EDB

User's Manual

Embedded Performance, Inc.

P/N 0380-0162-10 Rev 05

May, 2002

EPI has made every attempt to ensure that the information in this document is accurate and complete. However, EPI assumes no responsibility for any errors, omissions, or for any consequences resulting from the use of the information included herein or the equipment it accompanies. EPI reserves the right to make changes in its products and specifications at any time without notice.

Embedded Performance, Incorporated
606 Valley Way
Milpitas, California 95035
USA
(408) 957-0350

Acknowledgments:

CDB is a trademark of Third Eye Software.
MS DOS and Windows are trademarks of Microsoft Corporation.
Ethernet is a trademark of XEROX.
EDB and EPI are trademarks of Embedded Performance, Inc.
All other trademarks are trademarks of their respective companies.

Contents

About this Manual

This is the user manual for the (EDB). The information in this manual is intended to serve both new and experienced users. It outlines the installation, configuration, and operation of the EDB.

Version

This manual covers version 2.3 and later of EDB.

Notational Conventions in this Manual

The following conventions are used in the syntax descriptions of this manual.

`Bold face` Bold is used for characters that must be entered exactly as shown.

`Italic` is used to indicate a general category of input that will be described in detail in the command operands section. Italic is also used when a new term or concept is first introduced, and for the title of other documents.

`monospaced` A non-proportional type face is used for the names of registers, processor and emulator signals, and configuration options.

sans serif A sans serif type face is used for the names of windows, dialog boxes, menus, and fields.

`<key>` Angle brackets indicate that the item enclosed within the brackets is the name of a special key on the host's keyboard. Some examples are <enter>, <backspace>, and <esc>. In some cases several keys must be held down simultaneously, which is indicated with a dash between them. For example, <`ctrl-C`>.

[ ] Square brackets are used to enclose an optional operand or group of operands. The brackets are not to be entered in the command.

{ } Curly braces are used for grouping purposes. These are not to be entered in the command. They either enclose a list of alternatives, one of which must be chosen, or they enclose a group of operands that are to be taken together in the context of a list of alternatives or a subsequent repetition.

... An ellipses (three dots in succession) is used to indicate that the preceding operand, or group of operands if enclosed by [ ] or { }, may optionally be repeated one or more times.

| A vertical bar is used to indicate that the operand, or group of operands if enclosed by [ ] or { }, on either side of the bar may be entered, but not both.

.. Two dots in succession indicate the inclusion of sequential items between given start and stop points. For example, a..z refers to the entire alphabet including a and z.

Symbols Used in this Manual

This manual uses the following symbols to help identify notes, cautions, and warnings:

Cautions or Warnings: !

Notes: i

Support

Before contacting EPI for EDB support please check the Frequently Asked Questions (FAQ) list to see if your question has already been answered. Select the FAQ option from our Web site:

http://www.epitools.com

If you still need help and cannot find the information via the FAQ or online documentation, you can either send email to EPI at the Internet address below, call us at 408 957-0350, orFAX us at 408 957-0307.

support@epitools.com

Be sure to include your name, the version number of the product in question, your target processor type and vehicle type (ICE or SIM), your Internet address and phone number with all correspondence.

NOTE: The best way to show us your product information is to capture the screen output of the debuggers `DI` command and send it in an email or FAX.

If your question is sales-related please do not send email to our support department. Send it to the Internet address below:

sales@epitools.com

Documentation Feedback

If you have comments on how to improve EDB's documentation, send us e-mail at this Internet address:

edb@epitools.com

Be sure to include your name, the version number of EDB (in the About box), your target processor type, target vehicle type (ICE or SIM), your Internet address, and phone number with all correspondence.

You can also use this address for EDB product suggestions.

NOTE: Sorry, we cannot answer technical support questions from this address. Please send all support-related email to: support@epitools.com

We would especially like to get your thoughts in these areas:

- What important information is missing or hard to find?
- Does the Contents section help you locate the information you need? What groups of topics do you recommend that we add?
- Do you have suggestions for improvements?

Your comments will become the property of Embedded Performance Inc.

Introducing EDB 1

EDB is the generic name for EPI's source level debugger. The debugger can be used to control and debug programs running on a number of target system types supported by EPI.

This chapter provides an overview of EDB and describes the main features of EDB's Graphical User Interface. This chapter includes the following sections:

EDB History

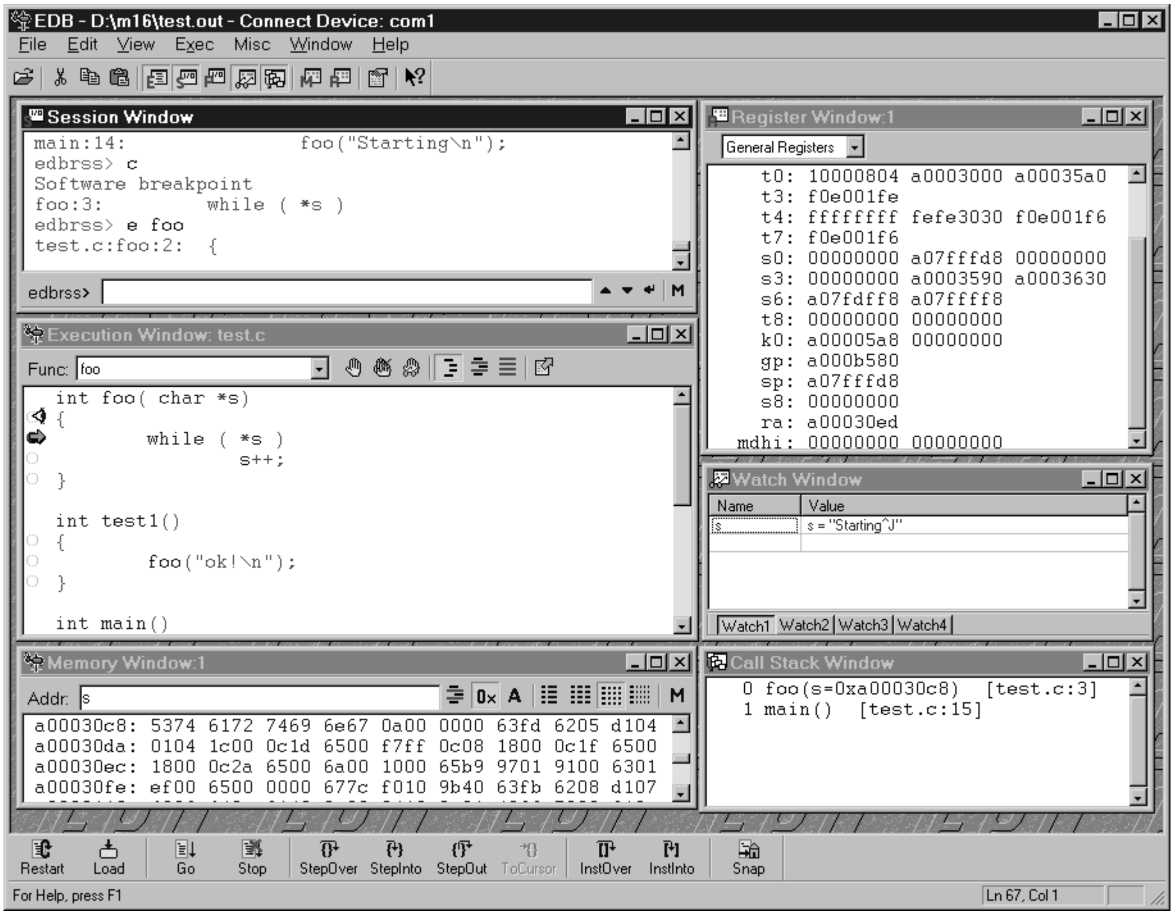
EPI's source level debuggers are derived from CDB, a widely available debugger originally developed by Third Eye Software. EPI has extensively modified CDB to support remote debugging of embedded systems. EDB is EPI's windowing interface and is built on top of both CDB and EPI's low level debugger MON. The command line user interfaces for both of these debuggers is available from inside EDB via the Session Window.

Depending on the options ordered, EDB may be used to debug programs which are running on the EPI Instruction Set Simulator on a host computer, or programs running remotely on an actual embedded hardware system via either EPI's remote debug kernel or an In-circuit Emulator.

The EDB Graphical User Interface

EDB provides standard Windows menus, toolbars, tool tips, and a status bar. The status bar provides expanded information about menus and toolbar buttons. Hovering the mouse cursor over a toolbar button also provides a few extra words about the underlying icon or object. This is known as a *tooltip*.

The following figure shows the EDB Graphical User Interface with a number of windows open in the workspace.



```
main:14:                foo("Starting\n");
edbrss> c
Software breakpoint
foo:3:          while ( *s )
edbrss> e foo
test.c:foo:2:   {
```

edbrss>

Func: foo

```
int foo( char *s)
{
        while ( *s )
                s++;
}

int test1()
{
        foo("ok!\n");
}

int main()
```

General Registers

```
  t0: 10000804 a0003000 a00035a0
  t3: f0e001fe
  t4: ffffffff fefe3030 f0e001f6
  t7: f0e001f6
  s0: 00000000 a07fffd8 00000000
  s3: 00000000 a0003590 a0003630
  s6: a07fdff8 a07ffff8
  t8: 00000000 00000000
  k0: a00005a8 00000000
  gp: a000b580
  sp: a07fffd8
  s8: 00000000
  ra: a00030ed
mdhi: 00000000 00000000
```

Name	Value
s	s = "Starting^J"

Watch1 Watch2 Watch3 Watch4

Addr: s

```
a00030c8: 5374 6172 7469 6e67 0a00 0000 63fd 6205 d104
a00030da: 0104 1c00 0c1d 6500 f7ff 0c08 1800 0c1f 6500
a00030ec: 1800 0c2a 6500 6a00 1000 65b9 9701 9100 6301
a00030fe: ef00 6500 0000 677c f010 9b40 63fb 6208 d107
```

```
0 foo(s=0xa00030c8)   [test.c:3]
1 main()   [test.c:15]
```

Title Bar

The Title bar is located along the top of a window. It contains the name of the application and document. To move the window, you can drag the Title bar. You can also move dialog boxes by dragging their Title bars.

The Title bar contains the common Windows elements (such as minimize button) as well as the application and document names, and the connection device (for ICE targets).

Scroll Bars

Displayed at the right and bottom edges of most windows. The scroll boxes inside the scroll bars indicate your vertical and horizontal location in the document. You can use the mouse to scroll to other parts of the document.

Toolbars

EDB provides several toolbars: a general toolbar, a toolbar to control the execution of your program (Execution toolbar), and a toolbar to change the processor context displayed (Context toolbar). By default the Execution toolbar displays a button label below the icon. These labels can be turned off or on via a checkbox in the Properties dialog. A smaller toolbar can be useful when you are short on screen real-estate. Also, note that each of these toolbars can be dragged around on the screen and docked to the sides of the workspace. The EDB toolbars are described in *General Toolbar Command* on page 79, *Execution Toolbar Command* on page 80, and *Context Toolbar Command* on page 81.

EDB Windows

EDB provides a number of windows to help with debugging your application. The details about these Windows and their usage are provided in *Navigating your Application* on page 10 and *Examining your Application Data* on page 12. Here we address some generic usage issues applicable to all EDB windows.

Window Size and Positioning

Every EDB window can be positioned and resized within the EDB desktop window. This positioning and other attributes can be saved across debug sessions via the Save Session Info item in the File menu, or by selecting the Save Layout at Exit and Restore Layout at Startup options in the Properties dialog.

NOTE: If you select the Save Session Info item in the File menu, you should turn off the Save Layout at Exit option in the Properties dialog. This way each time EDB is started it has the same session layout.

Window Toolbars

Most of the EDB windows contain toolbars inside the window. These toolbars provide buttons and fields which operate on the window itself. Some of these windows position their toolbar at the top of the window, others at the bottom. Any toolbar can be turned off via the window’s short-cut menu (see *Window Short-Cut Menus* below).

Window Short-Cut Menus

Every EDB window has a short-cut menu that can be accessed by clicking the right mouse button. Mostly the items in a short-cut menu mirror operations available via the toolbar. However, the Short-Cut menu can be a superset of the toolbar operations. Every Short-Cut menu also includes a window toolbar toggle and a Properties item that brings up the Properties dialog.

NOTE: Although the short-cut menu is normally brought up via a right mouse click, it can be called up via combining the <Shift-F10> keys. This may vary with the international version of Windows, so you may need to consult your Windows manual for details on bringing up a short-cut menu via the keyboard.

Starting EDB

EDB is a complex debug tool and supports many startup options and associated startup files. The details of these options can be found in *Invoking EDB* on page 24. The files are described in *EDB Startup Files* on page 5. Here we just touch on some of the basics.

EDB can be invoked either from a Windows icon or the command line. Typically, starting EDB from one the icon's setup by the installer will not work without first adjusting the program arguments to fit your environment. EDB comes in three different executable flavors. Each flavor is designed to execute with a different target environment. Depending upon your order you may only have a subset of these available to you:

Program name:	Target system type:
edbice	The EPI ICE version, HP-Probe. Supports all EPI ICE and some foreign ICEs.
edbsim	The EPI Instruction Set Simulator version.

NOTE: Throughout this manual, the debugger will be referred to as EDB. The only difference between the various versions is that the emulator-targeted versions provide access to the emulator's trace features.

Startup Arguments

Only a few arguments are needed to get started with EDB. Below are abbreviated descriptions of these arguments. Full descriptions can be found in *Invoking EDB* on page 24.

-v*cpu* cpu is a number specifying the target CPU type. Valid values for cpu are listed in your MON Assembly level debugger manual. Typically you can just specify your chip number: For example: **-v5000**.

-d *device_name* Specifies the name of the communications channel to be used to connect to a remote target. This option is not applicable with the Instruction Set Simulator. Typically this is the name of a serial device or an Ethernet host name. Ethernet host names must be followed by **:e** with one exception (if the ICE being connected to is an HP Software Probe, the Ethernet name must be followed by **:h**). The default device name is COM1.

NOTE: Using Ethernet for communication requires setting up IP and hostname addresses in your network.

-*n* If using an RS-232 serial channel for communication to an ICE, this argument specifies the speed of the channel. It does not apply to the simulator (EDBSIM). *n* is a digit in the range **0..7**, specifying a baud rate of 1200, 2400, 4800, 9600, 19,200, 38,400, 57,600, or 115,200. Not all hosts will be able to support the highest speeds. Your maximum speed may be limited for Remote Server Targets by the speed of the target system's serial chip.

EDB Invocation Examples

The following examples demonstrate invoking EDB from a DOS command window for various environments. Note that your window's icon can be modified to add any of these parameters.

Using an EPI ICE, connecting to it using COM2 (RS-232 serial channel) at 115.2K baud, and target contains an NEC 4300 or derivative:

```
edbice -t com2 -v4300 -7
```

Using an EPI ICE, connecting to it using Ethernet with IP address 115.1.2.234, and target contains an IDT 4700:

```
edbice -t 115.1.2.234:e -v4700
```

Using EPI's Instruction Set Simulator and the NEC 5400:

```
edbsim -v5400
```

EDB Startup Files

EDB has several startup files of significance to most users. They are listed below in the order of there usage (loading) by EDB. A complete startup file list and details can be found in sections under *EDB Files* on page 17.

cdb.rc This file is a simple EDB command file (that is, a text file of EDB commands) that is always loaded by EDB at startup. EPI supplies a default **cdb.rc** file in the same directory as the EDB executables.

start*XXX*.cmd Although this file is not a built-in EDB startup file, it is loaded by the default **cdb.rc** file shipped with EDB. *XXX* corresponds to the ICE (**ice**) and simulator (**sim**) flavors of EDB. This file typically contains startup configuration logic and is auto-loaded by EPI’s low level debugger MON. Many times a default **start*XXX*.cmd** file can be found that is already setup for use with a particular chip / target evaluation board. Because of this pre-customization work, users are encouraged to add new configuration / setup commands to the **ustrt*XXX*.cmd** file which is auto-loaded by all the sample **start*XXX*.cmd** files.

ustrt*XXX*.cmd This file is also not a built-in EDB startup file. It is loaded by the default **start*XXX*.cmd** file shipped with EDB. *XXX* corresponds to the ICE (**ice**) and simulator (**sim**) flavors of EDB. This file is typically empty of commands and is intended to be a convenient place for users to add new configurations or startup commands.

NOTE: Although EDB can be started via an icon and appears not to require a PATH variable, it may not be able to find all it's startup files unless yourPATH variable is configured to point at the EPI tools bin directory.

Application Program Startup File

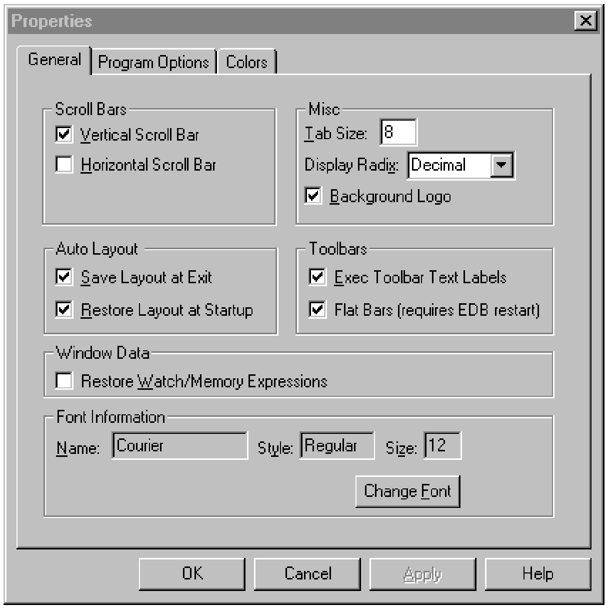
Whenever a new application program is opened in EDB (at startup, or by selecting Program to Debug from the File menu), EDB looks for a command file with the same base name (and in the same directory) as the program being debugged, plus an extension of **.rc**. Note that when you stop debugging an application and exit, EDB auto-creates this file and uses it to save breakpoints and other miscellaneous options considered specific to an application program.

Configuring EDB Options

EDB has many options and many ways to configure them. Here we touch on the basics of configuring these options via the various dialogs. Many options can also be configured via MON’s **EO** / **DO** commands. See *Enter Configuration Option* on page 66 and *Display Configuration Options* on page 65 for more details.

Properties Dialog

This dialog is brought up by selecting Properties from the Edit menu or from any Window Short-Cut menu. The dialog contains three tabs to access different option classes.



Scroll Bars
☑ Vertical Scroll Bar
☐ Horizontal Scroll Bar

Misc
Tab Size: 8
Display Radix: Decimal
☑ Background Logo

Auto Layout
☑ Save Layout at Exit
☑ Restore Layout at Startup

Toolbars
☑ Exec Toolbar Text Labels
☑ Flat Bars (requires EDB restart)

Window Data
☐ Restore Watch/Memory Expressions

Font Information
Name: Courier Style: Regular Size: 12

The General tab allows configuration of screen related data such as font size, scroll bars, etc. Configuring the font size to match the smallest size you can comfortably read will maximize usage of screen real-estate. Not turning on horizontal bars also allows a few more lines in each window. If you need to see a line extending past the screen edge, you can put the cursor on the line and hit the <End> key to move the window. Data configured via this tab is normally saved by selecting Save Layout from the File menu. Note that one of the items in the tab makes this automatically stored when you exit EDB. See *General Properties* on page 108 for more details.

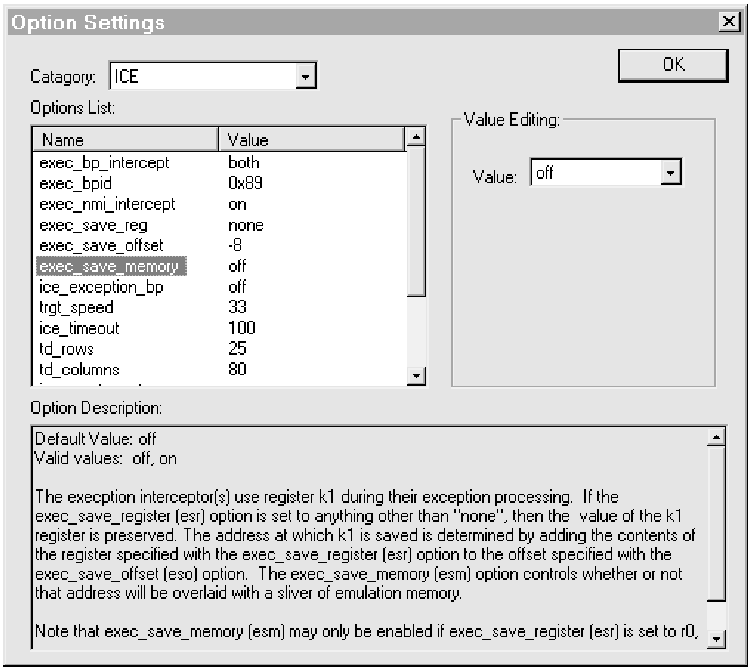
The Program Options tab allows configuration of data normally thought to relate to your application program being debugged. This includes the sections to download, program arguments, calling convention used by your program, etc. All the data configured via this tab can be stored in your `program.rc` file. See *Program Options Properties* on page 109 for more details.

The Color tab is also screen related data. It allows you to change the colors of various screen items. Note that colors used on the Session and Program I/O windows cannot be configured via this screen. Instead, you must use the *Option Settings Dialog* on page 102 to do this.

Option Settings Dialog

This dialog is accessed via the Option Settings item in the View menu. It allows you to view and edit various options. All the options displayed here can also be viewed and edited via **`EO`**/**`DO`** commands (see the MON manual for details on these commands). Note that a few of the options configurable via this dialog can also be configured via the Properties dialog (described above and in detail in *Properties Dialog* on page 108).

The general toolbar contains a button ▣ to access the Option Settings dialog, shown below.



Catagory: ICE

Options List:

Name	Value
exec_bp_intercept	both
exec_bpid	0x89
exec_nmi_intercept	on
exec_save_reg	none
exec_save_offset	-8
exec_save_memory	off
ice_exception_bp	off
trgt_speed	33
ice_timeout	100
td_rows	25
td_columns	80

Value Editing:

Value: off

Option Description:

Default Value: off
Valid values: off, on

The execption interceptor(s) use register k1 during their exception processing. If the exec_save_register (esr) option is set to anything other than "none", then the value of the k1 register is preserved. The address at which k1 is saved is determined by adding the contents of the register specified with the exec_save_register (esr) option to the offset specified with the exec_save_offset (eso) option. The exec_save_memory (esm) option controls whether or not that address will be overlaid with a sliver of emulation memory.

Note that exec_save_memory (esm) may only be enabled if exec_save_register (esr) is set to r0,

NOTE: You can use this dialog to browse through the options details. Use the option list box on the left side to select options. The option description box gives help on the selected option.

Opening and Loading an Application

Before you can begin debugging your application program you must tell EDB its name and use the Load button to download the programs data to your target.

Selecting the Program to Debug

Below are the steps to select and load into EDB your application program. Your application program should be a linked executable compiled with debug information turned on (most compilers do this via the **-g** option).

1. Select Program to Debug from the File menu. EDB maintains a recently used program list which can be used to quickly select previously debugged program.
2. Type in or browse to the application program you wish to run and/or debug. Note there is no standard default file extension for

embedded application programs and none is provided by the dialog (EPI's own sample application programs are built with no extension).

After you select the application program, EDB reads it and converts the application's debug information from your native compilers debug format into a format EDB uses. This data is stored in the file created by taking the base-name of your application program and adding **`.cdb`**. Once this file is created this conversion step is not done again until your application program changes.

EDB also looks for and loads a file *`program`***`.rc`** as your program but with **`.rc`** appended. This file is used by EDB to store saved breakpoints and some program specific options. The Session Window shows the loading and execution of this command file commands (if any).

Finally, EDB updates the execution window to point to the function symbol `main`. This means the module of your application that contains the function `main` is loaded into EDB and appears in the Execution window. Note, if EDB cannot find your source file, it will ask to supply its directory.

NOTE: If you purchased EPI's compilation tools you can select one of our sample programs. **`cdbdemo1`** is a good first choice.

Loading your Program

The Execution Toolbar's Load button is used to download your application's target code to your target system. Many times the application code is already in the target system (flash, etc.) in which case you still need to click the load button, but first, you configure the sections downloaded as none. This is done via the Program Options tab in the Properties dialog (see *Properties Dialog* on page 108). Note that this download configuration is saved in the *`program`***`.rc`** mentioned above.

Executing and Controlling your Application

Starting, Stopping, and Continuing

To run your program, click the GO button or select Go from the Exec menu. Execution continues until one of several events occurs:

- A breakpoint is hit.
- Your program completes via an `exit()` call.
- An exception occurs (bus error, etc.).
- The stop button is hit .

The GO button also continues execution after stopping. The Restart or Load buttons are used to restart from the beginning of your program.

Stepping

EDB supports many forms of stepping. Typically, source level single step and step over are the most commonly used. As you single through your program EDB execution window is updated to show progress through your program.

Breakpoints

Breakpoints can be extremely helpful in looking into the run details of your program. EDB provides both an easy to use breakpoint mechanism and an extensive breakpoint dialog, both accessible through the Execution window, shown in *Navigating with the Execution Window* on page 10.

The Execution window provides a code / breakpoint field to the left of every source line. Lines where the compiler produced *associated target machine code* are identified by an little gray circle. You can double-click this circle to toggle a breakpoint on the line. The circle turns red and fills when a breakpoint is set. For details, see *Execution Window* on page 91.

The View/Edit Breakpoints button brings up the Breakpoints dialog. This dialog allows viewing / editing of complex breakpoints. For details, see *Break Points Dialog* on page 86.

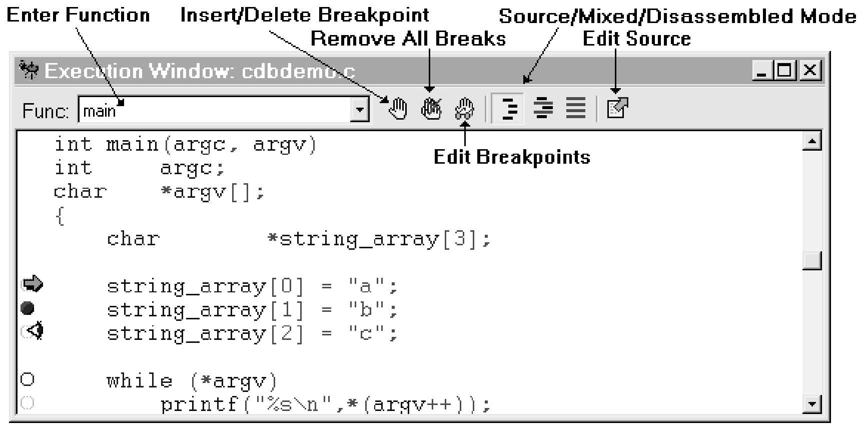
Navigating your Application

EDB has one main window used to browse through your source code. This is the Execution Window. Several other windows provide a feature called *hyper-linking*, which provides a quick way to move the Execution Window's viewpoint to a source line reference available in other windows. (For more information on hyper-linking, see *Context View Point* on page 27.)

