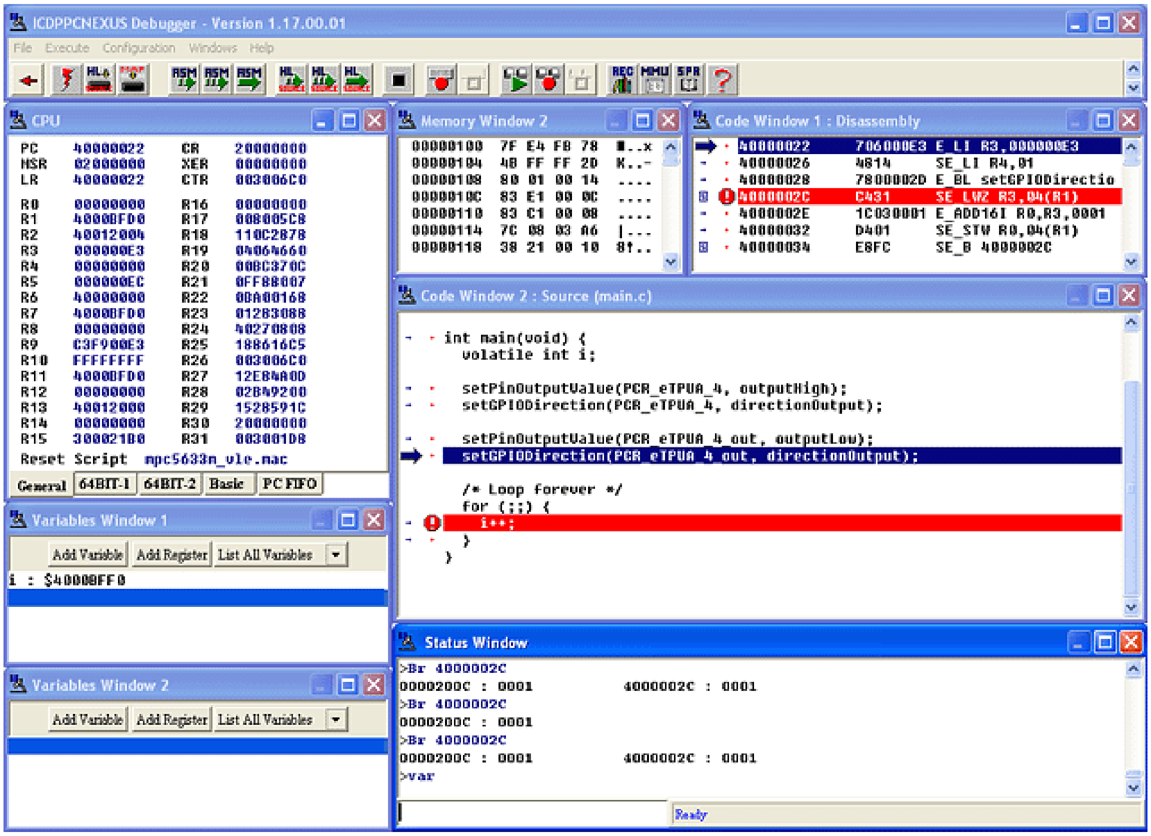
ICDPPCNEXUS User Guide

1 Introduction

PEmicro's ICDPPCNEXUS for Windows is a powerful tool for debugging code on NXP’s MPC55xx-57xx or STMicroelectronics’ SPC5 processors. The debugger comes with PEmicro's PROGPPCNEXUS flash programmer, which is documented separately. ICDPPCNEXUS supports BookE cores, VLE cores, and multiple cores.The debugger uses the processor's JTAG/Nexus debug mode, via a hardware interface, to give the user access to all on-chip resources.

Figure 1-1: ICDPPCNEXUS Debugger


ICDPPCNEXUS Debugger - Version 1.17.00.01
File Execute Configuration Windows Help
CPU
Memory Window 2
Code Window 1 : Disassembly
Code Window 2 : Source (main.c)
Status Window
Variables Window 1
Variables Window 2
General 64BIT-1 64BIT-2 Basic PC FIFO
Add Varible Add Register List All Variables
Ready

```
PC     40000022    CR     20000000
MSR    02000000    XER    00000000
LR     40000022    CTR    003006C0

R0     00000000    R16    00000000
R1     4000BFD0    R17    000005C8
R2     40012004    R18    110C2878
R3     000000E3    R19    04064660
R4     00000000    R20    00BC370C
R5     000000EC    R21    0FF88007
R6     40000000    R22    00A00168
R7     4000BFD0    R23    01283088
R8     00000000    R24    40270808
R9     C3F900E3    R25    188616C5
R10    FFFFFFFF    R26    003006C0
R11    4000BFD0    R27    12EB4A0D
R12    00000000    R28    02849200
R13    40012000    R29    1528591C
R14    00000000    R30    20000000
R15    30002100    R31    003001D8
Reset Script  mpc5633m_vle.mac
```

```
i : $4000BFF0
```

```
00000100  7F E4 F8 78  ■..x
00000104  48 FF FF 2D  K..-
00000108  80 01 00 14  ....
0000010C  83 E1 00 0C  ....
00000110  83 C1 00 08  ....
00000114  7C 08 03 A6  |...
00000118  38 21 00 10  8!..
```

```
40000022   706000E3 E_LI R3,000000E3
40000026   4814     SE_LI R4,01
40000028   7800002D E_BL setGPIODirectio
4000002C   C431     SE_LWZ R3,04(R1)
4000002E   1C030001 E_ADD16I R0,R3,0001
40000032   D401     SE_STW R0,04(R1)
40000034   E8FC     SE_B 4000002C
```

```
int main(void) {
  volatile int i;

  setPinOutputValue(PCR_eTPUA_4, outputHigh);
  setGPIODirection(PCR_eTPUA_4, directionOutput);

  setPinOutputValue(PCR_eTPUA_4_out, outputLow);
  setGPIODirection(PCR_eTPUA_4_out, directionOutput);

  /* Loop forever */
  for (;;) {
    i++;
  }
}
```

```
>Br 4000002C
0000200C : 0001          4000002C : 0001
>Br 4000002C
0000200C : 0001
>Br 4000002C
0000200C : 0001          4000002C : 0001
>var
```

1.1 Trial Edition

You may evaluate ICDPPCNEXUS by downloading the PKGPPCNEXUS 64K Trial Edition from pemicro.com. Please note that this trial download includes some extra software, such as a C compiler, which is not part of ICDPPCNEXUS.

1.2 ICDPPCNEXUS Features At A Glance

- Full-speed in-circuit debug
- Breakpoints with counters on the Nth execution
- Variables window
- Support for multi-core as well as VLE, BOOKE, and mixed mode devices.
- Startup and Macro files for automating the debug process
- Context-sensitive help for all commands
- Symbolic register files to allow decoding of on-chip peripheral registers
- MMU memory map display

- SPR display/modification
- Flash programming integrated into the debugger
- C source-level debugging based on ELF/DWARF 2.0 format

2 Command Line Parameters

To setup ICDPPCNEXUS to run with certain command line parameters, highlight the ICDPPCNEXUS_PRO icon and select PROPERTIES from the Program Manager File Menu.

Syntax:

ICDPPCNEXUS_PRO [option] ... [option]

[option] Optional parameters are as follows:

lpt1…lpt3 Chooses lpt1, lpt2, or lpt3. The software will remember the last setting used.

pci1…pci6 Chooses which PCI card to communicate with. The software will remember the last setting used.

pci_delay n Sets speed of PCI card shift clock, where n = 0…255. The equation for the PCI card shift clock frequency is 33 * 10^6 / (5 + 2n).

running Starts ICD with CPU running (see **Section 10 - Running**).

io_delay_cnt n Causes the background debug mode clock to be extended by 'n' Cycles where 1□=n□=64k. Used when using a very fast PC or a slow CPU clock. (default = 1).

-scriptfile f A macro file that runs at the beginning of debug session. Check **Section 11.49 - MACRO or SCRIPT - Execute a Batch File** for more details

-resetfile f A macro file that runs each time the device resets. You can manually change the script by going to Configuration Menu -> Automatic Script Options.

quiet Starts the ICD without filling the memory windows and the disassembly window. Can be used for speed reasons or to avoid DSACK errors on startup until windows are positioned or chip selects enabled.

-or-

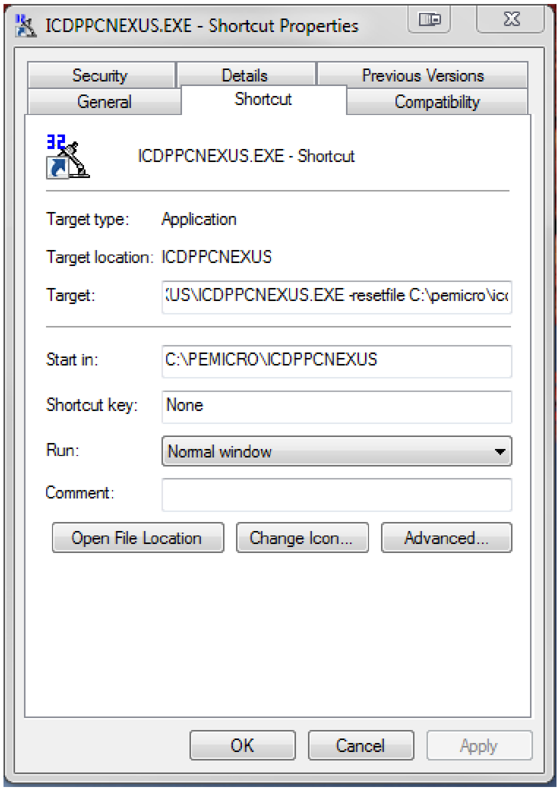
path A DOS path to the directory containing the source code for source level debug or a DOS path to a source file to be loaded at startup (path part is also saved).

Note: If more then one option is given, they must be separated by spaces.

Examples:

ICDPPCNEXUS_PRO lpt2 io_delay_cnt 2 Chooses lpt2, Causes the background debug mode clock to be extended by 2 Cycles.

Figure 2-1: Debugger Properties Dialog



Target type: Application

Target location: ICDPPCNEXUS

Target: (US\ICDPPCNEXUS.EXE -resetfile C:\pemicro\icc

Start in: C:\PEMICRO\ICDPPCNEXUS

Shortcut key: None

Run: Normal window

Comment:

Additionally, if a file named STARTUP.ICD exists in the current directory, it will be run as a macro at startup. See the MACRO command for more information.

3 Nomenclature

Note the following:

n any number from 0 to 0FFFFFFFF (hex). The default base is hex. To enter numbers in another base use the suffixes 'T' for base ten, 'O' for base eight or 'Q' for base two. You may also use the prefixes '!' for base ten, '@' for base 8 and '%' for base two. Numbers must start with either one of these prefixes or a numeric character. Example: 0FF = 255T = 377O = 11111111Q = !255 = @377 = %11111111

add any valid address (default hex).

[ ] optional parameter.

PC Program Counter points to the next instruction.

str ASCII string.

; Everything on a command line after and including the “;” character is considered a comment. This helps in documenting macro (script) files.

4 User Interface

Please note that certain user interface components may differ when using a multi-core device. For more information, please view **Section 5 - Dual Core Considerations**.

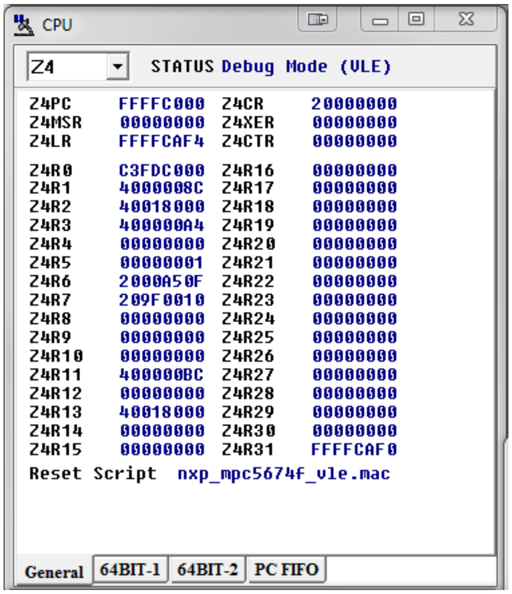
- CPU Window
- Status Window
- Memory Window
- Variables Window
- Code Window
- Colors Window

4.1 CPU Window

The CPU Window displays the current state of the registers.

Note: The appearance of the CPU Window differs slightly when working with multi-core devices. You may view Multi-Core CPU Window to see an example.

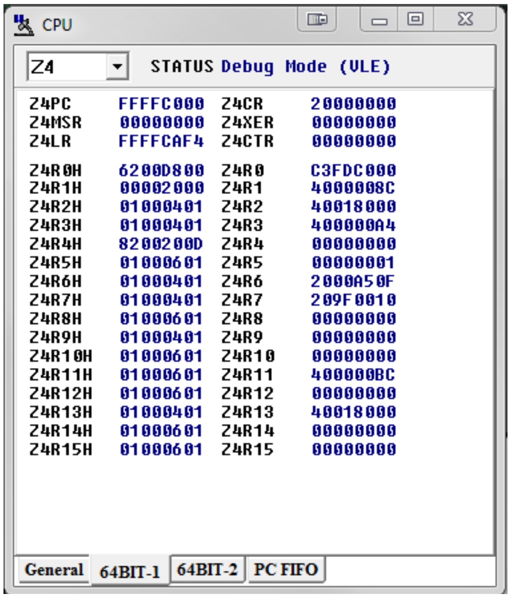
Figure 4-1: CPU Window



```
STATUS Debug Mode (VLE)

Z4PC     FFFFC000  Z4CR     20000000
Z4MSR    00000000  Z4XER    00000000
Z4LR     FFFFCAF4  Z4CTR    00000000

Z4R0     C3FDC000  Z4R16    00000000
Z4R1     4000008C  Z4R17    00000000
Z4R2     40018000  Z4R18    00000000
Z4R3     400000A4  Z4R19    00000000
Z4R4     00000000  Z4R20    00000000
Z4R5     00000001  Z4R21    00000000
Z4R6     2000A50F  Z4R22    00000000
Z4R7     209F0010  Z4R23    00000000
Z4R8     00000000  Z4R24    00000000
Z4R9     00000000  Z4R25    00000000
Z4R10    00000000  Z4R26    00000000
Z4R11    400000BC  Z4R27    00000000
Z4R12    00000000  Z4R28    00000000
Z4R13    40018000  Z4R29    00000000
Z4R14    00000000  Z4R30    00000000
Z4R15    00000000  Z4R31    FFFFCAF0
Reset Script  nxp_mpc5674f_vle.mac
```

Use the tabs at the bottom of the window to display the 64 bit SPE registers.

Figure 4-2: 64-Bit SPE Registers



```
STATUS Debug Mode (VLE)

Z4PC      FFFFC000  Z4CR      20000000
Z4MSR     00000000  Z4XER     00000000
Z4LR      FFFFCAF4  Z4CTR     00000000

Z4R0H     6200D800  Z4R0      C3FDC000
Z4R1H     00002000  Z4R1      4000008C
Z4R2H     01000401  Z4R2      40018000
Z4R3H     01000401  Z4R3      400000A4
Z4R4H     8200200D  Z4R4      00000000
Z4R5H     01000601  Z4R5      00000001
Z4R6H     01000401  Z4R6      2000A50F
Z4R7H     01000401  Z4R7      209F0010
Z4R8H     01000601  Z4R8      00000000
Z4R9H     01000401  Z4R9      00000000
Z4R10H    01000601  Z4R10     00000000
Z4R11H    01000601  Z4R11     400000BC
Z4R12H    01000601  Z4R12     00000000
Z4R13H    01000401  Z4R13     40018000
Z4R14H    01000601  Z4R14     00000000
Z4R15H    01000601  Z4R15     00000000
```

Double-clicking on any of these registers displays a popup window where the user can modify the register value. Commands are also associated with each of these registers that can be entered in the Status Window. Special Purpose registers can be changed through the SPR command.

