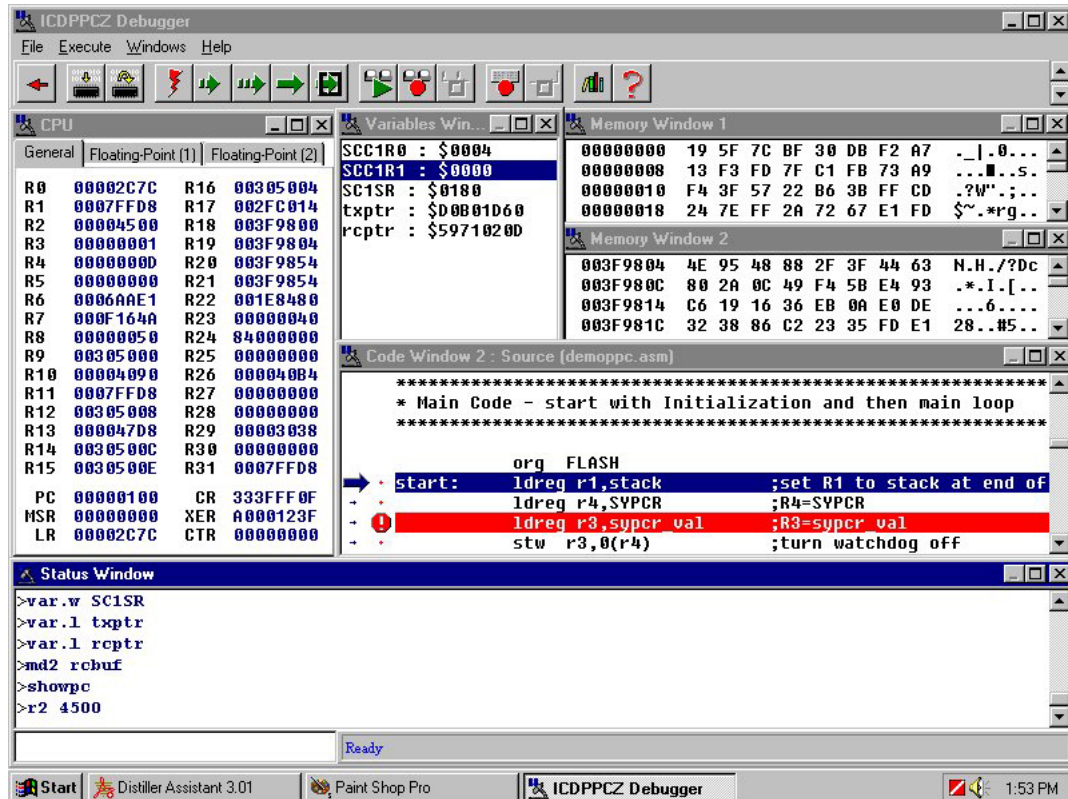


# ICDPPCZ User Guide

## 1 Introduction

PEmicro's ICDPPCZ for Windows is a powerful tool for debugging code on an NXP MPC5xx/8xx processor. The debugger uses the processor's JTAG debug mode, via a hardware interface, to give the user access to all on-chip resources.

Figure 1-1: ICDPPCZ Debugger



### 1.1 ICDPPCZ Features At A Glance

- Full-speed in-circuit debug
- Breakpoints with counters on the Nth execution
- Variables window showing bytes, words, strings, and long words
- Startup and Macro files for automating the debug process
- Symbolic register files to allow decoding of on-chip peripheral registers
- Full assembly source-level debugging
- C source-level debugging based on ELF/DWARF 2.0 format

## 2 Command Line Parameters

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

### Syntax:

ICDPPCZ\_PRO [option] ... [option]