Navigating with the Execution Window

From this window you can easily browse / navigate your program source code, associated machine code, set / delete breakpoints, and examine program variables. The window also contains a configurable Edit button that allows you to send current source file name and line number to your editor of choice. This allows a convenient way to make source changes via the editor you are familiar with.

The Execution Window, shown below, displays one source file at time. If the current viewpoint in your program does not have any source code associated with it, then the window will only display disassembled code. For details, see *Execution Window* on page 91.



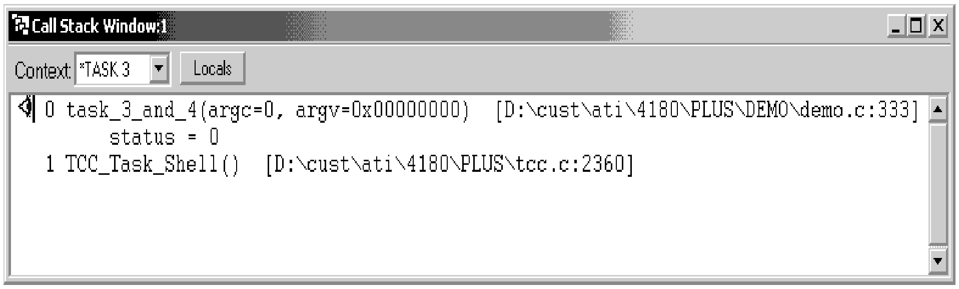
Func: main

```
int main(argc, argv)
int      argc;
char     *argv[];
{
    char        *string_array[3];

    string_array[0] = "a";
    string_array[1] = "b";
    string_array[2] = "c";

    while (*argv)
        printf("%s\n",*(argv++));
```

The Func dropdown listbox in the Execution Window toolbar provides a quick and easy way to move around inside your program from function to function. Normally the listbox shows only functions that were compiled with debug turned on, and therefore, should have source code available. The listbox can also be configured to display all your program functions or your program modules names. The configuration is set via the Short-Cut menu's Select Box Mode option.

Navigating with the Call Stack Window

The Call Stack window displays your current execution call stack. Use the button to bring it up. (See also *Examining via the Call Stack Window* on page 14, and for details, see *Call Stack Window* on page 90.)



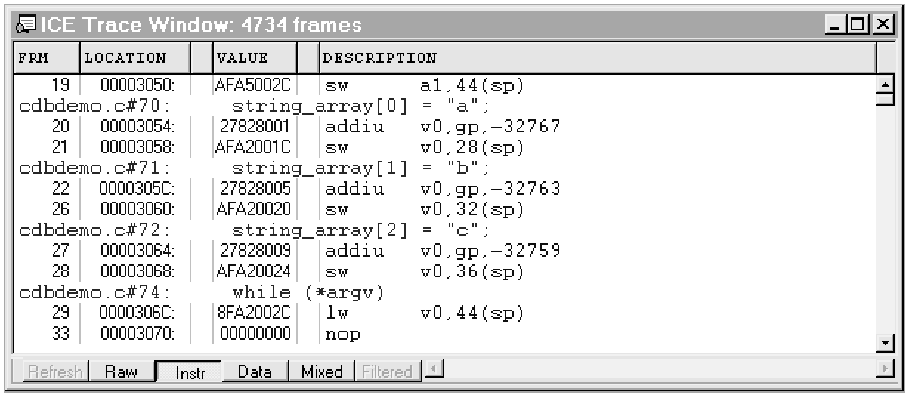
Context: *TASK 3

```
0 task_3_and_4(argc=0, argv=0x00000000)  [D:\cust\ati\4180\PLUS\DEMO\demo.c:333]
      status = 0
1 TCC_Task_Shell()  [D:\cust\ati\4180\PLUS\tcc.c:2360]
```

Like the Execution Window, it contains a clickable viewpoint icon field on the leftmost screen edge. Double clicking in this field will move the Execution Window's viewpoint (hyper-link) to the execution point within the selected function.

Navigating with the Trace Window

When in mixed or inst display mode, the Trace Window displays sources lines mixed in with the associated machine instructions. Note that the Trace Window is only available when using ICE targets with tracing capabilities. (For details, see *ICE Trace Display Window* on page 96.)



```
FRM  LOCATION       VALUE      DESCRIPTION
  19   00003050:    AFA5002C   sw       a1,44(sp)
cdbdemo.c#70:        string_array[0] = "a";
  20   00003054:    27828001   addiu    v0,gp,-32767
  21   00003058:    AFA2001C   sw       v0,28(sp)
cdbdemo.c#71:        string_array[1] = "b";
  22   0000305C:    27828005   addiu    v0,gp,-32763
  26   00003060:    AFA20020   sw       v0,32(sp)
cdbdemo.c#72:        string_array[2] = "c";
  27   00003064:    27828009   addiu    v0,gp,-32759
  28   00003068:    AFA20024   sw       v0,36(sp)
cdbdemo.c#74:        while (*argv)
  29   0000306C:    8FA2002C   lw       v0,44(sp)
  33   00003070:    00000000   nop
```

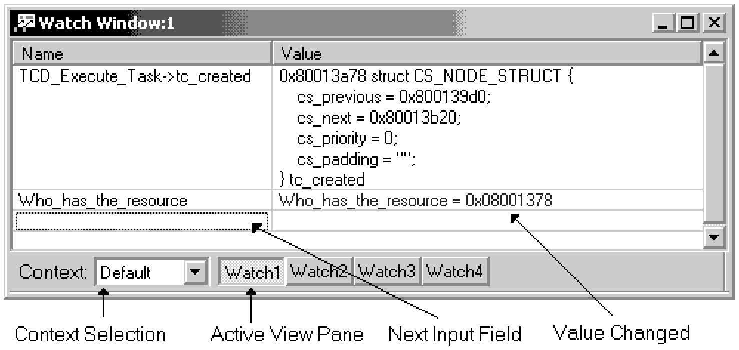
The Trace Window Short-Cut menu contains a hyper-link option which, when selected on a source line, causes the Execution Window's viewpoint to move to the selected function. Note that if the function / line being selected is currently in the call stack you cannot examine local variables via this kind of hyper-linking. Use the Call Stack Window for context hyper-linking.

Examining your Application Data

EDB provides several types of windows for examining your programs data. Typically program variables are best view via the Watch Window, whereas blocks of memory or registers are best viewed via the Memory and Register Windows.

Examining via the Watch Window

This window provides the most convenient way to view your program data. It contains four different panes (views) via the toolbar buttons (watch1, watch2, etc.). When debugging, the multiple panes can be used to group data objects based on context relevance. (For details, see *Watch Window* on page 115.)



Name	Value

TCD_Execute_Task->tc_created	0x80013a78 struct CS_NODE_STRUCT {
	cs_previous = 0x800139d0;
	cs_next = 0x80013b20;
	cs_priority = 0;
	cs_padding = "";
	} tc_created

Who_has_the_resource	Who_has_the_resource = 0x08001378

The watch data display is divided into two columns. The left side contains an editable name field. The right side displays the object data. See *Display Formats* on page 32 for details on data formatting options.

Using the Keyboard

New object names can be inserted via the keyboard by editing an existing left side box. Position the cursor at the last empty box and type or paste in your object name.

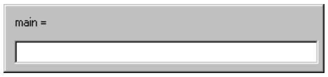
Using Drag and Drop

Many windows also support dragging selected data via the mouse. This can be a convenient way to add objects to the Watch window. Dragging variable names from the Execution Window is particularly useful.

NOTE: The window records the previous text output of each value and upon the next stop of execution identifies those values that have changed by displaying them in a different color.

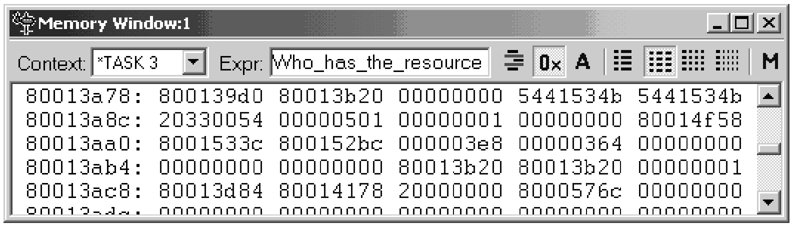
Editing Watch Data

A double-click on the right side of an object display allows the object's value to be changed via a simple edit dialog, shown below.



Examining via the Memory Windows

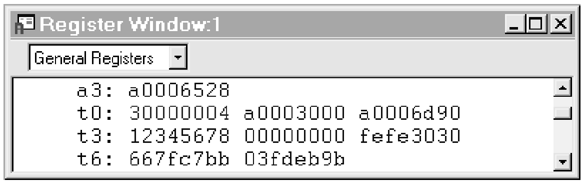
This window is used to view blocks of memory in raw data formats. Both the memory display format and object width are selectable via the Memory Window's toolbar or Short-Cut menu. You can open any number of Memory Windows. (For details, see *Memory Window* on page 99.)



```
80013a78: 800139d0 80013b20 00000000 5441534b 5441534b
80013a8c: 20330054 00000501 00000001 00000000 80014f58
80013aa0: 8001533c 800152bc 000003e8 00000364 00000000
80013ab4: 00000000 00000000 80013b20 80013b20 00000001
80013ac8: 80013d84 80014178 20000000 8000576c 00000000
80013adc: 00000000 00000000 00000000 00000000 00000000
```

Examining via the Register Windows

The Register Window groups and displays registers according to their type. The window's toolbar contains a listbox to configure the class of registers displayed in the window. (For details, see *Register Window* on page 112.)



```
      a3: a0006528
      t0: 30000004 a0003000 a0006d90
      t3: 12345678 00000000 fefe3030
      t6: 667fc7bb 03fdeb9b
```

NOTE: The window records the previous values of registers and upon the next stop of execution identifies those registers that changed by displaying them in a different color.

Examining via the Call Stack Window

In its default mode, this window displays your call stack and function arguments. However, it can be configured (via the Short-Cut menu) to display each function's local variables as well. This can provide a convenient way to view all of a function's local variables. (See also *Navigating with the Call Stack Window* on page 11 and, for details, see *Call Stack Window* on page 90.)

Using the EDB 2

This chapter provides information on using the debuggers with the JEENI. It includes the following sections:

EDB Caveats

This sections lists important caveats that will help you use EDB effectively.

- If you are using **#define** macros that take arguments, do not break the argument list with a new line during invocation. The standard Unix preprocessor (**cpp**) will generate the correct expansion, but all line numbers for the rest of the file will be off by one. This will cause EDB to report the wrong location in the code. This is a preprocessor problem and affects only Unix users who force the compiler to use the Unix preprocessor rather than the built-in preprocessor.
- The debugger depends on the symbol table for information about how machine instructions map to high-level language source lines. Most C compilers only issue line symbols at the end of each statement or line, whichever is greater. For example, if the two statements **a=0; b=1;** are on one line, the only way to put a break after the assignment to **a** and before the one to **b**, is to use the disassembler to spot the right location, and use the **bi** command, or use the Mixed or Disassembled Execution Window modes.
- Multi-line statements (if's with a new line in the middle of the condition) sometimes only have a line symbol generated at the end of the list of conditions. If you try to set a break on any but the last

line of this statement, the break may actually be set on the preceding statement. You can tell when this happens, because EDB will tell you which line it set the break on and whether it is different (less) than what you expected. In windowing versions of EDB, this problem does not arise because the Execution Window indicates the source lines where a breakpoint can be set.

- Some C “statements” do not put the code where you would expect it. For example, assume:

  ```
  5: for (i=0 ; i<9 ; i++)
  6:    {
  7:       foo( i );
  8:    }
  ```

  A breakpoint placed on line 5 will probably be hit only once. The code for incrementing is usually placed at line 8. This problem is especially severe under optimization, and you will have to experiment to become familiar with your compiler's code generation habits. EPI recommends that you use single stepping to see the order in which the source lines print out.

- String constants entered from the command line are stored in a magic buffer provided by the object file **cdb.o** that should be linked with your program (see *Executable Files* on page 17). EDB starts storing strings at the beginning of this buffer, and moves along as more assignments are made. If EDB reaches the end of this buffer, it will go back and reuse the buffer from the beginning. Usually, this will not cause any problems, however, if you are using very long strings, or you assign a string as a constant to a global pointer, you may find that things are shifting under your feet. You can provide more string space by modifying a copy of **.../lib/src/cdb.s** to increase the size of the block at label **__cdbbuffer**, and link in the resulting **cdb.o** file instead of the supplied version.

- Source file names are limited in length by the limitations of the object file format being used by the compiler. For COFF this limit is 14 characters, not including the directory path.

- If your object file format does not support path names to source files (for example, COFF) then you can have a problem debugging modules that have non-unique names. To avoid this limitation, you should ensure that all your modules have unique names. You can also turn on debug information (**-g**) only for the module you are interested in debugging.

- In some cases, it may not be possible to access local variables in the current procedure if you are stopped at the procedure entry point. This can happen if you step into the procedure or if you set a breakpoint at the procedure entry with *procname* **b**. A Step command will execute the procedure prologue, setting up the stack frame for the procedure and allowing all local variables to become “live”.

- If you are using **#include** directives, please note that the contents of the included file cannot be debugged.

EDB Files

Unless otherwise specified, EDB searches for the following files by looking first in your current directory, then in the startup directory (current directory at time of startup, then in the directory where the EDB program is located, then in each directory named in your PATH environment variable. Some categories of files are listed below:

- *Executable Files*
- *EDB Symbol Files* on page 18
- *Initialization Files* on page 18
- *Command Files* on page 19
- *RTOS DLL File* on page 20
- *Custom Registers and Register Windows* on page 20

The RTOS DLL file is an optional DLL file that implements RTOS specific integration features. The default name of this file is `rtos_api.dll` which is designed to work with ATI Nucleus Plus RTOS. Normal Windows DLL search rules are used to locate this file.

Executable Files

EDB can load executable files produced by EPI's compilation tools. Some GNU compilers are also supported (contact the EPI sales department for details). In order for EDB to be able to do source level operations and to access static and local variables, the C compiler must be directed to produce full symbolic debug information. For most compilers, this is done by specifying the **`-g`** command line option. In order to allow string literals or functions in your program to be used in expressions evaluated by EDB, a special object file (**`cdb.o`**) must be linked with the rest of your program. If you bought your compiler and assembler tools from EPI, the compiler driver program will automatically link in **`cdb.o`** if you use it to run the linker. Source code for **`cdb.o`** is provided in the **`./lib/src/cdb.s`** file.

Currently, EDB only supports debugging one program (executable file) at any one time. You can download multiple programs' contents, but you cannot have debug information loaded in EDB for more than one program. This is normally no problem, since in most cases the entire embedded system is implemented in a single executable image that is eventually burned into firmware.

EDB does have an option which allows users to separate their boot code from their application code. The **`EO`** option **`load_osboot`** controls this and defaults to off. If enabled, EDB automatically attempts to find and load a program called **`osboot`** whenever you load your application program. Note that EDB's restriction on only one porgram with source level information still applies, so no source level information is loaded for **`osboot`**.

If osboot or its equivalent has been burned into ROM, or is linked with the application code into a single executable, EDB's Load Osboot option in the Option Setting dialog box should be disabled (see *Option Settings Dialog* on page 102).

EDB Symbol Files

To enhance its portability, EDB maintains symbolic information in its own internal format. An auxiliary program, **`cdbtrans`**, is used to convert symbolic information in a COFF, ELF, or other supported executable file, to EDB's format. **`cdbtrans`** reads the COFF or ELF file and writes out a file with the input file's name and **`.cdb`** appended. On MS/DOS hosts, **`.cdb`** will replace any specified file name extension.

EDB will run **`cdbtrans`** automatically if the **`cdb`** file does not exist or if it is older than the executable file being debugged. On MS/DOS machines, there may not be enough memory remaining after EDB is loaded to allow **`cdbtrans`** to be run, especially if the COFF file has a large symbol table. If this happens, run **`cdbtrans`** manually before running EDB, giving it the name of the target program (for example, **`cdbtrans hello`**).

Initialization Files

EDB has three types of initialization files: Target initialization, User Interface initialization, and Command History.

edb*xxx*.ini

Since opening and positioning the windows pertinent to your particular debugging task may become tedious, windowing versions of EDB allow you to save your screen layout by choosing the Save Layout item in the File menu. This information is normally stored in the Windows directory under the name **edb*xxx*.ini** where *xxx* identifies the type of target, either ICE or simulator (sim).

If the appropriate file cannot be found at startup, EDB defaults to opening only the Session and Execution windows.

The table below shows the initialization file that is associated with each EDB executable:

Program Name	Initialization File
`edbsim`	**`startsim.ini`**
`edbice`	**`startice.ini`**

`startedb.hst`

This is the command line history file name. Normally, it is created in your working directory upon debugger shutdown. However, if at startup the file is found on your search path, then it will be updated to the same file / path.

EDB has one other special case startup file that is used only when referenced on the command via a **-m** option.

opt_file This is an ASCII text file containing EDB command line options for one or more specific target programs, and / or a set of default command line options. If the filename is given with a **-m** option, EDB will scan the file looking for the name of the current target program starting in column 1. If a match is found, the rest of that line and each subsequent line that starts with a tab character will be processed as EDB command line options. If no match for the program name is found, the default options will be processed if there is a section beginning with an asterisk (*****) in column 1.

The following example shows a default options file:

```
vrtxapp  -e first_task -d epiice3:e
         -a ./task1 -d ./task2
boot     -d com2 -5
*        -d epiice0:e
```

Command Files

Command files contain one or more debugger commands and may be read or written by EDB (see the < and > commands under *Record and Playback Commands* on page 57). When a Command file is read, EDB will execute each command as if it was entered at the keyboard. Command files may be specified on the command line when the debugger is invoked via the **-p** option. If this is done, the commands contained in the file are executed and their results displayed automatically.

Automatic Startup Command Files

On startup, EDB looks for two command files to playback global and program specific initialization commands. These files are played back before any command files specified with the **-p** option.

.cdbrc cdb.rc

This is the global startup command file. There can only be one copy of this file for each user; if it exists, it is played back at the start of every EDB session. The global startup command file should contain anything you always want done (for example, like setting up command aliases or configuring options, etc.).

The file name is **.cdbrc** on Unix hosts, and **cdb.rc** on MS-DOS hosts. On all hosts, the file must be located in the directory specified by the environment variable **HOME**. **HOME** is maintained automatically by the command shell on Unix systems. On MS-DOS systems you will have to add the command **SET HOME=***drive*:*path* to your **AUTOEXEC.BAT** or other setup batch file. If you would like the **HOME** directory to be your current working directory, add the command **SET HOME=.** to your setup batch file. If there is no **HOME** environment variable, EDB will not look for the global startup command file.

*program***.rc**

Whenever a new program is opened in EDB (at startup, or via the File Program to Debug menu), EDB will look for a command file with the same base name (and in the same directory) as the program being debugged, plus an extension of .rc. This file should contain commands specific to the program you are debugging. The user can cause EDB to create this file by choosing Yes at the Save Session Info exit prompt, or by selecting Save Session Info from the File menu. When creating the file, EDB will write the necessary commands to recreate your current list of search directories, breakpoints, assertions, and load option settings. When starting up, EDB processes this file and issues a warning if the file is older than the program being debugged, since it may contain breakpoint commands that are no longer valid. Breakpoints are saved with absolute addresses (not symbolic addresses).

RTOS DLL File

Starting with version 1.4, EDB has RTOS integration features. These features allow for RTOS object browsing and thread specific breakpoints. The latter can be very useful for setting breakpoints on OS interfaces used by multiple threads. The logic to gather the RTOS data is encapsulated in a dynamic link library (DLL) file. This DLL is designed to be easily customized to third party or customer operating systems. This DLL file should be stored in the same directory as the EDB executable file(s). Note that only one pre-configured DLL is supported at this time (for ATI's Nucleus Plus RTOS). If you are customizing your own DLL then you will need to replace this `rtos_api.dll` with your own version. The specific DLL loaded (if any) can be identified via EDB's help / about dialog.

If you are licensed for the RTOS feature, source code is available for the default DLL. Users can customize it to work with currently unsupported off the shelf or custom operating systems. For a complete list of supported operating systems contact `sales@epitools.com`.

Custom Registers and Register Windows

EDB 2.4 introduces the customizable register file, which is of particular importance to SOC designers. This feature allows new registers (memory based or CPU based) to be added to the EDB and MON command language. It also allows new register window types to be created and any defined registers to be added.

EDB and MON search for the register file in a sub-directory based on the chip’s architecture family type. **mips** for MIPS architecture based targets and **arm** for ARM based targets. The register file name is created from the selected chip type (see the definition of the debuggers **-v** switch, described in *Invoking EDB* on page 24).

Include files are supported to allow common processor elements to be placed in one file. The INCLUDE command, shown below, begins reading from the referenced file and returns to the calling file when done. Nested include files are also allowed.

```
INCLUDE "filename"
```

Character names (*space_name*) can be used for predefined debugger spaces (*space_id*). Using a character name provides more readable register definitions. Also, you can set more than one *space_name* for any *space_id*, including memory spaces.

```
CPU_SPACE = space_name space_id
```

Where:

space_name is an alphabetic name.

space_id is one of the pre-defined space numbers used by EDB and MON.

New register names are definable with the information below:

```
REG = reg_name offset space_name byte_size
      [SEQ first last obj_inc inc]
```

Where:

reg_name is an alphabetic name.

byte_size is one of: {1|2|4|8}.

first is a numeric value. This value is the first number appended to *reg_name* to form a sequence. Sequences make it easy to represent a consecutive set of like named registers (for example: r0..r31).

last is the last number in the sequence (see *first*).

obj_inc determines the object increment (number of *byte_size* objects) to add for each register in the sequence.

inc allows the register number sequence to increment by the specified increment value.

For Example: Let’s say we want to have a sequence of 4 byte-size registers mapped to memory at 0, with each register in the low byte of successive machine words (32 bits). If the designer chooses to name these registers z1, z3, z5, ..., then the definition would be:

```
REG=z 0x0 MEMORY_P 1 SEQ 1 7 4 2
```

Registers can also be broken down into displayable fields. Any previously defined register or register sequence can be set up as field encoded. Note that if a field breakdown is given for a register sequence, the fields apply to every register in the sequence. Fields of more than one bit are displayed as *field_name = hexidecimal value*. One bit fields are displayed as an uppercase or lowercase *field_name* where uppercase means a TRUE or 1 value.

REG_FIELD = *reg_name field_spec*

Where:

reg_name is an alphabetic name previously defined via a REG statement. Note that for sequence registers a full sequence register name must be given (including the number).

field_spec = *field_name high_bit low_bit* [, *field_spec*]

field_name is an alphabetic name.

high_bit is a numeric value in the bit range of the given register. Must be >=*low_bit*.

low_bit is a numeric value in the bit range of the given register. Must be <=*high_bit*.

EDB also supports adding additional Register window types (panes). A window definition is simply a list of register name pairs. All the registers logically contained between, and including the two referenced registers, are included in the list. Registers within the window are logically broken down into groups based on the name. Sequence registers are displayed in groups with wrapping occurring at the right screen edge. Registers with field symbol definitions always display one per line. The only supported display format for registers is hex.

REG_WINDOW_CLASS = *class_name reg_name_list*

Where:

class_name is an alphabetic name.

reg_name_list = {*reg_name reg_name*} | [, *reg_name_list*]

reg_name is an alphabetic name previously defined via a REG statement. Note that for sequence registers, a full sequence register name must be given (including the number).

Predefined spaces

In the debugger's architecture sub-directory you can find a file called **spaces.rd**, which defines names for the debugger's space designators. This file is automatically read at startup time. You can refer to it for definitions of coprocessors' spaces etc.

Below is a sample of a MIPS **spaces.rd** file:

```
// EDB Register Definition File (space.rd): defines various
// common MIPS access spaces.
SPACE=MEMORY_V   0x0    // Virtual Memory
SPACE=MEMORY_P   0x9    // Physical Memory
SPACE=GR         0x100  // r0 - r31
SPACE=TLB        0x200  // TLB registers  0..?
SPACE=MR         0x300  // mdhi, mdlo, (acc0-2 54xx)
SPACE=LX         0x400  // Lexra specific
// Coprocessor access spaces
SPACE=CP0_CTL    0x900  // Some newer MIPS32 chips use this space
SPACE=CP0_GEN    0x800  // Coprocessor control register (cause, sr, etc)
SPACE=CP1_CTL    0xA00  // floating point control
SPACE=CP1_GEN    0xB00  // floating point
SPACE=CP2_CTL    0xC00  // CP2 Typically not used
SPACE=CP2_GEN    0xD00  // "
SPACE=CP3_CTL    0xE00  // Mips I/II architecture chips only
SPACE=CP3_GEN    0xF00  // Mips I/II architecture chips only
// MIPS III and above (including MIPS32/64) architectures
SPACE=ICT        0xE00  // Instruction Cache tags
SPACE=DCT        0xF00  // Data Cache tags
```

Sample ARM **spaces.rd** file:

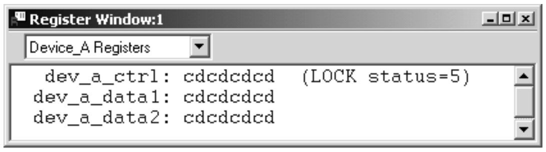
```
// EDB Register Definition File (space.rd): defines various
// common ARM access spaces.
SPACE=MEMORY_V   0x0       // Virtual Memory
SPACE=MEMORY_P   0x9       // Physical Memory
SPACE=CRNT       0x100     // r0 - r15
SPACE=USER       0x200     / User/System
SPACE=SVC        0x300
SPACE=IRQ        0x400
SPACE=FIQ        0x500
SPACE=ABORT      0x600
SPACE=UNDEF      0x700
SPACE=STATUS     0x800     // cpsr, spsr{svc,abort,undef,irq,fiq}
// coprocessers (bits 0..3) define coproc #, (16 registers per)
SPACE=COPROC0    0xf00
SPACE=COPROC1    0xf01
```

Sample Register Definition file

The example below demonstrates a definition for some memory mapped registers (common in hardware designs).

```
// Sample Register Definition File - Demonstrates the
// declaration of new registers, register fields, and an EDB
// register window for them.

INCLUDE "spaces.reg"

// Map device "a"'s registers -- contains three 32 bit registers
REG=dev_a_ctrl 0xFF00A000 MEMORY_P 4
REG=dev_a_data1 0xFF00A004 MEMORY_P 4
REG=dev_a_data2 0xFF00A008 MEMORY_P 4

REG_FIELD=dev_a_ctrl status 2 0, lock 3 3

REG_WINDOW=Device_A dev_a_ctrl dev_a_data2
```

Below is the resulting EDB Register Window:



```
  dev_a_ctrl: cdcdcdcd   (LOCK status=5)
 dev_a_data1: cdcdcdcd
 dev_a_data2: cdcdcdcd
```

User Register Definition files:

Although EPI provides some register definition files for standard complex CPU's, many users are creating cores with custom peripherals or need register based access to their target board's peripherals. Custom Register Definition files can be created by users and read in via our debuggers File Read (**FR**) command. Please see the description of this command in the EDB command section for more details.