KEYSTROKES

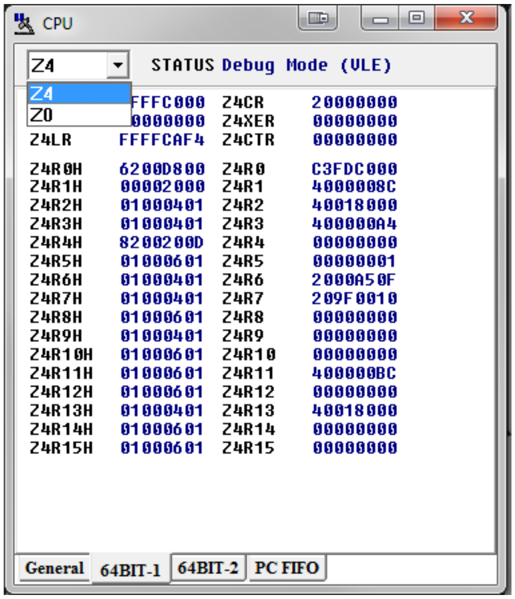
The following keystrokes are valid while the CPU window is the active window:

F1	Shows this help topic
ESC	Make the STATUS window the active window

4.1.1 Multi-Core CPU Window

The appearance of the CPU Window differs slightly when working with multi-core devices. Examples showing the Multi-Core CPU Window with the General tab selected or a core-specific tab selected are shown below:

Figure 4-3: CPU Window & Multi-Core CPU Window



```
STATUS Debug Mode (VLE)

            FFFC000  Z4CR     20000000
            0000000  Z4XER    00000000
Z4LR       FFFFCAF4  Z4CTR    00000000

Z4R0H      6200D800  Z4R0     C3FDC000
Z4R1H      00002000  Z4R1     4000008C
Z4R2H      01000401  Z4R2     40018000
Z4R3H      01000401  Z4R3     400000A4
Z4R4H      8200200D  Z4R4     00000000
Z4R5H      01000601  Z4R5     00000001
Z4R6H      01000401  Z4R6     2000A50F
Z4R7H      01000401  Z4R7     209F0010
Z4R8H      01000601  Z4R8     00000000
Z4R9H      01000401  Z4R9     00000000
Z4R10H     01000601  Z4R10    00000000
Z4R11H     01000601  Z4R11    400000BC
Z4R12H     01000601  Z4R12    00000000
Z4R13H     01000401  Z4R13    40018000
Z4R14H     01000601  Z4R14    00000000
Z4R15H     01000601  Z4R15    00000000
```

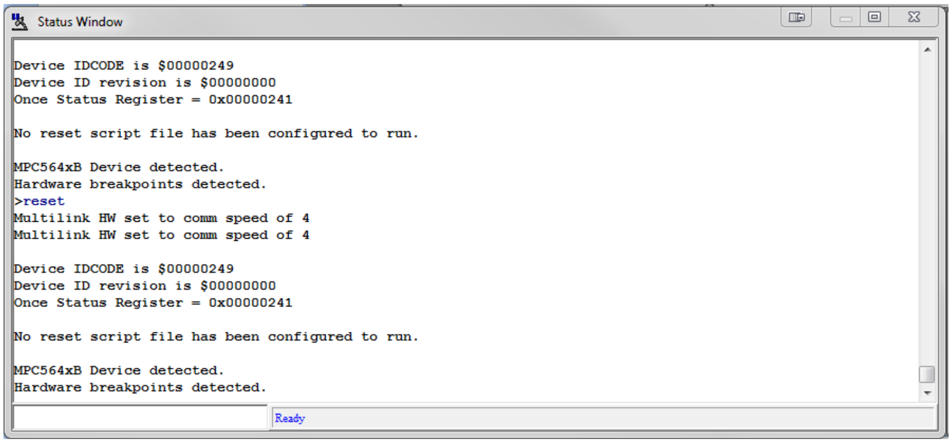
4.2 Status Window

The Status Window serves as the command prompt for the application. It takes keyboard commands given by the user, executes them, and returns an error or status update when needed.

Commands can be typed into the window, or a series of commands can be played from a macro file. This allows the user to have a standard sequence of events happen the same way every time. Refer to the MACRO command for more information.

It is often desirable to have a log of all the commands and command responses which appear in the status window. The LOGFILE command allows the user to start/stop the recording of all information to a text file that is displayed in the status window.

Figure 4-4: Status Window



```
Device IDCODE is $00000249
Device ID revision is $00000000
Once Status Register = 0x00000241

No reset script file has been configured to run.

MPC564xB Device detected.
Hardware breakpoints detected.
>reset
Multilink HW set to comm speed of 4
Multilink HW set to comm speed of 4

Device IDCODE is $00000249
Device ID revision is $00000000
Once Status Register = 0x00000241

No reset script file has been configured to run.

MPC564xB Device detected.
Hardware breakpoints detected.
```

POPUP MENU

By pressing the RIGHT MOUSE BUTTON while the cursor is over the status window, the user is given a popup menu which has the following options:

Help...

Displays this help topic.

KEYSTROKES

The following keystrokes are valid while the status window is the active window:

UP ARROW	Scroll window up one line
DOWN ARROW	Scroll window down one line
HOME	Scroll window to first status line
END	Scroll window to last status line
PAGE UP	Scroll window up one page
PAGE DOWN	Scroll window down one page
F1	Shows this help topic

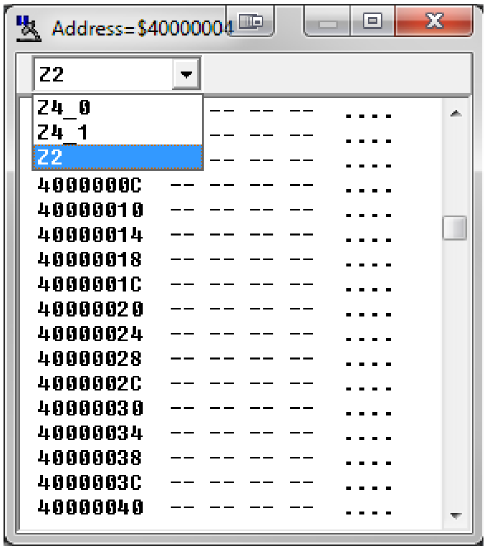
To view previous commands and command responses, use the scroll bar on the right side of the window.

4.3 Memory Window

The Memory Window is used to view and modify the memory map of a target. View bytes by using the scrollbar on the right side of the window. In order to modify a particular set of bytes, just double click on them. Double-clicking on bits brings up a byte modification window.

When working with a multi-core device the user can toggle between cores using the drop-down menu, shown in the upper left of the Memory Window below.

Figure 4-5: Memory Window



Address=$40000004

```
Z4_0            -- -- --   ....
Z4_1            -- -- --   ....
Z2              -- -- --   ....
4000000C  -- -- -- --   ....
40000010  -- -- -- --   ....
40000014  -- -- -- --   ....
40000018  -- -- -- --   ....
4000001C  -- -- -- --   ....
40000020  -- -- -- --   ....
40000024  -- -- -- --   ....
40000028  -- -- -- --   ....
4000002C  -- -- -- --   ....
40000030  -- -- -- --   ....
40000034  -- -- -- --   ....
40000038  -- -- -- --   ....
4000003C  -- -- -- --   ....
40000040  -- -- -- --   ....
```

POPUP MENU

By pressing the RIGHT MOUSE BUTTON while the cursor is over the memory window, the user is given a popup menu that has the following options:

Set Base Address

Sets the memory window scrollbar to show whatever address the user specifies. Upon selecting this option, the user is prompted for the address or label to display. This option is equivalent to the Memory Display (MD) Command.

Show Memory and ASCII

Sets the current memory window display mode to display the memory in both HEX and ASCII formats.

Show Memory Only

Sets the current memory window display mode to display the memory in HEX format only.

Help...

Shows this help topic.

KEYSTROKES

The following keystrokes are valid while the memory window is the active window:

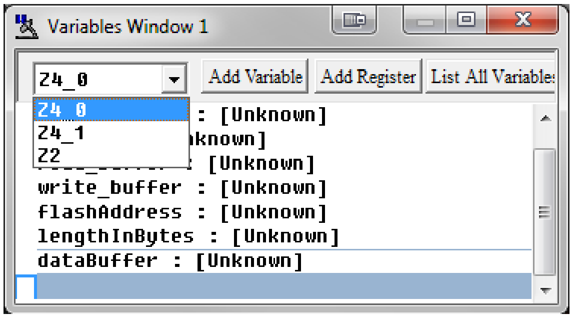
UP ARROW	Scroll window up one line
DOWN ARROW	Scroll window down one line
HOME	Scroll window to address $0000
END	Scroll window to last address in the memory map.
PAGE UP	Scroll window up one page
PAGE DOWN	Scroll window down one page
F1	Shows this help topic
ESC	Make the STATUS WINDOW the active window

4.4 Variables Window

Note: The appearance of the Variables Window differs slightly when working with multi-core devices. You may view Multi-Core Variables Window to see an example.

The variables window allows the user to constantly display the value of application variables. The following window shows a display of selected variables in the demonstration application:

Figure 4-6: Variables Window



```
: [Unknown]
known]
: [Unknown]
write_buffer : [Unknown]
flashAddress : [Unknown]
lengthInBytes : [Unknown]
dataBuffer : [Unknown]
```

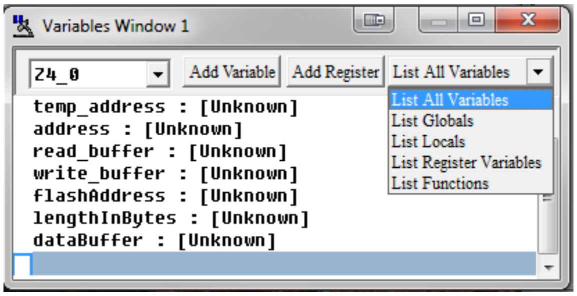
Variables that are pointers are displayed in red. Normal variables are displayed in black. Real-time variables are displayed in blue. A real-time variable is a variable that is updated even while the processor is running.

Listing valid variables

After the debugger loads a valid ELF/DWARF2.0 file, several dialogs become available in the List

Variables drop-down menu.

Figure 4-7: Variables Drop-Down Menu



```
temp_address : [Unknown]
address : [Unknown]
read_buffer : [Unknown]
write_buffer : [Unknown]
flashAddress : [Unknown]
lengthInBytes : [Unknown]
dataBuffer : [Unknown]
```

The following dialogs are available:

List All Variables

Any valid symbol that is listed in the DWARF debugging infomation appears here. Double-click an entry to add to the Variables window.

List Globals

Global symbols appear. Double-click an entry to add to the Variables window.

List Locals

Local symbols that are currently in scope appear. Double-click an entry to add to the Variables window.

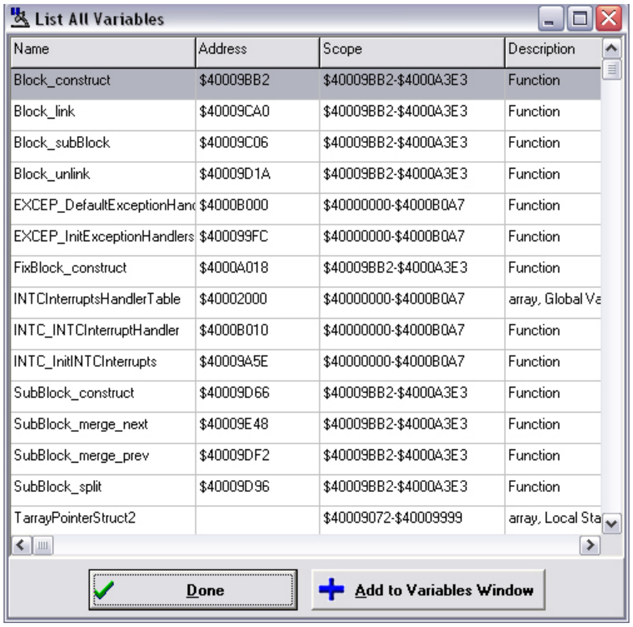
List Register Variables

If a variable resides in a register, rather than memory, it appears here regardless of scope. These variables do not appear with the global or local symbols. Double-click an entry to add to the Variables window.

List Functions

All functions, regardless of scope, appear on this dialog. Double-click an entry to show the source code, if available.

Figure 4-8: List Functions Window



Name	Address	Scope	Description
Block_construct	$40009BB2	$40009BB2-$4000A3E3	Function
Block_link	$40009CA0	$40009BB2-$4000A3E3	Function
Block_subBlock	$40009C06	$40009BB2-$4000A3E3	Function
Block_unlink	$40009D1A	$40009BB2-$4000A3E3	Function
EXCEP_DefaultExceptionHanc	$4000B000	$40000000-$4000B0A7	Function
EXCEP_InitExceptionHandlers	$400099FC	$40000000-$4000B0A7	Function
FixBlock_construct	$4000A018	$40009BB2-$4000A3E3	Function
INTCInterruptsHandlerTable	$40002000	$40000000-$4000B0A7	array, Global Va
INTC_INTCInterruptHandler	$4000B010	$40000000-$4000B0A7	Function
INTC_InitINTCInterrupts	$40009A5E	$40000000-$4000B0A7	Function
SubBlock_construct	$40009D66	$40009BB2-$4000A3E3	Function
SubBlock_merge_next	$40009E48	$40009BB2-$4000A3E3	Function
SubBlock_merge_prev	$40009DF2	$40009BB2-$4000A3E3	Function
SubBlock_split	$40009D96	$40009BB2-$4000A3E3	Function
TarrayPointerStruct2		$40009072-$40009999	array, Local Sta

4.4.1 Using The Variables Window

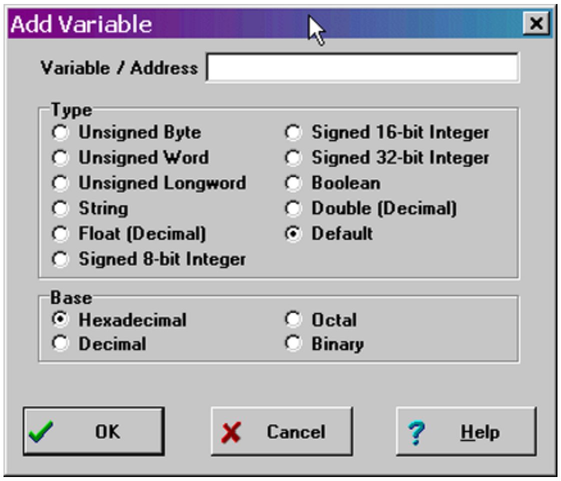
- Adding And Deleting Variables
- Modifying A Variable's Value

- Modifying A Variable's Properties
- Multi-Core Variables Window

4.4.1.1 Adding And Deleting Variables

Variables may be added via the VAR command in the status window, or by right clicking the variables window and choosing “Add a variable.” Variables may be deleted by selecting them and choosing delete. When adding variables, the user is presented with the following dialog:

Figure 4-9: Add/Delete Variable



Add Variable

Variable / Address

Type
○ Unsigned Byte ○ Signed 16-bit Integer
○ Unsigned Word ○ Signed 32-bit Integer
○ Unsigned Longword ○ Boolean
○ String ○ Double (Decimal)
○ Float (Decimal) ● Default
○ Signed 8-bit Integer

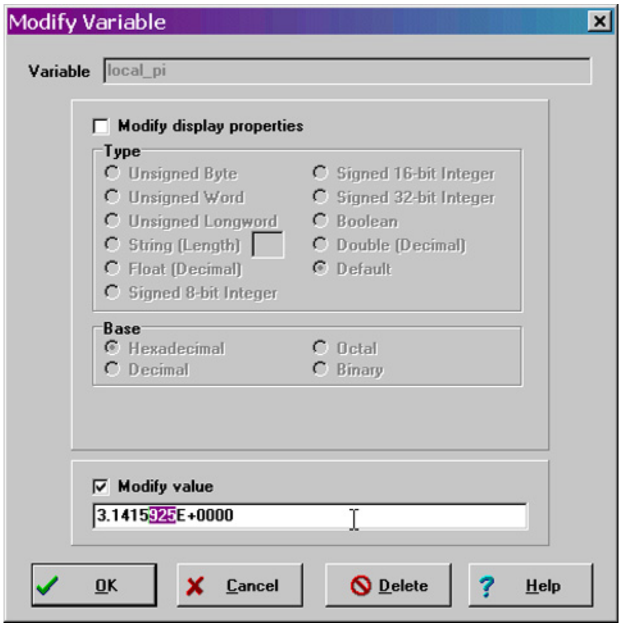
Base
● Hexadecimal ○ Octal
○ Decimal ○ Binary

In the variable field, the user should input the address or name of the variable that they would like displayed in the variables window. The type of the variable should most often be set to “Default,” which means that the variable will be shown as it is defined in the compiled/loaded application. When adding a variable the user may also specify the numeric base in which the variable should be displayed.

4.4.1.2 Modifying A Variable's Value

To modify the current value of a variable, double-click the variable name in the variables window. If the debugger supports modification of this type of variable, the variable modification dialog will be displayed. Make sure to check the “Modify value” checkbox. At this point the value may be altered by the user. When the OK button is clicked, the variable value in the processor’s memory will be updated and the variable window will be refreshed to display this value. Note that some user-defined types, such as enumerated types, may not be editable in this fashion.

Figure 4-10: Modify Value



Modify Variable

Variable: local_pi

□ Modify display properties

Type

- ○ Unsigned Byte
- ○ Unsigned Word
- ○ Unsigned Longword
- ○ String (Length)
- ○ Float (Decimal)
- ○ Signed 8-bit Integer
- ○ Signed 16-bit Integer
- ○ Signed 32-bit Integer
- ○ Boolean
- ○ Double (Decimal)
- ● Default

Base

- ● Hexadecimal
- ○ Decimal
- ○ Octal
- ○ Binary

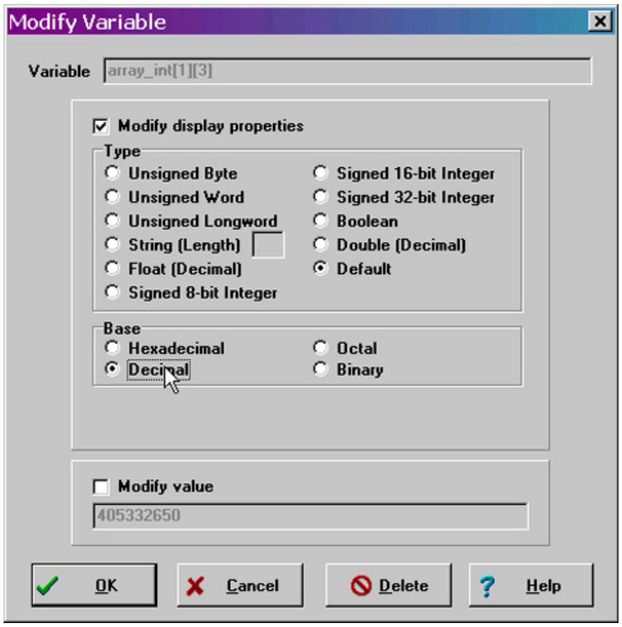
☑ Modify value

3.14159265E+0000

4.4.1.3 Modifying A Variable's Properties

To modify a variable’s display properties, such as the type or numeric display base, double-click the variable in the variables window. Check “Modify display properties” in the dialog that is then displayed. At this point the type and base may be modified. When the OK button is clicked, the variable in the variables window will update its value according to the new settings.