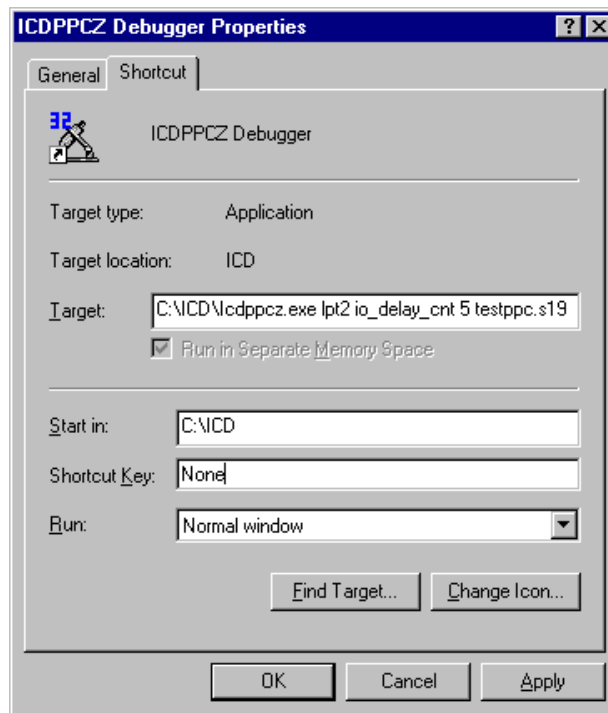
[option]	Optional parameters are as follows:
lpt1...lpt3	Chooses lpt1, lpt2, or lpt3. The software will remember the last setting used.
pci1...pci6	Chooses which PCI card to communicate with. The software will remember the last setting used.
pci_delay n	Sets speed of PCI card shift clock, where n = 0...255. The equation for the PCI card shift clock frequency is $33 * 10^6 / (5 + 2n)$ .
running	Starts ICD with CPU running (see RUNNING help)
io_delay_cnt n	Causes the background debug mode clock to be extended by 'n' Cycles where $1 \leq n \leq 64k$ . Used when using a very fast PC or a slow CPU clock (default = 1);
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 than one option is given, they must be separated by spaces.

**Examples:**

ICDPPCZ\_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**



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.

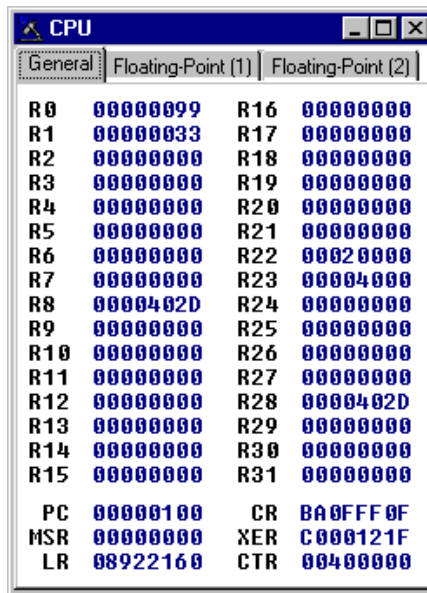
### 4 User Interface

- 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.

Figure 4-1: CPU Window



The CPU Window has three tabs for three different sets of registers: General, Floating-Point 1, and Floating-Point 2. The General Register screen shows all general-purpose registers along with the PC, CR, MSR, XER, LR, and CTR registers. The Floating-Point 1 Register screen shows floating point registers FR0-FR15 and the FPSCR register. The Floating-Point 2 Register screen shows floating point registers FR16-FR31. 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

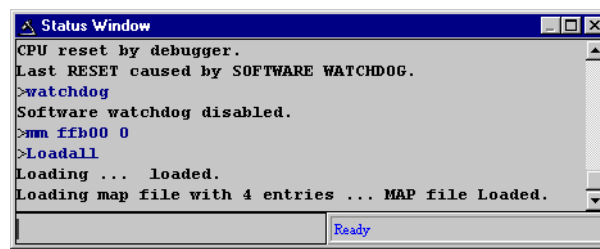
## 4.2 Status Window

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 4-2: ICDPPCZ Status Window**



**POPUK 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:

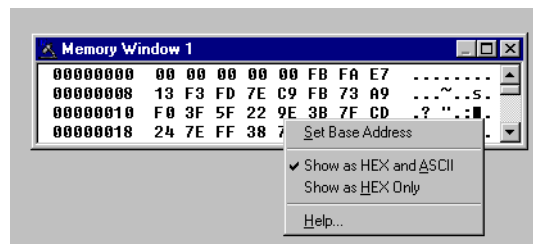
- UP ARROW            Scroll window up one line
- DOWN ARROW        Scroll window down one line
- HOME                Scroll window to first status line
- END                 Scroll window to last status line
- PAGE UP             Scroll window up one page
- PAGE DOWN         Scroll window down one page
- 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.

**Figure 4-3: Memory Window 1**



**POPUK 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**

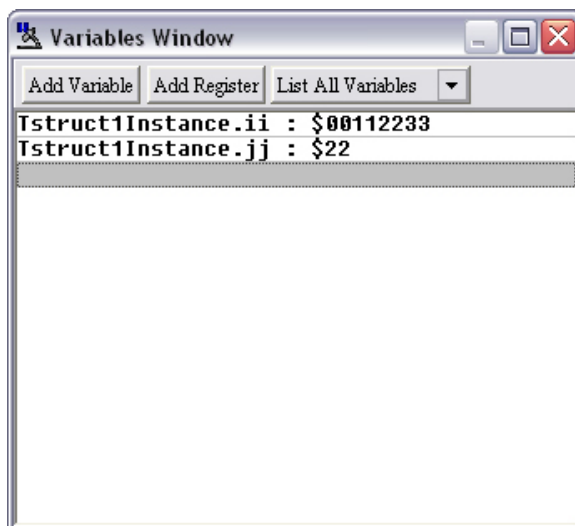
The following keystrokes are valid while the memory window is the active window:

