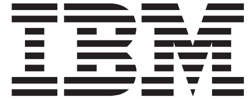
IBM® Distributed Debugger, Version 9



Distributed Debugger for Workstations

Note!

Before using this information and the product it supports, be sure to read the general information under **Notices**.

Edition notice

This edition applies to Version 9 of the IBM Distributed Debugger and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1999, 2002. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Distributed Debugger 1
Distributed Debugger: Source and Control Panes . . 1
Distributed Debugger: Value Panes 2
Recursion and debugging 3

Chapter 2. Preparing a program for debugging. 5
Writing a program for debugging 5
Compiling a program for debugging 5
UNIX call handling during debugging. 5
  exec() handling 5
  fork() handling 5
  system() handling. 6
  Optimized code debugging 6

Chapter 3. Starting the Distributed Debugger 9
Setting environment variables for the Debugger . . 9
Starting the debugger for local debugging 9
Starting the debugger for debugging compiled languages remotely 9
Remote debugging 10
Starting the Distributed Debugger user interface daemon. 11
Attaching to a running process 11
  When to attach 11
  Attaching to a local running process 11
  Attaching to a remote running process 12

Chapter 4. Working with breakpoints 15
Breakpoints 15
Setting breakpoints 15
  Setting a line breakpoint 15
  Setting an address breakpoint 16
  Setting a function breakpoint 16
  Setting a storage change breakpoint 17
  Setting a load occurrence breakpoint 17
  Setting a conditional breakpoint 18
  Setting a deferred breakpoint 18
  Setting multiple breakpoints. 18
  Viewing set breakpoints 19
  Modifying breakpoint properties 19
  Enabling and disabling breakpoints 19
  Deleting a breakpoint 20

Chapter 5. Controlling program execution 23
Running a program. 23
Exception Handling 23
  Selecting debugger recognized exceptions . . . 24
Stepping through a program. 24
Step commands 25
Skipping over sections of a program 26
Halting execution of a program. 27
Restarting a program 27

Chapter 6. Inspecting data 29
Inspecting variables 29
  Adding a variable or expression to the Monitors pane. 29
  Viewing the contents of a variable or expression 29
  Changing the contents of a variable 30
Inspecting registers. 30
  Viewing the contents of a register 30
  Changing the contents of a register 31
  Adding a register to the Monitors pane 31
Inspecting storage 32
  Viewing a location in storage 32
  Changing the representation of storage contents 33
  Changing the contents of a storage location. . . 33
  Adding a new Storage Monitor pane for an expression or register 33
  Heap errors 34
Enabling and disabling a monitored variable, expression or register 36
Enabling tool tip evaluation for variables 36
Changing the representation of monitor contents . . 37

Appendix A. Distributed debugger commands 39
idebug command 39
  Warning: Temporary Level 3 Header 39
irmtdbgc command. 42

Appendix B. Optional breakpoint parameters 45

Appendix C. Step commands 47

Appendix D. Environment variables . . 49
PATH environment variable 49
INCLUDE environment variable 49
LIBPATH environment variable. 49
DER_DBG_CASESENSITIVE environment variable 49
DER_DBG_PATH environment variable 49
DER_DBG_TAB environment variable 50
DER_DBG_TABGRID environment variable . . . 50
DER_DBG_DEEP_STEP_DEBUG 50

Appendix E. Postmortem debugging . . 51
Errors during UNIX workstation postmortem debugging. 51
Postmortem debugging on AIX 51
Debugging Dump Files 51

Appendix F. Limitations 53
Remote debug limitations 53

© Copyright IBM Corp. 1999, 2002

Limitations during postmortem debugging 53
Unusual debugger behavior 53

Appendix G. Program profiles 55

Appendix H. C/C++ expressions supported 57
C/C++ supported data types 57
C/C++ supported expression operands 57
C/C++ supported expression operators 58
C/C++ compiler options on workstation UNIX platforms 59

Notices 61
Programming interface information 63
Trademarks and service marks 63

Chapter 1. Distributed Debugger

The Distributed Debugger is a client/server application that enables you to detect and diagnose errors in your programs. This client/server design makes it possible to debug programs running on systems accessible through a network connection as well as debug programs running on your workstation.

The debugger server, also known as a *debug engine*, runs on the same system where the program you want to debug runs. This system can be your workstation or a system accessible through a network. If you debug a program running on your workstation, you are performing *local debugging*. If you debug a program running on a system accessible through a network connection, you are performing *remote debugging*.