Invoking EDB

EDB is executed by the following command line:

edb [[-*options*]... [*binfile*]]

This command causes the debugger to load symbolic information for a target program and prepare it for execution. The program will not be downloaded to the target system memory until you issue a command to load the file or to begin execution. Replace *edb* with the name of the specific version of EDB you want to run. See Chapte r1, *Introducing EDB*, on page 1 for details.

This command can be specified with one or more of the following *options*. Each *option* must begin with a dash (**-**) and may be followed by parameters.

Options	Description
-a *directory*	Names an alternate directory to search for source files. Multiple **-a** options can be specified, and the alternate directories are searched in the order specified. If a file is not found in an alternate directory, the current directory is searched.
-c *name*	Adds *name* to the title line of EDB. This can be helpful to identify multiple instances of EDB running on the desktop. *name* can be any number of characters, symbols, and numbers. If the *name* includes spaces, it must be enclosed in double quotes.
-d *device_name*	Specifies the name of the communications channel to be used to connect to a remote target. This option is not applicable with the Instruction Set Simulator. Typically this is the name of a serial device or an Ethernet host name. Ethernet host names must be followed by **:e** with one exception (if the ICE being connected to is an HP Software Probe, the Ethernet name must be followed by **:h**). Note that this option can be saved in the target initialization file. The default device name is **COM1** on MS-DOS hosts and **/dev/ttya** on Unix hosts. There is also a special option for serial device channels. **:s,d**[*n*], which causes communication to start at the baud rate given by either the n option following the **d** or a completely separate *-n* option. Without this option, the default startup rate is always 9600 baud and a serial driver must support baud rate changing to allow for faster rates.
-e *entry*	Specifies the name of the first procedure of the program (the default is **main**). This is the point in your program where debugging should begin and is normally the first high-level language procedure that will get control. If you begin execution with a source level single-step operation, EDB will execute up to this location. This is not usually the actual entry point address where execution will start. EDB normally starts execution at your program's entry point, as configured in your ELF or COFF execution file. This behavior can be controlled via the **EO** option **load_entry_pc**.
-i	Intrusive Startup mode (default). Resets the processor and clears any breakpoints. Use **-ni** to connect to a target without loosing this target state information. This option is only available for targets that support concurrent target debugging.

Options	Description
-l	Little Endian Startup mode. Indicates that the target system being connected to or configured is little endian.
-m *opt_file*	Causes a default options file to be read. This file may contain various EDB command line options that will be applied depending on the name of the program being debugged. See *Initialization Files* on page 18 for a description of this file.
-ni	Non-Intrusive Startup mode. Recovers the state of a running system (for example, breakpoints, application running or stopped). If the target is currently executing in interactive mode, the debugger will enter interactive mode and not disturb the running program. This command is only available for targets that support concurrent target debugging.
-p *file*	After EDB initializes and processes any *opt_file*, it will play back the commands from *file*. Multiple **-p** options can be used. The effect of using multiple **-p** options is as if each file begins with a **<** *file* command to invoke the next one. In other words, the command files will be played back in the opposite order in which they appear on the command line. In any case, **-p** command files will be played back after any automatic startup command files (see *Command Files* on page 19).
-r *file*	Record all commands to *file*.
-R *file*	Record all output Session window output to *file*.
-v *cpu*	*cpu* is a number specifying the target CPU type. Valid values for *cpu* are listed in your MON Assembly level debugger manual. You can also see the CPU list supported by your version of EDB by starting EDB with the option **-v** *cpu*, exiting EDB, and then examining the file sesslog.txt created in the current directory.
-z	Standalone Startup mode.
-*n*	Specifies the speed of the serial communications channel used to connect to a remote target. This option does not apply to the Instruction Set Simulator (edbsim). *n* is a digit in the range **0..7**, specifying a baud rate of 1200, 2400, 4800, 9600, 19,200, 38,400, 57,600, or 115,200. Not all hosts will be able to support the highest speeds. In particular, the maximum valid speed for Unix hosts is **-5** (38,400 baud). Your maximum speed may be limited for Remote Server Targets by the speed of the target systems serial chip.

Options	Description
`binfile`	The path name of the executable file to be loaded. EDB supports the loading of EPI COFF and ELF files produced by any compatible linker. If `binfile` is not specified, you will be prompted to enter the program name. **NOTE**: Arguments to the program being debugged are specified either via the Program Properties dialog (see *Program Options Properties* on page 109), or on the **`r`** (run) command. Program arguments are not specified on the EDB command line.

Conventions

Context View Point

EDB supports the concepts of *current file*, *current procedure*, and *current line*. This is referred to as the *context view point* and affects the scoping of expression data references from your program. Normally the context view point matches the execution point (the current program counter location), but can be changed via the **e** (Enter), **p** (Print source), and **vc** (View Context) session commands, the Func box in the Execution window toolbar (see *Execution Window* on page 91), the Default Context box in the Context Toolbar (see *Context Toolbar Command* on page 81), and by hyper-linking.

Hyper-linking is the means by which the Execution Window's context point can be quickly moved to a particular procedure. The Call Stack Window supports hyper-linking by right-clicking in the far left column which causes the Execution Window to change context to the current execution point within the referenced function. This feature is also available from the Call Stack Window Shortcut menu.

When you use the **p**, **w**, or **W** commands to display lines of source, the *context point* (line) is preceded by a > character. See *Execution Window* on page 91 for details on how the context and execution points are identified there.

Debugger Command Operands

In the command descriptions that follow, the following operands are used.

`cmds` One or more valid EDB commands, separated by semicolons. For example, **`t ; c`**.

`exp` Any expression. Where exp is allowed but not required, the default is 1 unless otherwise stated. See *Expressions* on page 28 for more information.

`file` A file name.

`format` A string specifying how to display the results of an expression evaluation. See *Display Formats* on page 32 for details.

`line` A line number. Where *`line`* is allowed but not required, the default is the current line.

`number` A single number (for example, **`9`**, not **`4+5`**). As in C, *`number`* is a sequence of decimal digits, or **`0`** followed by octal digits, or **`0x`** followed by hex digits.

`proc` The name of a procedure (function) in your program.

`stack` A number indicating a stack depth, as reported by the **`t`** (stack trace) command. This is used to identify a particular activation of a procedure.

`var` The name of a variable in your program, or a special EDB variable. See *Variables* on page 29 for ways to specify variables in hidden scopes.

Expressions

Expressions in EDB may in general be composed of any combination of variables, constants, function calls, and C operators, with the following special considerations:

- Do not type a <CR> in the middle of an expression. EDB is an interactive program, not a compiler, and <CR> signifies the end of the command. There is currently no provision for continuation lines.
- If the program is not currently active (you haven't done an **`r`** command yet, or the program has terminated), expressions can contain only constants and global address expressions.
- Use two slashes (`//`) for division instead of a single slash (`/`). This is to avoid confusion with the *`exp / format`* command.
- Do not start an expression at the beginning of a line with a dash (**`-`**) or pound sign (**`#`**), or it will be confused with the dash (**`-`**) or pound sign (**`#`**) command. You can make the expression unambiguous by wrapping it in parentheses (for example, **`-3`** will not be treated as move to 3 lines before the current line).
- A term of the form *`proc#line`* evaluates to the memory address of the code associated with the given line number in the file containing the given procedure. Similarly, (`#`*`line`*) means the memory address of the code associated with the given line number in the current file.

NOTE: C expression semantics apply, including issues of type. If you use the name of a simple variable (like an **`int`**), the value you get is the contents of that variable. If you use the name of an array without all its subscripts, or a function name without its (possibly empty) argument list, the value you get is the address of the object.

Of course you can get the address of a simple variable via the C address of operator (**&**).

Procedure Calls

Functions in your program may be invoked from the command line as part of an expression. For example:

```
foo = AddArgs( 1, 2 ) * 3
```

NOTES:

- Procedures that return more information than a **`long`** are generally not handled correctly if the return value is to be used, as in the above example.
- Using procedures in expressions requires that the helper file **`cdb.o`**, be linked into your target program.

If there are any breakpoints encountered during command line procedure invocation they will be processed as usual. This means that command line procedure calls can be a very useful tool. Once a fault has been traced to a specific function, appropriate breakpoints can be set and the function invoked with various combinations of arguments.

The list procedures command (**`l p`**) can be used to get a list of procedures in your program. If, while debugging, you would like to be able to invoke library procedures that are not referenced anywhere in your program (and are therefore not loaded), you can add a dummy function to your program that contains phony calls to the procedures.

Variables

Variable names under the debugger are exactly as you named them in your source code, but EDB may be instructed to ignore alphabetic case when searching the symbol table. The **`Z`** command (see *Miscellaneous Commands* on page 64) can be used to toggle case sensitivity in all searches.

When your program has stopped, EDB can access every variable that is active at that point: globals and statics, and local variables in each active procedure. This is in spite of the fact that you may have many instances of a variable hidden by nested scopes. The following naming conventions are supported to allow you to access the correct version of any variable:

`var` Standard C scope rules: *`var`* must be visible at the current line of the current procedure. Locals, local statics, and parameters of the current procedure are checked, then file scope statics, then globals, and finally EDB special variables. Remember, it is the current viewing point and not the location where execution last stopped that controls which is the current line, procedure, and file.

proc#*var*

Search the stack for the most deeply nested occurrence of procedure *proc*. If found, use that procedure activation's stop point and perform a scope search as above. If *proc* is not currently active, the search will begin with file scope statics for the file containing *proc*. This allows file scope statics that are hidden by a duplicate name at a lower scope to be found.

stack#*var*

Specifies the procedure activation that is at stack frame *stack* as the starting point for the scope search. When this form is used, *var* must be a local, local static, or parameter of the procedure. This is the only way to locate a local variable in a recursive procedure activation that is not the current procedure nor the most deeply nested.

:*var* Searches for a global variable *var*.

$*var* Searches for an EDB special variable var (see *Special Variables* on page 30).

. (dot) The dot is the *name* for the last thing you looked at. It has the same type that you used to view it. This means that if you look at a **long** as a **char** (for example, **long_var / cb**), then **.** will be considered to be a **char**. This is useful for functioning on things in a very different way than the default type allows, like changing only the second highest byte of a **long**. Dot may be assigned a value, used in another expression, etc.

NOTE: The dot in EDB is not the same as in some other debuggers, where it is simply a number. If you use it, it will be de-referenced like any other name, and if you enter **.+30**, it will perform whatever arithmetic is appropriate for that type of dot. For example, if **i** is an **int** variable and **cp** is a pointer to **char**, then the sequence of commands:

```
i ; . = . + 10 ; cp ; .[30]
```

is equivalent to:

```
i ; i = i + 10 ; cp ; cp[30]
```

Special Variables

EDB supports the concept of special variables. Special variables are names for things that are normally not directly accessible, either because they are machine registers or because they are internal to the debugger. If you simply type a name (for example, **foo**), the normal scope search will scan for a special variable of that name only after failing to find a match in the user program, and you will get an error if it is not found. If you preface the name with a **$**, then only the special variable list is searched, and if a matching name is not found, one will be created.

These user special variables have the same type as the last expression that was assigned to them. For example, entering **$mumble = 3*4** will create special variable **$mumble**, assign it the value 12 and make it type **int**. Special variables may be used like any other variable except that you may not meaningfully take their address.

Pre-Defined Special Variable Names

The following variables are always defined by EDB:

Register names — These are target-dependent. See *MIPS Target Differences* on page 35, for a discussion of available register names.

result — This can be used to check the return value of a procedure.

There are a number of debugger internal special variables. Modifying their values will change the way the debugger works. You can list them (along with user specials) with the command **l s**. The currently defined debugger variables are:

SIGNAL — The signal number that will be passed back to the target program (not actually meaningful for current target programs).

R_SIGNAL — The signal number that caused the program to stop (likewise not currently meaningful).

DELTA — This tells the disassembler the maximum distance an address can be from the nearest label or global and still produce the form **foo+0x02bc**.

LOAD_TEXT — Indicates whether the Text section type should be downloaded. A value of zero indicates that no sections of type Text should be downloaded. A non-zero value indicates that this section type should be downloaded.

LOAD_DATA — Indicates whether the Data section type should be downloaded. A value of zero indicates that no sections of type Data should be downloaded. A non-zero value indicates that this section type should be downloaded.

LOAD_BSS — Indicates whether the BSS section type should be downloaded. A value of zero indicates that no sections of type BSS should be downloaded. A non-zero value indicates that this section type should be downloaded.

LOAD_LIT — Indicates whether the LIT section type should be downloaded (RDATA section for mips targets). A value of zero indicates that no sections of type LIT should be downloaded. A non-zero value indicates that this section type should be downloaded.

screen_length The current screen length. Not used in EDB.

Debugger special variables beginning with an underscore (_) are not normally listed. They represent the internal state of the debugger, and should not be changed. To see them all, type **l s _**.

_LANGUAGE Shows which expression evaluator is in use. -1 indicates auto-select based on the type of the current file, 0 indicates C, and 1 indicates FORTRAN.

_FILE The name of the current file.

_PROCEDURE The name of the current procedure.

_LINE The current line number.

_BREAK The current breakpoint number.

_TIP_TYPE The type of the current debugger target vehicle environment. The possible values are listed below:

Unknown	0
ICE	1
Simulator	2

_TIP_SUBTYPE The subtype of the current debugger target vehicle environment. The possible values are categorized according to _TIP_TYPE below:

_TIP_TYPE = ICE (1)	
Unknown	0
HP Probe	5
MAJIC	6
_TIP_TYPE = Simulator (2)	
Unknown	0
Normal	1
Custom	2
Tracing	3

Display Formats

When you use an expression evaluation command (See Chapte r3, *Debugger Commands*, on page 39), the optional format operand may be used to control the number of items displayed, the size of each item, and the display format to use. The syntax of the format operand is:

```
[count][fmt][size]
```

where:

count Optional. Is the number of times to apply the format.

fmt Is the format style.

size Optional. Indicates the number of bytes to be formatted.

For example, **foo/4x2** would print, starting at **foo**, four 2-byte numbers in hex.

If present, *count* must be a number and *size* must be a number or one of the following mnemonic letters:

b	1 byte.
s	2 bytes (**short**). (Must be used with a format designator)
l	4 bytes (**long**).

For example, **foo/xb** prints a hex byte. The default value for *count* is **1**, and the default for *size* depends on the type of the expression. For instance, if the expression consists of a variable of type **char**, *size* will default to **1**, but if it is an **int** variable or expression, *size* will default to **4**.

A common use of *count* is to specify how many locations to display when dumping an area of memory. For example, ***0x28000/20xb** will display the 20 bytes starting at address 0x28000 in hex.

NOTE: *count* makes sense only when the expression evaluates to an addressable argument or with formats that interpret the expression as an address. For example, **0/3x** will display **0x0**, **0x4**, and **0x8** - probably not what was intended.

The format style *fmt* is one of the following letters:

a Display a string using the expression as the address of the first byte. This will print up to the first null character or 128 characters, whichever occurs first. The *size* value can be used to force printing of a given number of bytes, regardless of the occurrence of null characters.

B Display the expression value in binary. The capital **B** is used to avoid conflict with the *size* **b**.

c Display the expression value as a character.

d Display the expression value in decimal. This is the default for things of integer type.

e Display the expression value in **%e** floating point notation. When using the **e** or **f** formats, *size* specifies the precision desired. If unspecified, a reasonable default is used. The actual object size is implied by the expression type (4 for **float**, 8 for **double**).

f Display the expression value in **%f** floating point notation. When using the **e** or **f** formats, *size* specifies the precision desired. If unspecified, a reasonable default is used. The actual object size is implied by the expression type (4 for **float**, 8 for **double**).

g Display the expression value in **%g** floating point notation. (Default for expressions with float or double type).

i Disassemble a machine instruction at the address specified by the expression.

I Also disassembles a machine instruction, but if the address maps evenly to a line number in the source, the source line is also displayed. This is useful for viewing what the compiler generated for a line of source. For example, the command **foo#3 / 5I** will display line 3 in procedure **foo**, followed by the corresponding 5 machine instructions.

n Use the normal format, based on type. This is what one gets if a *format* operand is not specified.

N This is the same as **n**, except that the feature of calling a user-supplied function to display structures and unions is suppressed (see *Structure Formatting* on page 34).

o Display the expression value in octal.

p Display the name of the procedure containing the address specified by the expression. Also displays the file name and the source line or instruction that map to the address.

s Display a string using the expression as the address of a pointer to the first byte. Same as ***exp*/a**.

S Do a formatted dump of a structure. This is the default for things of type struct. See *Structure Formatting* below. The capital **S** is used to avoid conflict with the *size* **s**.

t Display the type of the object. In this case, the expression must consist of an addressable object (variable or function). Structure and procedure objects will include the names and types of their members and parameters.

u Display the expression value in unsigned decimal. This is the default for things of unsigned integer type.

x Display the expression value in hexadecimal.

The formats that print numbers allow an upper case character to be synonymous with appending the letter **l** to specify *size*. For example, **O** will print in long octal, as will **ol**. Of course, this seldom matters on most RISC systems where a plain **int** is the same size as a **long**.

Structure Formatting

When EDB dumps a **struct** or a **union**, it uses a straightforward formatting algorithm. For large, complex, or bizarre structures, this may not provide the kind of display you want to see. To allow complete flexibility in displaying structure contents, EDB provides a mechanism for you to supply your own display routine for any **struct** or **union** type.

Before using its default formatting rules to display a **struct** or **union** with tag name **FOO** (for example), EDB checks for a procedure in the target program with the name **_FOO** (that is, the tag name with an underscore character prepended.) If found, it will invoke that procedure, passing it two arguments: a pointer to the **struct** or **union** and the *size* parameter (default value is -1, user passable values are **0** to **INT_MAX**). You can code the procedure in your program to display the passed-in **struct** or **union** any way you like.

For example, if **fooFirst** was a node in a tree, you might type **fooFirst/n2** (for structures **n** is the same as **S**). The routine **_FOO** would then dump the members of **fooFirst** and its first two levels of children (or whatever is appropriate). This feature may be overridden by using the **N** format, instead of **S** or **n**.

If you are using EPI's compiler and library, and have included support for the host operating system interface, you can simply use **printf()** to display the contents of the **struct** or **union** and the output will appear on the debugger console as if it was generated by EDB. Support for passing operating system requests back to the debugger is automatically available in Instruction Set Simulator and Remote Server target types.

NOTE: Using procedures in expressions requires that the helper file **cdb.o** be linked into your target program.

MIPS Target Differences

This section documents aspects of the debugger that are specific to the MIPS processor types.

Register names

pc, **sp**, **a0-a3**, etc.

These are the names for the registers, program counter, stack pointers, argument registers, etc. Virtually all the register names defined in the chip manual for your specific Mips chip are available. To see a complete list use the **l r** command, but be prepared for a lot of output - the R3K family processors have a lot of registers, and some of them have two names (for example, **ra** and **r31**). A partial list of the register mnemonics is given below. Registers are usually 32 to 64-bits depending on your chip. Note that modifying the contents of registers when you are debugging a high-level program can produce undesirable results.

r0 - **r31** Processor zero's general registers.

sr, **cause**, ... Common names for coprocessor registers are also supported. These vary widely from chip to chip. Consult your CPU User's Manual for details.

c*X*_0 - c*X*_31 & **g*X*_0 - g*X*_31**
Coprocessor *X* registers, where *X* is **0..2**.

fcr0, **fcr31** Another name for coprocessor one's control registers zero and thirty-one. .

result, **resultf** These can be used to inspect the return value of a procedure. **result** is another name for **r2**, the register used to return the first (and usually only) word of a function result. **resultf** is another name for **f0**, the register used to return floating point function results.

tl0 - tl63, **th0 - th63**
TLB registers. These registers are not available on Mips chips that do not support virtual memory.

ARM Target Differences

This section documents aspects of the debugger that are specific to the ARM processor types.

Register names:

pc, **sp**, **lr**, **r0-r15**, **spsr**, etc.

These are the names for the registers, program counter, stack pointers, argument registers, etc. Virtually all the register names defined in the chip manual for your specific ARM chip are available. To see a complete list use the `l r` command, but be prepared for a lot of output. A partial list of the register mnemonics is given below. Register names are not case sensitive. ARM registers are 32 bits. Note that modifying the contents of registers when you are debugging a high-level program can produce undesirable results.

r0 - r15, **sp**, **lr** Processor's current general registers. Note that **sp**, **lr** and **pc** are synonyms for **r13**, **r14** and **r15** respectively. The ARM architecture has a banked register scheme which means the value of these registers depends upon your current mode of execution. Each of the banked registers can be accessed using the mode qualifiers as shown below.

r8_M - r14_M, **sp_M**, **lr_M**
Processor's user mode general registers where **M** is either execution mode **user** or **fiq** . (e.g. **r8_user**, **sp_fiq**, …).

r13_M - r14_M, **sp_M**, **lr_M**
Processor's user mode general registers where **M** is one of the following execution modes: **svc**, **abort**, **undef**, **irq**

cpsr, **spsr** ARM's status registers.

cpsr_M, **spsr_M** ARM's mode relative status registers, where **M** is either **svc**, **abort**, **undef**, **irq**, **fiq**.

c*X*_0 - c*X*_15 Coprocessor *X* registers, where *X* is `0..15`.

`result`	These can be used to inspect the return value of a procedure. **`result`** is another name for **`r0`**, the register used to return the first (and usually only) word of a function **`result`**. Note that this name is not available within the MON command language.

Debugger Commands 3

EDB has a large number of commands for viewing and manipulating the program being debugged.

EDB has two Command Processing modes. The default mode is EDB's native mode and is fully described here. The alternate mode uses our low-level debugger (MON) command interface. For ICE users, many of the MON level ICE only commands such as **`DT`** (Display Trace, etc.) are also available from EDB command mode. Please refer to the MON manual for details on these commands.

Because of the richness of the EDB windowing interface, you may never need to use most of these commands, since the vast majority of debugger functions can be performed by menu selections and mouse actions. There are a few operations that can only be performed via commands, such as searching the source code for a string of text, or listing the typedef table. Also, it is sometimes quicker and easier to perform an action by typing the corresponding command in the Session Window. For example, it may be quicker to type **`e main`** than to select main from the Func list box.

Many commands have various forms that are not representable in a single unambiguous line of syntax, even with the square bracket / curly brace / vertical bar notation to indicate optional entries and choices. In these cases, more than one syntax line is given to specify the alternate forms.

Most command arguments may be omitted, as indicated in the corresponding command syntax. Omitted arguments are defaulted as indicated in the discussion of each command.

Multiple commands, separated by a semi-colon (;), may be entered on a single command line. In fact such a command list, delimited by curly braces, can be given as an argument to certain EDB commands.

The command descriptions in this chapter are grouped into sections, with each section documenting a category of related commands. The categories are:

- *Command Summary* on page 40
- *Viewing Commands* on page 44

- *Program Control Commands* on page 47
- *Breakpoint Commands* on page 50
- *Assertion Commands.* on page 56
- *Record and Playback Commands* on page 57
- *History Commands* on page 63
- *Miscellaneous Commands* on page 64

Command Summary

Name	Syntax	Description
Again	<CR>	Repeat previous command, if possible.
Create Assertion	**a** *cmds*	Create a new assertion with the given command list.
Modify Assertion	*exp* **a** {**a** | **d** | **s**}	Activate (**a**), delete (**d**), or suspend (**s**) assertion number *exp*.
	A [**a** | **s**]	Toggle, activate (**a**), or deactivate (**s**) the overall state of the assertions mechanism.
Set Breakpoint	[*line*] **b** [*cmds*]	Set breakpoint at line **b**.
	[*exp*] **b**[**i**] [*cmds*]	Set breakpoint at address *exp*.
	[*stack*] **b**{**u** | **U** | **b** | **B**} [*cmds*]	Set breakpoint at procedure return point (**bu**) or beginning (**bb**).
List Breakpoints	**B**	List current breakpoints.
Continue	[*exp*] **c**[**i**] [*line*]	Continue from breakpoint with pass count *exp*, to line *line*. **i** means continue in interactive mode.
	[*stack*] **c**{**u** | **U**}	Continue from breakpoint to procedure return point.
Display Alias	**da** [***** | *alias*]	Shows the name and replacement text for one or all currently defined aliases.
Display Options	**do** [***** | *string* | *cfg_opt*]	Display the configuration options.
Delete Breakpoint	**D**	Delete all breakpoints.
	[*number*] **d**	Delete breakpoint *number*, or current breakpoint.
Enter Procedure	**e** [*proc* | *file*]	Enter procedure *proc* or file *file* within the current execution context.
	[*stack*] **e**	Enter active procedure at stack *stack*, or print the current file, procedure, and line number (**e** alone).

Name	Syntax	Description
Enter Alias	**ea** *alias cmd_list*	Creates an alias (synonym) for a list of one or more commands.
Enter Options	**eo** *cfg_opt* **=** *value*	Provides a mechanism to configure the operation of the debugger and emulator.
Address Format	**f** ["*printf-format*"]	Change address display format.
Open File	**fo**	Opens a new COFF or ELF file to debug.
File Read	**fr c** *file_name* [*p_value*] **fr m** *file_name* [*addr*] **fr** { **I** | **RD** | **TD** | **TF** | **TS** } *file_name*	Reads in a command file. Reads in memory binary image file. Reads in an initialization, trace display, trace format, or trace specification file.
File Write	**fw** [**a** | **o**] { **c** | **o** } *file_name* **fw** [**a** | **o**] **m** *file_name range* **fw** [**o**] { **I** | **TD** | **TF** | **TS** } *file_name* **fw** { **c** | **o** } { **-** | **+** }	Opens a command file or output file for writing. Writes out a memory image file. Writes out an initialization, trace display, trace format, or trace specification file. Temporarily suspends (**-**) or resumes (**+**) logging output to a command or output file.
Fix-it	**F**	Find and fix bug.
Go from line	**g** *line*	Go from line *line* (in the current procedure).
Goto	**goto** *label*	Used to change the order of command execution when playing back commands from a command file via the **fr** command.
List History	{ **h** | **history** }	List the 20 most recently entered commands.
Help	**help**	Provides online help for EDB.
If	**if** *exp* { *cmds* } [{ *cmds* }]	Conditionally execute commands.
Indent	**i** *number*	Sets the indent level (tab stop) to every number columns.
Info	**I**	Displays status information.
Kill Program	**k**	Terminates the current program, so that it can be restarted from the beginning.
Kill Alias	**ka** { ***** | *alias* }	Deletes the name and replacement text for one or all of the currently defined command aliases.
Load File	**lf**	Downloads the target program.