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**

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 4-4: Variables Window**



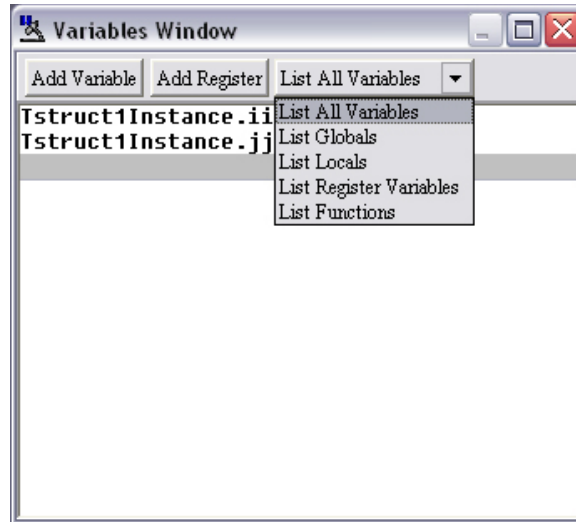
Variables that are pointers are displayed in red. Normal variables are displayed in black. Real-time variables are displayed in blue. A real-time variable is a variable that is updated even while the processor is running.

**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 4-5: Variables Drop-Down Menu**



The following dialogs are available:

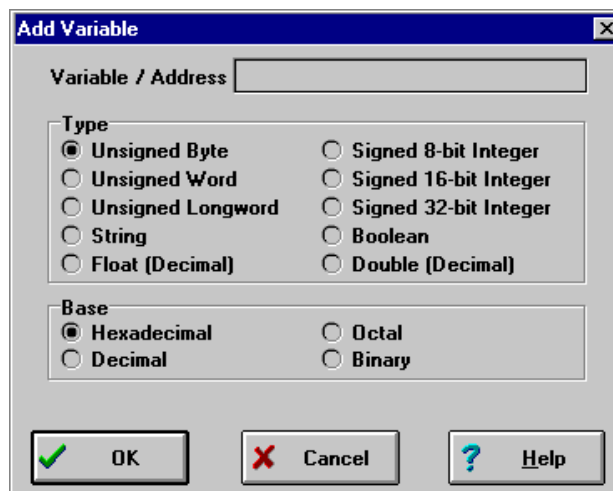
**List All Variables**

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

**'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 4-1: Add Variable Box**



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.

**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.

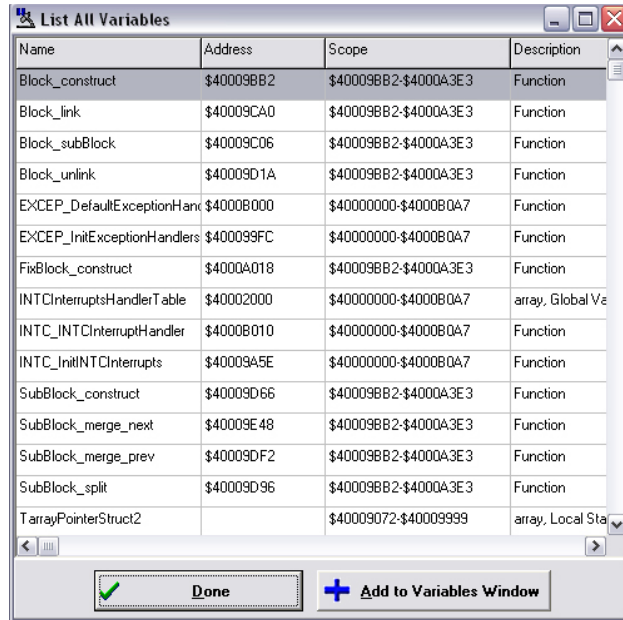
### List Register Variables

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 4-2: List Functions Window



#### 4.4.1 Using The Variables Window

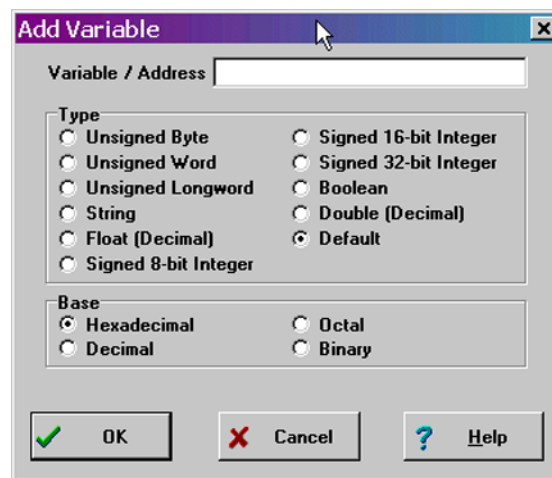
- Adding And Deleting Variables
- Modifying A Variable's Value
- Modifying A Variable's Properties

