
void s1_init(unsigned char m,unsigned char b, unsigned char* ibuf,int isiz, unsigned char* obuf,int osiz, COM *c);
void s1_close(COM *c);
int puts1(unsigned char *str, unsigned char n, COM *c);        int putsers1(unsigned char *str, COM *c);
int putser1(unsigned char outch, COM *c);        unsigned char getser1(COM *c);
int serhit1(COM *c);        unsigned char serout1(void);
int ee_wr(int addr, unsigned char dat);        int ee_rd(int addr);
void timer0_init(unsigned char mode, unsigned int md0, unsigned int tm0);
unsigned int timer0_rd(unsigned char i);
void timer1_init(unsigned char mode, unsigned int md1, unsigned int tm1);
unsigned int timer1_rd(unsigned char i);
void lcd_init(void);    void lcdcmd(int cmd);        void lcddat(int dat);
void lcd_wait(void);        void lputc(unsigned char ch);
void lcd_clr_line(unsigned char code);        void lcd2_init(void);        void lcd2cmd(int cmd);
void lcd2dat(int dat);        void lcd2_wait(void); void lputc2(unsigned char ch);
void lcd2_clr_line(unsigned char code);        void printf("xxxx");
void sw_led(unsigned char led, unsigned char onoff);        void sw_do(unsigned char o, unsigned char k);
unsigned char sw_di(unsigned char i);        void sw_hv(unsigned char hv, unsigned char k);
void sw_relay(unsigned char k);        void sw_beep(int t, int l);
int sw_kb_scan();        int sw_ee_wr(int addr, unsigned char dat);        int sw_ee_rd(int addr);
void sw_ad10(int* ad);        void sw_da10(int dat);        unsigned char adch(unsigned char ch);
int sw_ad12(unsigned char ch, unsigned char mode);        int sw_ad12a(unsigned char ch, unsigned char mode);
unsigned int sw_ptc_rd(unsigned char chnnl, unsigned char rmd);
```

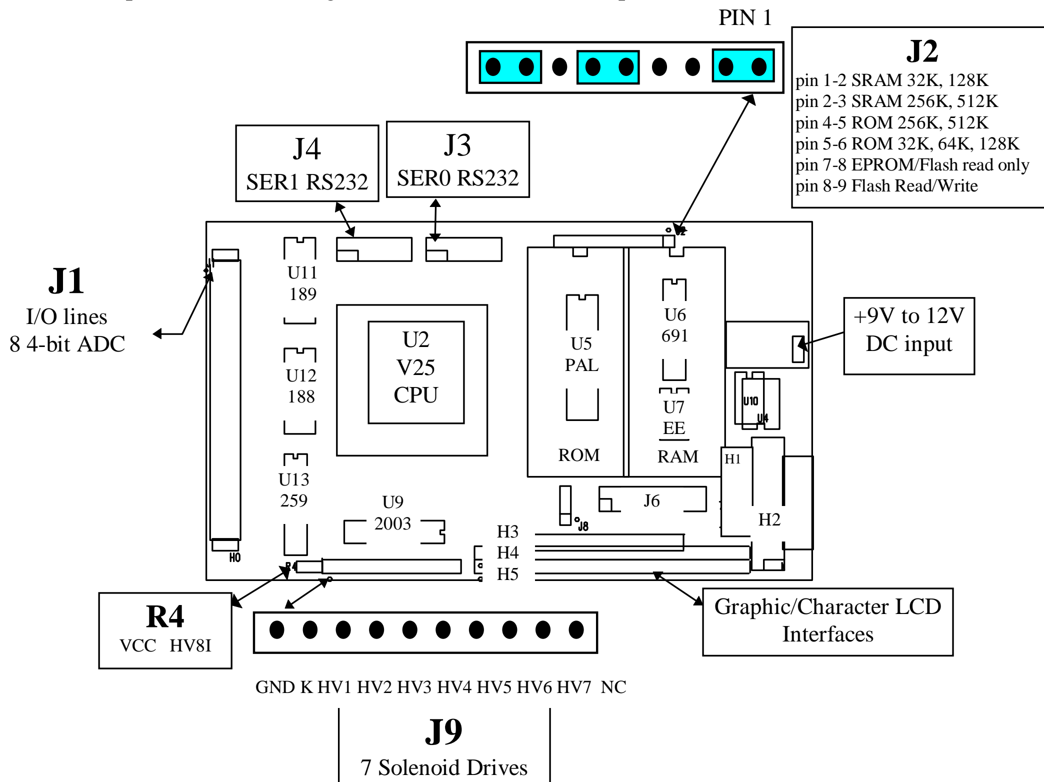
**** Paradigm DEBUG in Intel Extended Hex file for loading to ACTR™:***

pdrem32.hex
pdrem128.hex
pdrem512.hex

Appendix E. Interface a RELAY7™ with MiniDrive™

There are seven High-voltage, High-current Drivers on the MiniDrive. There are 7 12V power relays on the RELAY7.