The Distributed Debugger client is a graphical user interface where you can issue commands used by the debug engine to control the execution of your program. For example, you can set breakpoints, step through your code and examine the contents of variables. The Distributed Debugger user interface lets you debug multiple applications, which may be written in different languages, from a single debugger session. Each program you debug is shown on a separate program page with a tab on each page displaying program identification information such as the name of the program being debugged. The type of information displayed depends on the debug engine that you are connected to.

Each program page is divided into different sections, called *panes*. Each pane displays different information about your program. There are panes to display your source code, breakpoints, the program's call stack and various monitors. The types of control and value panes available on a program page depend on the program you are debugging.

For more information on the control and value panes available in the Distributed Debugger user interface, see the related topics below.

Distributed Debugger: Source and Control Panes

Selected entries in the Control Panes have a direct relationship with what is displayed in the Source Pane. For example, if you expand the stack view and click on one of the entries, the source pane displays the source for the selected entry. Similarly, if you click on a source file in the modules pane, that source file appears in the source pane.

The types of control panes displayed when debugging a program depend on the programming language used. The following control panes are available in the Distributed Debugger user interface:

Stacks pane

The Stacks pane provides a view of the call stack for each thread in the program you are debugging. Each thread in your program appears as a node in a tree list. Expanding a node will display the names of active functions for that thread. The thread you are debugging is highlighted. When debugging interpreted Java, the name of the thread will be displayed.

© Copyright IBM Corp. 1999, 2002

Breakpoints pane

The Breakpoints pane contains a view of information about the breakpoints you have set in the program you are debugging. Use the Breakpoints pane to view breakpoints set in your program, modify their properties, enable or disable breakpoints, delete them, or add new ones.

Source pane

The Source pane provides a view of the source code for the program you are debugging. If your program was compiled with debugging information, you have three choices as to how to view it: by its source code, its disassembled machine code, or a combination of the two. To view source code, the source code must be accessible to the debugger engine or the debugger user interface, either on a local or network drive. If the source code file is not found, or your program was not compiled with debugging information, only a disassembled machine code view is available.

Note: Disassembly is only shown for compiled language programs.

Modules pane

The Modules pane displays a list of modules loaded while running your program. The items in the list can be expanded to show compile units, files and functions.

The remaining panes are value panes. For more information on value panes, see the related topics below.

Note: When viewing a Control pane with expandable items, choosing the **Expand All** option from the pane's menu in the debugger main menu bar will expand one level of the tree at a time. You must choose **Expand All** for each subsequent level of the tree that you wish to expand. Choosing **Collapse All** will collapse all levels of the tree.

Distributed Debugger: Value Panes

Depending on the language you are debugging, the Distributed Debugger provides you with value panes to observe various aspects of your program. The following value panes are available in the Distributed Debugger user interface:

Variables and Expressions (Monitors pane)

The Monitors pane shows variables and expressions that you have selected to monitor. You can enter the variables or expressions in a dialog box or select them from the Source pane. Use the Monitors pane to monitor global variables or variables you want to see at all times during your debugging session. From the Monitors pane you can also modify the content of variables, or change the representation of values.

Tip: Enabling tool tip evaluation for variables provides a quick way to view the contents of variables in the Source pane. When you point at a variable, a pop-up appears displaying the contents of that variable. If hover help for variables is disabled and you want to enable it, see the related topic below.

Local Variables (Locals pane)

The Locals pane helps you monitor all local variables in scope at the current execution point of your program. The Locals pane is updated after each Step or Run command to show what variables are currently in scope and the contents of those variables. It is also used to modify the content of variables or change the representation of values.

Registers (Registers pane)

The Registers pane allows you to view and change the contents of processor registers for the threads in your program. Registers values are unique for each thread of your program. The registers are categorized, so you only need to expand the category of registers that you wish to view.

Storage (Storage pane and Storage Monitor panes)

The Storage pane and Storage Monitor panes let you view and change the contents of storage areas used by your program. You can also change the address range to view and modify the contents of storage, and change the representation the debug engine uses to display storage.

The initial Storage pane shows the storage areas used by your program at its starting address.

You can add additional Storage Monitor panes that start at the address of storage allocated to a register, variable, array, class object or expression.

Note: When viewing a Value pane with expandable items, choosing the **Expand All** option from the pane's menu in the debugger main menu bar will expand one level of the tree at a time. You must choose **Expand All** for each subsequent level of the tree that you wish to expand. Choosing **Collapse All** will collapse all levels of the tree.

Recursion and debugging

Recursion does not have to involve a routine calling itself directly; for example: FUNC1 calls FUNC2 calls FUNC3 calls FUNC1. After the call to FUNC3, each time you step into one of these routines, the entry for that call shows a recursion count one higher than the previous entry for that call on the Stacks pane.