##### 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 4-1: Add/Delete Variable**

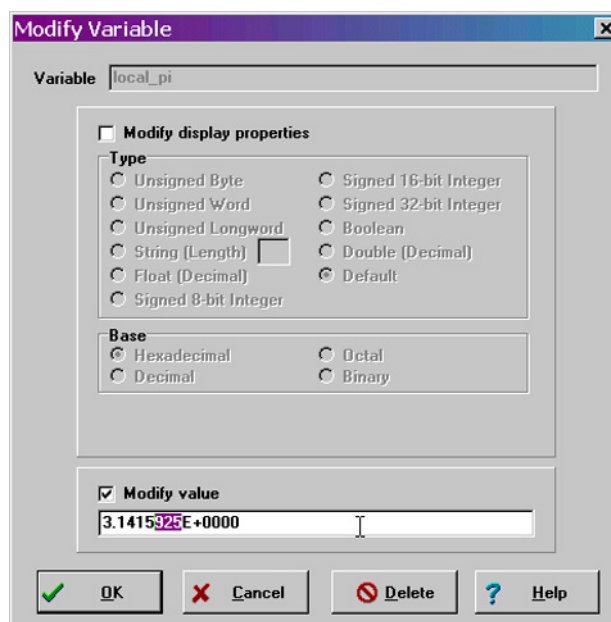


In the variable field, the user should input the address or name of the variable that they would like displayed in the variables window. The type of the variable should most often be set to “Default,” which means that the variable will be shown as it is defined in the compiled/loaded application. When 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 4-2: Modify Value**