Name	Syntax	Description
List objects	**l** {**a** | **b** | **d** | **f** | **g** | **p** | **r** | **s** | **t**} [*string* | *****] [*/format*]	List assertions, breakpoints, directories, files, globals, procedures, registers, specials, or typedefs.
	l [*proc* | *stack*]	List all parameters and locals of a procedure.
Snap	**L**	"Snaps" the viewing point back to the current context's execution location. Same as **0 e**.
MON subsystem	**mon**	Invokes MON subsystem.
Number Format	**n** *number*	Set the default number base used for displaying integer values.
Print source	[*line*] [**p** [*exp*]]	Print *exp* source lines starting at *line*.
	[**+** | **-**] [*exp*]	Print line *exp* lines before (**-**) or after (**+**) current line.
	[*line*] {**w** | **W**} [*exp*]	Print "window" of source lines surrounding *line* or the current line.
Quiet	**Q**	Quiet breakpoint reporting.
Quit	**q**	Quit debugger.
Run	**R**	Run target program with no arguments.
	r [*arguments*]	Run target program with specified or previous arguments.
Shift / Unshift	**shift** [*number*] **unshift** [*number* | *****]	Changes the correspondence between the arguments supplied on an **FR C** *filename* command and the parameter strings within the command file.
Step	[*exp*] {**s** | **S** | **si** | **Si**}	Single-step, following (**s** and **si**) or not following (**S** and **Si**) procedure calls.
Stop	**sp**	Halts a currently executing program in interactive mode.
Trace stack	[*exp*] {**t** | **T**}	Display a stack trace with (**T**) or without (**t**) locals.
Directory	**u** {*string* | *"string"*}	Add directory *string* to search list.
Comment	**v**	Call external editor with current source module (if any) and line number as arguments. See the **edit_callout** option in the Option Settings dialog for details on the setup of the editor callout configuration string. (See *Option Settings Dialog* on page 102.)

Name	Syntax	Description
View Context	**vc** [*exp* | **exec** | *thread _name*]	This command is used to display context information or change the current global view context. By default the global view context is set to the execution context when execution stops for any reason.
Exit assertion	*exp* **x**	Force an exit from assertion mode, possibly finishing the command list.
Comment	**Y** *comment_text*	Introduces a full line comment.
Toggle case	**Z**	Toggle case sensitivity in searches.
Cmd playback	<[<] *file*	Read commands (possibly with single step) from *file*.
Cmd recording	> [*file* | **t** | **f** | **c**]	Record commands to *file*, or turn recording on, or off, or close recording file.
Output recording	>> [*file* | **t** | **f** | **c**]	Record screen output to *file*, or turn recording on, or off, or close recording file.
String search	{/ | **?**}*string*	Search forwards (**/**) or backwards (**?**) in file for *string*.
Print string	"*any string*"	Print the string.
Evaluate expr	*exp* [{/ | @} *format*] ^ [*format*]	Display value of expression using specified format (or **/n**). Re-display value of previous object
Quiet Mode	{+ | -} **q**	Enables (**+**) or disables (**-**) quiet mode of command file playback when using the **fr c** command.
MON Mode	{+ | -} **mon**	Enables (**+**) or disables (**-**) the MON command sub-system regardless of the current command mode. Useful in command files that ensure a particular mode of operation.
EDB Mode	{+ | -} **edb**	Enables (**+**) or disables (**-**) the EDB command sub-system. Useful in command files that ensure a particular mode of operation.
Execute history	**#**[# | *number* | *string*] [*string*]	Re-executes a previous command.
Edit history	**%**[% | *number* | *string*] [*string*]	Edit a previous command.
Shell	**!** [*command-line*]	Execute operating system command shell.
Comment	//[*text*]	Starts a C++ style comment.

Viewing Commands

The commands in this section pertain to displaying source code and debugger or target program objects. All the commands that display source code move the current viewing point to a new location in the source file. There are five viewing commands:

- *Enter Procedure*
- *List Objects*
- *View Context* on page 45
- *String Search* on page 46
- *Trace Stack* on page 46
- *Evaluate Expression* on page 46

Enter Procedure

Syntax:

```
e [proc|file]
[stack]e
L
```

Description: This command is used to display or change the current procedure. The first form is used without an argument (**e** by itself) to display the current file, procedure, and line number. For example, **`test.c:PrintLine:28`**.

The first form is used with *`proc`* or *`file`* to change the current *viewing point* to the first executable source line in *`proc`* or the first line in *`file`*. The selected line is displayed.

The third form (**`L`**) is a synonym for **`0 e`**, which is a quick way to display the next line to be executed.

In all cases, the Execution Window will be updated to display the new context point. (See *Execution Window* on page 91.)

NOTE: This command does not change the global view context.

List Objects

Syntax:

```
l {a|b|d|f|g|p|r|s|t} [string|*] [/format]
l [proc|stack]
```

Description: The list command is used to display any number of things. The first form is used to list, depending on the following letter:

a	Assertions
b	Breakpoints
d	Directories to be searched for source files

f Source files for which debug information is available, or all matching procedures if *string* or ***** is specified

g Global variables

p Procedures for which debug information is available, or all matching procedures if *string* or ***** is specified. The procedure instruction mode is also displayed (**EXEC** or **EX16** for MIPS16 and Thumb).

r Register names and values (can be a very long list).

s Special EDB variables

t Typedef definitions known to EDB.

string and *****
are valid with all except **a**, **b**, and **d**. If string is supplied, only those entries in the list with the same initial characters are displayed. For example, **l g list_** will display only those global variables that begin with **list_**, and **l p get_** will display all procedures that begin with **get_**. ***** is equivalent to no string (all names match) except when used with **l p** or **l f** to include procedures or files which were not compiled with source level debug information.

/format is valid only with **r** and **g** (registers and global variables), and is provided since register contents will often be desired in hex format. Normally, all data is displayed using the **/n** format.

The second form is used to list all the local variables and parameters of the current procedure, or the specified procedure if *proc* or *stack* is supplied. This requires an active target program. Depending on your target processor type, you may need to have executed past the procedure prologue for this to work. For instance, if you set a breakpoint directly on the procedure entry point (via *procname* **b**) you may need to do a source level step before using **l** to list locals.

View Context

Syntax: vc [*exp* | **exec** | *thread _name*]

Description: This command is used to display context information or change the current global view context. By default the global view context is set to the execution context when execution stops for any reason. The **vc** command support two types of conceptual contexts (CPU contexts for chips with multiple register banks, and RTOS based contexts for threads).

If no arguments are given, then the command simply displays the global view context, execution context, and the range of contexts available for selection. If *exp* is given, then the context is set to this CPU context number. The keyword **exec** is used to identify the current execution context. *thread_name* is the name of a particular thread whose context is to be selected. Note that *thread_name*s are only supported if a suitable RTOS_API.DLL has been set up to match your operating system.

If a change of global view context is made, all of EDB's windows will refresh their data contents. Changing the context affects the display of the general (r0 - r31) and PC registers. This generally results in a different call stack walk-back (C command). Note that this command only affects the view context and does not change the actual execution context. A Step or Go command following a context change does not start from the location display by the PC register. Instead it starts from the execution context's PC register.

String Search

Syntax: `{/|?}`*`string`*

Description: Search through the current file for *string*. If **/** is used, the search is forward, starting with the line after the current line. If **?** is used, the search is backward, starting with the line before the current line. Searches wrap around the beginning or end of the file, and obey the current case sensitivity setting (see *Miscellaneous Commands* on page 64). If a string is not given, the previous one is used. If a match is found, the current line is changed to the line containing the match and it is displayed.

Trace Stack

Syntax: [*exp*] {**t**|**T**}

Description: This command is used to display a stack trace for the first exp levels (default 20). If **T** is specified, the display includes local variables using the default format **/n**. Of course, there must be a currently active target program in order to have a stack to trace.

The Trace stack display begins with the currently executing procedure at level zero, and proceeds back up the call stack until the top is reached or the specified number of levels have been displayed. Each line of the display includes the stack level and the procedure name and arguments. The stack level displayed by this command is the source of the stack argument in Enter and Breakpoint commands.

Evaluate Expression

Syntax: *exp* [{/|@} *format*]
^ [*format*]

Description: Any expression appearing by itself, or followed by an optional display-formatting modifier, is evaluated by EDB and the result is displayed.

Be careful of expressions starting with one letter variable names, as they may be taken as a command rather than as a variable. If an expression begins with a variable that might be mistaken for a command, just eliminate any white space between the variable and the first operator. For example, use **k= 9** instead of **k = 9**, use **p/n** instead of just **p**. Alternately, expressions can be wrapped in parentheses. To differentiate an expression beginning with unary minus from the *-exp* command described in *Record and Playback Commands* on page 57, use (*-exp*).

If *exp* is followed by */ format*, the value of the expression will be displayed using the specified format. For example, **foo/x** would print the contents of **foo** as an integer, in hexadecimal.

If *exp* is followed by *@ format*, the address of the expression will be displayed using the specified format. For example, **foo@o** would print the address of **foo** in octal.

The second form is used to back up to preceding memory location (based on the size of the last thing displayed). If *format* is not supplied, the previous format will be used. This form should only be used if the preceding expression evaluation produced an addressable result. For example, **foo** or **array[10]**, not **foo + 10** or **array[10] * 3**.

NOTE: Expression evaluation allows dumping of arbitrary regions of memory. A single location is displayed by de-referencing its address. A block of locations are displayed by using a *format* that includes a count. For example, ***0x28000/20x** will display the 20 words starting at address 0x28000 in hex.

Expressions can also be continually watched by entering them in the Watch Window. (See *Watch Window* on page 115.)

Program Control Commands

There are seven program control commands:

- *Load File* on page 48
- *File Open* on page 48
- *Run* on page 48
- *Continue* on page 49
- *Go From Line* on page 49
- *Kill Program* on page 50
- *Step* on page 50

Load File

Syntax: **lf**

Description: This command causes the target program to be downloaded. Only those section types specified in the Load Options dialog box (accessed by menu: Exec->Load Options) will be downloaded.

File Open

Syntax: **fo**[*file*]

Description: This command is used to open a new executable COFF or ELF file to debug. All the symbolic information associated with your current program will be deleted, and the new symbols will be loaded. Note that this command will not download the program to the target system memory until you issue a command to load the file or to begin execution. When you open a new file, EDB will give you the option of deleting your current source directory search paths, or appending to them.

Run

Syntax: **r**[**i**] [*arguments*]
R[**i**]

Description: This command is used to initialize static and global data and begin execution of the target program from the current Program Counter location. If the target program is already active, it is terminated and restarted, re-initializing all data areas.

The first form is used to run or re-run the program with arguments. *arguments* is the set of command line arguments that are to be passed to the program, not including the program name. If *arguments* is not specified, those specified by the previous command, if any, are used again.

If you have not yet started program execution, the **r** command is the same as clicking on in the Execution Toolbar. (See *Execution Toolbar Command* on page 80.)

The second form is also used to run or re-run the program, but with no command line arguments (**argc = 1**, **argv[0] = <program name>**, **argv[1] = NULL**).

The **i** option starts execution in interactive mode. This mode is only available if your target supports concurrent debug and allows a subset of debugger commands to be used while the target is executing. The **sp** (Stop) command interrupts the running program and returns EDB to normal debug mode. Note that EDB's interactive mode command prompt differs from the normal prompt (for example, EDB(r) means the target is running).

Continue

Syntax:

```
[exp]c[i] [line]
[stack]c{u|U}
```

Description: This command is used to continue program execution after a breakpoint or an interrupt.

In the first form, if *exp* is given, the current breakpoint will have its pass count set to this value. If line is given, a temporary breakpoint is set at that line number. This is shorthand for the pair of commands:
`line` **`b;`** [*`exp`*] **`c`**.

The **`i`** option starts execution in interactive mode. This mode is only available if your target supports concurrent debug and allows a subset of debugger commands to be used while the target is executing. The **`sp`** (Stop) command interrupts the running program and returns EDB to normal debug mode. Note that EDB's interactive mode command prompt differs from the normal prompt (for example, EDB(r) means the target is running).

The second form is used to set an up-level (step-out) breakpoint before continuing. This form is equivalent to the pair of commands [**`stack`**] **`b`** {**`u`**|**`U`**}**`;`** **`c`**. Thus **`cu`** sets a sticky breakpoint while **`cU`** sets a non-sticky breakpoint. **`cU`** is especially useful if you accidentally single-stepped into a procedure you meant to step over, or you want execution to proceed to some place further up the stack. Note that interactive (concurrent) mode is not supported for up-level (step-out) breakpoints.

The **`c`** command by itself is the same as clicking on in the Execution Toolbar, if you have already started program execution. Similarly, **`cU`** is the same as clicking on in the Execution Toolbar. (See *Execution Toolbar Command* on page 80.)

Go From Line

Syntax: **`g`***`line`*

Description: Go from the specified line. This changes the program counter so that line *`line`* is the next execution point, where execution will begin when the next Continue command is given. This command is commonly used within an **`if`** command in a breakpoint command list to conditionally affect program execution. For example, to force exit from a loop that is not terminating normally.

NOTE: While *`line`* is file relative, it must refer to a line in the currently executing procedure. Also, this command should not be used if the procedure prologue has not yet been completely executed or if part of the procedure epilogue has already been executed. Also note that there is no interactive (concurrent mode) form for the **`g`** command.

Kill Program

Syntax: **k**

Description: Terminates the current program, so that it can be restarted from the beginning by a Run or Step command. The Run command will automatically kill the current program, so this command is needed only in the unlikely event that you want to restart execution using the Step command.

After a Kill command, restarting execution by either method will cause the program to be downloaded to the target system. This is because the data areas need to be re-initialized as required by the C language.

The Kill command is seldom used, and is retained for compatibility reasons. A more natural alternative is to use the Load File (**lf**) command or button to kill the current program and re-download in one step.

Step

Syntax: [*exp*] {**s**|**S**|**si**|**Si**}

Description: Single-steps the program by exp source lines or machine instructions. If *exp* is not supplied, the default is 1. Successive <CR>'s will repeat with a count of 1. **s** and **S** step by source statements and display the next statement to be executed. **si** and **Si** step by machine instructions and display the next instruction to be executed in disassembled form. **s** and **si** step into called procedures, except that **s** will not step into a procedure for which debug information is not available. If you accidentally step into a procedure you do not care about, use the **cU** command to set a temporary up-level break and continue. **S** and **Si** step over called procedures, executing calls as a single statement.

See the *Execution Toolbar Command* on page 80 for button equivalents for these commands.

Breakpoint Commands

EDB provides a number of commands for setting and removing code breakpoints. Other than its address, a breakpoint can have many other items associated with it. These are shown below in the *b_opts* syntax description. Breakpoints also may be temporary. All temporary breakpoints are deleted the next time execution stops, whether they were taken or not. Although these Session commands are available, most people choose to use the Break Points dialog (see *Break Points Dialog* on page 86) or the breakpoint control mechanisms.

count is the number of times the breakpoint location must be reached before the break is actually taken. *count* is decremented each time. If *count* is negative, the breakpoint will be deleted after it is taken. If *count* is greater than 1, it is decremented each time the breakpoint instruction is executed. Once *count* reaches 1, the break will be taken and *count* will remain 1 until explicitly set to a different value by a **c** command. A *count* of 0 is used by the single step mechanism and means that the breakpoint is to be deleted the next time the program stops for any reason. Currently, the only way *count* can be set using the command language is by specifying it on the **c** command that continues from a breakpoint.

b_opts The breakpoint command supports a large number of options that configure additional breakpoint qualifiers. Note that these option are really intended to be used by EDB in saving and restoring breakpoints. Users setting their own complex breakpoints will find the Break Points dialog a much easier way to setup these kind of breakpoints (see *Break Points Dialog* on page 86). The syntax is shown below with a brief option description of each type.

```
{
 c {cmds} |
 s {8|16|32|64} |
 {ae|am|asid|ce|p|v|ve|vm} {exp|{exp}} |
 {ba|dis|dr|drw|dw|hwi|i16|tp}
} [b_opts]
```

c The c command specifies an optional command list. See *cmds* below for more details.

s Specifying a size limits the breakpoint to stopping only when an access of the specified size occurs.

ae Works in conjunction with the breakpoint address to provide a range of addresses to break on.

am Allows the supplied Location/Addr to be masked, which is useful for breaking on ranges and/or multiple memory segments (high address masking). The default value for this field is `0xffffffff.ffffffff`.

asid Allows an ASID qualifier to be specified. Typically, the ASID field is only available for targets which have dynamically mapped memory (TLBs).

`ce` You can make the current breakpoint conditional by entering an arbitrary expression with this option. All variables used in the condition must be active at the breakpoint location

`p` Allows you to specify how many times the breakpoint location must be hit before the program is actually halted.

`v` The value to match before the break hits.

`ve` Works in conjunction with the breakpoint value match (`v`) to provide a range of value to break on.

`vm` Allows the supplied breakpoint value match (`v`) to be masked, which is useful in breaking on interesting ranges of values (like odd numbers, etc). The default value for this field is `0xffffffff.ffffffff`.

`ba` Identifies the breakpoint address as a bus address rather than a normal mapped or virtual address. This option is only valid on data (`dw`, `dr`, `drw`) and hardware instruction (`hwi`) breakpoints.

`dis` Identifies the breakpoint as initially disabled.

`dr`, `dw`, `drw` Identifies the breakpoint as a data read, write or read / write break.

`hwi` Identifies a hardware level instruction breakpoint. Useful for setting instruction breakpoints in ROM areas because the location does not need to be writable.

`i16` Identifies an alternate (16 bit) instruction mode breakpoint (examples of architectures where this is valid: MIPS16 and ARM's Thumb mode).

`tp` Changes the break point into a trace point. A trace point does not stop the target. Instead it triggers the processors trace trigger features. This option is only valid on data (`dw`, `dr`, `drw`) and hardware instruction (`hwi`) breakpoints.

`cmds` contains EDB command(s) to be executed when the break is taken. Individual commands are separated by a semi-colon (**`;`**) and may be enclosed in braces (**`{}`**) to delimit the list given to the breakpoint when some other command follows the **`b`** command. If the first character is anything other than open brace (**`{`**), the rest of the line is given to the breakpoint as commands. If no commands are specified when the breakpoint is set, EDB will simply interrupt execution of the program and prompt for more user commands. If cmds is

specified, the command list may include the c command to cause program execution to continue immediately. For example: **`Third b Second b {}; c`** will set a breakpoint at procedure **`Third`** with the command list **Second b {}; c**. When the break is taken, the command list will set a breakpoint at **`Second`** and then continue execution.

NOTE: EDB's Windows interface now provides a more powerful and easier to use breakpoint model. Pass counts can be set when the breakpoint is created and reset each time the break is taken. Also, the breakpoint can be made conditional directly, rather than relying on an **`if`** command in *`cmds`*. See *Break Points Dialog* on page 86 for more details.

There are several breakpoint commands:

- *Set Breakpoint*
- *List Breakpoints* on page 54
- *Delete Breakpoint* on page 54

The following commands, while not strictly part of the breakpoint group, are used almost exclusively in the command list of breakpoints.

- *If* on page 55
- *Print Source* on page 55
- *Quiet* on page 55
- *Print String* on page 56

Set Breakpoint

Syntax:

```
[line] b [b_opts]
[stack]b{u|U|b|B} [b_opts]
[exp] b[i] [b_opts]
```

Description: Sets a breakpoint at a source line or an arbitrary address.

The first form is used to set a breakpoint at a particular source line. If *`line`* is not specified, then the breakpoint is set at the *current line*. Be aware that the C compiler may not associate the selected line with a code address. In this case, the breakpoint will actually be set at the first following line that corresponds to generated code. See *EDB Caveats* on page 15.

The second form is used to set a breakpoint at a procedure's return point or entry point. The procedure is identified by its call stack level, so you must have a currently active program to use this form. The type of break is specified by the second character of the command (following the **`b`**). An upper case letter causes the breakpoint to be temporary, rather than permanent.

Specifying **bu** or **bU** sets a permanent or temporary up-level breakpoint that will be taken immediately on return to the specified level. If *stack* is not supplied, the current procedure's caller is assumed. This is commonly done after accidentally single-stepping into a procedure. The commands **bU ; c** will execute the remainder of the current procedure at full speed, breaking on return to the caller ready for further single-stepping. The command **cU** accomplishes the same thing.

Specifying **bb** or **bB** sets a permanent or temporary breakpoint at the beginning of the specified procedure. The breakpoint is set after the procedure prologue, at the first executable line. If *stack* is not supplied, the current procedure is assumed.

The third form is used to set a breakpoint at an arbitrary machine instruction. *exp* may be any legal expression, especially one involving procedure names or other text labels. If no expression is given, then the address of the last thing you looked at with the **/i** or **/I** display mode will be used. For example, **printf+0x14 bi** would set a break 20 bytes into procedure **printf**.

NOTE: On targets that have multiple execution modes (that is, MIPS16 and ARM's Thumb mode) the breakpoint type is automatically. determined from symbolic information loaded from the *program*.**cdb** file.

See *Execution Window* on page 91 for information on setting, clearing, and editing breakpoints with the mouse.

List Breakpoints

Syntax: **B**

Description: All breakpoints are listed. This is the same as the **l b** command. Each breakpoint is identified by procedure name and relative line number, along with its corresponding index number, enabled / disabled status, breakpoint type, current pass count (if any), and command list.

Example:

```
0 E EXEC foo:12  -1 <t;i/D>
1 E EX16 bar:29   5 <Q;if *argv==1 {"Trashed argv again"}{c}>
```

Breakpoint type is one of **EXEC** (normal execution breakpoint), **EX16** (MIPS16 or ARM-Thumb type breakpoint), and **HARD** (hardware based breakpoint).

Delete Breakpoint

Syntax: **D** | { [*number*] **d** }

Description: This command deletes one or more breakpoints. The **D** command deletes all breakpoints. The **d** command deletes a specific breakpoint identified by the index number listed by the **B** command. If *number* is not supplied,

d will try to delete a breakpoint at the current line. If no breakpoint is found, all current breakpoints will be listed.

If

Syntax:

```
if exp {cmds} [{cmds}]
```

Description: Conditional command execution. If exp evaluates to a non-zero value, the first group of commands will be executed. Otherwise, the second group, if present, will be executed. This command is nest-able (that is, another **if** command can appear in either command list). It is also useful in command files, where the **GOTO** command can be used to conditionally alter the flow of control.

Print Source

Syntax:

```
[line] [p [exp]]
[+|-] [exp]
[line] {w|W} [exp]
```

Description: Display one or more lines of source code. These commands are seldom used in windowing versions of EDB, since they display the source lines in the Session Window and do not affect the Execution Window except for moving the viewing point indicator.

The first form displays *exp* lines (default 1), starting from line *line* (default is the current line). The viewing point is moved, with the new current line becoming the last line displayed.

The second form moves the viewing point to *exp* (default 1) lines before (-) or after (+) the current line, and displays the new current line.

The third form optionally moves the viewing point to *line*, and displays a window (*exp* lines big) of text centered around the new current line. The default for *exp* is 11 lines for **w**, and 22 lines for **W**. This command is used most frequently without arguments to display some context around the current viewing point, which is usually the current execution point.

Quiet

Syntax:

```
Q
```

Description: EDB normally indicates when a break has been taken by displaying the breakpoint (*procedure*:*linenumber*: *source_line*). If this command appears as the first command in a breakpoint's command list, the normal announcement is not made. This allows quiet checks of variables, and so on to be made without cluttering up the screen.

Print String

Syntax: `"any string"`

Description: Prints the string. The string may have the standard C character escapes in it. This command can be useful for commenting things in breakpoint commands

Assertion Commands.

Assertions (a slight misnomer) are lists of commands that are executed before every statement. This means that if there is even one active assertion, the program will be automatically single- stepped. This means it will run very slowly. A primary use for assertions is tracking down bugs involving someone stepping on a global variable (or others). Some examples will follow the command descriptions.

The following assertion commands are described below:

- *Create Assertion*
- *Modify Assertion*
- *Exit Assertion*

Create Assertion

Syntax: **a** *cmds*

Description: Create a new assertion with the given command list. Assertions, like breakpoints, are assigned a reference number in the order they are created. See the examples under *Exit Assertion* below.

Modify Assertion

Syntax:

```
exp a {a|d|s}
A [a|s]
```

Description: The first form changes the current state of assertion number *exp*. Depending on the letter following *exp* **a**, the assertion will be: activated (**a**), deleted (**d**), or suspended (**s**). Suspended assertions continue to exist but are not in use (that is, their command lists are not executed).

The second form changes the current state of the overall assertion mechanism. Depending on the letter following **A**, the assertion mechanism will be: toggled (no parameter), activated (**a**), or suspended (**s**). This is a quick way to turn off assertions temporarily without having to suspend each individual assertion.

Exit Assertion

Syntax: *exp* `x`

Description: Force an end to assertion processing. If no expression is present, or if it evaluates to 0, then exit immediately, otherwise the current assertion finishes executing. If any assertion executes an **`x`** command, the program will be stopped and the assertion doing the **`x`** will be identified.

Example:

```
a if (foo!=$foo) {$foo=foo; foo/d; if (foo>9) {x}}
```

This command will create an assertion to display the value of some global variable (**`foo`**) whenever it changes, and stop if it exceeds some value. It uses a debugger special variable to keep track of the old value of **`foo`**.

```
a L; if (foo > (bar-9)*10) {A; 1 x; c} {bar -= 10}
```

This assertion prints the line about to be executed, then checks the condition. If it is false, **`bar`** is decremented by 10. If it is true, assertions are suspended, assertion mode is ended, and the program continues at normal speed. Without the non-zero number before the **`x`** command, the **`c`** command would not have been seen and the program execution would not have continued.

Record and Playback Commands

EDB contains a record and playback feature to help recreate program states. This is particularly useful for bugs requiring long setups.

Although an attempt has been made to do things reasonably, it is possible to fake out the recording mechanism. Be particularly careful about trying to playback from a file currently open for recording or vice-versa. This may cause unpredictable results.

The following record and playback commands are described below:

- *Command Recording* on page 58
- *Output Recording* on page 58
- *Command Playback* on page 58
- *Set Quiet Mode* on page 58
- *Set Command Sub-System Mode* on page 59
- *File Read* on page 59
- *File Write* on page 60
- *Goto* on page 62
- *Shift / Unshift* on page 62

Command Recording

Syntax: `>` [*`file`*|**`t`**|**`f`**|**`c`**]

Description: If no parameter is specified, report the current command and output recording status. If *`file`* is given, set the command recording file to *`file`* and activate recording. Otherwise turn command recording on (**`t`**) or off (**`f`**), or close the current recording file (**`c`**). To avoid confusing (or even recursive) results, any command line beginning with > or < will not be placed in the current recording file. This can be overridden by simply beginning the line with a space.

Output Recording

Syntax: `>>` [*`file`*|**`t`**|**`f`**|**`c`**]

Description: If no parameter is specified, report the current command and output recording status. If *`file`* is given, set the Session Window output recording file to *`file`* and activate recording. Otherwise turn output recording on (**`t`**) or off (**`f`**), or close the current recording file (**`c`**).