Figure 4-11: Modify Properties



Modify Variable

Variable: array_int[1][3]

☑ Modify display properties

Type

- ○ Unsigned Byte
- ○ Unsigned Word
- ○ Unsigned Longword
- ○ String (Length)
- ○ Float (Decimal)
- ○ Signed 8-bit Integer
- ○ Signed 16-bit Integer
- ○ Signed 32-bit Integer
- ○ Boolean
- ○ Double (Decimal)
- ● Default

Base

- ○ Hexadecimal
- ● Decimal
- ○ Octal
- ○ Binary

□ Modify value

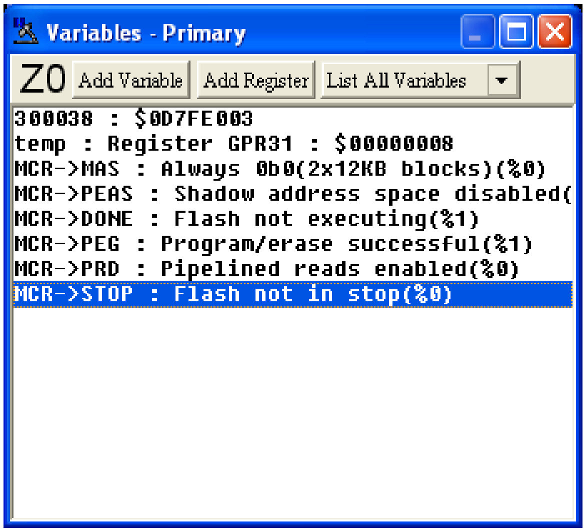
405332650

4.4.2 Multi-Core Variables Window

The appearance of the Variables Window differs slightly when working with multi-core devices.

The window will indicate from which core the data is being displayed. An example is shown below:

Figure 4-12: Multi-Core Variables Window



```
300038 : $0D7FE003
temp : Register GPR31 : $00000008
MCR->MAS : Always 0b0(2x12KB blocks)(%0)
MCR->PEAS : Shadow address space disabled(
MCR->DONE : Flash not executing(%1)
MCR->PEG : Program/erase successful(%1)
MCR->PRD : Pipelined reads enabled(%0)
MCR->STOP : Flash not in stop(%0)
```

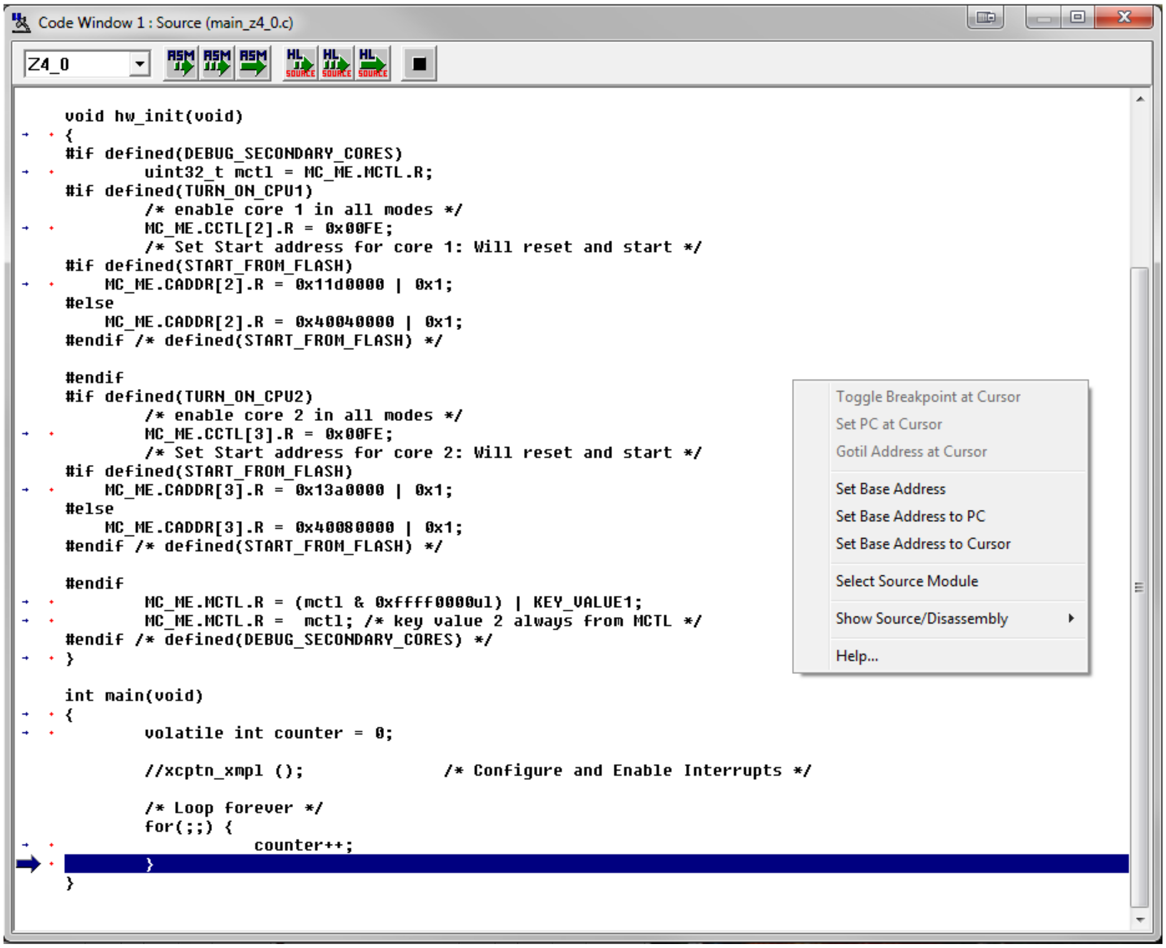
4.5 Code Window

Note: The appearance of the Code Window differs slightly when working with multi-core devices. You may view Multi-Core Code Window to see an example.

The Code Window displays either disassembled machine code or the user's source code if it is available. The "Disassembly" mode will always show disassembled code regardless of whether a source file is loaded. The "Source/Disassembly" mode will show source code if source code is loaded and the current PC points to a valid line within the source code, and shows disassembly otherwise. To show both modes at once, the user should have two code windows open and set one to "Disassembly" and the other to "Source/Disassembly".

Code windows also give visual indications of the Program Counter (PC) and breakpoints. Each code window is independent from the other and can be configured to show different parts of the user's code.

Figure 4-13: Code Window



```
void hw_init(void)
{
#if defined(DEBUG_SECONDARY_CORES)
        uint32_t mctl = MC_ME.MCTL.R;
#if defined(TURN_ON_CPU1)
        /* enable core 1 in all modes */
        MC_ME.CCTL[2].R = 0x00FE;
        /* Set Start address for core 1: Will reset and start */
#if defined(START_FROM_FLASH)
    MC_ME.CADDR[2].R = 0x11d0000 | 0x1;
#else
    MC_ME.CADDR[2].R = 0x40040000 | 0x1;
#endif /* defined(START_FROM_FLASH) */

#endif
#if defined(TURN_ON_CPU2)
        /* enable core 2 in all modes */
        MC_ME.CCTL[3].R = 0x00FE;
        /* Set Start address for core 2: Will reset and start */
#if defined(START_FROM_FLASH)
    MC_ME.CADDR[3].R = 0x13a0000 | 0x1;
#else
    MC_ME.CADDR[3].R = 0x40080000 | 0x1;
#endif /* defined(START_FROM_FLASH) */

#endif
        MC_ME.MCTL.R = (mctl & 0xffff0000ul) | KEY_VALUE1;
        MC_ME.MCTL.R =  mctl; /* key value 2 always from MCTL */
#endif /* defined(DEBUG_SECONDARY_CORES) */
}

int main(void)
{
        volatile int counter = 0;

        //xcptn_xmpl ();              /* Configure and Enable Interrupts */

        /* Loop forever */
        for(;;) {
                counter++;
        }
}
```

POPUP MENU

By pressing the RIGHT MOUSE BUTTON while the cursor is over the code window, the user is given a popup menu that has the following options:

Toggle Breakpoint at Cursor

This option is enabled if the user has already selected a line in the code window by clicking on it with the LEFT MOUSE BUTTON. Choosing this option will set a breakpoint at the selected location, or if there is already a breakpoint at the selected location, will remove it.

Set PC at Cursor

This option is enabled if the user has already selected a line in the code window by clicking on it with the LEFT MOUSE BUTTON. Choosing this option will set the Program Counter (PC) to the selected location.

Gotil Address at Cursor

This option is enabled if the user has already selected a line in the code window by clicking on it with the LEFT MOUSE BUTTON. Choosing this option will set a temporary breakpoint at the selected line and starts processor execution (running mode). When execution stops, this temporary breakpoint is removed.

Set Base Address

This option allows the code window to look at different locations in the user's code, or anywhere in the memory map. The user will be prompted to enter an address or label to set the code window's base address. This address will be shown as the top line in the Code Window. This option is equivalent to the SHOWCODE command.

Set Base Address to PC

This option points the code window to look at the address where the program counter (PC) is. This address will be shown as the top line in the Code Window.

Select Source Module

This option is enabled if a source-level map file is currently loaded, and the windows mode is set to "Source/Disassembly". Selecting this option will pop up a list of all the map file's source filenames and allows the user to select one. This file is then loaded into the code window for the user to view.

Show Disassembly or Show Source/Disassembly

This option controls how the code window displays code to the user. The "Show Disassembly" mode will always show disassembled code regardless of whether a source file is loaded. The "Show Source/Disassembly" mode will show source-code if source code is loaded and the current PC points to a valid line within the source code, and shows disassembly otherwise.

In disassembly mode, the base address of the code window is the first line showing in the window when the scrollbar is at the top. Due to the nature of disassembly, you cannot scroll backwards arbitrarily, and hence you must have a starting point. This starting point is the base address. The base address can be set using the SHOWCODE command or by using the popup menu of the code window. The base address has no meaning in source-level mode unless the user tries to change it (again, refer to the *showcode* command).

Help

Displays this help topic.

KEYSTROKES

The following keystrokes are valid while the code window is the active window:

UP ARROW	Scroll window up one line
DOWN ARROW	Scroll window down one line
HOME	Scroll window to the Code Window's base address.
END	Scroll window to last address the window will show.
PAGE UP	Scroll window up one page
PAGE DOWN	Scroll window down one page
F1	Shows this help topic
ESC	Make the STATUS window the active window

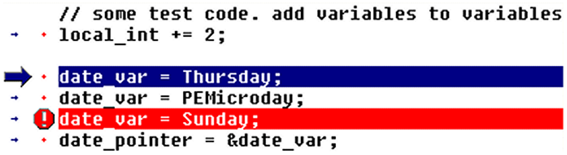
4.6 Using The Code Window

- Using Code Window Quick Execution Features
- Using Code Window Popup Debug Evaluation Hints
- Multi-Core Code Window

4.6.1 Using Code Window Quick Execution Features

In the source code window, there will be a tiny red dot and a tiny blue arrow next to each source instruction that has underlying object code. If a large blue arrow is shown on a source line, this indicates that the program counter (PC) points to this instruction. If a large red stop sign appears on the source line, this indicates that a breakpoint is set on this line. A close-up of the code may be seen below:

Figure 4-14: Code Close-Up



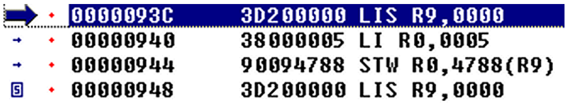
```
// some test code. add variables to variables
local_int += 2;

date_var = Thursday;
date_var = PEMicroday;
date_var = Sunday;
date_pointer = &date_var;
```

The user may set a breakpoint at an instruction by double-clicking the tiny red dot,. When the user issues the HGO command or clicks the high-level language GO button on the debugger button bar, execution will begin in real-time. If the debugger encounters a breakpoint, execution will stop on this source line. If a breakpoint is not encountered, execution will continue until the user presses a key or uses the stop button on the debugger button bar. To remove a breakpoint, double-click the large red stop sign.

By double-clicking the tiny blue arrow, the user will be issuing a GOTIL command to the address of this source line. A GOTIL command will set a single breakpoint at the desired address, and the processor will begin executing code in real-time from the point of the current program counter (PC). When the debugger encounters the GOTIL address, execution will stop. If this location is not encountered, execution will continue until the user presses a key or uses the stop button on the debugger button bar. Note that all set breakpoints are ignored when the GOTIL command is used.

The disassembly window also supports double-clicking of the red and blue symbols, and there is an additional symbol that may appear: a small blue S enclosed in a box. This indicates that a source level instruction starts on this disassembly instruction. An image of this is shown here:

Figure 4-15: Source Level Instruction

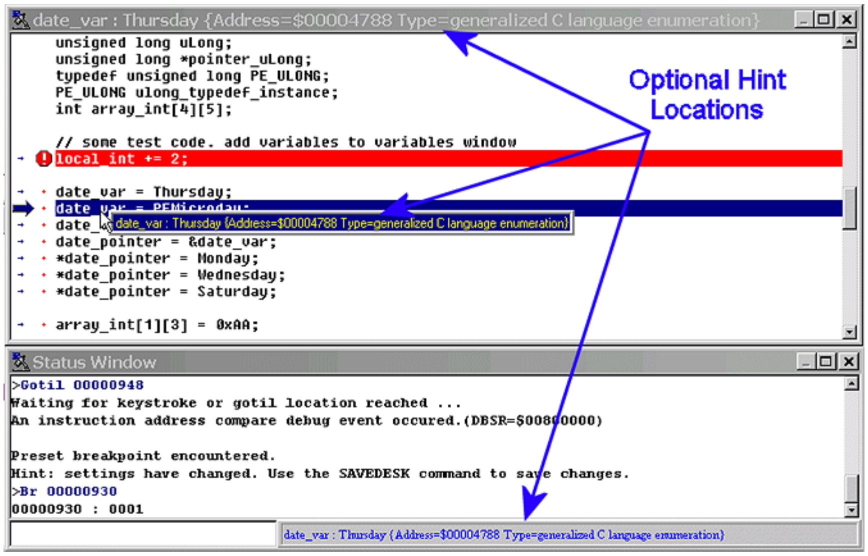


```
0000093C      3D200000 LIS R9,0000
00000940      38000005 LI R0,0005
00000944      90094788 STW R0,4788(R9)
00000948      3D200000 LIS R9,0000
```

4.6.2 Using Code Window Popup Debug Evaluation Hints

When debugging source code, it is often advantageous to be able to view the contents of a variable that appears in the source code. The in-circuit debugger has a feature called “debug hints” which, when active, will display the value of a variable while the mouse cursor is held still over the variable name in the code window. The hint may be displayed in one of three locations, as shown below:

Figure 4-16: Hint Locations



date_var : Thursday {Address=$00004788 Type=generalized C language enumeration}

```
unsigned long uLong;
unsigned long *pointer_uLong;
typedef unsigned long PE_ULONG;
PE_ULONG ulong_typedef_instance;
int array_int[4][5];

// some test code. add variables to variables window
local_int += 2;

date_var = Thursday;
date_var = PEMicroday;
date_
date_pointer = &date_var;
*date_pointer = Monday;
*date_pointer = Wednesday;
*date_pointer = Saturday;

array_int[1][3] = 0xAA;
```

date_var : Thursday {Address=$00004788 Type=generalized C language enumeration}

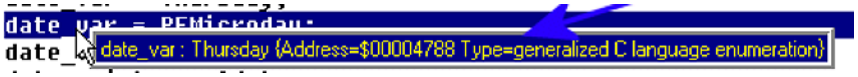
```
>Gotil 00000948
Waiting for keystroke or gotil location reached ...
An instruction address compare debug event occured.(DBSR=$00800000)

Preset breakpoint encountered.
Hint: settings have changed. Use the SAVEDESK command to save changes.
>Br 00000930
00000930 : 0001
```

date_var : Thursday {Address=$00004788 Type=generalized C language enumeration}

The three configurable locations are the code window title bar, the status window caption bar, or a popup that is displayed until the mouse is moved. The hint can be displayed in any combination of the three locations. Locations where the popup hints are displayed are set in the configuration menu of the debugger.