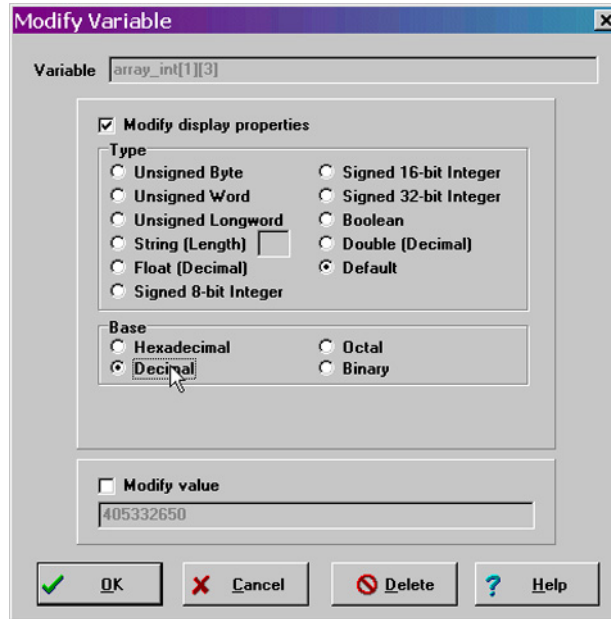


#### 4.4.1.3 Modifying A Variable's Properties

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 4-3: Modify Properties**

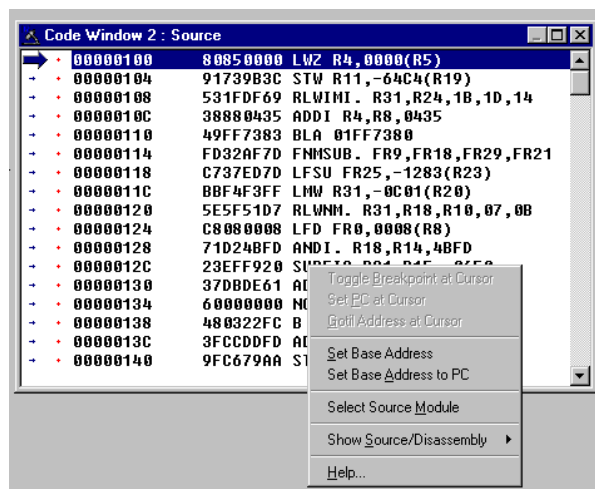


#### 4.5 Code Window

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 4-4: Code Window**



#### 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

#### 4.5.1 Using The Code Window

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

##### 4.5.1.1 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 4-1: 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 4-2: Source Level Instruction**

```

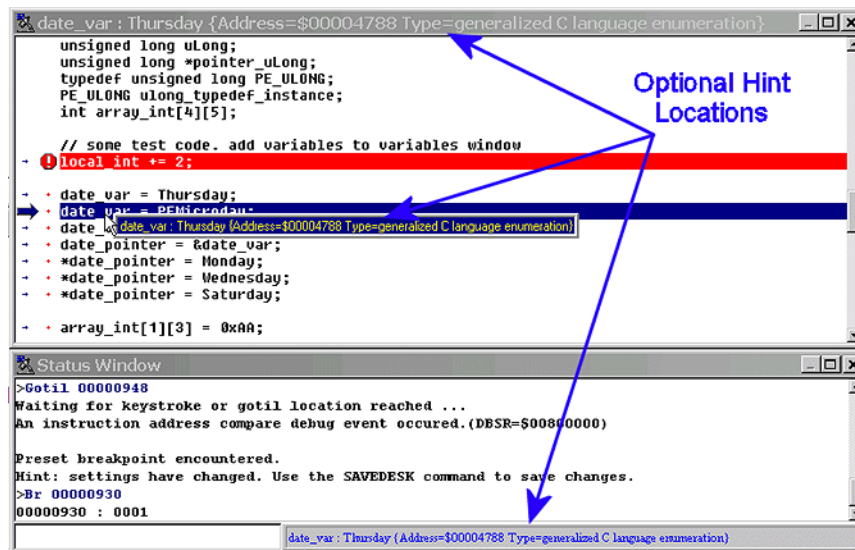
➔ • 0000093C 3D200000 LIS R9,0000
- • 00000940 38000005 LI R0,0005
- • 00000944 90094788 STW R0,4788(R9)
☐ • 00000948 3D200000 LIS R9,0000

```

##### 4.5.1.2 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:

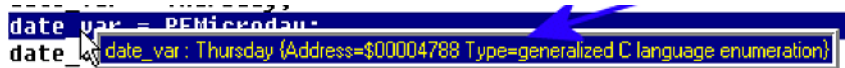
Figure 4-1: Hint Locations



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 4-2: Hint Box



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

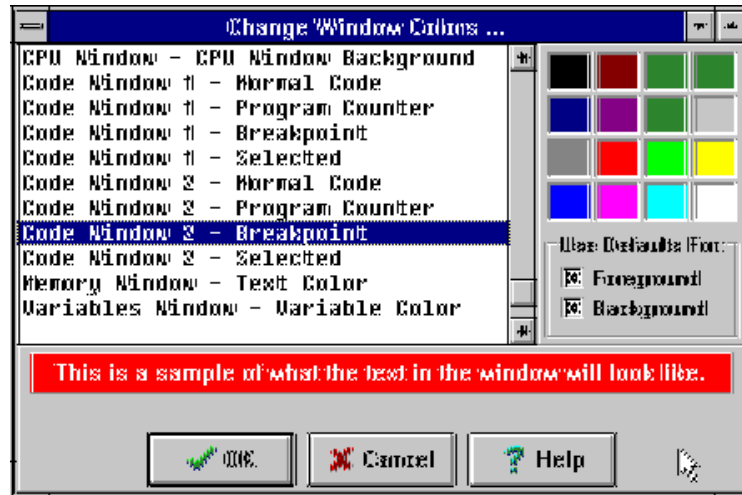
#### 4.6 Colors Window

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 4-3: Colors Window**



## 5 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.

## 6 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



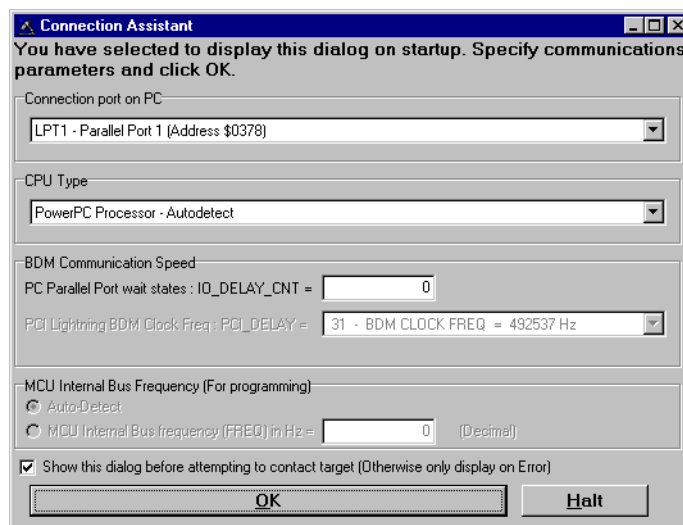
R0H ... R31H  
 PC  
 CR  
 MSR  
 XER  
 LR  
 CTR

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

## 7 Connection To Target

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

**Figure 7-1: Connection Assistant Dialog**



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.