You can use the recursion value in the stack frame properties box to detect unintentionally recursive calls.

Limits to debugging recursive function calls

Only the copy of the variables from the most recent invocation of a function can be monitored. Variables from previous invocations of the recursive function cannot be monitored.

Chapter 2. Preparing a program for debugging

Writing a program for debugging

You can make your programs easier to debug by following these simple guidelines:

- Do not hand-tune your source code for performance until you have fully debugged and tested the untuned version. Hand-tuning may make the logic of your code harder to understand.
- Where possible, do not put multiple statements on a single line, because some Distributed Debugger features operate on a line basis. For example, you cannot step over or set line breakpoints on more than one statement on the same line.
- Assign intermediate expression values to temporary variables to make it easier to verify intermediate results by monitoring the temporary variables.

To debug programs at the level of source code statements, you must specify the compiler options that generate debug information. In some cases, you must specify additional options that enable the debug engine to work properly with your code.

Compiling a program for debugging

In order to debug your program at the source code level, you need to compile your program with certain compiler options that instruct the compiler to generate symbolic information and debug hooks in the object file. See your compiler reference documentation on how to compile your program with debug information.

UNIX call handling during debugging

exec() handling

When a process calls `exec()`, a new program is loaded to replace the current program.

The debugger suspends program execution at this point and opens a dialog, which allows you to choose whether to debug program initialization or whether to use a program profile. The name of the new program is shown, but, you cannot change the name. After you select **OK**, the debugger stops at the first instruction of the new program's runtime (if you asked to debug program initialization), or at the first instruction or statement in the new program.

fork() handling

When a process calls `fork()`, an exact copy of that process is created. The process that forked is called the parent, and the new process is called the child. If a process being debugged forks, the Distributed Debugger stops both the parent and child processes, and opens a dialog box that lets you choose whether to continue debugging the parent process or switch to the child process.

Whichever choice you make (**Parent** or **Child**), the Distributed Debugger ignores the process you did *not* choose, and allows it to continue running. The debugger will halt the selected process, which then allows you to perform debugging activities, such as adding a breakpoint. The process remains halted until explicitly

© Copyright IBM Corp. 1999, 2002

restarted .Breakpoints set in the process you did not choose are ignored. Execution stops in the process that you chose to debug.

If the process you did *not* choose performs an `exec()`, the Distributed Debugger will allow
you to debug the new process. Look under the exec() handling related reference for additional information.

system() handling

AIX SOLARIS Restriction: This is supported on AIX and Solaris only.

When a program running in a UNIX environment starts another program using a call to `system()`, the `system()` function calls both `fork()` and `exec()`. The following describes the Distributed Debugger's behavior after you perform a **Step Over** command on a line containing a `system()`call, and tells you what actions you should take to begin debugging the child process.

1. The `system()` function calls `fork()`. The Distributed Debugger stops execution and raises a Process fork action dialog.
2. At this point you should choose to debug the child process. Once the Process fork action dialog closes, issue the **Run** command to continue debugging the child process.
3. The new child process calls `exec()`to load `/bin/sh`, and the debugger opens a New process dialog and the active Distributed debugger Source pane shows a disassembly view of the initial runtime entry point of `/bin/sh`.
4. Click **OK** to start debugging the child process.
5. The Distributed Debugger stops in the main function of `/bin/sh`.
6. Issue the **Run** command.
7. AIX The /bin/sh process calls exec() to load the program specified in the call to system() in the original program. The Distributed Debugger opens a New process dialog and the active Distributed Debugger Source pane shows a disassembly view of the initial runtime entry point of the program specified in the call to `system()`.
8. Click **OK**. The Distributed Debugger stops at `main()`. From here you can continue debugging.

Optimized code debugging

Problems that only surface during optimization are often an indication of logic errors or compile errors that are exposed by optimization, for example using a variable that has not been initialized. If you encounter an error in your program that only occurs in the optimized version, you can usually find the cause of the error using a binary search technique to find the failing module:

1. Begin by optimizing half the modules and see if the error persists.
2. After each change in the number of optimized modules, if the error persists, optimize fewer modules; if the error goes away, optimize more modules. Eventually you will have narrowed the error down to a single module or a small number of modules.
3. Debug the failing module. If possible, turn off the instruction scheduling optimizations for that module. Look for problems such as reading from a variable before it has been written to, and pointers or array indices exceeding the bounds of storage allocated for the pointer or array.

When you debug optimized code, information in debugger panes may lead you to suspect logic problems that do not actually exist. You should bear in mind the points below:

Local variables are not always current

Do not rely on the Local variables monitor to show the current values of variables. Numeric values, character values and pointers may be kept in processor registers. In the optimized program, these values and pointers are not always written out to memory; in some cases, they may be discarded because they are not needed.