The information displayed in the hint box is similar to the information displayed in the variables window. A close-up image of this hint box is shown here:

Figure 4-17: Hint Box



```
date_var = PEMicroday;
date_
```

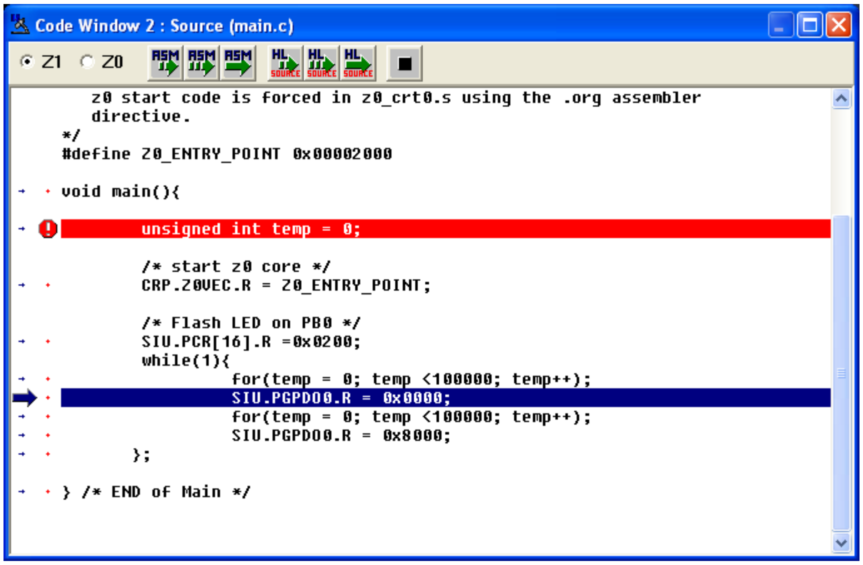
date_var : Thursday {Address=$00004788 Type=generalized C language enumeration}

The information shown is the variable name (date_var), value (Thursday), and type (generalized C language enumeration).

4.6.3 Multi-Core Code Window

The appearance of the Code Window differs slightly when working with multi-core devices. Note the radio buttons which allow the user to toggle between cores. An example of the Multil-Core Code Window is shown below:

Figure 4-18: Multi-Core Code Window



Code Window 2 : Source (main.c)

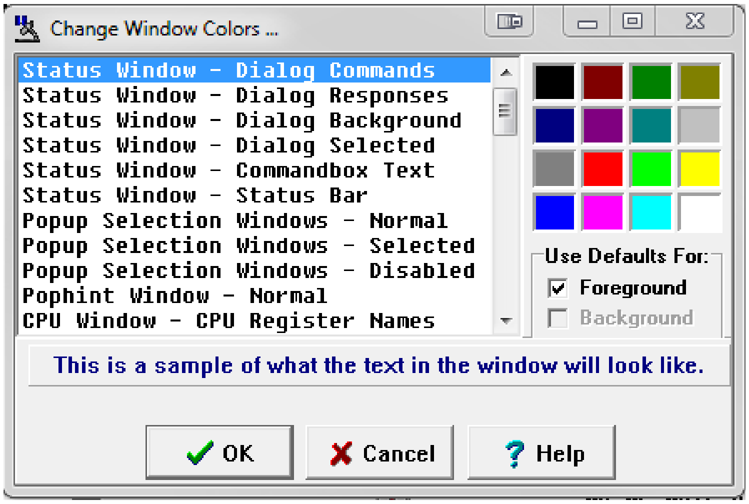
● Z1 ○ Z0

```
    z0 start code is forced in z0_crt0.s using the .org assembler
    directive.
 */
 #define Z0_ENTRY_POINT 0x00002000

 void main(){

         unsigned int temp = 0;

         /* start z0 core */
         CRP.Z0VEC.R = Z0_ENTRY_POINT;

         /* Flash LED on PB0 */
         SIU.PCR[16].R =0x0200;
         while(1){
                   for(temp = 0; temp <100000; temp++);
                   SIU.PGPDO0.R = 0x0000;
                   for(temp = 0; temp <100000; temp++);
                   SIU.PGPDO0.R = 0x8000;
        };

 } /* END of Main */
```

4.7 Colors Window

The Colors Window shows the colors that are set for all of the debugger windows. In order to view the current color in a window, select the item of interest in the list box and view the text in the bottom of the window. To change the color in a window, select the item and then use the left mouse button to select a color for the foreground or use the right mouse button to select a color for the background. Some items will only allow the foreground or background to be changed. Press the OK button to accept the color changes. Press the Cancel button to decline all changes.

Figure 4-19: Colors Window



Change Window Colors ...

Status Window - Dialog Commands
Status Window - Dialog Responses
Status Window - Dialog Background
Status Window - Dialog Selected
Status Window - Commandbox Text
Status Window - Status Bar
Popup Selection Windows - Normal
Popup Selection Windows - Selected
Popup Selection Windows - Disabled
Pophint Window - Normal
CPU Window - CPU Register Names

Use Defaults For:
☑ Foreground
□ Background

This is a sample of what the text in the window will look like.

5 Dual Core Considerations

Some of the ICD’s windows and procedures differ slightly when working with a multi-core processor.

CPU Window - The CPU window displays the CPU registers for both the e200z1 and e200z0 cores. ICD displays each core's register set on a separate tab.

Code Window - The Code Window displays source code for either the e200z1 or e200z0 core. The user may switch between views using the radio buttons in the upper left corner of the Code Window.

Memory Window - The Memory Window displays data for either the e200z1 or e200z0 core. The user may switch between views using the radio buttons in the upper left corner of the Memory Window.

Variables Window - There are two Variables Windows, one for the primary e200z1 core and another for the secondary e200z0 core.

Commands - Most ICD Status Window commands operate on the primary e200z1 core. Use the "Z0" prefix to issue ICD commands to the secondary e200z0 core only.

Example, Source Step:

HSTEP - high-level source step the primary e200z1 core.

Z0HSTEP - high-level source step the secondary e200z0 core.

Example, CPU Values and Names:

PC 100 - sets program counter on primary e200z1 core to 0x100.

Z0PC 100 - sets program counter on secondary e200z0 core to 0x100.

6 Command Recall

You can use the PgUp and PgDn keys to scroll through the past 30 commands issued in the debug window. Saved commands are those typed in by the user, or those entered through macro (script) files. You may use the ESC key to delete a currently entered line including one selected by scrolling through old commands.

Note that only "command lines" entered by the user are saved. Responses to other ICD prompts are not. For example, when a memory modify command is given with just an address, the ICD prompts you for data to be written in memory. These user responses are not saved for scrolling; however, the original memory modify command is saved.

7 CPU Values & Names

CPU Values

Any CPU register with an alphabetic name (R1, PC etc.) may be changed by entering the name of the register followed by a value and <Enter> key in the Status Window will result as:

```
>R1 7654321
```

This will change the value of the low 32 bits of the R1 register to hex value $7654321.

```
>R1H 23456789
```

This will change the value of the high 32 bits of the R1 register to hex value $23456789

```
>PC 100
```

The value of Program Counter register (PC) will be changed to hex value $100

```
>MSR 2000
```

This will turn on the floating point registers.

CPU Names:

REGISTERS

R0 ... R31

R0H ... R31H

PC

CR

MSR

XER

LR

CTR

Note: The SPR Command may be used to change the value of any special purpose register.

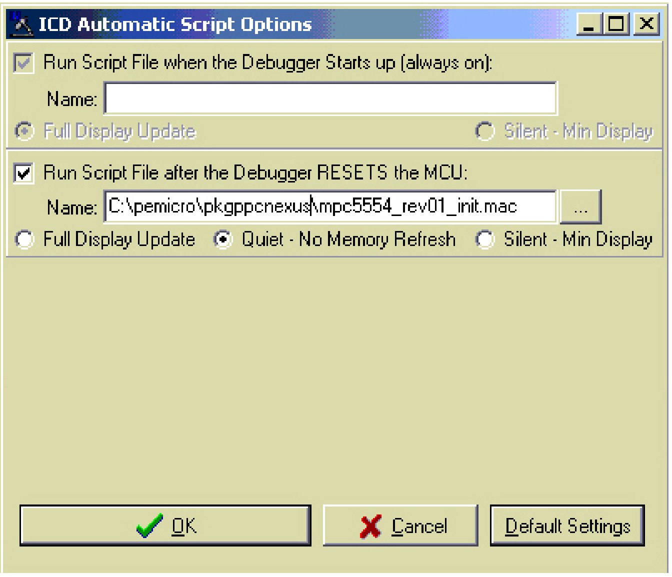
8 Boot Assist Module

The processor Boot Assist Module (BAM) would normally initialize the the processor. However, when running the target application from the debugger, the BAM functionality is disabled. To account for this, the debugger must run a script file on reset. The script initializes the memory of the processor similar to the way in which the BAM would initialize the processor. The user can view and modify all the script's initialization tasks.

The appropriate macro file should automatically run on reset. To run the script file manually, type the MACRO command in the debugger status window or click the play macro ▶ button on the debugger toolbar. Select the macro file MPC5500_any_init.mac, located in the installation directory. The macro file MPC5500_any_init.mac does not initialize the external bus. You may run the macro files MPC5500_ebi_32.mac or MPC5500_ebi_16.mac to configure the external bus for 32 bit or 16 bit memory. When the script file runs, if the debugger has not been configured to run in silent mode, you will see a significant number of status messages scroll down the status window of the debugger.

To configure the debugger macro to execute automatically on reset, select the debugger Configuration menu, Automated Script Options dialog, shown here:

Figure 8-1: Automated Script Options Dialog



ICD Automatic Script Options

☑ Run Script File when the Debugger Starts up (always on):

Name:

● Full Display Update ○ Silent - Min Display

☑ Run Script File after the Debugger RESETS the MCU:

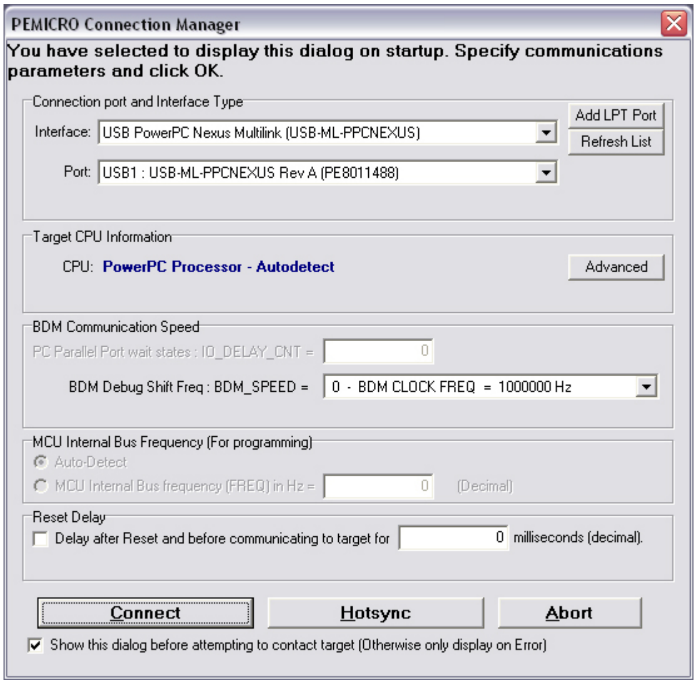
Name: C:\pemicro\pkgppcnexus\mpc5554_rev01_init.mac

○ Full Display Update ● Quiet - No Memory Refresh ○ Silent - Min Display

9 Connection To Target

When you first run the software, you will see the Connection Assistant dialog box:

Figure 9-1: Connection Assistant Dialog



PEMICRO Connection Manager

You have selected to display this dialog on startup. Specify communications parameters and click OK.

Connection port and Interface Type

Interface: USB PowerPC Nexus Multilink (USB-ML-PPCNEXUS)

Port: USB1 : USB-ML-PPCNEXUS Rev A (PE8011488)

Target CPU Information

CPU: PowerPC Processor - Autodetect

BDM Communication Speed

PC Parallel Port wait states : IO_DELAY_CNT = 0

BDM Debug Shift Freq : BDM_SPEED = 0 - BDM CLOCK FREQ = 1000000 Hz

MCU Internal Bus Frequency (For programming)

● Auto-Detect

○ MCU Internal Bus frequency (FREQ) in Hz = 0 (Decimal)

Reset Delay

☐ Delay after Reset and before communicating to target for 0 milliseconds (decimal).

☑ Show this dialog before attempting to contact target (Otherwise only display on Error)

You may use this dialog box to connect to your target using one of PEmicro's debugging interface products. The dialog allows you to specify the type of PEmicro debugging interface hardware. You may also select the CPU type, communications speed, and reset delay. If you wish, you may disable the dialog box so that it will appear only on error.

Use the Connect button to reset the target. Use the Hotsync button to connect to the target without resetting it.

10 Running

Sometimes it is desirable to leave the CPU running and exit the ICD debug software. To do this, use the GOEXIT command. To re-enter the ICD debug software, use the option RUNNING as a parameter on the start up command line (see STARTUP). This option causes the debugger to not do a RESET at startup and to ignore any STARTUP.ICD macro file. In order to use this option, the CPU must have previously been left executing by the debugger.

Another way to re-enter the ICD without resetting the target is to use the Hotsync button on the Connection Assistant.

It is possible to remove the ICD cable from your target system and then reconnect it provided that the following sequence is observed:

1) Exit the ICD by using the GOEXIT command.

2) Disconnect cable without removing power connections.

3) Remove power from cable.

4) Do whatever...

5) Connect power to cable.

6) Start the ICD software using the RUNNING option.

7) Connect cable to your target.

11 Commands

NOTE: If using a multi-core device, a prefix may be used to issue a core-specific command. Please see Dual Core Considerations for more information about command usage for multi-core devices.

11.1 ADDSPR Command – Set User-Defined SPR Name

Sets user-defined SPR name to equal user-defined SPR number.

Note: Let str be the name of special purpose register <n>.

Syntax:

ADDSPR str <n>

Examples:

>ADDSPR MCSR 572T	Machine Check Syndrome Register

11.2 ASCIIF3 and ASCIIF6 Commands – Toggle Memory Window

Toggles the memory windows between displaying [data only] // [data and ASCII characters].

ASCIIF3 toggles memory window 1.

ASCIIF6 toggles memory window 2.

Syntax:

ASCIIF3

Example:

>ASCIIF3	Toggles memory window 1 between displaying [data only] // [data and ASCII characters].

11.2.1 ASM Command - Assemble Instructions

The ASM command assembles instruction mnemonics and places the resulting machine code into memory at the specified address. The command displays a window with the specified address (if given) and current instruction, and prompts for a new instruction. If an instruction is entered and the ENTER button is pressed, the command assembles the instruction, stores and displays the resulting machine code, then moves to the next memory location, with a prompt for another instruction. If there is an error in the instruction format, the address will stay at the current address and an 'assembly error' flag will show. To exit assembly, press the EXIT button. See Instruction Set for related information on instruction formats.

Syntax:

ASM [<address>]

Where:

<address> Address where machine code is to be generated. If you do not specify an <address> value, the system checks the address used by the previous ASM command, then uses the next address for this ASM command.

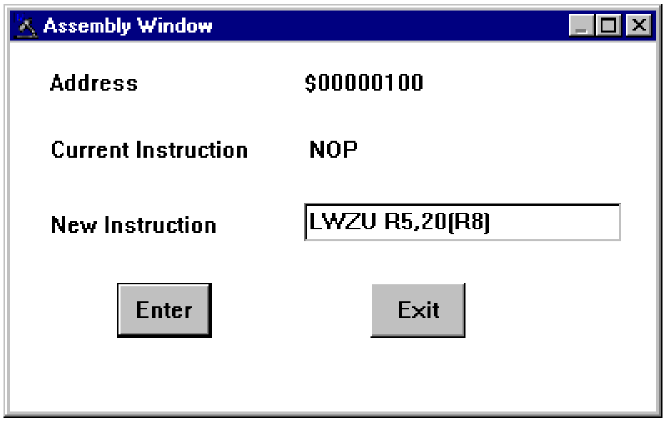
Examples:

With an address argument:

>ASM 100

The following window appears:

Figure 11-1: Connection Assistant Dialog



Assembly Window

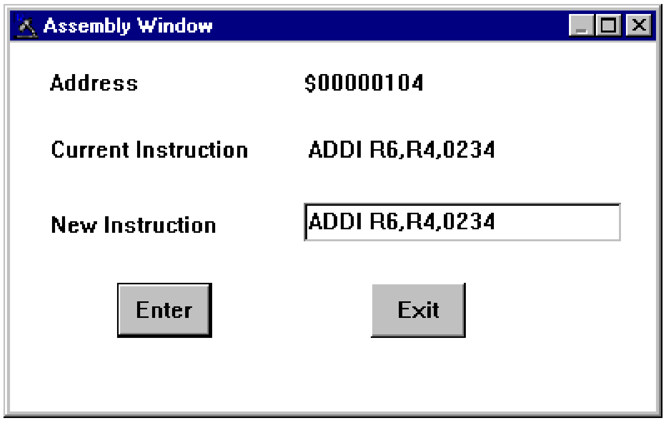
Address $00000100

Current Instruction NOP

New Instruction LWZU R5,20(R8)

The user can type a new instruction in the edit box next to the 'New Instruction' text. In this example, the instruction 'LWZU R5,20(R8)' is typed and then the ENTER button is pressed. As soon as ENTER is click the following window will appears

Figure 11-2: Connection Assistant Dialog



Address $00000104

Current Instruction ADDI R6,R4,0234

New Instruction ADDI R6,R4,0234

This window shows that address is incremented by 4 and the instruction at address is ADDI R6,R4,0234. You can either enter another instruction or click EXIT to get out of this window.