## 8 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.

## 9 Commands

### 9.1 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.

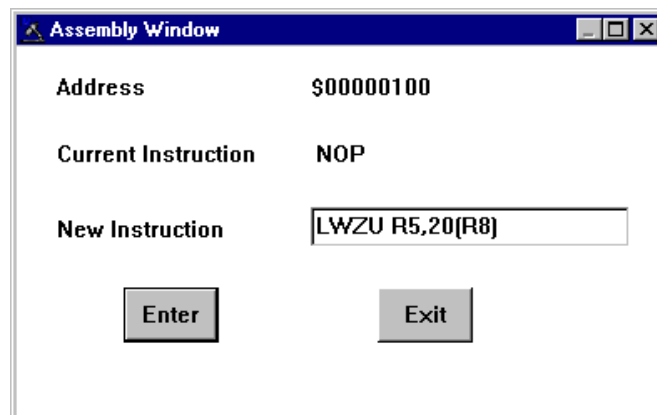
#### Examples:

With an address argument:

```
>ASM 100
```

The following window appears:

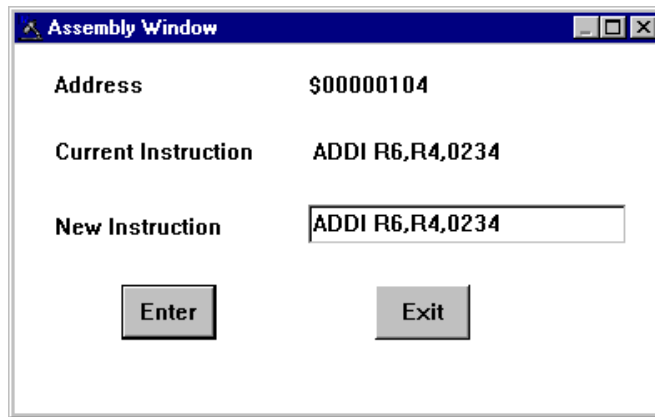
**Figure 9-1: Connection Assistant Dialog**



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 9-2: Connection Assistant Dialog**



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.

## 9.2 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 C0 CF AA	Store hex value AA in bytes at addresses C0-CF.
>FILL.B C0 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

### 9.3 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.

### 9.4 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.

### 9.5 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.

## 9.6 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.

## 9.7 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.

## 9.8 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.

### 9.9 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

### 9.10 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.

### 9.11 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.

### 9.12      **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.

### 9.13      **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.

### 9.14      **G, GO, 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.

**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.

**9.15 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.

**9.16 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.

**9.17 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.

**9.18 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.

**9.19 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.

**9.20 HSTEP - 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.

**Syntax:**

HSTEP

**Examples:**

> HSTEP Step one source instruction.

**9.21 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

**9.22 LOAD Command (Legacy)**

LOAD is a legacy command. Use the HLOAD command.

**9.23 LOADALL Command (Legacy)**

LOADALL is a legacy command. Use the HLOAD command.

**9.24 LOADV Command (Legacy)**

LOADV is a legacy command. Use the HLOAD command.

**9.25 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.

**9.26 LOADMAP Command (Legacy) - Load Map File**

LOADMAP is a legacy command. Use HLOADMAP command.

**9.27 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.

**9.28 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.

## 9.29 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.

## 9.30 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.

**9.31      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.

**9.32      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

**9.33      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

### 9.34 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.

### 9.35 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.

### 9.36 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:**

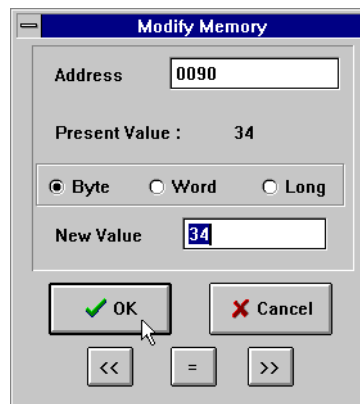
<address>        The address of the first memory location to be modified.  
 <n>                The value(s) to be stored (optional).

**Examples:**

With only an address:

>MM 90            Start memory modify at address \$90.

**Figure 9-3: Connection Assistant Dialog**



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.

**9.37 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.

### 9.38 NOBR Command - Clear All Breakpoints

Clears all break points.

**Syntax:**

NOBR

**Example:**

>NOBR Clears all break points.