Command Playback

Syntax: `<`[`<`] *`file`*

Description: Starts command playback from file *`file`*. If **`<<`** is used, the single-stepping feature of command playback is used (see the **`-Z`** *`N`* command line option). When the end of the playback file is reached, commands are again read from the console.

Command playback files can be nested up to 20 levels deep. When the end of a nested file is reached, command execution resumes with the next command in the calling file.

If you wish to pass arguments to a command file, you must use the **`FR C`** command.

Set Quiet Mode

Syntax: {`+`|`-`}**`Q`**

Description: This command is used to enable (**`+`**) or disable (**`-`**) the Quiet mode of command file playback. Normally, debugger prompts and commands read from the command file are displayed just as they would be if the commands were entered from the keyboard. But when Quiet mode is active, the debugger does not display prompts and commands while reading commands from a file. Note that Quiet mode is automatically turned on while a command alias is being executed. When the alias command is finished, the original state of Quiet mode is restored.

Note that Quiet mode of command file playback can only be used when a command file is read back via the **FR** command.

Set Command Sub-System Mode

Syntax: **{+|-}{mon|edb}**

Description: These commands are used to enable (**+**) or disable (**-**) a particular command sub-system. They are most useful in command files that need to ensure a particular mode of operation to allow seamless operations from either command mode. For example a command file consisting of:

```
+mon
dw pc
-mon
```

will work from either command mode and restore the command mode that was active with the command file was started.

File Read

Syntax:

FR C *file_name* [*p_value...*]

FR M *file_name* [*addr*]

FR {I|RD|TD|TF|TS} *file_name*

Description: This command is used to read files of various types. The type of file is specified by the first operand. The valid file type operands for a File Read are described below:

Operand	File type	Default Extension
C	Command file	**.cmd**
M	Memory binary image or S-Record file	**.mem**
RD	Register Definition file	**.rd**
TD	Trace Display file	**.td**
TF	Trace Format file	**.tf**
TS	Trace Specification file	**.ts**

file_name
is the name of the file to be read. If *file_name* does not include an extension, the debugger will supply the default file name extension. If no extension is desired, *file_name* should end with a period (**.**), which will be removed before opening the file.

addr The starting address where the memory image will be loaded. It is not necessary that this address match the starting address used to write the file. *addr* is optional on memory image files containing Motorola S-Records. If given, *addr* specifies a load address relative to the addresses in the S-Record file.

p_value An argument to be passed to the Command file. Each *p_value* is an arbitrary string of text delimited by white-space (blank or tab). When each line of the command file is read in, it will be scanned for parameter strings of the form **$*** or **$***n*, where *n* is a one or two digit decimal number. **$0** will be replaced with the number of arguments, **$1** will be replaced with the text of the first argument, **$2** with the second, etc. The **shift** and **unshift** commands can be used to change which argument string parameter **$1** actually refers to, making it possible to process a variable number of arguments in a loop. **$*** will be replaced with the entire list of arguments from the File Read command. The replacement text can be pasted into a larger token by using \ as a delimiter character. For example, **$1\text**, **my\$2\ident**, **temp\$3**.

Note that for MIPS targets, you must use **$$***n* instead of just **$***n* since **$***n* conflicts with certain register names.

Note that command files may contain File Read commands that execute other command files. Command file reads may be nested up to 20 levels deep. If an **FR C** command is not the last command of a multi-command line, the rest of that command line will be executed after the contents of the command file.

Remember that empty lines in a command file are equivalent to hitting <Enter> at the debugger prompt. That is, they may cause the previous command to be repeated if it was a repeatable command.

When **FR TS** is used to read a trace specification, all pre-existing conditions, events, states, and filters are first killed. But if **FR C** is used instead, the new conditions, events, and states are added to the current trace specification, overwriting when appropriate. **FR TF** does not kill pre-existing trace formats, it just adds to them, overwriting when appropriate.

File Write

Syntax:

```
FW[O]   {I|TD|TF|TS} file_name
FW[A|O]      M       file_name range
FW[A|O] {C|O}        file_name
FW      {C|O} {-|+}
```

Description: This command is used to write files of various types. The type of file is specified by the first operand. The valid file type operands for a File Read are described below:

Operand	File type	Default Extension
`C`	Command file	`.cmd`
`M`	Memory image file	`.mem`
`O`	Output capture file	`.out`
`TD`	Trace Display file	`.td`
`TF`	Trace Format file	`.tf`
`TS`	Trace Specification file	`.ts`

`[A|O]` Normally, if the file specified exists, you will be prompted for permission to overwrite or (for command and output files) append to it. To avoid that prompt, use the **`FWA`** command to append without prompt or the **`FWO`** command to overwrite without prompt.

`file_name`

is the path and filename of the file to be written, relative to the current working directory. If *`file_name`* does not include an extension, the debugger will supply the default file name extension. If no extension is desired, *`file_name`* should end with a period (`.`), which will be removed before opening the file.

`range` specifies the region of memory (or block of registers) to be written to the Memory image file.

`{+|-}` Once writing to a Command or Output file has been initiated, output may be temporarily suspended with minus (`-`) and later resumed with plus (`+`).

Doing a File Write to the Output (**`O`**) file type causes each line printed to the console (including the echo of commands entered) to be logged into the specified file. This allows a permanent record to be made of a debugging session. Whereas the Command (**`C`**) file type records only the commands entered, but not the prompts and responses from the debugger. This is a convenient way to create script files that can be used to automate repetitive command sequences or to quickly recreate an interrupted debugging session.

The initialization file saves most debugger invocation options (such as communication port and target endianness) as well as the current configuration of the **`MC`** and **`OM`** commands. Memory image files contain raw binary data uploaded from the target. They normally represent the contents of some range of memory at the time they were created, but they can also contain a dump of the processor's register contents. The file contains no control information (such as the original address *`range`* written to the file), so a Memory file can be written from one location and later read back into a different location. Writing a Trace Display file also suspends tracing, resets the current trace control specification to state 0, and causes any subsequent tracing to begin with a cleared timestamp and trace buffer. If Execution Tracing mode (**+te**) is active, trace control is automatically re-enabled when execution later resumes. But if Trace Always mode is desired, it must be explicitly re-enabled with **+t**.

Goto

Syntax: **goto** *label*

Description: The **goto** command is used to change the order of command execution when playing back commands from a command file. It causes the command file reader to jump to the line following the specified label. Labels are defined in the command file by a line of the form:

```
:label
```

where label is any valid identifier string. The colon does not have to be in the first column of the line, but there must be no white space between the colon and the label. **goto** commands may precede or follow the corresponding label definition. Label definitions and **goto** commands have no effect when reading commands from the console, but they will be saved in a command output file if command logging is in effect.

Note that the **goto** command can only be used when reading command files with the **fr** command, not the **<** command.

Shift / Unshift

Syntax:

shift [*number*]
unshift [*number* | *****]

Description: The **shift** and **unshift** commands change the correspondence between the arguments supplied on an **FR C** *filename* command and the parameter strings within the command file.

Normally, the first argument is substituted for **$1**, the second argument for **$2**, and so forth. The **shift** command increments the argument number that corresponds to each parameter number, effectively shifting the argument array so that a given range of parameter numbers refer to a higher range of arguments. The **unshift** command reverses this effect.

number is the number of arguments to shift or unshift.

***** is valid only for **unshift**, and it restores the arguments so that **$1** again refers to the first argument.

i **NOTE**: The **$0** parameter is also affected by shifting: if there were originally 10 arguments, after a **shift 2** command **$0** will be replaced with 8. **$*** is not affected by shifting, it is always replaced with the entire argument list.

Note that for MIPS targets, you must use **$$***n* instead of just **$***n* since **$***n* conflicts with certain register names.

The **shift** and **unshift** commands can only be used when reading command files with the **fr** command, not the **<** command.

Example:

```
mon
if ($0 < 2) { dv "Expected address and count\n"; goto done }
:loop
    dw $1 L $2
    shift 2
    if ($0 >= 2) { goto loop }
:done
q
```

Argument shifting is very useful when you want to perform the same series of actions repetitively on an unknown number of argument (or groups of arguments). The previous command file displays the contents of a series of ranges using **MON** commands. It expects an argument list of the form:

addr count [*addr count*]...

History Commands

EDB has a powerful history mechanism via the Command Input toolbars in the Session Window and Program I/O Window. EDB also supports an older style history mechanism (shown below) that remembers the last 20 commands. Most users will find the new style history mechanism more convenient to use.

The following history commands are described below:

- *List History*
- *Execute History* on page 64
- *Edit History* on page 64

List History

Syntax: {**history** | **h**}

Description: Displays the current history list (the 20 most recently entered commands). Each command is preceded by a reference number.

Execute History

Syntax: `#[#|number|string1] [string2]`

Description: Re-executes a previous history command. If the first parameter (which must not be separated from the first `#`) is not given, or is `#`, the last command is re-executed. If *`number`* is given, the specified command is re-executed. If *`string1`* is given, the command that starts with *`string1`* is re-executed. In any case, if the second parameter, *`string2`*, is given, it is appended to the command before it is re-executed.

Edit History

Syntax: `%[%| number|string1] [string2]`

Description: This command selects a history command just as with `#`, but it passes the command to a simple line editor before executing it. Directions for its use are available upon invocation.

Miscellaneous Commands

The following miscellaneous commands are described below:

- *Again* on page 65
- *Shell* on page 65
- *Display Alias* on page 65
- *Display Configuration Options* on page 65
- *Enter Alias* on page 66
- *Enter Configuration Option* on page 66
- *Address Format* on page 67
- *Fix-It* on page 68
- *Indent (tab size)* on page 68
- *Info* on page 68
- *Kill Alias* on page 68
- *MON Subsystem* on page 68
- *Number Format* on page 69
- *Quit* on page 69
- *Source Directory* on page 69
- *v* on page 69
- *Yak (comment line)* on page 70
- *Toggle Case* on page 70

Again

Syntax: **`<CR>`**

`<Ctrl-D>`

Description: <CR> will attempt to repeat the last command, possibly with an appropriate increment. <Ctrl-D> is similar, but will repeat the command 10 times. Neither form will have any effect if the previous command is not obviously repeatable.

Shell

Syntax: **`!`**[*`command-line`*]

Description: Invokes an operating system command shell. If *`command-line`* is present, it is executed and the shell will return immediately; otherwise, the shell returns when exited (via the **`exit`** command on Unix and MS/DOS systems, or **`logout`** on Vax/VMS systems).

Display Alias

Syntax: **`da`** [**`*`**|*`alias`*]

Description: The Display Alias command shows the name and replacement text for one or all currently defined aliases. If the command is entered without a parameter, all aliases are displayed.

`*`	display all aliases. This is the default.
`alias`	the name of a command alias defined with the **`EA`** command.

See the description of the **`EA`** command for more information about creating an alias.

Display Configuration Options

Syntax: **`do`**[**`v`**] [**`*`**|*`string`*|*`cfg_opt`*]

Description: This command displays the configuration options. If the command is entered without parameters, all options are displayed. The **`v`** option displays help text about the given option. If string is supplied, only those *`cfg_opts`* with the same initial characters are displayed. For example, **`do l*`** will display only those configuration options that begin with **`l`**. If **`*`** is specified, all options will be displayed. This is the default.

Note that you can also see and edit EO/DO options via the Option Settings dialog. (See *Option Settings Dialog* on page 102.)

Enter Alias

Syntax: **ea** *alias cmd_list*

Description: This command creates an alias (synonym) for a list of one or more commands. It is normally used to create a short (one or more characters) abbreviation for a longer command or sequence of commands that are frequently needed.

alias identifier
the new command name that is being created or re-defined.

An *identifier* consists of alphanumeric characters plus underscore (_) and dollar sign (**$**), and must start with an alphabetic character or underscore (_). Alias names are not case sensitive.

cmd_list command [*;command*] ...
one or more debugger commands, separated by semicolons. If the last command in *cmd_list* is not complete (missing some parameters at the end) they must be provided when the alias is used.

When a command is being processed, the debugger first checks to see if the command name matches an existing *alias* name, ignoring alphabetic case. If a match is found, the alias name is replaced by the text of *cmd_list*, and the command is re-scanned. If a match is not found, the debugger checks for built-in commands. This means that aliases can be used to re-define existing built-in commands, and the alias replacement text can contain other alias names. Recursive alias references are not supported, however.

If *cmd_list* includes an **FR C** *file* command, the command file will be read in Quiet mode to provide the illusion that the alias name is a built-in command.

Aliases can be displayed with the Display Alias command, and removed with the Kill Alias command.

Example:

```
EA DTS DC;DE;DS        // display trace specification.
ea rc fr c             // read command file without echo.
```

Enter Configuration Option

Syntax: **eo** *cfg_opt* = *value*

Description: This command provides a mechanism to configure the operation of the debugger and emulator. The options available are dependent upon your target environment. For a complete description of each option use the Option Settings dialog or **dov** command to display the help information available for each option.

Wherever a *`cfg_opt`* is called for, either its full name or just its initials (as an abbreviation) may be entered.

Note that you can also see and edit EO/DO options via the Option Settings dialog. (See *Option Settings Dialog* on page 102.) Below are some of the more generic options available.

Full Name	Initials	Description	Range	Default
`dp_color`	**`dc`**	Color or black and white display	**`off`** | **`on`**	**`off`**
`dp_color_output`	**`dco`**	Display Output Color	*`color1`*	**`green`**
`dp_color_input`	**`dci`**	Display Input Color	*`color`*	**`white`**
`dp_color_prompt`	**`dcp`**	Display Prompt Color	*`color`*	**`cyan`**
`dp_color_standout`	**`dcs`**	Display Standout Color	*`color`*	**`yellow`**
`dp_color_backgnd`	**`dcb`**	Display Background Color	*`color2`*	**`black`**
`dp_color_error_msg`	**`dcem`**	Display Error Message Color	*`color`*	**`red`**
`load_absolute_syms`	**`las`**	Include absolute-valued symbols in loads	**`off`** | **`on`**	**`off`**
`calling_convention`	**`cc`**	Function Calling Convention	**`o32`** | **`n32`** | **`o64`**	**`o32`**
`serial_speed`	**`ss`**	Serial Communication Speed	**`0..7`**	**`on`**
`edb_go_interactive_mode`	**`egim`**	Go interactive (concurrent) mode	**`off`** | **`on`**	**`off`**
`edit_callout`	**`ec`**	EDB edit callout configuration	*`string`*	**`cw32 %f -G%l`**
`sym_delta`	**`sd`**	Maximum *offset* in *symbol+offset* address display	**`0..ffffffff`**	**`ffff`**

Address Format

Syntax: **`f`** [`"`*`printf-style-format`*`"`]

Description: This command may be used to alter the default display format for address expressions. *`printf-style-format`* is a conversion specification string as defined by the C printf function. If there is no argument, this defaults to **`%#lx`**, which prints the address in long hex. All addresses are treated as if they are long, so you should handle all 4 bytes to get something meaningful. This is for viewing memory addresses in decimal, octal, etc.

Fix-It

Syntax: `F`

Description: Find and fix bug - just what you've been looking for! Actually, this is CDB's version of "joke of the day".

Indent (tab size)

Syntax: **`i`** *`number`*

Description: Tab characters in the source file will align the text to 8 column boundaries by default. This command can be used to change the tab stops to every *`number`* column. In windowing versions of EDB the new setting will affect the display of source lines in all windows, not just the Execution Window. Note that this setting is saved in an EDB startup file. See the menu option Save Layout for details. (See *Execution Window* on page 91 and *Save Layout Command* on page 72.)

Info

Syntax: `I`

Description: Displays various information about the state of EDB.

Kill Alias

Syntax: **`KA *`**

`KA` *`alias`*

Description: The Kill Alias command deletes the name and replacement text for one or all of the currently defined command aliases. Command aliases are created with the Enter Alias command.

`*`	Remove all aliases.
`alias`	Remove just *`alias`* from the command alias list.

MON Subsystem

Syntax: `mon`

Description: This command invokes a command subsystem that provides a subset of the commands described in the User's Manual for your target system (Simulator, Emulator, etc.). This subsystem has its own help file, and provides complete control of the trace control, profiling, and hardware breakpoint features of EPI emulator's. The Quit command exits the MON subsystem and returns to the standard EDB command interpreter.

Number Format

Syntax: **n** *number*

Description: This command can be used to alter the default number base (radix) used to display integer expressions. The normal default is decimal (**n 10**). Note that this setting is saved in an EDB startup file. See the menu option Save Layout for details. (See *Save Layout Command* on page 72.)

Quit

Syntax: **q**

Description: Exits the EDB program. Before exiting, you are prompted to confirm what you want to do. Answers to the prompt are **y** (Exit, the default), **s** (save breakpoints, assertions, source file search paths, tab size, default radix, and load option settings, then exit), or **n** (stay in EDB). If you select **s**, and the program you are debugging is **foo**, the save file will be **foo.rc**. The commands in this file will be executed automatically the next time you run EDB to debug **foo**.

Source Directory

Syntax: **u** {*string* | *"string"*}

Description: Adds the directory whose path name is given by *string* to the source file search list. Alternate directories will be searched in the order given. If a file is not found in the current directory, the alternate directories are searched in order. To view the current directory list, use the **l d** command. If the *string* includes spaces, *string* must be enclosed in double quotes.

V

Syntax: **v**

Description: This command calls up an external editor supplying the current module and line number as arguments. The editor and options supplied to the editor are configured via an option configuration string. See the **edit_callout** option in the Option Settings dialog for details on the setup of the editor callout configuration string. (See *Option Settings Dialog* on page 102.)

Yak (comment line)

Syntax: `Y` *arbitrary_text*

Description: This command echoes whatever text follows the `Y` to the screen. It can be used to add comments to command files. The Print String command (see *Print String* on page 56) can also be used for this, but the `Y` command behaves a little differently. It is intended to be the only command on the line, and the command file playback system will recognize it and not pause after executing it even if the **-Z***N* option is in effect.

Toggle Case

Syntax: `Z`

Description: This command toggles case sensitivity in source and symbol table searches. This affects everything: file names, procedure names, variables, and string searches. EDB starts out as case sensitive. The new case sensitivity setting is displayed. The current setting is displayed by the Info command.

Menu and Window Reference 4

This chapter describes the EDB menus, windows, and dialog boxes. The menus are described below, and the windows and dialogs are described in *Windows and Dialogs* on page 86.

EDB Menus

EDB uses seven menus:

- *File Menu*
- *Edit Menu* on page 73
- *View Menu* on page 75
- *Exec Menu* on page 82
- *Misc Menu* on page 85
- *Window Menu* on page 85
- *Help Menu* on page 85

File Menu

The File menu offers the following commands:

Program to Debug	Select Program to Open / Debug. Accelerator key: <Ctrl-O>
Save Layout	Save current window layout.
Restore Layout	Restore window layout from save file.
Save Session Info	Save program session data (breakpoints, etc.).
Recent File List	Select Most Recently used Programs.
Exit	Exit EDB.

Program to Debug Command

Use this command to select a program to open or debug. Programs must be COFF or ELF files to be recognized. EDB also requires that a *program*.**cdb** file exist for the given program. You can create this file by running **cdbtrans** on the COFF or ELF executable file as follows:

```
cdbtrans program
```

If EDB cannot find the *program*.**cdb** file or finds that its date is earlier than the COFF or ELF program files date, then EDB will run the **cdbtrans** program for you.

After a valid selection is made EDB will look for a startup session file associated with the program. The session file is normally in the same directory as the program file and is named *program*.**rc**. For more information on this file see *Save Session Info Command* on page 73.

Note that you cannot debug multiple programs at once. Selecting a new program to debug will remove debug information on the previously loaded program.

Shortcuts:

Toolbar:	📂
Keys:	<Ctrl+O>

Save Layout Command

Use this command to save the screen layout in the EDB startup file. The file is normally created and kept in your Windows directory and is composed of the full name of your program (**edbice**, **edbsim**, etc.) with **.ini** tacked on the end (for example, **edb***xxx***.ini**). Note that besides just screen layout, this file also contains your most recently used program list and Memory/Watch Window expressions, status of the various fields in the General Properties dialog (for example, Font type, tab size, display radix, scrollbar settings, etc.), and color setup.

EDB also has an automatic save and restore layout at exit feature that can be enabled or disabled via the General Properties dialog.

To restore a screen layout see the File menu command *Restore Layout Command* below.

Restore Layout Command

Use this command to restore a previously saved screen layout. To save a screen layout or for more details on the file content location see the File menu command *Save Layout Command* above.

Save Session Info Command

This command saves the current breakpoints, source file search path, and assertions in the currently loaded program's session information file `program.rc`. Session files are automatically restored (read) by EDB upon selection of a program to debug. EDB will also prompt you upon exit to save any session information to this file.

Note that program session files contain EDB/CDB commands and can be edited via any text editor.

Recent File List Command

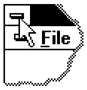
Use the numbers and filenames listed at the bottom of the File menu to select the last four programs you debugged. Choose the number that corresponds with the program you want to debug or open.

Exit Command

Use this command to end your EDB session. You can also use the Close command on the application Control menu. If you have a program loaded, EDB prompts you to save the session information (see *Save Session Info Command* above) and exit (Yes), or simply exit (No), or cancel the exit operation (Cancel).

Shortcuts:

Mouse: Double-click the application's Control menu button.



Keys: <Alt+F4>

Edit Menu

The Edit menu offers the following commands:

Undo Reverse previous editing operation. Accelerator key: <Ctrl-Z>

Cut Deletes data from the document and moves it to the clipboard. Accelerator key: <Ctrl-X>

Copy Copies data from the document to the clipboard. Accelerator key: <Ctrl-C>

Paste Pastes data from the clipboard into the document. Accelerator key: <Ctrl-V>

Select All Select the entire contents of the window. Note that this operation is only valid for source display mode. Accelerator key: <Ctrl-A>

Properties Bring up Properties dialog. Allows display and edit of program configuration.

Undo Command

Use this command to reverse the last editing action, if possible. The name of the command changes, depending on what the last action was. The Undo command changes to Can't Undo on the menu if you cannot reverse your last action. If the Undo option is grayed-out, then Undo is not supported for the last operation.

Shortcuts:

Toolbar: ↶

Keys: <Ctrl+Z> or <Alt-Backspace>

Cut Command

Use this command to remove the currently selected data from the windows/control and put it on the clipboard. This command is unavailable if there is no data currently selected or the window/control does not support editing.

Cutting data to the clipboard replaces the contents previously stored there.

Shortcuts:

Toolbar: ✂

Keys: <Ctrl+X>

Copy Command

Use this command to copy selected data onto the clipboard. This command is unavailable if there is no data currently selected.

Copying data to the clipboard replaces the contents previously stored there.

Shortcuts:

Toolbar: ⧉

Keys: <Ctrl+C>

Paste Command

Use this command to insert a copy of the clipboard contents at the insertion point. This command is unavailable if the clipboard is empty.

Shortcuts:

Toolbar: 📋

Keys: <Ctrl+V>

Select All Command

Use this command to select the entire contents of a window. Note that some windows (at times) may represent an extremely large data set, and thus, full data selection is not supported for these cases.

Shortcuts:

Toolbar:	none
Keys:	<Ctrl-A>

Properties Command

Use this command to view or change the display and program properties. Program properties are broken down into General Properties, Program Properties (properties that relate to the program under debug), and Color Properties. See *Properties Dialog* on page 108.

Shortcuts:

Toolbar:	none
Keys:	none Most Windows Shortcut (right-click pop-up) menus have a properties entry.

View Menu

The View menu offers the following commands:

Session	View Session Window.
Program I/O	Create / View Program Input/Output Window.
Execution	View Execution Window.
Memory	View Memory Window.
Watch	View Data Watch Window.
Registers	Create / View new Registers Window.
Call Stack	View Call Stack Window.
ICE Trace	(ICE targets only) View ICE Trace Window.
Profiler Data	(ICE targets only) View Profiler Data Display Window.
RTOS	View RTOS Window.
Breakpoints	View and edit breakpoints.
Option Settings	View and edit configuration option settings.
ICE Trace Spec	(ICE targets only) View and edit ICE trace specifications.
Profiler Setup	(ICE targets only) View Profiler Setup dialog.

General Toolbar	Show or hide the General toolbar.
Execution Toolbar	Show or hide the Execution toolbar.
Context Toolbar	Show or hide the Context toolbar.
Status Bar	Show or hide the Status bar.

Session Command

Use this command to open or view the Session Window, shown in *Session Window* on page 114. The toolbar allows you to enter commands and recall previous commands.

Shortcuts:

Toolbar: S I/O

Keys: none

Program I/O Command

Use this command to open or view the debugger's Program Input/Output Window, shown in *Program Input/Output Window* on page 107. Input to program user input requests can be entered via this window's toolbar input box.

Shortcuts:

Toolbar: P I/O

Keys: none

Execution Command

Use this command to open or view the Execution Window. See *Execution Window* on page 91 for more details.

Shortcuts:

Toolbar: none

Keys: none

Memory Command

Use this command to open or view the Memory Window. The window provides dumps (in various formats) of memory at a specified address/expression. See *Memory Window* on page 99 for more details.

Shortcuts:

Toolbar: M

Keys: none

Watch Command

Use this command to open or view the variable Watch Window. See *Watch Window* on page 115 for more details.

Shortcuts:

Toolbar:

Keys: none

Register Command

Use this command to open or view the Register Window. See *Register Window* on page 112 for more details.

Shortcuts:

Toolbar:

Keys: none

Call Stack Command

Use this command to open or view the Call Stack Window. See *Call Stack Window* on page 90 for more details.

Shortcuts:

Toolbar:

Keys: none

ICE Trace Command

Use this command to open or view the ICE Trace Window. See *ICE Trace Display Window* on page 96 for more details.

Shortcuts:

Toolbar:

Keys: none

Profiler Data Command (ICE targets only)

Use this command to open or view the Profiler Data window. This menu item is only available if you are using a profiler equipped EPI ICE. See *Profiler Data Window* on page 103 for more details.

Shortcuts:

Toolbar:

Keys: none

RTOS Command

Use this command to open or view the RTOS Object Browser window. See *RTOS Window* on page 113 for more information.

Shortcuts:

Toolbar: ▣

Keys: none

Breakpoints Command

Use this command to open or view the Breakpoints Dialog. Breakpoints can be added, deleted, and modified via this dialog box. See *Break Points Dialog* on page 86 for more information.

Shortcuts:

Toolbar: none

Keys: none

Option Settings Command

Use this command to open or view the Option Settings dialog. Breakpoints can be added, deleted, and modified via this dialog box. See *Option Settings Dialog* on page 102 for more information.

Shortcuts:

Toolbar: ▣

Keys: none

ICE Trace Spec Command (ICE targets only)

Use this command to open or view the Trace Specification dialog. Trace specifications can be viewed, edited, parsed, and downloaded via this dialog box. See *ICE Trace Specification Dialog (ICE targets only)* on page 98 for more information.

Shortcuts:

Toolbar: ▣

Keys: none

Profiler Setup Command (ICE targets only)

Use this command to open or view the Profiler Setup dialog. This menu item is only available if you are using a profiler equipped EPI ICE. See *Profiler Setup Dialog* on page 105 for more information.