11.3 BELL Command - Sound Bell

The BELL command sounds the computer bell the specified hexadecimal number of times. The bell sounds once when no argument is entered. To turn off the bell as it is sounding, press any key.

Syntax:

BELL [<n>]

Where:

<n>	The number of times to sound the bell.

Example:

>BELL 3	Ring PC bell 3 times.

11.4 BF or BLOCK FILL Command - Fill Block

The BF or FILL command fills a block of memory with a specified byte, word or long. The optional variant specifies whether to fill the block in bytes (.B, the default), in words (.W) or in longs (.L). Word and long must have even addresses.

Syntax:

BF[.B | .W | .L] <startrange> <endrange> <n>

FILL[.B | .W | .L] <startrange> <endrange> <n>

Where:

<startrange>	Beginning address of the memory block (range).
<endrange>	Ending address of the memory block (range).
<n>	Byte or word value to be stored in the specified block.

The variant can either be .B, .W, .L, where:

.B Each byte of the block receives the value.

.W Each word of the block receives the value.

.L Each word of the block receives the value.

Examples:

>BF C0 CF FF	Store hex value FF in bytes at addresses C0-CF.
>FILL C0 CF FF	Store hex value FF in bytes at addresses C0-CF
>BF.B CO CF AA	Store hex value AA in bytes at addresses C0-CF.
>FILL.B CO CF AA	Store hex value AA in bytes at addresses C0-CF.
>BF.W 400 41F 4143	Store word hex value 4143 at addresses 400-41F.
>FILL.W 400 41F 4143	Store word hex value 4143 at addresses 400-41F.
>BF.L 1000 2000 8F86D143	Store long hex value 8F86D143 at address 1000-2000
>FILL.L 1000 2000 8F86D143	Store long hex value 8F86D143 at address 1000-2000

11.5 BGND_TIME Command - Log Time Since BGND Instruction

First, the processor execution is started at the current PC. Then, each time a BGND instruction is encountered, the time since the last BGND instruction is logged in memory. Up to n points (default = 500 and max = 500 data points) may be logged. The accuracy is somewhere in the microsecond range. There is some positive time error to get in and out of background mode. In addition, while the ICD software is storing the information, the target processor is not running which introduces a real time error. One can determine the amount of time spent by the ICD to go into and out of BGND mode by timing the execution of a string of BGND instructions and deducting this from the times given. The data logging stops when 500 points have been logged or the operator presses a key. The logged points are then written to the debug window and also to the capture file if enabled.

Syntax:

BGND_TIME [n]

Where:

nnumber of points logged

Example:

>BGND_TIME 4

The above command will give you four time differences (t1,t2,t3,t4).

```
  PC--------->BGND1----------->BGND2----------------------->BGND3----------------------->BGND4
    <-----t1------><---------t2-----------><------------t3---------------><---------------t4----------------->
```

11.6 BR Command - Set Or Clear Breakpoint

Sets or clears a breakpoint at the indicated address. Break happens if an attempt is made to execute code from the given address. There are at most 7 breakpoints. They cannot be set at a misaligned address. Typing BR by itself will show all the breakpoints that are set and the current values for n.

Note: The first 2 breakpoints set are hardware breakpoints; after that the breakpoints are set through software. Software breakpoints can only be set in RAM. Hardware breakpoints can be set anywhere in memory.

Syntax:

BR [add] [n]

Where:

add	Address at which a break point will be set.
n	If [n] is specified, the break will not occur unless that location has been executed n times. After the break occurs, n will be reset to its initial value. The default for n is 1.

Examples:

```
>BR           ; Shows all the breakpoints that are set and the current values for n.
>BR  100      ; Set break point at hex address 100.
>BR   200  5  ; Break will not occur unless hex location 200 has been executed 5 times.
```

11.7 CAPTURE Command - Open Capture File

Opens a capture file named 'filename'. Most outputs to the debug window are also sent to the capture file. The user is prompted for information as to appending to or deleting the 'filename' file if it already exists.

Syntax:

CAPTURE <filename>

Where:

<filename>	Name of the file where commands and messages are stored.

Example:

>CAPTURE testfile	Capture all the command and messages displayed at the debug window into the file "TESTFILE.CAP".

11.8 CAPTUREOFF Command - Turn Off Capture

Turns off capturing of commands and messages at the debug window and closes the current capture file.

Syntax:

CAPTUREOFF

Example:

>CAPTUREOFF	Turns off capturing of commands and messages at the debug and window closes the current capture file.

11.9 CLEARMAP Command - Clear Map File

The CLEARMAP command removes the current MAP file from memory. This will force the debugger to show disassembly in the code windows instead of user source code. The user defined symbols, defined with the SYMBOL command, will not be affected by this command. (The NOMAP command is identical to CLEARMAP.)

Syntax:

CLEARMAP

Example:

>CLEARMAP   Clears symbol and source information.

11.10 CLEARSYMBOL Command - Clear User Symbols

The CLEARSYMBOL command removes all the user defined symbols. The user defined symbols are all created with the SYMBOL command. The debug information from MAP files, used for source level debugging, will be unaffected. The NOSYMBOL command is identical.

Note: Current user defined symbols can be listed with the SYMBOL command.

Syntax:

CLEARSYMBOL

Example:

>CLEARSYMBOL   Clears user defined symbols.

11.11 CLEARVAR Command - Remove All Variables

The CLEARVAR command removes all the variables from the variables window.

Syntax:

CLEARVAR

Example:

CLEARVAR   Removes all the variables from the variables window.

11.12 CODE Command - Show Disassembled Code

Shows disassembled code in the code window starting at address add. If you specify an address in the middle of an intended instruction, improper results may occur.

Syntax:

CODE <add>

Where:

<add>   Address where your code begins.

Example:

>CODE 100   Shows the disassembled code in the code window starting at hex address 100.

11.13 COLORS Command - Set Colors of Simulator

The COLORS command brings up a popup window, the Colors Window, that allows the user to choose the text and background colors for all windows in the debugger. Once colors are selected for the windows, use the SAVEDESK command to save them for all further debugging sessions. See Colors Window for more information.

Syntax:

COLORS

Example:

>COLORS	Open the colors window.

11.14 COUNT Command - Display Execution Count

The COUNT command tells the user how many times each address in the internal counter table is executed. If no address parameters are provided, the processor will execute from the current Program Counter until an existing breakpoint is encountered, or the user presses a key. If the user provides a starting address [add1], the processor will begin executing from this address until it reaches the second address [add2], or if that parameter is not given, until an existing breakpoint is encountered, or the user presses a key. When a breakpoint or keypress occurs, you are put into the "Show Count" window. The count locations set in the source code window are shown in descending order of executions. The percent is a rough percent of all counts. You may scroll in this window using the cursor keys and return to the debug window by hitting F1.

The addresses in the internal counter table are set using the COUNTER command.

Syntax:

COUNT [add1] [add2]

Where:

add1	Go from first address.
add2	Set breakpoint at second address.

Example:

>COUNT 100 200	Start execution of the program at address 100 and stops at address 200.

11.15 COUNTER Command - Add Or Subtract From Counter Table

Adds or subtracts a location from the internal counter table. The user may then use the COUNT command to count how many times each of the locations in the table executes. Using the COUNTER command with no address shows the current table of counters.

Syntax:

COUNTER [add]

Where:

add	Address to be added to, or removed from, the internal counter table.

Example:

>COUNTER 100	Add (or remove) a counter at hex location 100.
>COUNTER	Shows all the current internal counters.

11.16 CR Command - Condition Register

The CR command sets the condition register (CR) to the specified hexadecimal value.

Syntax:

CR <n>

Where:

<n>   The new hexadecimal value for the CR.

Example:

>CR $C4   Assign the value C4 to the CR.

11.17 CTR Command - Counter Register

The CTR command sets the counter register (CTR) to the specified hexadecimal value.

Note: The counter register is used by the CPU for looping purposes. This register is also a special purpose register.

Syntax:

CTR <n>

Where:

<n>   The new hexadecimal value for the CTR.

Example:

>CTR $100   Assign the value $100 to the CTR.

11.18 DASM Command - Disassemble Memory

The DASM command disassembles machine instructions, displaying the addresses and their contents as disassembled instructions in the status window. The memory locations between the first and second addresses (add) are uploaded to the screen in the form of Bytes, Words, or Long Words. The first address must be on an even boundary for Words or Long Words. If the capture feature is active, the lines of dumped data are also sent to the capture file. Data is read as Bytes, Words, or Long Words from the data space.

> If the command includes an address value, one disassembled instruction is shown, beginning at that address.

> If the command is entered without any parameter values, the software finds the most recently disassembled instruction then shows the next instruction, disassembled.

> If the command includes startrange and endrange values, the software shows disassembled instructions for the range.

Note: If the DASM command is entered with a range, sometimes the disassembled instructions scroll through the status window too rapidly to view. Accordingly, the LF command can be entered, which records the disassembled instructions into a logfile, or use the scroll bars in the status window.

Syntax:

DASM <address1> [<address2>] [n]

Where:

<address1> The starting address for disassembly. <address1> must be an instruction opcode. If you enter only an <address1> value, the system disassembles three instructions.

<address2> The ending address for disassembly (optional). If you enter an <address2> value, disassembly begins at <address1> and continues through <address2>. The screen scrolls upward as addresses and their contents are displayed, leaving the last instructions in the range displayed in the window.

n The optional parameter n determines the number of Bytes, Words, or Long Words which are written on one line.

Examples:

```
>DASM 300
          0300          A6E8          LDA #0E8
          0302          B700          STA PORTA
          0304          A6FE          LDA #FE

>DASM 400 408
          0400          5F            CLRX
          0401          A680          LDA #80
          0403          B700          STA PORTA
          0405          A6FE          LDA #FE
          0407          B704          STA DDRA
```

11.19 DUMP Command - Dump Data Memory to Screen

The DUMP command sends contents of a block of data memory to the status window, in bytes, words, or longs. The optional variant specifies whether to fill the block in bytes (.B, the default), in words (.W), or in longs (.L).

Note: When the DUMP command is entered, sometimes the memory contents scroll through the debug window too rapidly to view. Accordingly, either the LF command can be entered, which records the memory locations into a logfile, or the scroll bars in the status window can be used.

Syntax:

DUMP [.B | .W | .L] <startrange> <endrange> [<n>]

Where:

<startrange> Beginning address of the data memory block.

<endrange> Ending address of the data memory block (range).

<n> Optional number of bytes, words, or longs to be written on one line.

Examples:

```
>DUMP  C0  CF              Dump array of RAM data memory values, in bytes.
```

>DUMP.W 400 47F	Dump ROM code from data memory hex addresses 400 to 47F in words.
>DUMP.B 300 400 8	Dump contents of data memory hex addresses 300 to 400 in rows of eight bytes.

11.20 DUMP_TRACE Command - Dump Trace Buffer

Dumps the current trace buffer to the debug window and to the capture file if enabled.

Syntax:

Dump_Trace

Example:

>Dump_Trace

11.21 EVAL Command- Evaluate Expression

The EVAL command evaluates a numerical term or simple expression, giving the result in hexadecimal, decimal, octal, and binary formats. In an expression, spaces must separate the operator from the numerical terms.

Note that octal numbers are not valid as parameter values. Operand values must be 16 bits or less. If the value is an ASCII character, this command also shows the ASCII character as well. The parameters for the command can be either just a number or a sequence of : number, space, operator, space, and number. Supported operations are addition (+), subtraction (-), multiplication (*), division (/), logical AND (&), and logical OR (^).

Syntax:

EVAL <n> [<op> <n>]

Where:

<n>	Alone, the numerical term to be evaluated. Otherwise either numerical term of a simple expression.
<op>	The arithmetic operator (+, -, *, /, &, or ^) of a simple expression to be evaluated.

Examples:

```
>EVAL 45 + 32
004DH  077T  000115O  0000000001001101Q  "w"
```

```
>EVAL 100T
0064H  100T  000144O  0000000001100100Q  "d"
```

EXECUTE_OPCODE Treats numeric parameter as an opcode and executes it.

11.22 EXECUTE_OPCODE Command - Treats Numeric Parameter As Opcode And Executes

Treats numeric parameter as an opcode and executes it.

Syntax:

EXECUTE_OPCODE <n>

Where:

<n> Numeric opcode.

Examples:

```
>EXECUTE_OPCODE $7C0007A4
; tlbwe
```

11.23 EXIT or QUIT Command - Exit Program

The EXIT command terminates the software and closes all windows. If the debugger is called from WINIDE it will return there. The QUIT command is identical to EXIT.

Syntax:

```
EXIT
```

Example:

```
>EXIT         Finish working with the program.
```

11.24 FPSCR Command - Floating Point Status And Control Register

The FPSCR command sets the condition register (CR) to the specified hexadecimal value.

Syntax:

```
FPSCR <n>
```

Where:

<n>   The new hexadecimal value for the CR.

Example:

```
> FPSCR $C4    Assign the value C4 to the CR.
```

11.25 FR(X) Command - Set Floating Point Register

The FR(x) command sets the value of the 32-bit Floating Point Register FR(x), where (x) is a value from 0 to 31.

Syntax:

```
FR(x) [n]
```

Where:

(x)   Value from 0 to 31, corresponding to which register the user intends to write.

[n]   Value to be written to register.

Example:

```
FR3 20.345234    Writes value of 20.345234 to Floating Point Register FR3.
```

11.26 G, GO/GOALL, or RUN Commands

The G or GO or RUN command starts execution of code in the Debugger at the current Program Counter (PC) address, or at an optional specified address. When only one address is entered, that address is the starting address. When a second address is entered, execution stops at that address. The G or GO or RUN commands are identical. When only one address is specified, execution continues until a key or mouse is pressed, a breakpoint set with a BR command occurs, or an error occurs.

The GOALL command performs the same action (G, GO, RUN) on each core of a multi-core device.

Syntax:

GO [<startaddr> [<endaddr>]]

Where:

<startaddr>	Optional execution starting address. If the command does not have a <startaddr> value, execution begins at the current PC value.
<endaddr>	Optional execution ending address.

Examples:

>GO	Begin code execution at the current PC value.
>GO 346	Begin code execution at hex address 346.
>G 400 471	Begin code execution at hex address 400. End code execution just before the instruction at hex address 471.
>RUN 400	Begin code execution at hex address 400.
>GOALL 400	Begin code execution at hex address 400 on each core of a multi-core device..

11.27 GOEXIT Command - Begin Program Execution W/O Breakpoints & Terminate Debugger

Similar to GO command except that the target is left running without any breakpoints and the debugger software is terminated.

Syntax:

GOEXIT [add]

Where:

add	Starting address of your code.

Example:

>GOEXIT 100	This will set the program counter to hex location 100, run the program and exit from the background debugging mode.

11.28 GOTIL Command - Execute Program until Address

The GOTIL command executes the program in the Debugger beginning at the address in the Program Counter (PC). Execution continues until the program counter contains the specified ending address or until a key or mouse is pressed, a breakpoint set with a BR command occurs, or an error occurs.

Syntax:

GOTIL <endaddr>

Where:

<endaddr>	The address at which execution stops.

Example:

>GOTIL 3F0	Executes the program in the Debugger up to hex address 3F0.

11.29 GOTILROM Command - Execute Fast Single Step

Executes fast single steps without updating the screen, until the address is reached. This is the fastest way to breakpoint in ROM.

Syntax:

GOTILROM [add]

Where:

add	Starting address of your code.

Example:

>GOTILROM 1000	This will do fast single steps from the location where your program counter is set at and stops at hex location 1000 which in this example is the starting location of the ROM. Starting location of the ROM depends on the memory map of your system. After reaching hex 1000 you can do single step to debug the code.

11.30 HELP Command - Open Help File

The HELP command opens the Windows help file for the program. If this command is entered with an optional parameter, help information specifically for that parameter appears. If this command is entered without any parameter value, the main contents for the help file appears.

An alternative way to open the help system is to press the F1 key.

Syntax:

HELP [<topic>]

Where:

<topic>	a debug command or assembly instruction

Examples:

>HELP	Open the help system

11.31 HGO Command - Begin Program Execution

The HGO command starts execution of code in the debugger at the current program counter (PC) address. Execution continues until a key or mouse is pressed, a breakpoint set with a BR command occurs, or an error occurs. If a key is pressed, real-time execution will stop and the debugger will stop the processor until it reaches the next source instruction.

Syntax:

HGO

Examples:

>HGO Step one source instruction.

11.32 HGOALL Command - Begin Program Execution (Multi-Core)

The HGOALL command is for Multi-Core devices It starts execution of code in the debugger on each core at the current program counter (PC) address. Execution continues until a key or mouse is pressed, a breakpoint set with a BR command occurs, or an error occurs. If a key is pressed, real-time execution will stop and the debugger will stop the processor until it reaches the next source instruction.

Syntax:

HGO (or HGOALL)

Examples:

> HGO Step one source instruction.

> HGOALL Step one source instruction on each core of a multi-core device.

11.33 HLOAD Command - Load ELF/DWARF Object

The HLOAD command allows the user to load an ELF/DWARF, S19, or PEmicro map file.

Syntax:

HLOAD [<filename>]

Where:

<filename> The name of the object or debug file to be loaded.