### 9.39 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.

### 9.40 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

### 9.41 QUIT or EXIT Command - Quit Program

Quits the program. Identical to EXIT command.

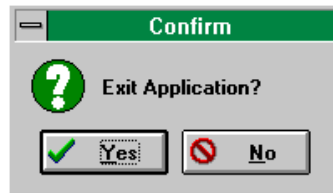
**Syntax:**

QUIT

**Example:**

>QUIT Exit the application

**Figure 9-4: Confirm Exit Button**

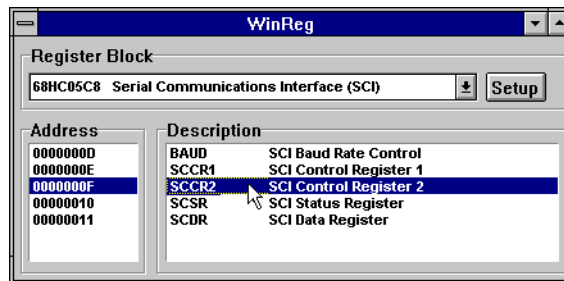


**9.42 R Command - Use Register Files**

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.

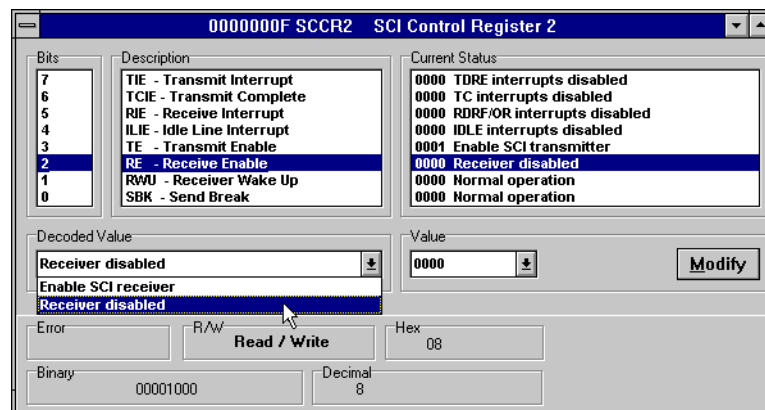
Entering this command opens the register files window, which initially shows a list of register files. Selecting a file brings up a display of values and significance for each bit of the register.

**Figure 9-5: Register Display**



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 9-6: Modify Specific Register**



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.

**9.43 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.

**9.44 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.

**9.45 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.

#### 9.46 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

#### 9.47 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

#### 9.48 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

#### 9.49 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
```

## 9.50 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.
```