Shortcuts:

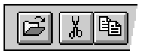
Toolbar: none

Keys: none

General Toolbar Command

Use this command to display or hide the General toolbar, which includes buttons for many commands in EDB, such as Program to Debug, Open Windows, Copy, etc. A check mark appears next to this menu item when the General toolbar is currently displayed.

Many of the View menu items are also available from the General toolbar, shown below.



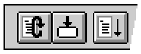
The General toolbar is displayed across the top of the application window, below the menu bar, and provides quick mouse access to many operations and tools used in EDB. To hide or display the General toolbar, choose General toolbar from the View menu <Alt, V, G>.

Click	To
📂	Program to Debug / Open. EDB displays the Open dialog box, in which you can locate and open the desired COFF or ELF program file.
✂	Remove selected data from the document and store it on the clipboard.
⎘	Copy the selection to the clipboard.
📋	Insert the contents of the clipboard at the insertion point.
S I/O	View Session Input/Output Window.
P I/O	View Program Input/Output Window.
👓	View / Edit variable Watch Window.
▦	View Call Stack Window.
🔍	View ICE Trace Window.
🔍	View ICE Profiler Data Window.
M	View new Memory Window.
R	View new Register Window.
📝	View / Edit Option Settings dialog.
⟳	View ICE Trace Specification dialog.
+t	Toggle ICE Trace Execution mode on / off.
+te	Toggle ICE Trace All mode on / off.
↖?	Context sensitive help.

Execution Toolbar Command

Use this command to display and hide the Execution toolbar, which includes buttons for some of the most common execution related commands in EDB. A check mark appears next to this menu item when the Execution toolbar is currently displayed.

Many of the View menu items are also available from the General toolbar. The Execution Toolbar is shown below:

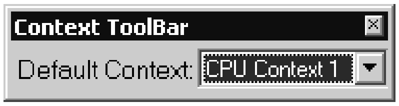


The execution toolbar is displayed across the top of the application window, below the menu bar, and provides quick mouse access to many execution operations used in EDB. To hide or display the General toolbar, choose Execution toolbar from the View menu <Alt, V, T>. Note that the *Properties Dialog* on page 108 contains a check box item to view this toolbar with or without text labels.

Click	To
	Restart Program. See *Restart Command* on page 82.
	Download program data to target. See *Load Command* on page 82.
	Go (start program execution from current location). You may also see this as Go(i). This alternate form means interactive Go mode is set. For more details see *Go or Go Interactive Command* on page 83.
	Stop program execution. See *Stop Command* on page 83.
	Source level step over. See *Source Step Over Command* on page 83.
	Source level step / step into. See *Source Step Into Command* on page 84.
	Source level step out of function. See *Source Step Out Command* on page 84.
	Run to cursor location. See *Run to Cursor Command* on page 84.
	Instruction level step over. See *Instruction Step Over Command* on page 85.
	Instruction level step / step into. See *Instruction Step Into Command* on page 84.
	Snap to home location in current window. The home location can have different meanings for different windows. For the Execution Window, Snap moves the global context view point to the execution point and positions the window at this location. Most other windows move the window viewpoint to the top of the window data.

Context Toolbar Command

The Context toolbar is a specialized toolbar designed to display the current global (default) processor context and allow it to be changed. The execution context is identified with a * in front of the context name. The drop down combo box labeled Default Context provides this functionality and is the only element in the toolbar. The functionality is analogous to MON’s **vc** command.



In general, the setting of the global view context affects all the windows. Things like general registers, call stacks, local and file scope variables, global variables (if multiple programs involved), etc. Some windows have override settings that allow them to be fixed to a particular context.

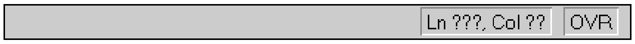
To hide or display the Context toolbar, choose Context Toolbar from the View menu <Alt, V, X>.

NOTE: This specialized toolbar is available only if you have installed an appropriate RTOS_DLL configured for your RTOS or with selected processors that support multiple CPU contexts and/or execution units. Lexra’s NetVortex chip is an example of a chip with multiple CPU contexts.

Status Bar Command

Use this command to display or hide the Status bar, which describes the action to be executed by the selected menu item or depressed toolbar button, line/column data and keyboard insert/over-type state. A check mark appears next to the menu item when the Status bar is displayed.

The Status Bar is shown below:



The status bar is displayed at the bottom of the EDB Window. To display or hide the Status bar, use the Status bar command in the View menu.

The left area of the status bar describes actions of menu items as you use the arrow keys to navigate through menus. This area similarly shows messages that describe the actions of toolbar buttons as you depress them, before releasing them. If after viewing the description of the toolbar button command you do not want to execute the command, then release the mouse button while the pointer is off the toolbar button.

The right areas of the status bar indicate line and column information (if applicable) and shows which of the following keys are latched down:

Indicator	Description

OVR	Status of Insert/Overtype mode (default is off, or insert mode)

Exec Menu

The Exec menu offers the following commands, which enable you to control the execution of your program in various ways:

Restart	Restart program.
Load	Load / Reload program.
Verify Load	Verify downloaded program data matches executable file data.
Go	Start program execution. Accelerator key: <F5>
Stop	Stop execution. Accelerator key: <Ctrl-Break>
Source Step Over	Source level step over. Accelerator key: <F10>
Source Step Into	Source level step Into. Accelerator key: <F8>
Source Step Out	Source level step out.
Run To Cursor	Go until execution point is at line with cursor.
Instr Step Into	Instruction level step into. Accelerator key: <F7>
Instr Step Over	Instruction level step over. Accelerator key: <F6>

Restart Command

Kills the current process and prepares to restart execution. This is the same as clicking on ⟲ in the Execution Window, or typing **K** in the Session Window. The program will be downloaded when you begin execution with your next Go or Step operation, unless downloading is suppressed with the **-z** command line option or by adjusting the Section Load option in the Properties dialog (see *Properties Dialog* on page 108).

Shortcuts:

Toolbar: ⟲

Keys: none

Load Command

Downloads the target program. Starts or resumes full speed program execution. This is the same as clicking on ⤓ in the Execution Window, or typing **lf** in the Session Window.

Shortcuts:

Toolbar: ⤓

Keys: none

Verify Load Command

The section types previously downloaded to the target are uploaded and checked against the original COFF or ELF files. The specific section types verified can be seen or adjusted via the Program Options dialog box.

Shortcuts:

Toolbar: none

Keys: none

Go or Go Interactive Command

Start or resume full speed program execution. This is the same as clicking on in the Execution Window, or typing either **`c`** or **`r`** in the Session Window. For interactive mode the button icon is and the commands are **`ci`** or **`ri`**.

Some of EPI's target execution vehicles support what we call Interactive Go mode. This mode is also referred to as *concurrent target debug mode*. When active, this mode allows usage of the various EDB data windows (like the Memory, Watch, and Trace) while a target program is executing. The state of this mode is control via the EO option **`edb_go_interactive_mode`**. See the *Option Settings Dialog* on page 102 or the **`EO`** command description (*Enter Configuration Option* on page 66) for more details. Note that the menu name and Execution button's Go icon change slightly with the changing of this option.

Shortcuts:

Toolbar: or

Keys: <F5>

Stop Command

Stops target execution. Choosing this menu item is the same as typing <Ctrl-C>, or clicking on the button in the Execution Window. This menu item can also be used to stop the execution of certain EDB and MON subsystem commands such as loading a file, doing a verify load, displaying memory, and doing a memory test.

Shortcuts:

Toolbar:

Keys: <Ctrl-Break>

Source Step Over Command

High-level language single-step (steps one source level instruction), stepping over any procedure calls. This is the same as clicking on in the Execution Window, or typing **`s`** in the Session Window.

Shortcuts:

Toolbar: {}↓

Keys: <F10>

Source Step Into Command

High-level language single-step, stepping *into* any procedure calls. This is the same as clicking on {↓} in the Execution Window or typing **s** in the Session Window.

Shortcuts:

Toolbar: {↓}

Keys: <F8>

Source Step Out Command

Resume program execution, stopping when the currently executing procedure returns to its caller. This is the same as clicking on {}↑ in the Execution Window or typing the command **cU** in the Session Window.

Shortcuts:

Toolbar: {}↑

Keys: none

Run to Cursor Command

Start or resume program execution until we reach the line that cursor is currently on. This is the same as clicking on →{} in the Execution Window or typing **c** into the Session Window's command line. If Interactive mode is set a **ci** is done. Note that, if the menu item and button are grayed-out, the cursor is not on an executable line.

Shortcuts:

Toolbar: →{}

Keys: none

Instruction Step Into Command

Machine-level single-step, stepping into any procedure calls. This is the same as clicking on [↓] in the Execution Window, or typing **si** in the Session Window.

Shortcuts:

Toolbar: [↓]

Keys: <F7>

Instruction Step Over Command

Machine-level step over (steps one machine level instruction), stepping over any procedure calls. This is the same as clicking on in the Execution Window, or typing **Si** in the Session Window.

Shortcuts:

Toolbar:

Keys: <F6>

Misc Menu

The Misc menu offers the following commands:

Refresh Window Data Refreshes (and reloads from target if needed) the data display within the window.

Window Menu

The Window menu offers the following commands, which enable you to arrange multiple views of multiple documents in the application window:

Cascade Arranges open windows in an overlapped fashion.

Tile Arranges open windows in non-overlapped tiles.

Arrange Icons Arranges icons of minimized windows at the bottom of the main window.

Split Split the active window into panes. You can then use the mouse or the keyboard arrows to move the splitter bars. When you are finished, press the mouse button or enter to leave the splitter bars in their new location. Pressing escape keeps the splitter bars in their original location. If this menu item is grayed-out, the window does not support splitting.

Window *1*, *2*, ... Go to the specified window. EDB displays a list of currently open document windows at the bottom of the Window menu. A check mark appears in front of the document name of the active window.

Help Menu

The Help menu offers the following commands, which provide you assistance with this application:

Help Topics Offers you an index to topics on which you can get help.

About Displays the version number and copyright information of this application.

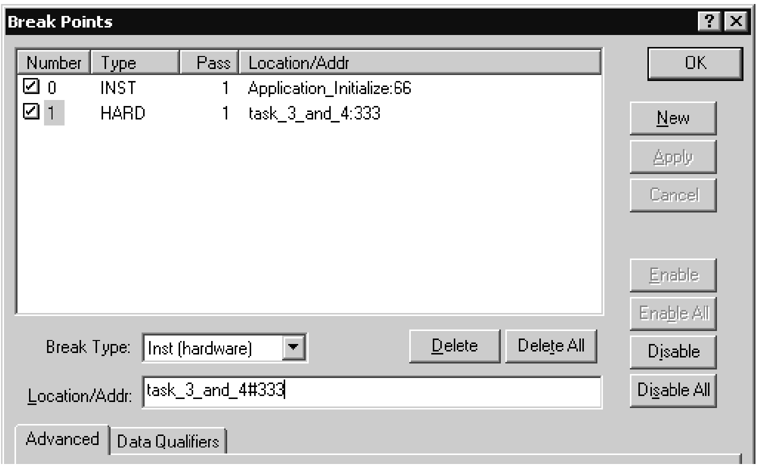
Windows and Dialogs

This section describes all windows and dialogs in alphabetical order.

- *Break Points Dialog* on page 86
- *Call Stack Window* on page 90
- *Execution Window* on page 91
- *ICE Trace Display Window* on page 96
- *ICE Trace Specification Dialog (ICE targets only)* on page 98
- *Memory Window* on page 99
- *Option Settings Dialog* on page 102
- *Profiler Data Window* on page 103
- *Profiler Setup Dialog* on page 105
- *Program Input/Output Window* on page 107
- *Register Window* on page 112
- *RTOS Window* on page 113
- *Session Window* on page 114
- *Watch Window* on page 115

Break Points Dialog

Breakpoints can be added, deleted, and modified via the dialog below.


Break Points

Number	Type	Pass	Location/Addr
☑ 0	INST	1	Application_Initialize:66
☑ 1	HARD	1	task_3_and_4:333

Break Type: Inst (hardware)

Location/Addr: task_3_and_4#333

The Breakpoints Listbox

The scrollable listbox within the dialog allows you to view, select, and modify all your breakpoints. Selection is performed via standard listbox mouse and key actions. Breakpoints with their number field highlighted (or grayed) are in the selected state. A left click in the checkbox column not only does the standard row (breakpoint) selection, but also toggles the breakpoint enable / disable state (the checkbox). A left double-click does a hyper-link (moves the execution windows view pointer to the breakpoint line). A right-click brings up the short-cut menu which simply gives access to many of the buttons available within the dialog. It also provides another way to do a hyper-link.

☑ This status box shows the enabled / disabled state of the breakpoint. Clicking on the box will toggle the state. A check indicates an enabled breakpoint.

Number The ID number of the breakpoint. Numbers are assigned automatically as breakpoints are created. The breakpoint number can be used in the command line interface to delete the breakpoint by typing *`number`* **`d`** in the Session Window. In addition to the number, the check box to the left of the number shows the enable disable state of each breakpoint. A checked box indicates the enabled state.

Type The type of breakpoint. Breakpoints can be one of six types: `INST`, `INST16`, `HARD`, `READ`, `WRITE`, `R/W`. `INST` identifies a normal instruction level breakpoint. `INST16` identifies an alternate instruction set form of a normal instruction breakpoint (for example, MIPS16 and ARM's Thumb mode). `HARD` identifies a hardware level instruction breakpoint (useful for setting instruction breakpoints in ROM areas). READ, WRITE, and R/W identify data access breaks. Depending on the capabilities of your hardware, data access breakpoints can be qualified with various options available in the data access tab. Note that data access options may also be valid for HARD type breakpoints.

Pass Count

The breakpoint pass count is the number of times the breakpoint is hit before the debugger will stop execution (by default this number is the initial pass count). The Initial Pass Count edit box will show the reload pass count of the selected breakpoint (see Edit boxes below).

Breakpoint location

The address of the breakpoint, usually expressed by the function name and line number. Breakpoints can also be set at arbitrary locations by specifying the address with an expression.

The Edit Boxes

The Edit boxes display more detailed information on the selected breakpoint and allow that information to be changed. If more than one breakpoint is selected then these boxes are disabled. Upon changing any data in these boxes, the Apply and Cancel buttons will enable you to make permanent changes or discard them.

Break Type This drop down list box shows the current breakpoint type and allows selection of other possible breakpoint types. Available breakpoints are `Inst` (Software), `Inst16` (software), `Inst` (Hardware), `Data Read`, `Data Write`, and `Data Read/Write`. Note that this closely matches the types displayed in the type column of the breakpoint list. The list abbreviates the names leaving off the word `Data` and changing `Read/Write` to `R/W`.

Location/Addr

This closely resembles the Location/Addr value in the list box, but matches the syntax required for editing. You can use this box to change the breakpoint location.

The Buttons

OK

Accepts any changes that have not been applied (via the Apply button) and closes the dialog.

New

Selecting this button adds a new breakpoint to the list. Focus is changed to Location/Addr edit box, allowing you to enter the breakpoint location. When you are done specifying the breakpoint data, click the Apply button to make it permanent.

Apply

Accepts the breakpoints new or changed data as permanent.

Cancel

Cancels a breakpoint edit or new operation.

Delete

Deletes the selected breakpoint(s).

Delete All

Deletes all the breakpoints (selected or not).

Enable

Enables the selected breakpoint(s).

Enable All

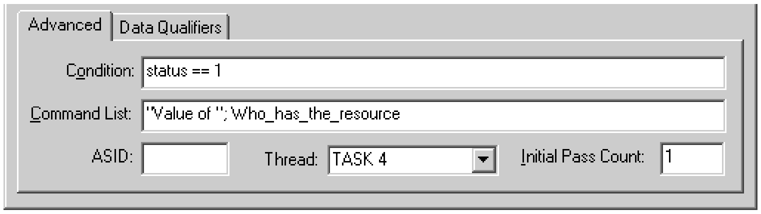
Enables all the breakpoints.

Disable

Disables the selected breakpoint(s).

Advanced Tab Box

This tab box provides access to more advanced breakpoint features such as pass counts and conditions, and a command list.



Condition: status == 1

Command List: "Value of "; Who_has_the_resource

ASID:

Thread: TASK 4

Initial Pass Count: 1

Condition

You can make the current breakpoint conditional by entering an arbitrary expression in the Breakpoint Condition input box. All variables used in the condition must be active at the breakpoint location.

NOTE: If the breakpoint is conditional and has a pass count, the conditions will be evaluated first, and if the conditions are true then the pass count will be decremented and checked.

Command List

This box allows you to add commands that are automatically executed upon hitting the breakpoint. Multiple commands can be entered by separating them with a semi-colon (;).

ASID

This box allows you to add an ASID qualifier. Typically, the ASID field is only available for targets that have dynamically mapped memory (TLBs). NOTE: For target support CPU contexts, this field can be used to give your breakpoint a context qualification (enter a CPU context number).

Thread

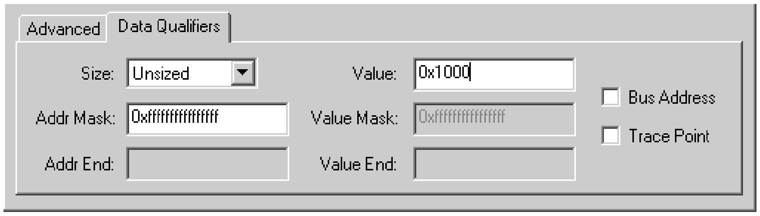
This list box allows you to qualify (or un-qualify) a breakpoint as specific to a particular active thread. Such a qualified breakpoint will only stop the running program if the specified thread is active. Such qualification is very useful for setting breakpoints on RTOS interface functions or applications that have multiple threads executing the same code. Since, thread data is not generally available until after the RTOS and applications are initialized, thread qualifications must be entered after such initialization. Also, since breakpoint data saved between debug sessions is restored long before application initialization, such qualifications are lost between EDB sessions.

Initial Pass Count

Pass counts allow you to specify how many times the breakpoint location must be executed before the program is actually halted. Pass shows the current status of the pass counter, that is how many more times the breakpoint will be reached before it is hit. Initial Pass Count is the starting pass count value that is reloaded into Pass automatically when the breakpoint is finally hit.

Data Qualifiers Tab Box

This tab box provides access to data breakpoint qualifiers. Items showed grayed out are not available in your debug environment.



Size: Unsized
Value: 0x1000
Addr Mask: 0xffffffffffffffff
Value Mask: 0xffffffffffffffff
Addr End:
Value End:
☐ Bus Address
☐ Trace Point

Size

Typically the default size of Unsized is best. Unsized simply means break on any access to the memory associated with this address. Note that for many systems this is the only choice available. Other possible choices are BYTE, HALF WORD, WORD and DOUBLE WORD. Specifying one of these sizes limits the break to only access the specified size.

Addr Mask

This field allows the supplied Location/Addr to be masked, which is useful for breaking on ranges and/or multiple memory segments (high address masking). The default value for this field is `0xffffffff.ffffffff`.

Addr End

This works in conjunction with your Location/Addr to provide a range of addresses to break on.

Value

The value to match before the break hits.

Value Mask

This field allows the supplied Value to be masked, which is useful for breaking on interesting ranges of values (like odd numbers, etc). The default value for this field is `0xffffffff.ffffffff`.

Value End

This works in conjunction with the Value field to provide a range of values to break on.

Bus Address

Checking this box identifies the Location/Addr as a bus address. Normally the Location/Addr field is treated as a user level (mapped or virtual) address.

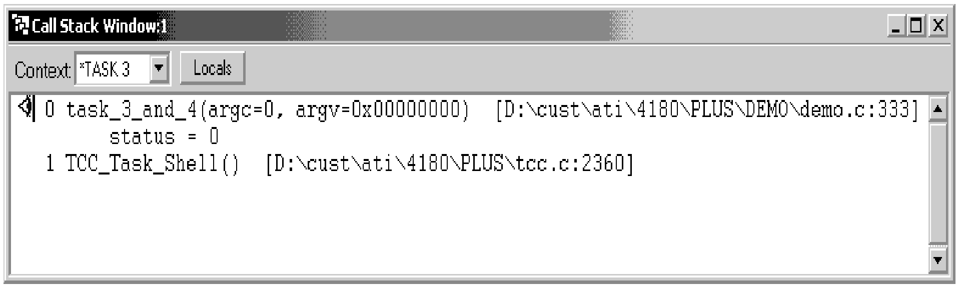
Trace Point

Checking this box changes the break point into a trace point. A trace point does not stop the target. Instead it triggers the processors trace trigger features. Note that because of this some breakpoints fields are not meaningful and appear disabled when trace point is checked. Please consult your target documentation for more information on your target's trace trigger feature.

Call Stack Window

The Call Stack Window shows the current call stack, identifying function calls, parameters, source files/lines, and optionally local variables. One of the most used features of the Call Stack Window is Hyper-Linking. Right-clicking in the far left column of this window causes the Execution

Window to change context to the current execution point within the referenced (clicked-on) function. An example of the Call Stack Window is shown below:



```
0 task_3_and_4(argc=0, argv=0x00000000)  [D:\cust\ati\4180\PLUS\DEMO\demo.c:333]
      status = 0
1 TCC_Task_Shell()  [D:\cust\ati\4180\PLUS\tcc.c:2360]
```

Call Stack Toolbar: Note that this toolbar defaults to off if no context data is available. See the Context Input Box description below for details.

Context Input Box: Allows you to override the default context configured via the global Context toolbar. The context setting can affect how the expression evaluates, if the expression references any context relevant data. Note that this input box is only available if you are using a CPU that supports CPU contexts, or you have an appropriate RTOS_API.DLL.

Locals Button: Optionally, the Calls Windows can be configured to display each function's local variables as well. This feature is a toggle via the Locals button or the window's Shortcut (right-click) menu. A check mark next to the menu item indicates that displaying local variables is on.

Short-Cut Menu

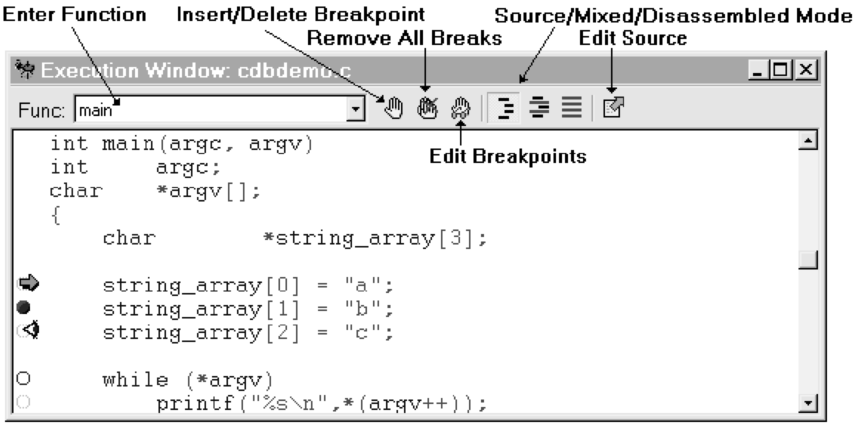
Hyper-Linking: Right-clicking in the far left column of this window causes the Execution Window to change context to the current execution point within the referenced function. This feature can also be used from the Shortcut menu. (For a definition of hyper-linking, see *Context View Point* on page 27)

Show Locals: See Locals Button above.

Execution Window

The Execution Window lets you watch your code as you step through execution, set and modify breakpoints, and examine data variables. It can display source code only, mixed source and assembly code, or pure assembly code.

In Source mode, the Execution Window looks like:



```
int main(argc, argv)
int     argc;
char    *argv[];
{
    char        *string_array[3];

    string_array[0] = "a";
    string_array[1] = "b";
    string_array[2] = "c";

    while (*argv)
        printf("%s\n",*(argv++));
```

➡ Identifies the execution point. It can also look like ➡ if the execution point is not exactly at the first address of code for this line.

◁ Identifies the context point. A context point is conceptually the line at which the debugger is currently looking. Any variable display references, etc., will use this line as a basis.

Is a combined execution and context point. It can also look like if the execution point is not exactly at the first address of code for this line.

● Identifies a line as having a breakpoint inserted. If the breakpoint looks like ○ , then it is a disabled breakpoint. Note that breakpoints will still show up even if they do not reside exactly on the first line of machine code for the source line.

○ Identifies a line as having machine code associated with it. Only lines with code will accept breakpoints.

Execution Window Toolbar Buttons

Inserts or deletes an existing breakpoint on the current source or disassembly line.

Removes all existing breakpoints.

Brings up the Breakpoint Edit dialog.

Sets the Execution Window Display mode to Source. If no source module is available the button still sets, but the mode will remain disassembled.

Sets the Execution Window Display mode to Mixed Source and Disassembled. If no source module is available the button still sets, but the mode will remain Disassembled.

Sets the Execution Window Display mode to Disassembled.

Calls up an external editor supplying the current module and line number as arguments. The editor and options supplied to the editor are configured via an option configuration string. See the **`edit_callout`** option in the *Option Settings Dialog* on page 102 for details on the setup of the editor callout configuration string.

Viewing Modes

Three viewing modes are supported: Source, Mixed, and Disassembled. The viewing mode is controlled by buttons on the window's toolbar, the Shortcut menu, or by hitting the <F3> accelerator key. Note the EDB will auto-switch to the disassembled mode if the viewing point does not have any source code debug information.

Instruction View Mode

These buttons control the Execution Window's View mode. The first is Source mode, which displays only high-level source lines from your source code. The second is Mixed mode, which displays both source instructions and the associated machine instructions directly underneath each source line. The third is Disassembled mode, in which only machine instructions are shown.

Short-Cut Menus

The Shortcut menu (right-click) has three operating modes: Left Column Breakpoints menu, Code menu, and the Selected Item menu. A right-click in the breakpoint column brings up the Breakpoint menu. A right-click on selected text brings up the Selected Item menu, otherwise the normal Code menu comes up. (See *Execution Window Shortcut Menus* below.)

Execution Toolbar

Although not part of the Execution Window, the Execution toolbar relates directly to contents of this window. See *Execution Toolbar Command* on page 80 for more details.

Execution Window Shortcut Menus

Breakpoint ShortCut Menu

This menu is entered by a right-click on the far left side of the Execution Window. Note that the mouse cursor will change to a breakpoint circle when you are in the right area. It has the following options:

Insert/Delete Breakpoint

Insert a breakpoint at the current line. If a breakpoint already exists here then it is deleted. Breakpoints can only be inserted on lines marked as having code.

Enable/Disable Breakpoint

Toggle the enable / disabled state of the breakpoint on the current line.

Edit Breakpoints
Brings up the Edit Breakpoints dialog. (See *Break Points Dialog* on page 86.)

Selected Text Shortcut Menu

This menu is entered by a right-click within some selected text. It has the following options:

Copy
Copy selection to the clipboard. (See *Copy Command* on page 74.) Accelerator key: <Ctrl-C>

Goto Function Definition
If the selected item matches a known function the Execution Window context will move to that function.