Examples:

> HLOAD MAIN.ELF Load the object and debug information in the ELF/DWARF file MAIN.ELF.

11.34 HLOADMAP Command - Load DWARF/MAP Debug Info

The HLOADMAP command loads a map file that contains source level debug information into the debugger. This command only loads debug info, it does not load object and debug info.

Syntax:

HLOADMAP [<filename>]

Where:

<filename> The name of a map file to be loaded. The .MAP extension can be omitted. The filename value can be a pathname that includes an asterisk (□) wildcard character□ If so, the command displays a lists of all files in the specified directory that have the .MAP extension.

Examples:

>HLOADMAP PROG.MAP   Load map file PROG.MAP into the host computer.

11.35 HSTEP/HSTEPALL - High-Level Language Source Step

The HSTEP command allows the user to step one high-level language source instruction. This is accomplished by rapidly single-stepping the processor on the assembly level.

The HSTEPALL command is for multi-core processors, and performs the same (HSTEP) action on each core of a multi-core device.

Syntax:

HSTEP (or HSTEPALL)

Examples:

> HSTEP   Step one source instruction.

> HSTEPALL   Step one source instruction on each core of a multi-core device.

11.36 HSTEPFOR - Step Forever (High-Level Language)

HSTEPFOR command continuously executes instructions, one at a time, beginning at the current Program Counter address until an error condition occurs, a breakpoint occurs, or a key or mouse is pressed. All windows are refreshed as each instruction is executed.

Syntax:

HSTEPFOR

Example:

>HSTEPFOR   Step through instructions continuously

11.37 LISTON Command - Show Info during Steps

The LISTON command turns on the screen listing of the step by step information during stepping. The register values and program instructions will be displayed in the status window while running the code. The values shown are the same values seen by the REG instruction.

To turn off this step display, use the LISTOFF command.

Syntax:

LISTON

Example:

>LISTON   Show step information.

11.38 LOAD Command (Legacy)

LOAD is a legacy command. Use the HLOAD command.

11.39 LOAD_BIN Command - Load A Binary File

Loads a binary file of bytes starting at address add. The default filename extension is .BIN.

Syntax:

LOAD_BIN [filename] [add]

Where:

filename	Name of the binary file
add	Starting address

Example:

>LOAD_BIN myfile.bin 100   Loads a binary myfile of bytes starting at hex address 100

11.40 LOADALL Command (Legacy)

LOADALL is a legacy command. Use the HLOAD command.

11.41 LOADV Command (Legacy)

LOADV is a legacy command. Use the HLOAD command.

11.42 LOADV_BIN Command - Load A Binary File & Verify

First performs the LOAD_BIN command and then does a verify using the same file.

Syntax:

LOADV_BIN [filename] [add]

Where:

filename	Name of the binary file
add	Starting address

Example:

>LOADV_BIN myfile.bin 100 Loads a binary myfile of bytes starting at hex address 100 and then does a verify using the same file.

11.43 LOADDESK Command - Load Desktop Settings

The LOADDESK command loads the desktop settings that set the window positions, size, and visibility. This allows the user to set how the windows are set up for the application. Use SAVEDESK to save the settings of the windows of the debugger into the desktop file.

Syntax:

LOADDESK

Example:

>LOADDESK Get window settings from desktop file.

11.44 LOADMAP Command (Legacy) - Load Map File

LOADMAP is a legacy command. Use HLOADMAP command.

11.45 LOADV Command (Legacy)

LOADV is a legacy command. Use HLOAD command.

LOAD_BIN Load a binary file of byte. The default filename extension is .BIN.

LOADV_BIN Perform LOAD_BIN command, verify using the same file.

11.46 LF or LOGFILE Command - Open / Close Log File

The LF command opens an external file to receive log entries of commands and copies of responses in the status window. If the specified file does not exist, this command creates the file. The LOGFILE command is identical to LF.

If the file already exists, an optional parameter can be used to specify whether to overwrite existing contents (R, the default) or to append the log entries (A). If this parameter is omitted, a prompt asks for this overwrite/append choice.

While logging remains in effect, any line that is appended to the command log window is also written to the log file. Logging continues until another LOGFILE or LF command is entered without any parameter values. This second command disables logging and closes the log file.

The command interpreter does not assume a filename extension.

Syntax:

```
LF [<filename>  [<R | A>]]
```

Where:

<filename> The filename of the log file (or logging device to which the log is written).

Examples:

>LF TEST.LOG R	Start logging. Overwrite file TEST.LOG (in the current directory) with all lines that appear in the status window.
>LF TEMP.LOG A	Start logging. Append to file TEMP.LOG (in the current directory) all lines that appear in the status window.
>LOGFILE	(If logging is enabled): Disable logging and close the log file.

11.47 LPTx

Specifies which PC compatible parallel port should be used for the debugger. The port must be fully PC compatible and a full 10-pin cable must be used.

Syntax:

```
LPTx
```

Where:

x	1, 2 or 3

Example:

LPT 2	Specifies that the debugger should use parallel port 2

11.48 LR Command - Link Register

The LR command sets the link register (LR) to the specified hexadecimal value.

Syntax:

LR <n>

Where:

<n> The new hexadecimal value for the LR.

Example:

>LR $C4 Assign the value C4 to the LR.

11.49 MACRO or SCRIPT - Execute a Batch File

The MACRO command executes a macro file, a file that contains a sequence of debug commands. Executing the macro file has the same effect as executing the individual commands, one after another. Entering this command without a filename value brings up a list of macro (.MAC) files in the current directory. A file can be selected for execution directly from this list. The SCRIPT command is identical.

Note: A macro file can contain the MACRO command; in this way, macro files can be nested as many as 16 levels deep. Also note that the most common use of the REM and WAIT commands is within macro files. The REM command displays comments while the macro file executes.

If a startup macro file is found in the directory, startup routines run the macro file each time the application is started. See STARTUP for more information.

Syntax:

MACRO <filename>

Where:

<filename> The name of a macro file to be executed, with or without extension .MAC. The filename can be a pathname that includes an asterisk(*) wildcard character. If so, the software displays a list of macro files, for selection.

Examples:

>MACRO INIT.MAC	Execute commands in file INIT.MAC.
>SCRIPT	Display names of all .MAC files (then execute the selected file).
>MACRO A:	Display names of all .MAC files in drive A (then execute the selected file).
>MACRO	Display names of all .MAC files in the current directory, then execute the selected file.

11.50 MACROEND Command - Stop Saving Commands to File

The MACROEND command closes the macro file in which the software has saved debug commands. (The MACROSTART command opened the macro file). This will stop saving debug commands to the macro file.

Syntax:

MACROEND

Example:

>MACROEND Stop saving debug commands to the macro file, then close the file.

11.51 MACROSTART - Save Debug Commands to File

The MACROSTART command opens a macro file and saves all subsequent debug commands to that file for later use. This file must be closed by the MACROEND command before the debugging session is ended.

Syntax:

MACROSTART [<filename>]

Where:

<filename> The name of the macro file to save commands. The .MAC extension can be omitted. The filename can be a pathname followed by the asterisk (□) wildcard character; if so, the command displays a list of all files in the specified directory that have the .MAC extension.

Example:

>MACROSTART TEST.MAC Save debug commands in macro file TEST.MAC

11.52 MACS Command - List Macros

Brings up a window with a list of macros. These are files with the extension .ICD (such as the STARTUP.ICD macro). Use the arrow keys and the <ENTER> key or mouse click to select. cancel with the <ESC> key.

Syntax:

MACS

Example:

>MACS Brings up a list of MACROS

11.53 MD Command - Set Memory Window 1 To Specific Address

The MD command displays (in the memory window) the contents of memory locations beginning at the specified address. The number of bytes shown depends on the size of the window and whether ASCII values are being shown. See Memory Window for more information. If a log file is open, this command also writes the first 16 bytes to the log file.

The MD and MD1 commands are identical.

Syntax:

MD <address>

Where:

<address>	The starting memory address for display in the upper left corner of the memory window.

Examples:

>MD 200	Display the contents of memory beginning at hex address 200.
>MD1 100	Display the contents of memory beginning at hex address 100.

11.54 MD2 Command - Set Memory Window 2 To Specific Address

The MD2 command displays the contents of 32 emulation memory locations in the second memory window. The specified address is the first of the 32 locations. If a logfile is open, this command also writes the first 16 values to the logfile.

Syntax:

MD2 <address>

Where:

<address>	The starting memory address for display in the memory window.

Example:

>MD2 1000	Display the contents of 32 bytes of memory in the second memory window, beginning at address 1000.

11.55 MM or MEM Command - Modify Memory

The MM command directly modifies the contents of memory beginning at the specified address. The optional variant specifies whether to fill the block in bytes (.B, the default), in words (.W), or in longs (.L).

If the command has only an address value, a Modify Memory window appears with the specified address and its present value and allows entry of a new value for that address. Also, buttons can be selected for modifying bytes (8 bit), words (16 bit), and longs (32 bit). If only that address is to be modified, enter the new value in the edit box and press the OK button. The new value will be placed at the location. If the user wishes to modify several locations at a time, enter the new value in the edit box and press the >> or << or = button. The new value will be placed at the specified address, and then the next address shown will be the current address incremented, decremented, or the same, depending on which button is pressed. To leave the memory modify window, either the OK or CANCEL buttons must be pressed.

If the MM command includes optional data value(s), the software assigns the value(s) to the specified address(es) (sequentially), then the command ends. No window will appear in this case.

Syntax:

MM [.B|.W|.L] <address>[<n> ...]

Where:

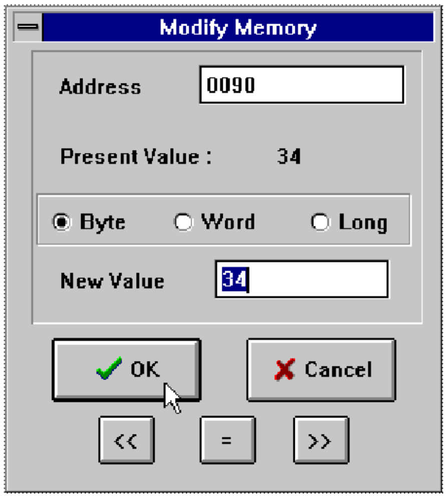
<address>	The address of the first memory location to be modified.
<n>	The value(s) to be stored (optional).

Examples:

With only an address:

>MM 90 Start memory modify at address $90.

Figure 11-3: Modify Memory Dialog



Modify Memory

Address 0090

Present Value : 34

● Byte ○ Word ○ Long

New Value 34

With a second parameter:

>MM 400 00 Do not show window, just assign value 00 to hex address 400.

>MM.L 200 123456 Place long hex value 123456 at hex address 200.

11.56 MSR Command - Machine Status Register

The MSR command sets the machine status register (MSR) to the specified hexadecimal value.

Syntax:

MSR <n>

Where:

<n> The new hexadecimal value for the MSR.

Example:

> MSR $C4 Assign the value C4 to the MSR.

11.57 NOBR Command - Clear All Breakpoints

Clears all break points.

Syntax:

NOBR

Example:

>NOBR Clears all break points.

11.58 PC Command - Program Counter

The PC command assigns the specified value to the program counter (PC). As the PC always points to the address of the next instruction to be executed, assigning a new PC value changes the flow of code execution.

An alternative way for setting the Program Counter if source code is showing in a code window is to position the cursor on a line of code, then press the right mouse button and select the Set PC at Cursor menu item. This assigns the address of that line to the PC.

Syntax:

PC <address>

Where:

<address> The new PC value.

Example:

>PC 0500 Sets the PC value to 0500.

11.59 QUIET Command

Turns off (or on) refresh of memory based windows. This command can be used on the startup command line. Default = on.

Syntax:

QUIET

Example:

>QUIET Turns off (or on) refresh of memory based windows

11.60 QUIT or EXIT Command - Quit Program

Quits the program. Identical to EXIT command.

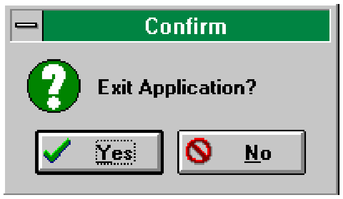
Syntax:

QUIT

Example:

>QUIT Exit the application

Figure 11-4: Confirm Exit Button



Confirm

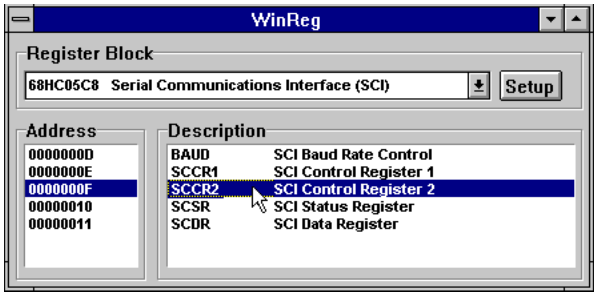
Exit Application?

11.61 R Command - Use Register Files

The R command open a processor's register files (sold separately by PEmicro) and starts interactive setup of such system registers as the I/O, timer and COP.

Entering this command opens the register files window, which initially shows a list of register files. Selecting a file brings up a display of values and significance for each bit of the register.

Figure 11-5: Register Display



WinReg

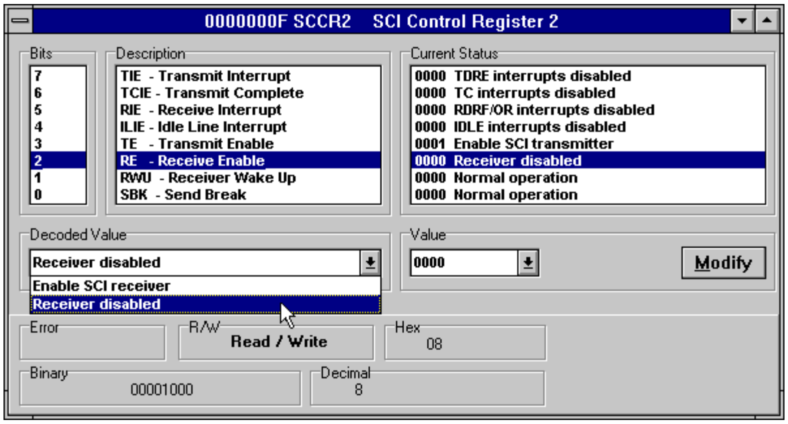
Register Block

68HC05C8 Serial Communications Interface (SCI)

Address	Description	
0000000D	BAUD	SCI Baud Rate Control
0000000E	SCCR1	SCI Control Register 1
0000000F	SCCR2	SCI Control Register 2
00000010	SCSR	SCI Status Register
00000011	SCDR	SCI Data Register

The user can view any of the registers, modify their values, and store the results back into Debugger memory. This is a good tool for gaining quick information on a register.

Figure 11-6: Modify Specific Register



0000000F SCCR2 SCI Control Register 2

Bits	Description	Current Status
7	TIE - Transmit Interrupt	0000 TDRE interrupts disabled
6	TCIE - Transmit Complete	0000 TC interrupts disabled
5	RIE - Receive Interrupt	0000 RDRF/OR interrupts disabled
4	ILIE - Idle Line Interrupt	0000 IDLE interrupts disabled
3	TE - Transmit Enable	0001 Enable SCI transmitter
2	RE - Receive Enable	0000 Receiver disabled
1	RWU - Receiver Wake Up	0000 Normal operation
0	SBK - Send Break	0000 Normal operation

Decoded Value: Receiver disabled

Enable SCI receiver

Receiver disabled

Value: 0000

Error:

R/W: Read / Write

Hex: 08

Binary: 00001000

Decimal: 8

An alternate way to bring up the register files window is to press the Register button.

Syntax:

R

Example:

>R Start interactive system register setup.

11.62 R(x) Command - Set R Register

The R(x) command sets the value of the 32-bit General Purpose Register R(x), where (x) is a value from 0 to 31. For targets that support the 64 bit SPE registers, use the H and L suffixes.

Syntax:

R(x)[H | L] [n]

Where:

(x) Value from 0 to 31, corresponding to which register the user intends to write.

[H | L] Indicates register with most significant bits H or least significant bits L. Default is L.

[n] Value to be written to register.

Example:

R3 $CF03D4AA	Writes value of $CF03D4AA to General Purpose Register R3.
R3H $CF03D4AA	Writes value of $CF03D4AA to General Purpose Register R3H.

11.63 REG or STATUS Command - Show Registers

The REG command displays the contents of the CPU registers in the status window. This is useful for logging CPU values while running macro files. The STATUS command is identical to the REG command.

Syntax:

REG

Example:

>REG   Displays the contents of the CPU registers.

11.64 REM Command - Place Comment in Macro File

The REM command allows a user to display comments in a macro file. When the macro file is executing, the comment appears in the status window. The text parameter does not need to be enclosed in quotes.

Syntax:

REM <text>

Where:

<text>   A comment to be displayed when a macro file is executing.

Example:

>REM Program executing   Display message "Program executing" during macro file execution.

11.65 RESET Command- Reset Emulation MCU

The RESET command forces a reset of the device into background mode and sets the program counter to the contents of the reset vector. This command does not start execution of user code.

Example:

>RESET   Reset the MCU into background mode.