The ULN2003(U9) on the MD has high voltage, high current Darlington transistor arrays, consisting of 7 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. All driver outputs are routed to J9. as shown below.



You must provide +12V on the MiniDrive to J9 pin 2 "K", in order to power the RELAY7.

A wire can be soldered on the back side of MD, between the LM7805 pin 1 to J9 pin2=K.

The interface signal between MD and RELAY7 are:

GND, K=12V, HV1, HV2, HV3, HV4, HV5, HV6, HV7.

You may connect MD and RELAY7 by solder directly, socket, or wire.
The common substrate G is routed to J9 pin1 GND. All currents sinking in must be return from J9 pin 1 GND. A heavy gage(20) wire may be used to connect GND terminal to external power supply ground return. K is connecting to the protection diodes. K should be tied to +12V in the system.

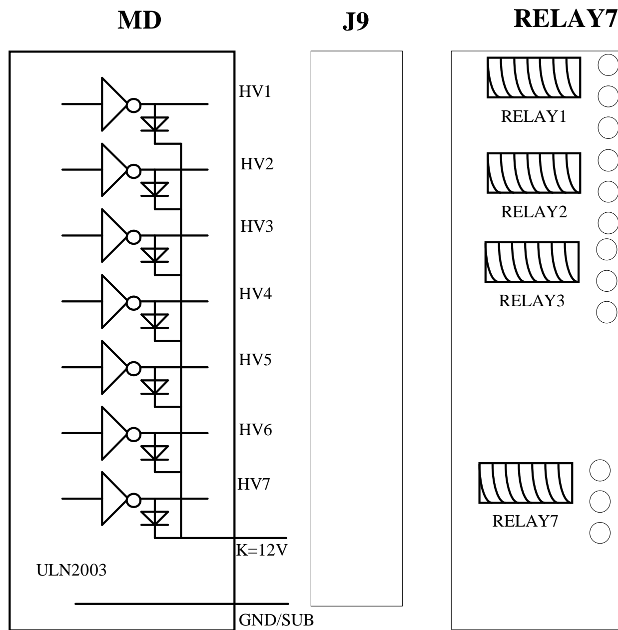
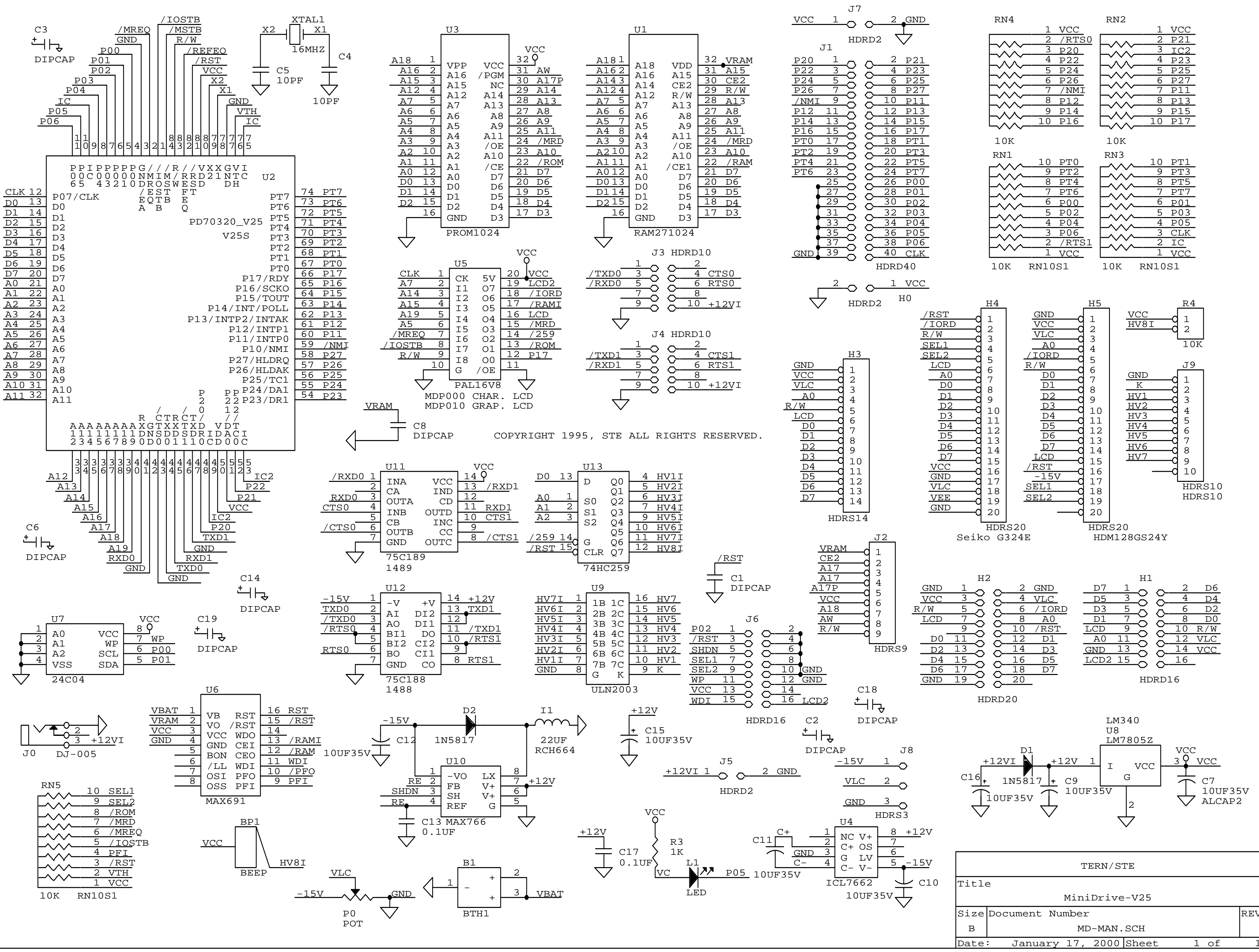