Add Watch
Add selection to Watch Window. Note that you can also drag the selection to the Watch Window.

Evaluate Selection
Evaluates the selection by copying it to the Session Window command input.

Default Shortcut Menu

This menu is entered by a right-click when none of the above conditions apply. It has the following options:

Copy
Copy selection to the clipboard. Accelerator key: <Ctrl-C>

Select All
Selects all the text in the window. Note that this operation is only valid in Source mode. Accelerator key: <Ctrl-A>

Edit Source
Calls an external editor passing the current source module name and line number as arguments. See the Allows for and Brings up the Breakpoint sub-menu items. See the Breakpoint Shortcut Menu above. See the **`edit_callout`** option in the *Option Settings Dialog* on page 102 for details on the setup of the editor callout configuration string.

Breakpoint
Brings up the Breakpoint sub-menu items. See the Breakpoint Shortcut menu above.

Select Function
Puts input focus to the Enter Function drop-down box on the Execution Window toolbar. Accelerator key: <Ctrl-F>. Note that if the Select Box mode is set to modules then you will only see module names, not functions.

Select Box Mode
Selects mode of Execution toolbar input box. See *Execution Toolbar Select Box Mode* below.

Run to Cursor
Continues execution until it encounters the code for the line with the cursor.

Set Context to Cursor
: Sets the debugger's context viewpoint to the current cursor line. This can affect the context of variables displayed in various EDB Windows. Use the Snap button to set the context back to the execution point.

Source Mode
: Selects Source mode as the default display mode for code.

Mixed Mode
: Selects Mixed Source/Disassembled mode as the default display mode for code.

Disassembled Mode
: Selects Disassembled mode as the default display mode for code.

Cycle Mode
: Cycles the display mode between Source, Mixed, and Disassembled. Accelerator key: <F3>

Code Coloring
: Toggles the state of code coloring to be on or off. The default is on.

Split
: Splits the execution into multiple views.

Toolbar
: Toggle the hidden / displayed state of the Execution Window toolbar.

Properties
: Bring up Properties dialog. Allows display and edit of program configuration. See *Properties Dialog* on page 108.

Execution Toolbar Select Box Mode

This menu is a pop-up under the normal Execution Window Shortcut menu. It has the following options:

Debug Functions
: Shows only functions compiled with debug information on (compiler's **-g** flag).

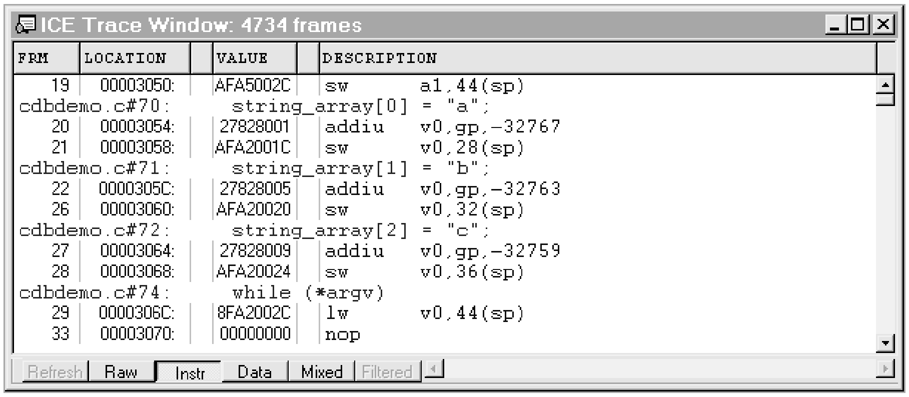
All Functions
: Shows all functions in the program.

Debug Modules
: Shows only modules compiled with debug information (compiler's **-g** flag).

ICE Trace Display Window

The Trace Display Window provides convenient scrollable access to any captured trace data in your emulator (ICE versions of EDB only).

The following example of the Trace Display Window shows disassembled instructions:



```
FRM  LOCATION     VALUE      DESCRIPTION
  19   00003050:  AFA5002C   sw       a1,44(sp)
cdbdemo.c#70:      string_array[0] = "a";
  20   00003054:  27828001   addiu    v0,gp,-32767
  21   00003058:  AFA2001C   sw       v0,28(sp)
cdbdemo.c#71:      string_array[1] = "b";
  22   0000305C:  27828005   addiu    v0,gp,-32763
  26   00003060:  AFA20020   sw       v0,32(sp)
cdbdemo.c#72:      string_array[2] = "c";
  27   00003064:  27828009   addiu    v0,gp,-32759
  28   00003068:  AFA20024   sw       v0,36(sp)
cdbdemo.c#74:      while (*argv)
  29   0000306C:  8FA2002C   lw       v0,44(sp)
  33   00003070:  00000000   nop
```

The text in the title bar includes the total number of captured frames (cycles) of trace data, the current display mode, and whether filtering is in effect.

The column header displays the field name and polarity. Both Color (white vs. black) and the normal negation over bar shows the active high or low state of the signal. Note that many signals may be grouped within one column field. This grouping is loosely based on MON's **ETF** command formatting information. A column break is inserted if a field is horizontal, has any space separation or inverse color change. Note that the time stamp header also functions as a mode display button. Clicking on the Time Stamp Header button cycles the display between absolute, relative, and delta. Note that other header buttons have any operation effect.

The Trace Display Toolbar buttons allow you to update the trace data, specify the type and format of trace data displayed, and enable/disable filtering.

Refresh Button

This button causes EDB to upload fresh Trace data from the emulator. This is necessary if you leave the Trace Display Window open while you resume execution of your target program. When execution next stops, the emulator's Trace Data buffer will be full of new information but the Trace Display Window will still be showing the data captured earlier.

Raw Displays the captured trace data in a raw binary format. All of the processor's busses and control signals can be seen. The signals displayed and their grouping can be changed with the **ETF** (Enter Trace Format) MON command.

Note that no bus information is available when using the HP Software Probe.

Instr — Displays valid instruction accesses in disassembled form. If the address of an instruction corresponds to a source line, the source line will also be displayed.

Data — Displays valid data accesses in a formatted form. (Not available with HP Software Probe.)

Mixed — Combination of Instr and Data modes. (Not available with HP Software Probe.)

Filtered — You can use filtering to reduce the number of frames displayed to those that match a specified list of conditions. This can be useful when trying to find a particular event or access in the trace data. You create the filter with the **EF** (Enter Filter) MON command, specifying one or more of your Trace Control Conditions. If your Trace Control Specification does not include the condition you need, you can create a new condition just for the filter.

Note that if you do not have a filter specification defined then the Filter button is displayed in Disabled mode.

The Shortcut menu provides access to all the buttons above (allowing you to turn off the toolbar and save some screen space). You can also configure the time stamp display state, mode, and do hyper-linking.

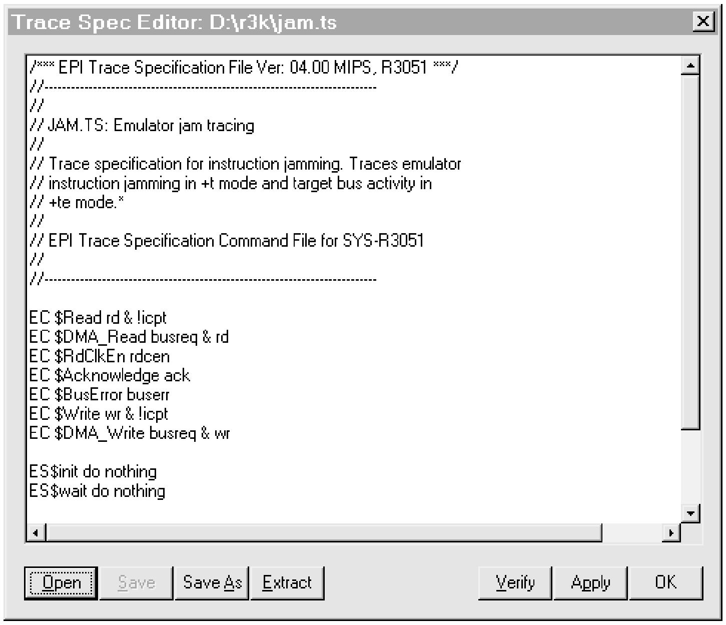
Display Time Stamp — This menu item toggles the state of the time stamp display. If checked, the time stamp is displayed after the frame number. Note that this mode mirrors the mode used by MON’s **dt** command and can also be configured via MON’s **ETF** command. The time stamp field on/off state defaults to on for RAW mode and off for formatted modes.

Time Stamp Mode — This menu item brings up a sub-menu with the Time Stamp Display Mode options. Note that these options can also be configured by clicking on the time stamp header field. By default, a time stamp is only displayed in RAW display mode. This can be configured via MON’s **ETF** command and the shortcut menu toggle Display Time Stamp.

HyperLink — Whenever source code is intermixed in the trace data, the Shortcut (right-click) menu will contain a hyper-link option which if activated will cause the Execution Window to move its context point and display the referenced line in code.

ICE Trace Specification Dialog (ICE targets only)

Trace specifications can be viewed, edited, parsed, and downloaded via this dialog box. Note that this feature is not available when using the HP Software Probe.



```
/*** EPI Trace Specification File Ver: 04.00 MIPS, R3051 ***/
//----------------------------------------------------------------------
//
// JAM.TS: Emulator jam tracing
//
// Trace specification for instruction jamming. Traces emulator
// instruction jamming in +t mode and target bus activity in
// +te mode.*
//
// EPI Trace Specification Command File for SYS-R3051
//
//----------------------------------------------------------------------

EC $Read rd & !icpt
EC $DMA_Read busreq & rd
EC $RdClkEn rdcen
EC $Acknowledge ack
EC $BusError buserr
EC $Write wr & !icpt
EC $DMA_Write busreq & wr

ES$init do nothing
ES$wait do nothing
```

The Edit Window

This Edit Window allows you to view and edit trace specification. It does not, by default, display the active trace specification unless you have previously used the Apply button to parse (and thereby, make active) this specification. You can also use the Extract button to fill the Edit Window with the active trace specification (see the Extract button description below). This window supports standard Cut, Copy, and Paste operations.

The Buttons

Open — Brings up an open file dialog. Trace specification files have the default file extension **.ts**.

Save — Saves the active trace specification using its current file name (displayed in the dialog header). This button is grayed-out if the current specification is not named, has not changed, or is empty.

SaveAs — Brings up the standard Windows SaveAs dialog. Trace Specification files have a default extension of **.ts**. Leaving off the extension when naming the file will cause the dialog to append this extension when saving.

Extract — Clears any trace specification in the Edit Window, extracts the currently active trace specification, and inserts it into the Edit Window. If no active trace specification exists, then the default trace specification is used. Please note that hitting the Apply button following this extract operation will cause you to lose any comments and non-default formatting that you may have in your specification.

Verify — Verifies the debugger's active trace specification. Note that this is not necessarily the trace specification that is in the Edit Window. Instead it operates on the active trace specification. You can use the Apply button to parse and make active your current Edit Window specification. Verify will then operate on this specification. Note that any verification errors appear in the Session Window.

Apply — Causes the trace specification in the Edit Window to be parsed and readied for download to the ICE. This process also makes this trace specification the active trace specification. Note that any parsing errors will appear in the Session Window.

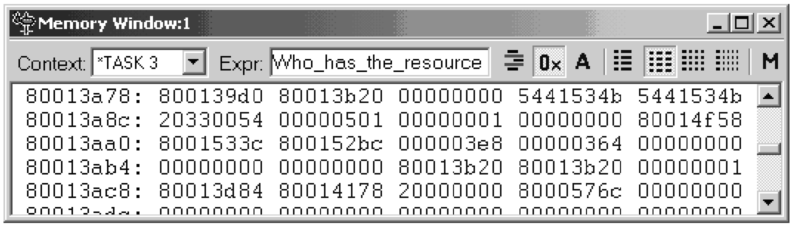
Ok — Does the same operation as the Apply button above and also closes the dialog. Note that your trace specification is saved such that reopening this dialog brings back your last specification. Closing EDB will cause any unsaved trace edits to be lost.

Memory Window

The Memory Window lets you examine and modify your target and program memory in a variety of formats. You can use multiple memory windows to examine different data at the same time, and scroll around inside the window to get different addresses.

Object editing is done on a per object basis. Simply position the cursor within the object and type a new value. Once you begin editing, an Edit box appears. The Edit box closes when you move the cursor off the current object, and the target memory is then updated (written) with the new value. An edit operation can be aborted by pressing the <ESC> key. Note that target memory is always written in the current object size (word, half words, etc).

The Execution Toolbar's snap button will bring the window back to the address result of the address expression. Also note that a data refresh (read) can be done via the Master Menus Misc/Refresh menu item or the Shortcut menu. The Memory Window is shown below:



Context: *TASK 3 Expr: Who_has_the_resource

```
80013a78: 800139d0 80013b20 00000000 5441534b 5441534b
80013a8c: 20330054 00000501 00000001 00000000 80014f58
80013aa0: 8001533c 800152bc 000003e8 00000364 00000000
80013ab4: 00000000 00000000 80013b20 80013b20 00000001
80013ac8: 80013d84 80014178 20000000 8000576c 00000000
80013adc: 00000000 00000000 00000000 00000000 00000000
```

Memory Window Toolbar

Context Input Box

Allows you to override the default context configured via the global context toolbar. The context setting can affect how the expression evaluates, if the expression references any context relevant data. Note that this input box is only available if you are using a CPU that supports CPU contexts, or you have an appropriate RTOS_API.DLL.

Address Input Box

Allows you to specify the beginning of a memory region to examine. This may be a symbolic name or expression, like the **`char_array`** example above, or a numeric value, like **`0x80001000`**. The Address Expression box defaults to CDB style expressions. The M button allows you to specify MON style expressions. MON style expressions have the advantage of being able to access the full address space (see Spaces in the MON manual for details), but also has the drawback of being limited to global symbols.

Note that the default input radix for expressions is decimal in CDB mode and hexadecimal in MON mode. The current mode can be determined via the state of the M button (MON mode) and the prompt **`Expr:`** (for EDB mode) and **`Addr:`** (for MON mode).

Format Buttons

Clicking on these changes the way the data is displayed.

≡ Displays the data in instruction (disassembled) format. Source lines from your program that match the addresses will be displayed intermixed with their associated assembly code. This mode is only valid for word and half-word width formats. Half-word instruction format is useful on processors that support alternate 16-bit instruction modes (MIPS16/ARM Thumb). Right-clicking on the far left of one of these source lines will cause the Execution Window to “home” to that part of your program, changing the context viewing point. This is called hyper-linking (see *Context View Point* on page 27). The Shortcut Pop-up menu (right-click) can also be used to perform hyper-linking.

0x Displays the data in hexadecimal format. Valid for any mode

A Displays printable ASCII characters with non-printable bytes displayed as a dot. Valid only for Byte Width mode.

Display Width Buttons

Clicking on these changes the object size of the displayed data and also affects which formats are available.

▤ Displays eight bytes (one double word) in hex.

▦ Displays four bytes (one word) objects in the selected display format.

▦ Displays two bytes (one half-word) objects in the selected format.

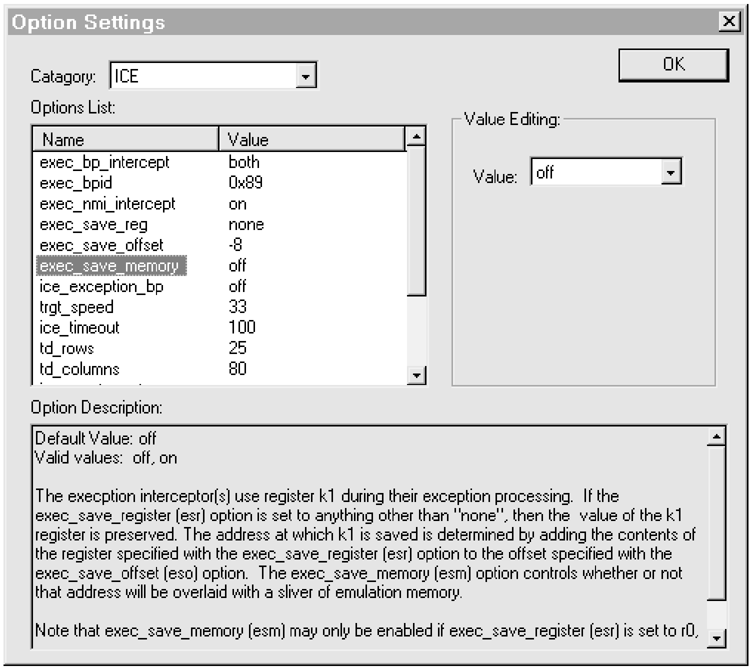
▦ Displays memory as hex bytes.

MON Expressions When active, the MON Expressions button M causes the address box data to be parsed by the MON expression parser. See Address Input Box above for more details.

Shortcut Menu The Memory Windows Shortcut (right-click) menu provides access to the standard Copy, Toolbar on/off, etc., and also allows toggling the MON Expressions button, causing a data refresh, and configuration of the display format.

Option Settings Dialog

Configuration Option Settings can be modified via the dialog below.



Catagory: ICE

Options List:

Name	Value
exec_bp_intercept	both
exec_bpid	0x89
exec_nmi_intercept	on
exec_save_reg	none
exec_save_offset	-8
exec_save_memory	off
ice_exception_bp	off
trgt_speed	33
ice_timeout	100
td_rows	25
td_columns	80

Value: off

Option Description:

Default Value: off
Valid values: off, on

The execption interceptor(s) use register k1 during their exception processing. If the exec_save_register (esr) option is set to anything other than "none", then the value of the k1 register is preserved. The address at which k1 is saved is determined by adding the contents of the register specified with the exec_save_register (esr) option to the offset specified with the exec_save_offset (eso) option. The exec_save_memory (esm) option controls whether or not that address will be overlaid with a sliver of emulation memory.

Note that exec_save_memory (esm) may only be enabled if exec_save_register (esr) is set to r0,

Category Selection Box

The category selection controls the contents of the Option List box. It defaults to the all selection which means the Option List contains all the available options for the given target environment. Other options in the category list limit the options to a small related set, making it easier to examine the settings.

Options List

This scrollable listbox within the dialog allows you to view and select for modification all the options displayed by MON's **DO** command. Selection is performed via standard listbox mouse and key actions. Only one item may be selected. A selected item expanded description appears in the Option Description box. This description is the same as that produced via MON's **DOV** command.

Name	The option name.
Value	The current value.

Option Description

This box displays text describing the details of the option, its range of allowed values, and its default value.

Value Editing Group Box

This box contains either an edit box and Apply button (allowing the editing of numeric options), or it contains a simple drop list selection for enumerated options.

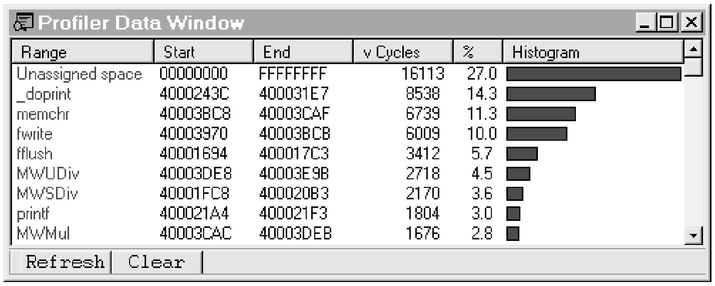
Edit Value Box — This edit box allows the configuration of numeric or address values. Upon making a change the apply button will de-gray and can be used to parse and accept the change (if valid). Switching from one item to another also cause an auto-apply (parsing and acceptance) to take place (if changes are made).

Drop Down Selection Box — This box allows the easy changing of options that have a limited set of enumerated values. The description of each value's meaning can be found in the Option Description box.

Apply Button — This button is available only for values that require direct keyboard editing (like numeric or address options). It turns non-gray after editing.

Profiler Data Window

The Profiler Data Window provides convenient scrollable access to any captured profiler data in your emulator (ICE versions of EDB only). The window's column headers provide not only column identification but allow you to sort by the column as well. Sorting is limited to the range, start, and cycles columns. The currently selected column is identified by a **^** or **v** in front of the column name. Clicking on the currently selected column will toggle the ascending / descending aspect of the sort. Note that the column widths can be manipulated by dragging the mouse on the divider lines.



Range	Start	End	v Cycles	%	Histogram
Unassigned space	00000000	FFFFFFFF	16113	27.0	
_doprint	4000243C	400031E7	8538	14.3	
memchr	40003BC8	40003CAF	6739	11.3	
fwrite	40003970	40003BCB	6009	10.0	
fflush	40001694	400017C3	3412	5.7	
MWUDiv	40003DE8	40003E9B	2718	4.5	
MWSDiv	40001FC8	400020B3	2170	3.6	
printf	400021A4	400021F3	1804	3.0	
MWMul	40003CAC	40003DEB	1676	2.8	

A short description of the columns follows:

Range — The name of the procedure and source code line(s) if any.

Start — The start address for the profiled range.

End — The end address for the profiled range.

Cycles — The total number of cycles counted while the processor was executing code in the range.

% — The percentage of the total cycles counted while executing in the range.

Histogram — A bar graph of the percentage.

Profiler Data Window Toolbar:

Refresh — This button causes EDB to upload fresh Profiler data from the emulator. This would be necessary if you leave the Profiler Data Window open while you resume execution of your target program. When execution next stops, the emulator's Profiler Data buffer will be full of new information but the Profiler Data Window will still be showing the data captured earlier. The new data is not uploaded automatically since you may still be viewing the earlier data and may not be ready to view new data. This button is also available on the Shortcut menu as Refresh Data.

Clear — When selected, this button clears any current profiler data.

Note that the toolbar can be turned on/off via the Shortcut menu's (right-click) toolbar item.

Profiler Data Window Shortcut Menu:

Refresh Data — This menu item operates the same as the toolbar's Refresh button. For details, see the Refresh button description above.

Clear — When selected, this button clears any current profiler data.

Hyper-Linking — Right-clicking in the far left column of a line in the window causes the Execution Window to change context to the referenced line and/or function. You can also access this feature via the Shortcut (right-click) menu. (For a definition of hyper-linking, see *Context View Point* on page 27.)

Ignore 0 Cycle Functions

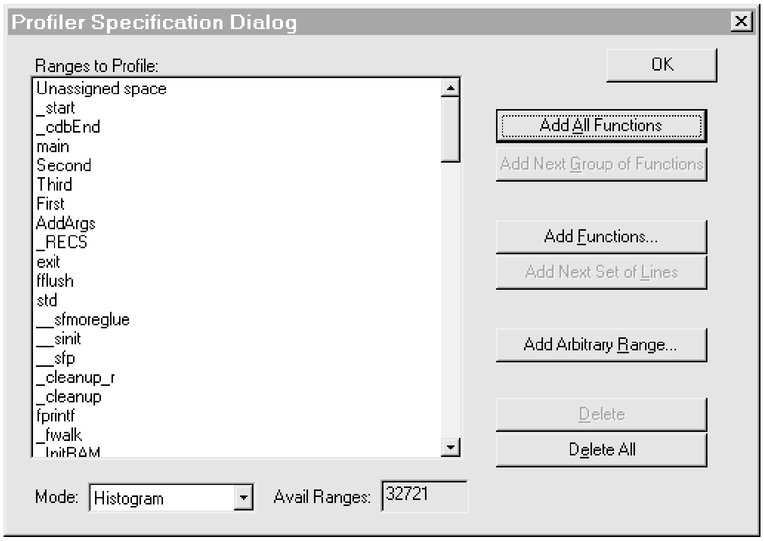
Suppresses display of ranges that have cycle counts of zero.

Combine RAM & ROM spaces

When selected, this choice causes the profiling hardware to ignore the control signal that differentiates between the Instruction ROM and Instruction RAM address spaces. It is applicable only to target systems using the AMD Am2900x and Am29050 processors, and should be activated if the target memory system ignores the IREQT signal. This menu item is a toggle.

Profiler Setup Dialog

This dialog provides setup access to the powerful profiling capability of EPI's In-Circuit Emulators (ICE versions of EDB only).



Profiler Specification Dialog

Ranges to Profile:

Unassigned space
_start
_cdbEnd
main
Second
Third
First
AddArgs
_RECS
exit
fflush
std
__sfmoreglue
__sinit
__sfp
_cleanup_r
_cleanup
fprintf
_fwalk
_InitRAM

Mode: Histogram

Avail Ranges: 32721

The dialog displays the functions, lines, and ranges to be profiled. The Available Ranges box displays the number of available profiler buckets (ranges) that can be added.

Profiler Setup Dialog Buttons and Controls:

Add All Functions

This button automatically creates a range for each function in your program, in ascending address order. If there are more functions than your emulator's profiler option supports, the first group will be defined and the Next Group of Functions button will be enabled. In this case, it will be necessary to execute your program multiple times to profile the entire executable image at the function level.

Next Group of Functions

This button is enabled only if a previous All Functions or Next Group of Functions ran out of profiler ranges before running out of functions. If so, clicking on this button resumes adding ranges where the previous operation left off.

Add Functions...

This button allows you to choose a specific function or list of functions for profiling, optionally line-by-line. See *Add Function(s)* below for more details.

Add Next Set of Lines

Similar to Next Group of Functions, this button will resume adding ranges for executable lines in the function, if profiling a function by lines previously overflowed the available number of profiler ranges. Note if you selected multiple functions, only the lines from the function that is left off will be added. Any other functions that did not previously get added will need to be re-added manually.

Add Arbitrary Range...

This button allows an arbitrary address range, optionally divided into a number of equal sized sub-ranges, to be added. It brings up the *Add Arbitrary Range Dialog* on page 107.

Delete

This button deletes the selected range(s).

Delete All

This button deletes all current ranges allowing a fresh specification to be created.

Mode

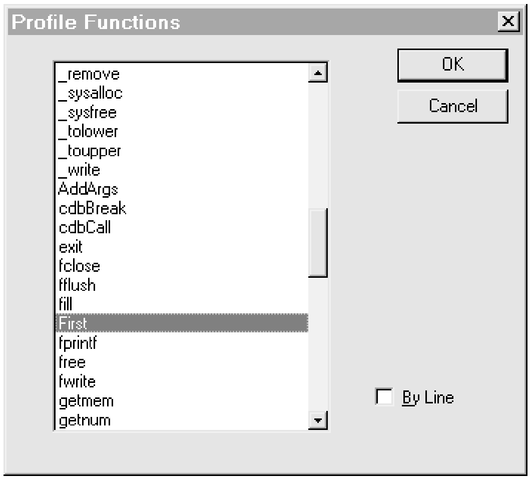
Displays and allows for the configuration of the profiler operation mode. The modes available will vary with the type of ICE in use. Please refer to your ICE manual for details on these modes.

OK

Accepts any changes made and closes the dialog.