Syntax:

RESET

11.66 RTVAR Command - Display Variable During Execution

The RTVAR command displays the specified address and its contents in the Variables Window for viewing during code execution and while the part is running (real time). Variants of the command display a byte, a word, a long, or a string. As the value at the address changes, the variables window updates the value. The maximum number of variables is 32. You may also enter the requisite information using the Add Variable box, which may be called up by double-clicking on the Variables Window or executing the RTVAR command without a parameter.

In the ASCII displays, a control character or other non-printing character is displayed as a period (.). The byte, word, long, or string variant determines the display format:

Byte (.B): hexadecimal (the default)

Word (.W): hexadecimal

Long (.L): hexadecimal

String (.S): ASCII characters

To change the format from the default of hexadecimal, use the Add Variable box.

The optional <n> parameter specifies the number of string characters to be displayed; the default value is one. The <n> parameter has no effect for byte, word, or long values.

Syntax:

```
RTVAR [.B|.W|.L|.S] <address> [<n>]
```

Where:

<address>	The address of the memory variable.
<n>	Optional number of characters for a string variable; default value is 1, does not apply to byte or word variables.

Examples:

>RTVAR	C0	Show byte value of address C0 (hex and binary)
>RTVAR.B	D4	Show byte value of address D4 (hex and binary)
>RTVAR.W	E0	Show word value of address E0 (hex & decimal)
>RTVAR.S	C0 5	Show the five-character ASCII string at hex address C0.

11.67 SAVEDESK Command - Save Desktop Settings

The SAVEDESK command saves the desktop settings for the application when it is first opened or for use with the LOADDESK command. The settings saved are window position, size, visibility, etc.

Syntax:

```
SAVEDESK
```

Example:

>SAVEDESK Save window settings for the application.

11.68 SERIAL Command - Set Up Parameters For Dumb Terminal

Sets up parameters for serial port. This port may then be attached to the Serial Port on your target for real-time debugging of communications software. See SERIALON command. COM1 or COM2, baud = 9600, 4800, 2400, 1200, 600, 300, 150 or 110, parity = N, E or O, data bits = 7 or 8, stop bits = 1 or 2. Example: SERIAL 1 9600 n 8 1

Syntax:

```
SERIAL (1 or 2) (baud) (parity) (data bits) (stop bits)
```

Where:

1 or 2	COM1 or COM2
baud	Baud rate ranging from 110 to 9600
parity	No, Even or Odd parity
data bits	7 or 8 data bits
stop bits	1 or 2 stop bits

Example:

>SERIAL 2 9600 E 8 2	Sets serial port to Com2 port with 9600 baud rate, even parity, 8 data bits and 2 stop bits

11.69 SERIALOFF Command - Disable Status Window As Dumb Terminal

Turns off serial port use during GO.

Syntax:

```
SERIALOFF
```

Example:

>SERIALOFF	Turns off serial port use during GO command

11.70 SERIALON Command - Enable Status Window As Dumb Terminal

Turns the communication window into a dumb terminal during a GO command using the serial port set up with the SERIAL command. To terminate the GO command from the keyboard, hit F1.

Syntax:

```
SERIALON
```

Example:

```
>SERIAL 2 9600 N 8 1
>SERIALON
>GO
```

11.71 SHOWCODE Command - Display Code at Address

The SHOWCODE command displays code in the code windows beginning at the specified address, without changing the value of the program counter (PC). The code window shows either source code or disassembly from the given address, depending on which mode is selected for the window. This command is useful for browsing through various modules in the program. To return to code where the PC is pointing, use the SHOWPC command.

Syntax:

```
SHOWCODE <address>
```

Where:

<address>	The address or label where code is to be shown.

Example:

>SHOWCODE 200	Show code starting at hex location 200.

11.72 SHOWMMU Command- Display MMU Information

Displays MMU information.

11.73 SHOWPC Command - Display Code at PC

The SHOWPC command displays code in the code window starting from the address in the program counter (PC). The code window shows either source code or disassembly from the given address, depending on which mode is selected for the window. This command is often useful immediately after the SHOWCODE command.

Syntax:

```
SHOWPC
```

Example:

>SHOWPC	Show code from the PC address value.

11.74 SHOWSPR Command - Display SPR Information

Displays SPR information.

11.75 SHOWTRACE Command - View Trace Buffer After TRACE Command

Example:

>SHOWTRACE	Displays the trace buffer logged during a previously executed TRACE command.

11.76 SNAPSHOT Command

Takes a snapshot (black and white) of the current screen and sends it to the capture file, if one exists. Can be used for test documentation and system testing.

Example:

>LOGFILE SNAPSHOT	This command will open a file by the name SNAPSHOT.LOG and store all the commands at the status window.
>SNAPSHOT	This command will take a snapshot of all the open windows of ICD and store it in SNAPSHOT.LOG file.
>LF	This command will close SNAPSHOT.LOG file

You can open the SNAPSHOT.LOG file with any text editor, such as EDIT.

11.77 SOURCEPATH Command - Search For Source Code

Either uses the specified filename or prompts the user for the path to search for source code that is not present in the current directory.

Syntax:

```
SOURCEPATH filename
```

Where:

filename Name of the source file

Example:

```
>SOURCEPATH d:\mysource\myfile.asm
```

11.78 SPR Command - Display/Modify Special Purpose Register

The SPR Command displays the value of the Special Purpose Register (x). The user can then enter a new value for the register or a simple carriage return to keep the same value. The addresses used are the same as for the MTSPR or MFSPR instructions. This is used to setup the LR, CTR, IMMR, and other special purpose registers.

Syntax:

```
SPR (x) [n]
```

Where:

(x) Value from 0 to 1023 corresponding to which register the user intends to write.

Note: The default debugger base is hexadecimal, so to force the register number to be base 10, add the character T as a suffix.

[n] Optional Value to be written to register.

Example:

SPR 638T Displays the IMMR special purpose register.

11.79 SS Command - Source Step

Does one step of source level code. Source must be showing in the code window.

Syntax:

```
SS
```

Example:

>SS Does one step of source level code.

11.80 ST, STEP/STEPALL or T Commands - Single Step

The ST or STEP or T command steps through one or a specified number of assembly instructions, beginning at the current Program Counter (PC) address value, and then halts. When the number argument is omitted, one instruction is executed. If you enter the ST command with an <n> value, the command steps through that many instructions.

The STEPALL command performs the same function (ST, STEP, T) on each core of a multi-core device.

Syntax:

```
STEP <n>
   or
```

ST <n>

or

T <n>

STEPALL <n>

Where:

<n>The hexadecimal number of instructions to be executed by each command.

Example:

>STEP	Execute the assembly instruction at the PC address value.
>ST 2	Execute two assembly instructions, starting at the PC address value.
>STEPALL 2	Execute two assembly instructions on each core of a multi-core device, starting at the PC address value.

11.81 STATUS or REG Command - Show Registers

The STATUS command displays the contents of the CPU registers in the status window. This is useful for logging CPU values while running macro files. The REG command is identical to the STATUS command.

Syntax:

STATUS

Example:

>STATUS Displays the contents of the CPU registers.

11.82 STEP Command - See ST, STEP/STEPALL, T

11.83 STEPALL Command - See ST, STEP/STEPALL, T

11.84 STEPFOR Command - Step Forever

STEPFOR command continuously executes instructions, one at a time, beginning at the current Program Counter address until an error condition occurs, a breakpoint occurs, or a key or mouse is pressed. All windows are refreshed as each instruction is executed.

Syntax:

STEPFOR

Example:

>STEPFOR Step through instructions continuously.

11.85 STEPTIL Command - Single Step to Address

The STEPTIL command continuously steps through instructions beginning at the current Program Counter (PC) address until the PC value reaches the specified address. Execution also stops if a key or mouse is pressed, a breakpoint set with a BR command occurs, or an error occurs.

Syntax:

STEPTIL <address>

Where:

<address> Execution stop address. This must be an instruction address.

Example:

>STEPTIL 0400 Execute instructions continuously until PC hex value is 0400.

11.86 SYMBOL Command - Add Symbol

The SYMBOL command creates a new symbol, which can be used anywhere in the debugger, in place of the symbol value. If this command is entered with no parameters, it will list the current user defined symbols. If parameters are specified, the SYMBOL command will create a new symbol.

The symbol label is case insensitive and has a maximum length of 16T. It can be used with the ASM and MM command, and replaces all addresses in the Code Window (when displaying disassembly) and Variables Window.

The command has the same effect as an EQU statement in the assembler.

Syntax:

SYMBOL [<label> <value>]

Where:

<label> The ASCII-character string label of the new symbol.

<value> The value of the new symbol (label).

Examples:

>SYMBOL Show the current user□defined symbols.

>SYMBOL timer_control $08 Define new symbol 'timer_control', with hex value 08. Subsequently, to modify hex location 08, enter the command 'MM timer_control'.

11.87 T Command - See ST, STEP/STEPALL, T

11.88 TIME Command - Displays Real Time Elapsed During Code Execution

Will give you an estimate of real time to execute the command from one address to another.

Set breakpoint at second address. Go from first address. If only one address given, it is the start address. If no stop address is given, the ICD will run forever or until a breakpoint is encountered or a key on the keyboard is hit. If no address is given the command is a "Time forever" command. When the command ends (either a break or a key) the debug window will show the amount of real-time that passed since the command was initiated.

Syntax:

TIME <[add1] [add2]>

Where:

add1	Starting address
add2	Ending address

Example:

>TIME 800 805 Will give you an estimate of real time to execute the command from hex location 800 to hex location 805.

11.89 _TR Command - Add Register Field Description to VAR Window

The _TR command is used to add a register field description to the VAR Window.

Syntax:

_TR

Example:

_TR This allows the user to add peripheral register fields to the variables window to monitor register settings.

11.90 TRACE Command - Monitors CPU Execution & Logs Instructions

The TRACE command is similar to the GO command except that execution does not occur in real-time. The ICD software monitors the execution of the CPU and logs the address of (up to) the last 256 instructions that have been executed into an internal array .

The trace executes from the first address until the breakpoint at the second address. If only one address given, it is the start address. If no stop address is given, the ICD will run forever or until a breakpoint is reached or a key on the keyboard is hit. If no address is given the command is a "Trace forever" command.

After execution, you may use the SHOWTRACE command or hit F7 to view the trace buffer.

Syntax:

TRACE <[add1] [add2]>

Where:

add1	Starting address
add2	Ending address

Example:

>TRACE 800 805 Will give you an estimate of real time to execute the command from hex location 800 to hex location 805.

11.91 UPLOAD_SREC Command - Upload S-Record to Screen

The UPLOAD_SREC command uploads the content of the specified program memory block (range), in .S19 object file format, displaying the contents in the status window. If a log file is opened, then UPLOAD_SREC will put the information into it as well. Same as P_UPLOAD_SREC.

Note: If the UPLOAD_SREC command is entered, sometimes the memory contents scroll through the debug window too rapidly to view. Accordingly, either the LOGFILE command should be used, which records the contents into a file, or use the scroll bars in the status window.

Syntax:

UPLOAD_SREC <startrange> <endrange>

Where:

<startrange> Beginning address of the memory block.

<endrange> Ending address of the memory block (range)

Example:

>UPLOAD_SREC 300 7FFUpload the 300□7FF memory block in .S19 format.

11.92 VAR Command - Display Variable

The VAR command displays the specified address and its contents in the Variables Window for viewing during code execution. Variants of the command display a byte, a word, a long, or a string. As the value at the address changes, the variables window updates the value. The maximum number of variables is 32. You may also enter the requisite information using the Add Variable box, which may be called up by double-clicking on the Variables Window or executing the VAR command without a parameter.

In the ASCII displays, a control character or other non-printing character is displayed as a period (.). The byte, word, long, or string variant determines the display format:

Byte (.B): hexadecimal (the default)

Word (.W): hexadecimal

Long (.L): hexadecimal

String (.S): ASCII characters

To change the format from the default of hexadecimal, use the Add Variable box.

The optional <n> parameter specifies the number of string characters to be displayed; the default value is one. The <n> parameter has no effect for byte, word, or long values.

Syntax:

VAR [.B|.W|.L|.S] <address> [<n>]

Where:

<address> The address of the memory variable.

<n> Optional number of characters for a string variable; default value is 1, does not apply to byte or word variables.

Examples:

>VAR C0 Show byte value of address C0 (hex and binary)

>VAR.B D4 Show byte value of address D4 (hex and binary)

>VAR.W E0 Show word value of address E0 (hex & decimal)

>VAR.S C0 5 Show the five-character ASCII string at hex address C0.

11.93 VERIFY Command - Compare Program Memory & S-Record

Compares the contents of program memory with an S-record file. You will be prompted for the name of the file. The comparisons will stop at the first location with a different value.

Syntax:

VERIFY

Example:

>LOADALL	test.s19
>VERIFY	As soon as you press <ENTER> key it will give you a message "Verifying...verified"

11.94 VERSION or VER - Display Software Version

The VERSION command displays the version and date of the software. VER is an alternate form of this command.

Syntax:

VERSION

Examples:

>VERSION	Display debugger version.
>VER	Display debugger version.

11.95 WATCHDOG Command - Disable Active Watchdog

Disables watchdog timer (toggles the state of the SWE bit in the SYPCR). Remember that this register may only be written once following a reset of the hardware. Reset enables the watchdog timer.

Syntax:

WATCHDOG

Example:

>WATCHDOG

11.96 WHEREIS Command- Display Symbol Value

The WHEREIS command displays the value of the specified symbol. Symbol names are defined through source code or the SYMBOL command.

Syntax:

WHEREIS <symbol> | <address>

Where:

<symbol> A symbol listed in the symbol table.

<address> Address for which a symbol is defined.

Examples:

>WHEREIS START	Display the symbol START and its value.
>WHEREIS 0300	Display the hex value 0300 and its symbol name if any.

11.97 XER Command - Integer Exception Register

The XER command sets the integer exception register (XER) to the specified hexadecimal value.

Syntax:

XER <n>

Where:

<n> The new hexadecimal value for the XER.

Example:

>XER $C4 Assign the value C4 to the XER.

12 Errors

Various errors may appear in the Status Window during your debugging. Most are self-explanatory. The following four errors are due to the debugger's interface with the background mode of the processor:

Warning Not ready response from chip.

Warning BERR Terminated bus cycle□□□Debugger Supplied DSACK

Warning Illegal command error from chip □□□ Debugger Supplied DSACK

All four errors are probably due to some type of problem accessing memory. The Debugger will fill most bad memory accesses with XX.

Note: The debugger rewrites the windows showing on the screen often. If a window is showing memory that does not exist, one of these errors will occur every time the debugger tries to update that window. This concerns the two memory windows and the code window. Additionally, reading or writing non-existing memory areas mapped internally may cause one of these errors.

In most cases the debugger will try to recover. If the system starts acting erratic after this message, it may be due to a fatal memory error and you may have to reset the system.

13 Variable Length Encoding (VLE)

The ICD disassembly windows display both the traditional PowerPC (BookE) and VLE sets of target instructions. While single stepping the target, ICD's disassembly windows will automatically update to display the appropriate instruction set.

14 Register Display

Are you tired of looking through manual pages for register descriptions? PEmicro has the ultimate solution for this problem.

The R command lets you look at and modify the registers and the register fields in both a symbolic and numeric format. When you type the R command, it searches the directory which contains the ICD program for any files with the .REG extension (REGister files). Each of these files describes a block of registers. The block name (1st line of .reg) file is displayed in a pick window. If you select a register block, each of the registers in that block has its address, name, and description listed in a pick window. If you select a register, its current value is read from the device and displayed both symbolically and numerically. You may then edit the register contents.

If you modified the displayed register contents and exit the register display, you are asked if you want to write the new value back to the device register. You should be careful of any "WRITE ONLY" registers, since they can not be read but will be written.

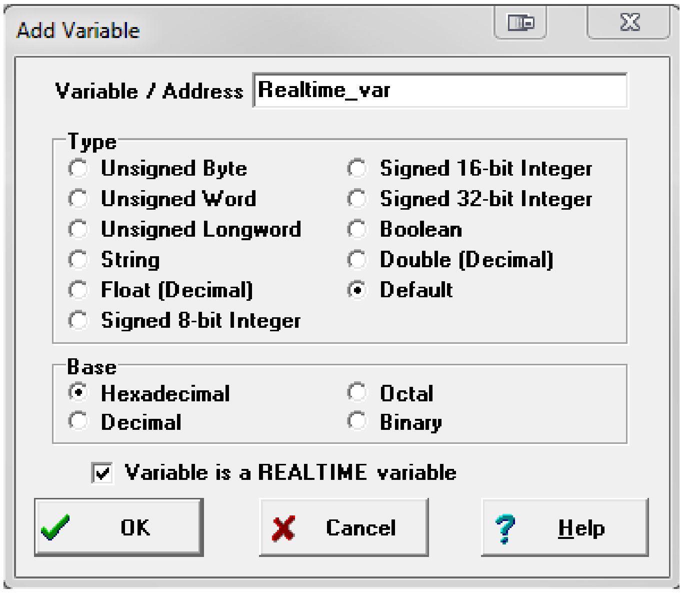
The .REG files for most Motorola modules are available at nominal cost from PEmicro

Microcomputer Systems. These files are in ASCII and it is possible to create your own files for devices not supported by PEmicro.

15 'Add Variable' Box

The Add Variable box allows the user to display a variable specified by label or memory address, and to choose the format and base of the data that is displayed. The Add Variable box pops up when the user double-clicks the Variables Window.