## 9.51 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.
```

## 9.52 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.

### 9.53 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

### 9.54 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.

### 9.55 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.

### 9.56 ST, STEP 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.

**Syntax:**

STEP <n>

or

ST <n>

or

T <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.

### 9.57 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.

### 9.58 STEP Command - See ST, STEP

### 9.59 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.

### 9.60 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.

## 9.61 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'.

## 9.62 T Command - See ST, STEP/STEPALL, T

## 9.63 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.

## 9.64 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.

## 9.65 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.

## 9.66 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.

## 9.67 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"



## 9.68 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.

## 9.69 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

## 9.70 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.

## 9.71 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.

## 10 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.

## 11 Register Display

Are you tired of looking through manual pages for register descriptions? PE micro 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 NXP modules are available at nominal cost from PE micro. These files are in ASCII and it is possible to create your own files for devices not supported by PE micro.

## 12 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 2.0.0

Note that the debugger will load DWARF version 2.0 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.

## 12.1 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.

## 12.2 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.

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.

The HSTEP, HSTEPFOR, and HGO commands are identical to the STEP, STEPFOR, and GO 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.

## 12.3 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

## 12.4 Examining A Variable

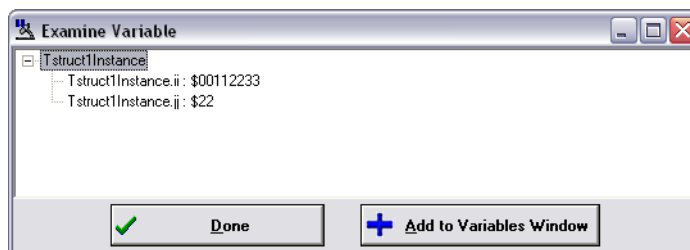
You may view the entire contents of a structure or array variable by double-clicking the entry in the Variables Window. Alternatively, right-click on an entry in the Variables Window and select "Examine Variable" in its popup menu.

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

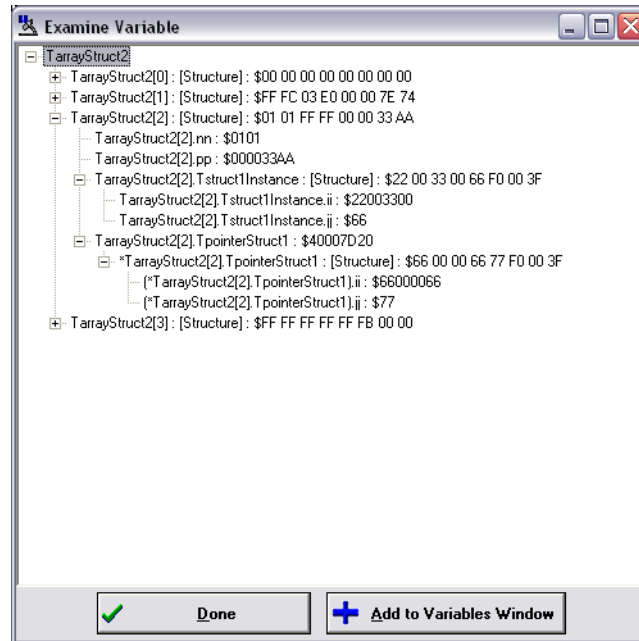
- Structures - Displays the members of the structure.
- Arrays - Displays elements of non-dynamic arrays.
- Pointer - Displays dereferenced pointer to non-dynamic variable.
- Other types - Dialog is not available.

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

**Figure 12-1: Simple Structure**



**Figure 12-2: More Complex Structure**



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.

## 12.5 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 PE micro 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.

## 12.6 LOADING AN ELF/DWARF 2.0 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:

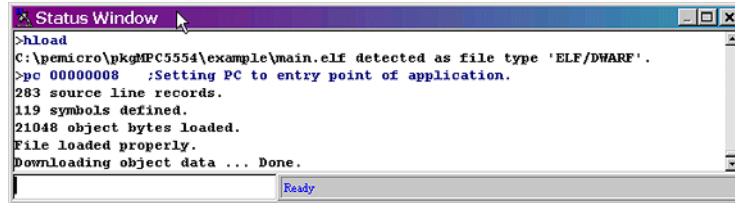
- (1) Binary image to be loaded into FLASH : If the binary image is to be loaded into flash, it must be done prior to entering ICD which does not program flash. PE micro's PROG flash programmer may be used to program the image into flash. Upon entering the ICD, with the binary image already resident 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 12-3: Status Window

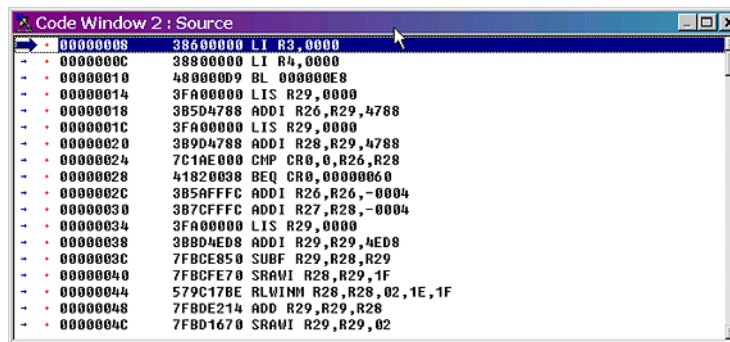


By default, when an Elf/Dwarf object is loaded, the program counter (PC) is set to point to the start of the demonstration application code.

## 12.7 RUNNING UNTIL THE START OF SOURCE CODE

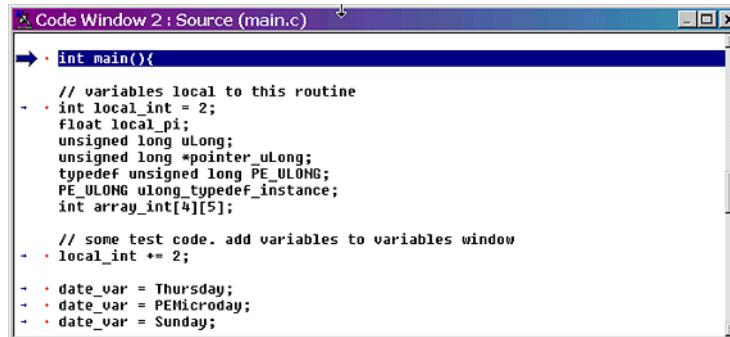
After loading an Elf/Dwarf file, it may be that the code window still points to disassembly and does not initially show the loaded applications source code:

Figure 12-4: Code Window: Source




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 12-5: Code Window: Source (main.c)



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.

## 12.8 STEPPING THROUGH C LEVEL INSTRUCTIONS

The PE micro 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 MPC5554 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 at 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.