Fig. Drive RELAY7 with MiniDrive high voltage/current drives.

The ULN2003 is driven by U13 74HC259. You can write the 74HC259 with ***outportb(0x80A0+hv, dat);***
where hv=0-6 for solenoid drivers HV1-7 on J9;
hv=7 for TTL output HV8I on R4
dat=0/1, off/on



TERN/STE
Title
MiniDrive-V25
Size B
Document Number
MD-MAN.SCH
REV
Date: January 17, 2000
Sheet 1 of 1
COPYRIGHT 1995, STE ALL RIGHTS RESERVED.
U2 PD70320_V25 V25S
U3 PROM1024
U1 RAM271024
U5 PAL16V8
MDP000 CHAR. LCD
MDP010 GRAP. LCD
XTAL1 16MHZ
C5 10PF
C4 10PF
U11 75C189 1489
U12 75C188 1488
U13 74HC259
U9 ULN2003
U7 24C04
U6 MAX691
U10 MAX766
D2 1N5817
I1 22UF RCH664
U8 LM7805Z LM340
D1 1N5817
U4 ICL7662
J0 DJ-005
BP1 BEEP
P0 POT
B1 BTH1
R3 1K
L1 LED
J1 HDRD40
J2 HDRS9
J3 HDRD10
J4 HDRD10
J5 HDRD2
J6 HDRD16
J7 HDRD2
J8 HDRS3
J9 HDRS10
H0 HDRD2
H1 HDRD16
H2 HDRD20
H3 HDRS14
H4 HDRS20 Seiko G324E
H5 HDRS20 HDM128GS24Y
R4 10K
RN1 10K RN10S1
RN2 10K
RN3 10K RN10S1
RN4 10K
RN5 10K RN10S1