Figure 15-1: Add Variable Box



Add Variable

Variable / Address: Realtime_var

Type

○ Unsigned Byte
○ Unsigned Word
○ Unsigned Longword
○ String
○ Float (Decimal)
○ Signed 8-bit Integer
○ Signed 16-bit Integer
○ Signed 32-bit Integer
○ Boolean
○ Double (Decimal)
● Default

Base

● Hexadecimal
○ Decimal
○ Octal
○ Binary

☑ Variable is a REALTIME variable

Use the Variable / Address input field to enter the label or memory location for the data you wish to be displayed.

Below, select from the Type and Base buttons to specify in which format and base you would like the data to be displayed.

16 ELF/DWARF SUPPORT

The In-Circuit Debugger supports loading files of type ELF/DWARF, which provides C source-level debugging. The ICD will process files that adhere to the following specifications:

Executable and Linking Format (ELF)

System V Application Binary Interface, edition 4.1

Debug With Arbitrary Record Format (DWARF)

DWARF Debugging Information Format Revision 2.0/3.0

Note that the debugger will load DWARF version 2.0/3.0 information only. No other DWARF version is compatible with the ICD.

Some processors have ELF/DWARF supplemental specifications in addition to the general specifications above. The ICD adheres to all available processor supplements.

16.1 SUPPORTED FEATURES

· Source-Level Code View

· Disassembly-Level Code View

· Single-Stepping Source

· Running and Stopping Source

· Loading Object Data to MCU Memory:

· Base Type Variable View

· Array Variable View

· Structure/Union Variable View

· Enumerated Type Variable View

· Typedef Variable View

· Global Variable View

· Location Relative (e.g. Stack Relative) Variable View

· Register Based Variable View

· Variable Scoping:

· Auto-typing of Variables in VAR Window

· Modifying Variables

Currently, the ICD does not support C macro information.

16.2 GETTING STARTED

Issue the BETA command in the debugger to enable the ELF/DWARF capabilities.

The HLOAD command loads ELF/DWARF source files.

The HLOADMAP command loads DWARF debugging information only. No executable object code is loaded.

The HSTEP command, or SS, single-steps one high-level source line. HSTEPALL performs this same action (HSTEP) on each core of a multi-core device.

The HSTEPFOR command continually steps high-level instructions until the user aborts by pressing a key.

The HGO command starts full-speed execution and attempts to stop on a high level instruction when aborted by the user. HGOALL performs this same action (HGO) on each core of a multi-core device.

The HSTEP/HSTEPALL, HSTEPFOR, and HGO/HGOALL commands are identical to the STEP/STEPALL, STEPFOR, and GO/GOALL commands with the following exceptions:

(1) While assembly instructions are executed between the high-level language lines, the Variable, Memory, and Source Code windows are not updated. The Disassembly and CPU windows are updated for every low-level instruction. At the next high-level instruction boundary, all windows are updated (unless in GO mode).

(2) When the user aborts the current execution command, the debugger executes up to 20 steps while trying to find the next high-level language instruction boundary. The debugger will attempt to show source automatically. If, in 20 steps, it can not find the boundary of a high-level source code instruction, it will stop and show disassembly. To see source again, use the HSTEP/HTEPALL command or right click on the Source Code window and select Show Source Module.

After loading the code, depending upon the software application, you may have to set the program counter (PC or IP) to the reset vector of your code. There is not usually high-level source code at this location. Instead, there is often compiler code used to initialize your variables, heap, and so forth. The reset code should call the "main" function. Use the command "GOTIL main" to execute code to the beginning of the "main" function. At this point, you should see source code. It is not correct to set the PC directly to the main routine, as this would skip the compiler's initialization.

16.3 DISPLAYING VARIABLES

The debugger Variable window will show global and static variables as well as location relative variables. A location relative variable is typically a local variables on the application stack.

However, the compiler may indicate other variable types as location relative and may use many different location schemes for variables. Some variables may change location depending on the value of the PC. The ICD supports all of the DWARF 2.0 location possibilities.

The debugger will show variables whose location is a register. Compilers will often store temporary variables in CPU registers of the processor. If you attempt to look at the address of a register variable by adding the symbol &I (where I is the variable) to the Variables window, the debugger should indicate the register in which the variable is stored.

The ICD supports scoping of variables. If you put a variable name in the Variables window, the debugger will show the variable of the same name that is currently in scope. If you had a global integer variable, temp, and a local float variable, temp, within the routine init_port, the float would be shown while you step through the init_port routine. Otherwise, the Variable window would display the integer variable. When a variable value equates to [Not Accessible], this means the variable specified is either out of scope or doesn't exist.

The following symbols may be added to a variable name in the Variables window. Note that pointer variables are displayed in red.

& dereference

* reference

. access to union or structure member

-> pointer access to union or structure member

[ ] array subscript

For example:

```
int TintGlobal;
int *ptrTint;
int TmultiArray[3][3][3];

struct Tstruct {
  int ii;
  int jj;
  short ll;
} TstructInstance;

union Tunion {
  unsigned long TuLongUnion;
  struct Tstruct TstructUnion
} TunionInstance;

union Tunion *TunionPointer;
```

ICD commands:

```
var TintGlobal
```

The value of TintGlobal

var &TintGlobal

The address of TintGlobal

var TmultiArray[0][1][2]

The value of this element of the array

var TstructInstance.ii

The value of this member of the structure

var TunionPointer->TuLongUnion

The value of this union member, the union pointed to by TunionPointer

16.4 Examining A Variable

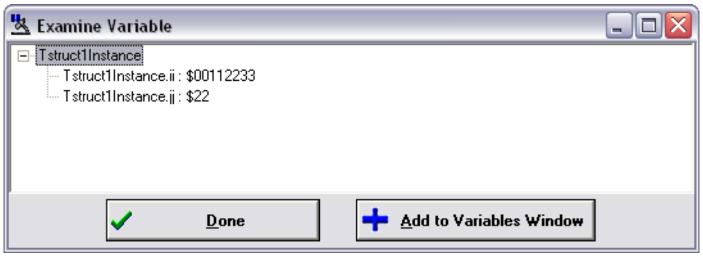
You may view the entire contents of a structure or array variable by double-clicking the entry in the Variables Window. Alternatively, right-click on an entry in the Variables Window and select "Examine Variable" in its popup menu.

The Examine Variable dialog is available for ELF/DWARF2.0 debugging for the following variable types:

Structures	- Displays the members of the structure.
Arrays	- Displays elements of non-dynamic arrays.
Pointer	- Displays dereferenced pointer to non-dynamic variable.
Other types	- Dialog is not available.

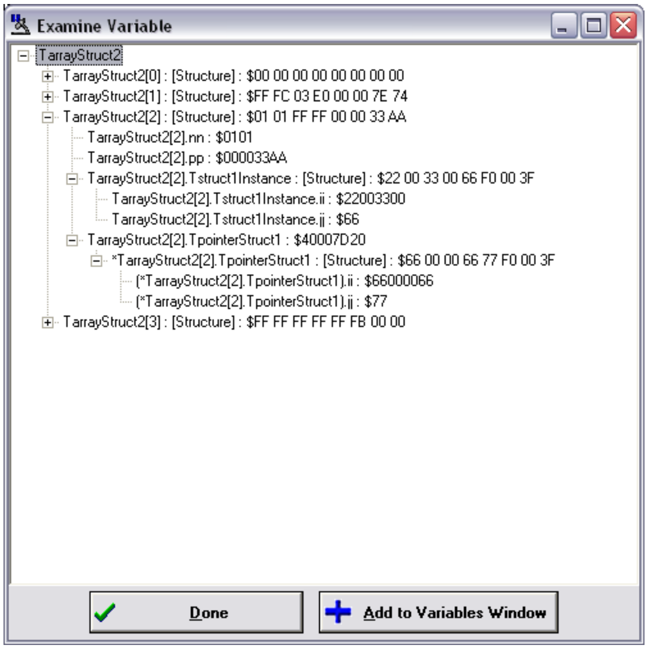
In the Examine Variable window, click the plus sign to expand the variable.

Figure 16-1: Simple Structure



```
⊟ Tstruct1Instance
    Tstruct1Instance.ii : $00112233
    Tstruct1Instance.jj : $22
```

Figure 16-2: More Complex Structure



```
⊟ TarrayStruct2
    ⊞ TarrayStruct2[0] : [Structure] : $00 00 00 00 00 00 00 00
    ⊞ TarrayStruct2[1] : [Structure] : $FF FC 03 E0 00 00 7E 74
    ⊟ TarrayStruct2[2] : [Structure] : $01 01 FF FF 00 00 33 AA
        TarrayStruct2[2].nn : $0101
        TarrayStruct2[2].pp : $000033AA
        ⊟ TarrayStruct2[2].Tstruct1Instance : [Structure] : $22 00 33 00 66 F0 00 3F
            TarrayStruct2[2].Tstruct1Instance.ii : $22003300
            TarrayStruct2[2].Tstruct1Instance.jj : $66
        ⊟ TarrayStruct2[2].TpointerStruct1 : $40007D20
            ⊟ *TarrayStruct2[2].TpointerStruct1 : [Structure] : $66 00 00 66 77 F0 00 3F
                (*TarrayStruct2[2].TpointerStruct1).ii : $66000066
                (*TarrayStruct2[2].TpointerStruct1).jj : $77
    ⊞ TarrayStruct2[3] : [Structure] : $FF FF FF FF FF FB 00 00
```

Add an array element or structure member to the Variables Window by double-clicking on the entry, or use the "Add to Variables Window" button.

16.5 ELF PROGRAM HEADERS

Program headers, included in every executable ELF/DWARF file, describe how the application object code is to be loaded to the target. Two values in each program header entry, defined by the System V ABI as p_paddr and p_vaddr, are available to provide a load address for a particular group of code. For executable files, the p_vaddr field is typically used to provide the load address. However, some compilers, such as the GNU compiler, may utilize the p_paddr field instead.

When the PEmicro debugger loads the ELF/DWARF file, it may detect the use of the non-standard p_paddr field in the ELF program header. In this instance, the debugger will display a dialog box that will ask the user what to do. Generally, when using the GNU tools, click "Yes" in the dialog box to load code using the non-standard p_paddr field.

For a complete description of ELF Program Headers, see the System V ABI (ELF) specification.

16.6 LOADING AN ELF/DWARF 2.0/3.0 APPLICATION (HLOAD/HLOADMAP)

The Elf/Dwarf 2.0 file contains two types of information:

(1) Elf Binary Image : This contains all the instructions and data which make up the application which will run on the target microprocessor. This part of the image will eventually reside in the flash memory of the target, but during debug may also be loaded into the RAM of the target.

(2) Dwarf Debug Information : The debug information is used by the debugger to allow the user to debug the binary image. This contains source file line number information, variable names, variable addresses, and so forth. This information is loaded into the debugger only and is not presented to the target microcontroller.

The two most common configurations for loading an Elf/Dwarf 2.0 file are:

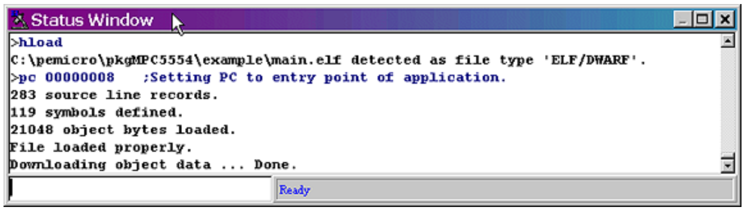
(1) Binary image to be loaded into FLASH : If the binary image is to be loaded into flash, it must be done prior to entering ICD which does not program flash. PEmicro's PROG flash programmer may be used to program the image into flash. Upon entering the ICD, with the binary image already resident in flash, the user would use the HLOADMAP command to load the debug information portion of the Elf/Dwarf file into the debugger.

(2) Binary image needs to be loaded into RAM : If the binary image of the application is to be loaded into RAM on the target, the HLOAD command is used. The HLOAD command loads both the binary image into the target microprocessor's RAM as well as loads the dwarf debug

information into the debugger. Before loading the binary image, the user should make sure that the RAM is turned on at the proper address.

The STATUS window will display the amount of debug and object information loaded from the Elf/Dwarf file, in a manner similar to the following window:

Figure 16-3: Status Window



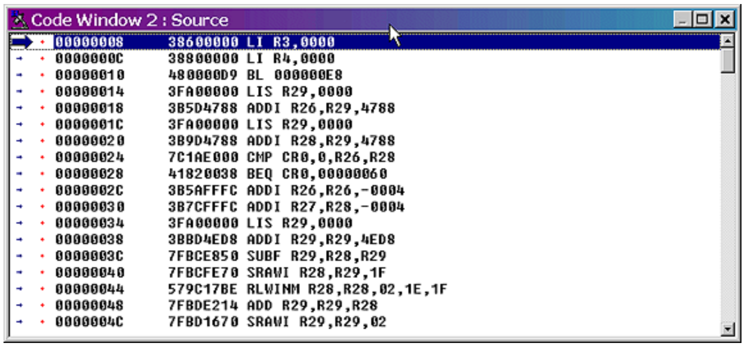
```
>hload
C:\pemicro\pkgMPC5554\example\main.elf detected as file type 'ELF/DWARF'.
>pc 00000008    ;Setting PC to entry point of application.
283 source line records.
119 symbols defined.
21048 object bytes loaded.
File loaded properly.
Downloading object data ... Done.
```

By default, when an Elf/Dwarf object is loaded, the program counter (PC) is set to point to the start of the demonstration application code.

16.7 RUNNING UNTIL THE START OF SOURCE CODE

After loading an Elf/Dwarf file, it may be that the code window still points to disassembly and does not initially show the loaded applications source code:

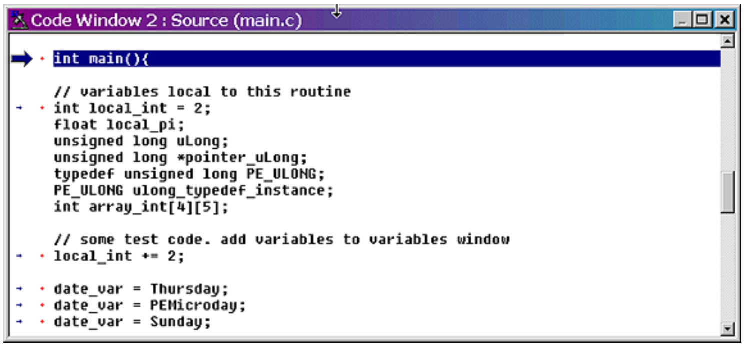
Figure 16-4: Code Window: Source



```
00000008    38600000 LI R3,0000
0000000C    38800000 LI R4,0000
00000010    480000D9 BL  000000E8
00000014    3FA00000 LIS R29,0000
00000018    3B5D4788 ADDI R26,R29,4788
0000001C    3FA00000 LIS R29,0000
00000020    3B9D4788 ADDI R28,R29,4788
00000024    7C1AE000 CMP CR0,0,R26,R28
00000028    41820038 BEQ CR0,00000060
0000002C    3B5AFFFC ADDI R26,R26,-0004
00000030    3B7CFFFC ADDI R27,R28,-0004
00000034    3FA00000 LIS R29,0000
00000038    3BBD4ED8 ADDI R29,R29,4ED8
0000003C    7FBCE850 SUBF R29,R28,R29
00000040    7FBCFE70 SRAWI R28,R29,1F
00000044    579C17BE RLWINM R28,R28,02,1E,1F
00000048    7FBDE214 ADD R29,R29,R28
0000004C    7FBD1670 SRAWI R29,R29,02
```

This is because, before running the user’s main() function, the compiler must first execute some initialization code which does not have corresponding debug information. To run past the compiler initialization code, issue the “gotil main” command in the status window. Note that the labels are case sensitive and that the main label should be all lowercase. This will set a breakpoint at the beginning of the main() routine in main.c and start the processor running. Execution should stop almost immediately and the PC should be pointing to valid source code. This source code will appear in the source code window, similar to the following:

Figure 16-5: Code Window: Source (main.c)



```
int main(){

// variables local to this routine
int local_int = 2;
float local_pi;
unsigned long uLong;
unsigned long *pointer_uLong;
typedef unsigned long PE_ULONG;
PE_ULONG ulong_typedef_instance;
int array_int[4][5];

// some test code. add variables to variables window
local_int += 2;

date_var = Thursday;
date_var = PEMicroday;
date_var = Sunday;
```

Also, instead of using the GOTIL command, the user could have stepped through the initialization code (using the STEP or HSTEP commands) and would have eventually reached the main() function.

16.8 STEPPING THROUGH C LEVEL INSTRUCTIONS

The PEmicro debugger implements a high-level language source step command, which may be executed by using the HSTEP command in the status window or by clicking the high-level step button on the debugger button bar. Each time the high-level language source is stepped, the debugger will rapidly single step assembly level instructions until the next source instruction is encountered, at which point execution will cease. While the debugger is fast single-stepping, the only on-screen value which will be updated is the PC (by default). When the debugger reaches the next source instruction, all visible windows will be updated with data read from the target. Note that using the HSTEP command does not run code in real-time. Real-time execution is described in the next section.

The user should step several source-level instructions as this point. Note that some instructions will take longer to step than others, because each C level instruction may consist of a greater or fewer number of underlying assembly instructions than others.