Add Function(s)

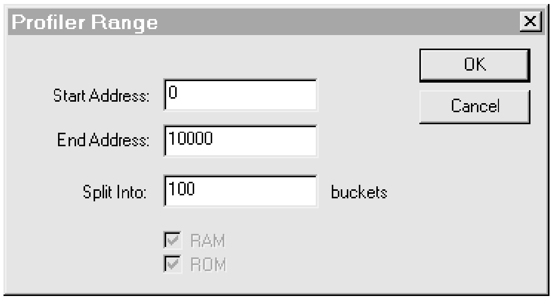
The Add Functions dialog box allows you to select function(s) to add to the profile list. If the function(s) is compiled with source level debug information turned on, you can check the By Line box to split the function into separate ranges by executable line. Below is an example of the dialog.



NOTE: If the By Line box is checked, and too many executable lines exist in the function(s) being profiled, as many ranges as possible will be added and the Add Next Set of Lines button will be enabled. However, if there are multiple functions being added at once, the remaining functions that were not added will be lost and must be entered again. You will get a warning message when this happens.

Add Arbitrary Range Dialog

This dialog box allows you to manually specify an address range to be added to the profiler list. Entering a number in the Split Into input box will split the range into that many equal-sized sub-ranges. If your target system has separate Instruction RAM and Instruction ROM address spaces, and you have not selected the Combine RAM & ROM Spaces menu item, you can specify which address space your range is in by checking the appropriate check box.



Start Address: 0

End Address: 10000

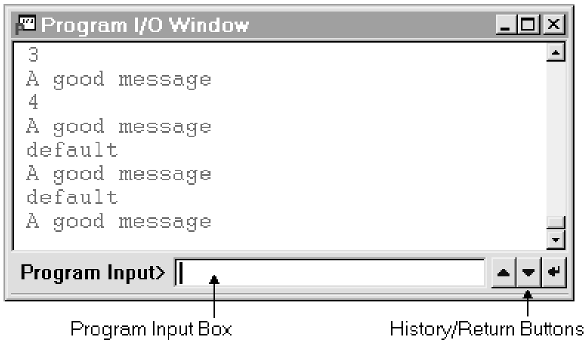
Split Into: 100 buckets

☑ RAM

☑ ROM

Program Input/Output Window

The Program I/O Window lets you interact with your target program through **stdin** and **stdout**. If your program is setup to use libraries for I/O that are compatible with MAJIC's interception methods, then the I/O is routed to your host via MAJIC/EDB. If the Program I/O Window is not open, input produced by your target program will force it open. Otherwise you can select Open from the menu or via a toolbar button (see the *Program I/O Command* on page 76).



```
3
A good message
4
A good message
default
A good message
default
A good message
```

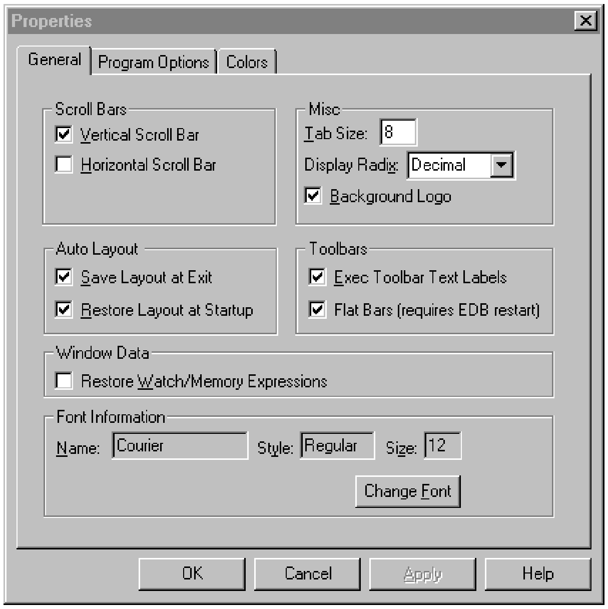
The Program I/O toolbar provides program input and history features. If you enter data in this input box it will be buffered (upon hitting <CR>) and provided to your target program to satisfy any calls for input from **stdin**. Note that input is always line based. If input is not available at the time your program requests it, then program execution will pause until you enter the data. Also note that if you are paused while waiting for input, any Stop requests will not be serviced until the current program input request is met. (See *Stop Command* on page 83.)

The Toolbar History buttons can be used to recall previous input, edit, and re-enter.

See also: *Session and Program I/O Shortcut Menu* on page 114, and *Session/ Program I/O Toolbar* on page 114.

Properties Dialog

General Properties



Scroll Bars
☑ Vertical Scroll Bar
□ Horizontal Scroll Bar

Misc
Tab Size: 8
Display Radix: Decimal
☑ Background Logo

Auto Layout
☑ Save Layout at Exit
☑ Restore Layout at Startup

Toolbars
☑ Exec Toolbar Text Labels
☑ Flat Bars (requires EDB restart)

Window Data
□ Restore Watch/Memory Expressions

Font Information
Name: Courier Style: Regular Size: 12

All the items below are saved in the **edb***xxx***.ini** file. See *Save Layout Command* on page 72 for more details about this.

Scrollbars — EDB's horizontal and vertical scrollbar can be set on (displayed) and off via these check boxes. This affects every EDB Window except the Watch and Profiler Data Windows.

Layout — These options control the saving and restoring of EDB's screen / window layout. We suggest setting the screen the way you like, saving it, and then turn off the save layout feature so that each time you start EDB it will have your setup defined. Layout information is normally kept in your Windows directory under the file named **edb***xxx***.ini**, where *xxx* is **sim** or **ice**.

Misc

Allows adjustment of EDB’s tab size, display radix (CDB’s **n** command), and background display. The tab size applies to any line where source code is displayed. The display radix effects program variable displays in both the Session and Watch Windows.

Toolbars

Selecting the Exec Toolbar Text Labels option turns on the display of text labels below the Execution Toolbar button icons (see *Execution Toolbar Command* on page 80). The text labels take up considerable screen space, so if you are using a small display screen you may want to leave this option off.

Selecting the Flat Bars option gives EDB’s toolbars a flat look and feel (sometimes referred to as *cool bars*). Changing this option requires you to restart EDB before the option change can take effect. This is due to bugs in the Microsoft Windows DLL files that provide the support for this feature.

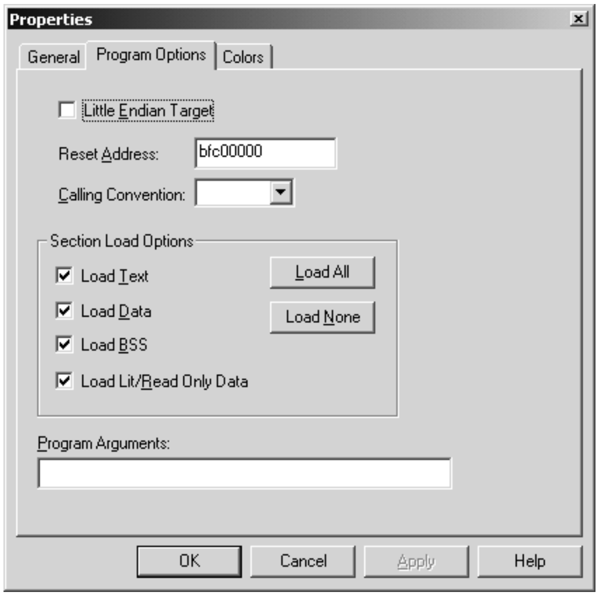
Window Data

The Restore Watch/Memory Expressions option allows you to control whether the expressions in these windows are saved and restored across debug sessions.

Font Information

This shows what EDB is using as the font for all its windows except the Watch Window. The Change Font button brings up a standard Windows Font dialog whereby you can adjust the font and font size.

Program Options Properties



□ Little Endian Target

Reset Address: bfc00000

Calling Convention:

Section Load Options

☑ Load Text

☑ Load Data

☑ Load BSS

☑ Load Lit/Read Only Data

Program Arguments:

Many of the options below may not be available on all types of execution vehicles. In such a case, the option will be grayed out.

Little Endian Target

Specifies whether the target is little endian or big endian.

Reset Address

Specifies the hexadecimal address at which the target begins execution when reset is released.

Calling Convention

This option allows you to tell the debugger which calling convention was followed by the compiler in the generation of the code in the program under test. o32 is the original MIPS standard R3000 calling convention. n32 is the newer MIPS standard for R4000 (MIPS 3 ISA and later) processors with 32-bit pointers. The EPI compiler generates n32 by default. o64 refers to the calling convention used by many GNU compilers for the R4000.

Note that failing to set this option correctly will cause both the Calls Window and the display of any stack-based variable to be incorrect.

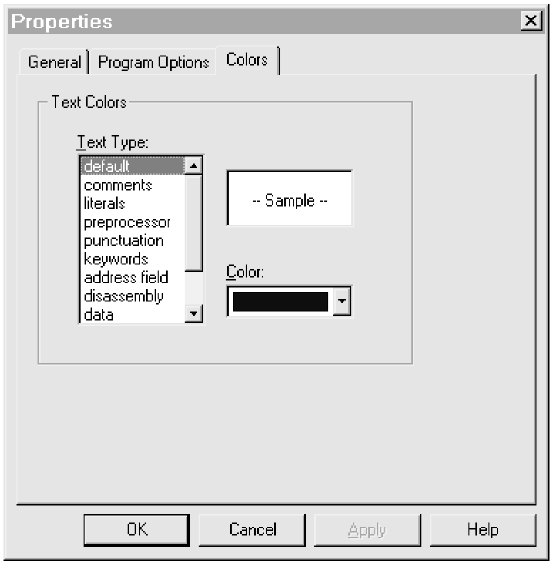
Section Load Options

This box lists the current load options, and allows you to modify them. Section types marked with a check will be downloaded the next time a download is required. Section types not marked with a check will not be downloaded. Load All and Load None buttons are simply shortcuts to turn all the check boxes on or off.

Program Arguments

Set or edit the command line arguments to be sent to your target program. For details, see the **r** command, described in *Run* on page 48.

Color Properties



This dialog allows you to configure the colors used for all various aspects of window data. However, data in the Session and Program I/O Windows is configured via MON's **EO** command. See the MON manual or MON's online help system for details.

Text Type

This list box identifies and allows you to select the current text type to examine or change its color. Below are listed the various text types and their usage.

default

Default text color applies to many items in different windows. Specifically, Watch Window names column, Register Window names column, Call Stack Window, Profiler Data, and the default color in the Session/ Program I/O Windows (when color is turned off).

comments

Color for code comments in Execution Window.

literals

Color for code literals (numbers) in Execution Window.

preprocessor

Color for C preprocessor directives in Execution Window.

punctuation

Color for C/C++ punctuation symbols in Execution Window.

keywords

Color for C/C++ keywords in Execution Window.

address field

Color for address fields (Execution and Memory Windows).

disassembly Color for disassembled code (Execution and Memory Windows).

data Color for general data display (Register, Trace, Watch (value column), and Memory Windows).

modified data Color for general modified data (Register, Trace, and Memory Windows).

symbols Color for symbols in the Memory Window.

errors Color for various error messages (does not apply to Session Window).

Colors This drop down list box allows you to configure the color for the selected text type.

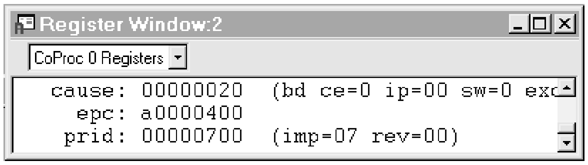
Sample This box shows an example of what the selected color will look like.

Register Window

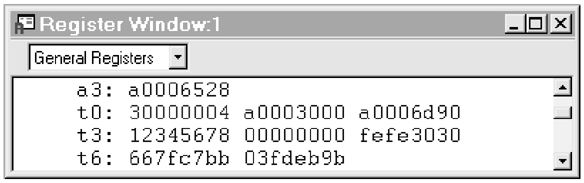
The Register Window lets you examine and modify your target's registers. The drop down box in the toolbar allows you to specify which logical set of registers you wish to view. Different processors will have different sets of registers available. You can also add custom registers and register windows (see *Custom Registers and Register Windows* on page 20).

Register editing is done on a per register basis. Simply position the cursor within the object and type a new value. Once you begin editing, an Edit box appears. The Edit box closes when you move the cursor off the current object, and the register is updated (written) with the new value. An edit operation can be aborted by pressing the <ESC> key.

You can have multiple register windows all displaying any register set available. Two Register Windows are shown below:



```
  cause: 00000020  (bd ce=0 ip=00 sw=0 exc
    epc: a0000400
   prid: 00000700  (imp=07 rev=00)
```



```
   a3: a0006528
   t0: 30000004 a0003000 a0006d90
   t3: 12345678 00000000 fefe3030
   t6: 667fc7bb 03fdeb9b
```

Register windows remember their contents across execution starts and stops and highlight registers that have changed value since the last time your program stopped. Note that the Register window does not allow the selection of RTOS based context views. However, the window's Register

Set drop-down box can be used to select CPU context views. RTOS users can use the global Default Context box to change the Register window's view, or manually display thread relative register sets by some simple commands in the Sessions Window, as follows:

```
   MON> dw r0 r31 ; vc <context>; dw r0 r31
```

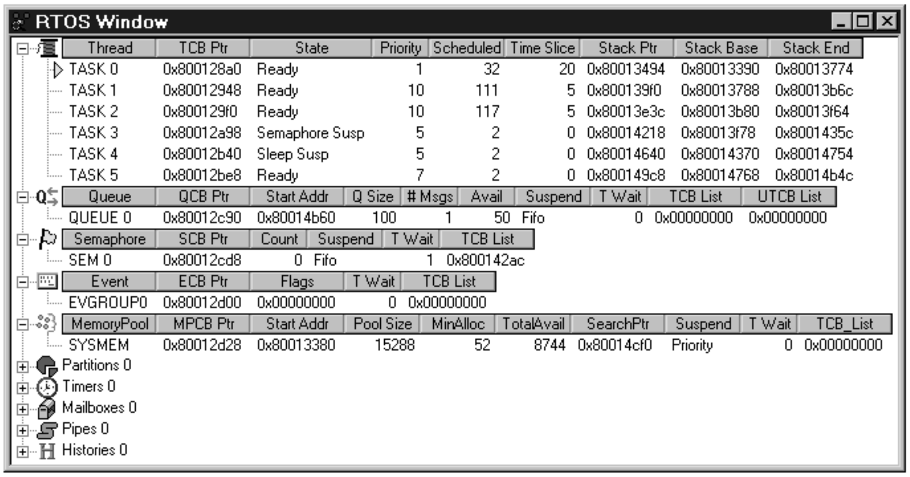
Register Window Toolbar

The Register Window toolbar provides a drop-down box that allows for selection of the logical set of registers to display. The available register sets vary depending upon the microprocessor in use.

The toolbar can be enabled or disabled via the Shortcut (right-click) menu.

RTOS Window

The RTOS Window, shown below, provides an easy way to browse through RTOS objects such as threads, queues, memory partitions, etc. The window presents each object class as a tree node with each object of that class in turn being a node of the class. A header line is displayed for each object class when opened. On the left of the Thread class list is a yellow arrow (current task pointer).


RTOS Window

Thread	TCB Ptr	State	Priority	Scheduled	Time Slice	Stack Ptr	Stack Base	Stack End
▷ TASK 0	0x800128a0	Ready	1	32	20	0x80013494	0x80013390	0x80013774
TASK 1	0x80012948	Ready	10	111	5	0x800139f0	0x80013788	0x80013b6c
TASK 2	0x800129f0	Ready	10	117	5	0x80013e3c	0x80013b80	0x80013f64
TASK 3	0x80012a98	Semaphore Susp	5	2	0	0x80014218	0x80013f78	0x8001435c
TASK 4	0x80012b40	Sleep Susp	5	2	0	0x80014640	0x80014370	0x80014754
TASK 5	0x80012be8	Ready	7	2	0	0x800149c8	0x80014768	0x80014b4c

Queue	QCB Ptr	Start Addr	Q Size	# Msgs	Avail	Suspend	T Wait	TCB List	UTCB List
QUEUE 0	0x80012c90	0x80014b60	100	1	50	Fifo	0	0x00000000	0x00000000

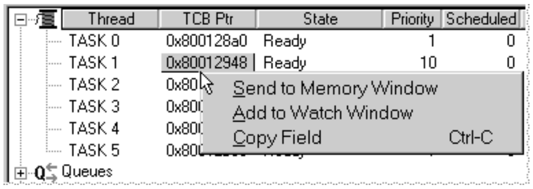
Semaphore	SCB Ptr	Count	Suspend	T Wait	TCB List
SEM 0	0x80012cd8	0	Fifo	1	0x800142ac

Event	ECB Ptr	Flags	T Wait	TCB List
EVGROUP0	0x80012d00	0x00000000	0	0x00000000

MemoryPool	MPCB Ptr	Start Addr	Pool Size	MinAlloc	TotalAvail	SearchPtr	Suspend	T Wait	TCB_List
SYSMEM	0x80012d28	0x80013380	15288	52	8744	0x80014cf0	Priority	0	0x00000000

Partitions 0
Timers 0
Mailboxes 0
Pipes 0
Histories 0

Data items that represent pointers to memory or objects in memory become buttons when the mouse cursor is hovered over the item. A left or right click on the selected button brings up the short-cut menu, shown below.



Thread	TCB Ptr	State	Priority	Scheduled
TASK 0	0x800128a0	Ready	1	0
TASK 1	0x80012948	Ready	10	0
TASK 2	0x80			
TASK 3	0x800			
TASK 4	0x800			
TASK 5	0x80			

Queues

The menu items perform the following functions:

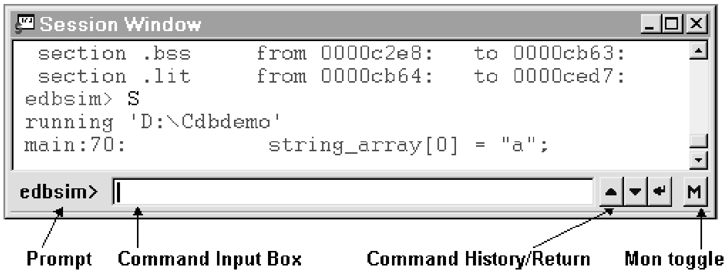
Send to Memory Window — This menu item opens a new memory window and configures it with the selected address

Add to Watch Window — If selectable, this menu item casts the given address to the proper data type and adds it to the watch window.

Copy Field — This menu items copies the selected data to the clipboard.

Session Window

The Session Window allows access to EDB's powerful command languages (CDB and MON) and gives visual feedback on many EDB operations.



```
  section .bss      from 0000c2e8:    to 0000cb63:
  section .lit      from 0000cb64:    to 0000ced7:
edbsim> S
running 'D:\Cdbdemo'
main:70:            string_array[0] = "a";
edbsim>
```

The session toolbar provides CDB command input and history features and full access to EPI's low level debugger MON via the MON toggle button. Note that a separate history buffer is used for both CDB versus MON Session mode.

Session and Program I/O Shortcut Menu

Shortcut Menu — The Shortcut (right-click) menu has the following commands:

Copy — Copy selection to the clipboard. Accelerator key: <Ctrl-C>

Select All — Selects all the text in the window. Accelerator key: <Ctrl-A>

Clear Window — Clear's the window's content.

Toolbar — Toggle the hidden / displayed state of the window's toolbar.

Properties — Brings up Properties dialog. Allows display and edit of program configuration.

Session/Program I/O Toolbar

This toolbar provides the Command Input/Edit box and History buttons. The Command Input/Edit box allows you to enter, edit, and recall both debugger commands (Session Window) and responses to program input requests (Program I/O Window). Normal Windows edit keys are supported including Cut, Copy, and Paste operations. The Up/Down arrow keys (or buttons) can be used to recall and edit previously entered commands.

The toolbar can be turned off via the Shortcut (right-click) menu. If a pending command input request is asked for that must be satisfied, the toolbar will automatically reactive itself.

For a summary of EDB command mode commands see Chapte r3, *Debugger Commands* on page 39.

Command History Recall Buttons

▲▼↵

These buttons allow you to recall and re-enter previously entered commands. The up and down arrows recall previous commands. The Execute (third) button operates the same as having pressed the <Enter> key.

Note that Session Window and Program I/O Window maintain separate command history buffers. Separate history buffers are also maintained in the Session Window for CDB vs. MON command mode.

Your history data is saved between debug sessions in a file called **`startedb.hst`**. Note that the MON and Program Input history data is relevant to and used by the command line version of MON as well.

MON Command Mode

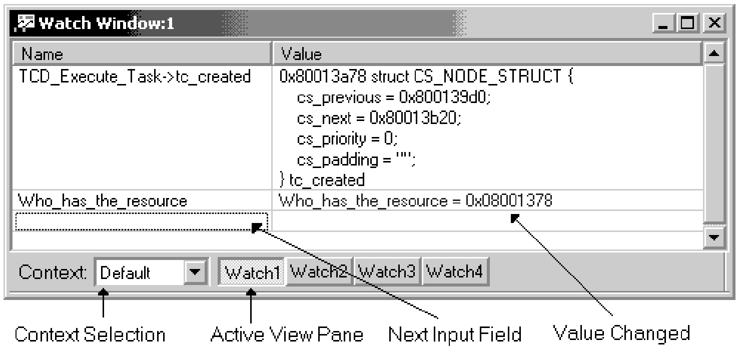
M

The Session Window supports two input modes. CDB commands (default) and MON commands. MON command mode is entered by typing the command **`mon`** at the prompt, or hitting the MON button.

Note that MON commands are not documented in this help file. Please refer to the hard copy MON manual or online help system via MON's **`h`** command.

Watch Window

The Watch Window provides a convenient way to do expression evaluation. Expressions entered in the Watch Window are updated each time your program stops executing. On each update the expression is compared against the old value and changes highlighted. If the expression result changes between updates the changed text is displayed in a different color. The last line of a Watch window is always the new item input line. You can select it and start typing to enter new items. The Watch Window supports multiple panes of data, and you can launch multiple Watch Windows. The Watch Window looks like:



Name	Value

TCD_Execute_Task->tc_created	0x80013a78 struct CS_NODE_STRUCT {
	cs_previous = 0x800139d0;
	cs_next = 0x80013b20;
	cs_priority = 0;
	cs_padding = "";
	} tc_created

Who_has_the_resource	Who_has_the_resource = 0x08001378

Context: Default

Adding Items

Items can be inserted in several ways. First, you can simply start typing on the row with focusor, <left-mouse-click> on a selected row. In either case an input box will pop-up to allow the editing to occur. Expressions can also be pasted from the clipboard via the standard Paste menu items or keys <Ctrl-V>. And, finally selected text from any EDB Window can be dragged and dropped on the Watch Window.

Selecting Items

The Watch window supports standard Windows list selection mechanisms. <Left-Click> selects an item and deselects any previous selections, <Ctrl-Left-Click> selects without deselecting any previous items, and shift-click extends a selection from the last <Left-Click> item to the new item. You can also use the arrow keys to move the selection around.

Deleting Items

Items can be deleted by selecting one or more items and pressing the <Delete> or <Ctrl-X> Cut keys. The <Cut> key will copy the items to the clipboard before the deletion occurs.

Editing Expressions

Expressions (the left side) can be edited by either hitting the mouse button on a selected expression or hitting the <CR> key on a selected. The relative size of the header fields to one another can be adjusted by a <Left-Click-Hold> and drag on the dividing line. Note that when sizing the Watch Window the columns will each resize proportionally.

Editing Values

Values (the right side) can be changed by clicking on a changeable value line. This brings up a Dialog Window allowing a new value to be entered. Lines of structures that do not display values (display syntax) are not editable.

Header Fields

The relative size of the header fields can be adjusted by a <Left-Click-Hold> and drag on the dividing line. Note that when sizing the Watch Window the columns will each resize proportionally.

Shortcut Menu

The Watch window shortcut (right-click) menu, described below, provides access to the standard features like Cut, Copy, etc., but also allows you to configure the display radix, and the actively displayed window pane (Watch 1..4).

Toolbar

The Watch window toolbar provides access to the four watch panes, and the ability to change to the view context associated with the window. (See *Watch Window Toolbar* on page 118.) The watch panes provide a convenient means to watch function relative values and display only when needed.

NOTE: Unlike most of EDB's windows, the Watch window's font and scroll bars are not affected by the Properties options (See *Properties Dialog* on page 108). Instead, the vertical scroll bar is always on and the horizontal scroll is off in favor of the Watch Tips feature: if a value in either the Name (expression) or Value (expression result) column overflows the column width, you can view the full text by hovering the mouse over the field.

Watch Window Shortcut Menu

The Watch Windows Shortcut menu is dependent upon a left or right column click. Below are the menu items common to both:

Cut

Cut the selection to the clipboard. Accelerator key: <Ctrl-X>

Copy

Copy selection to the clipboard. Accelerator key: <Ctrl-C>

Paste

Paste selection from the clipboard. Accelerator key: <Ctrl-V>

Clear

Delete or Clear selected items. Accelerator key: <Del>

Select All

Select all the items in the window. Accelerator key: <Ctrl-A>

Radix

(Pop-up menu - Hexadecimal, Decimal, Octal, Binary) Configures the global display radix for all expression output in EDB.

Watch 1..4

Display panes 1..4 of the Watch Window.

Toolbar

Toggle the hidden / displayed state of the Watch Windows toolbar.

Properties

Bring up Properties dialog (see *Properties Dialog* on page 108). Allows display and edit of program configuration.

Left Column Shortcut Menu Additions:

Dereference — De-reference the clicked-on expression.

Un-Dereference — Un-Dereference the clicked-on expression.

Right Column Shortcut Menu Additions:

Edit Value — Edit the clicked-on value (if allowed).

Dereference — De-reference the clicked-on value.

- and Add Item — De-reference the clicked-on value and add as a new expression.
- and Replace Item — De-reference the clicked-on value and replace current expression with new value.

Reference — Reference the clicked-on value.

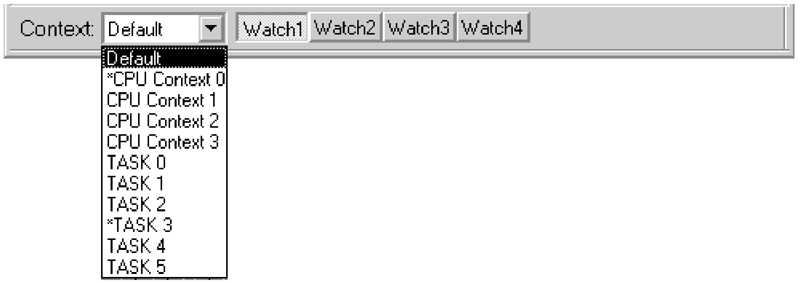
- and Add Item — Reference the clicked-on value and add as a new expression.
- and Replace Item — Reference the clicked-on value and replace current expression with new value.

Watch Window Toolbar

The Watch Window toolbar provides four buttons that each activates a different watch pane. Each pane contains its own set of watch expressions. This feature is useful to not only provide more room for watch data expressions, but also organize your watch data based on relevant scopes (for example, put data variables related to one function / breakpoint in one watch pane).

The Watch Window toolbar can be enabled or disabled via the Shortcut (right-click) menu.

Below is a sample Watch Window toolbar showing the Context drop-down list.



Index