Static and external variables are not always current

Within an optimized function, the values of static or external variables are not always written out to memory.

Registers and Storage monitors are always current

The Registers and Storage monitors are correct. Unlike a monitor that shows actual variables, such as the Locals Variables monitor, the Registers and Storage monitors are always up-to-date as of the last time execution stopped.

Source statements may be optimized away

Using the disassembly view or mixed view to see the machine code for your program, you may find, for example, that an assignment to a variable in your source code does not result in any disassembly code being produced; this may indicate that the variable's value is never used after the assignment.

Chapter 3. Starting the Distributed Debugger

Setting environment variables for the Debugger

The Distributed Debugger user interface running on a workstation uses environment variables to specify the settings for a variety of factors, such as the location of source files for the client application, the location of the executable for that application, and case-sensitivity when doing particular name comparisons.

You may want to set environment variables for the debug engine and Distributed Debugger client. You can set the environment variables based on your operating system. For instructions on setting environment variables refer to your operating system manuals.

All variables listed in the related references below can be set on the host where the debug engine is located. The following variables are also used by the Distributed Debugger client: DER_DBG_PATH, CLASSPATH, and PATH.

Starting the debugger for local debugging

AIX WIN **Restriction:** Local debugging is supported on AIX and Windows only.

To start debugging a program locally from the command line, issue the `idebug`command with local debug parameters at a command line prompt.

If you issue the `idebug`command without any options, the debugger will prompt you for the required information in the Load Program dialog.

Tip: If you have debugged a specific program before and do not want to use the previous profile options, make sure that **Use program profile** is not selected or use the `-p-` option of the `idebug` command.

Once the debugger user interface is running, you can debug other programs using the same debugger session by selecting **File > Load Program** from the main menubar.

Starting the debugger for debugging compiled languages remotely

AIX WIN SOLARIS Restriction: Remote debugging is only supported on AIX, Solaris, HP-UX, and Windows.

The debugger allows you to run the debugger user interface and the debug engine on separate machines. These separate machines can be running different operating systems. When you start the debugger for remote debugging, you first start a debug engine daemon. This daemon waits for a connection from the debugger user interface. Once a connection is established, you can begin to debug your program.

To start debugging a remote program from the command line:

1. On the remote system, start the debug engine with the `irmtdbgc`command at a command line prompt. For information on `irmtdbgc`command parameters, see the related topic below.

© Copyright IBM Corp. 1999, 2002

2. On the local system, use the `idebug`command to start the Debugger user interface and then select **TCP/IP connection** in the Load Program dialog and enter the name of the host where you started the debug engine to make the connection.

   Alternatively, you can use a command line after starting the debug engine to issue the `idebug`command, using the remote debug parameters. If you use this method to make the connection, you must specify the `-qhost` parameter and the `-qlang`parameter for the language you are debugging. For information on the `idebug`command parameters, see the related topic below.

Tip: The debug engine is terminated if the debugger cannot load the program you want to debug. Also, the debug engine is terminated when the program you are debugging runs to completion or is terminated manually. To prevent the debug engine from being terminated in these situations, use the `-qdaemon`option of the `irmtdbgc` command.

Remote debugging

Debugging a program running on one system while controlling the program from another system is known as remote debugging. The debugger supports remote debugging by allowing you to run the debugger user interface on one system, while running the debug engine on another system. The system running the debugger user interface is known as the *local* system. The system where the debug engine runs is known as the *remote* system.

When debugging a program remotely, you can start the debugger in one of two ways:

- Start a debug engine daemon, then start the debugger user interface.
- Start a debugger user interface daemon, then start a debug engine.

In both cases, a daemon will listen for a connection. Once a connection is made you can begin to debug your program.

Why use remote debugging
You might want to use remote debugging for the following reasons:

- *The program you are debugging is running on another user's system, and is behaving differently on that system than on your own.*

  You can use the remote debug feature to debug this program on the other system, from your system. The user on the system running that program interacts with the program as usual (except where breakpoints or step commands introduce delays) and you are able to control the program and observe the program's internal behavior from your system.
- *You want to debug a program that uses graphics or has a graphical user interface.*

  It is easier to debug an application that uses graphics or has a graphical user interface when you keep the debugger user interface separate from that of the application. Your interaction (or another user's interaction) with the application occurs on the remote system, while your interaction with the debugger occurs on the local system.
- *The program you are debugging was compiled for a platform that the debugger user interface does not run on.*

  You can use the remote debug feature to take advantage of the debugger user interface while debugging the remote application.

390 400 **Restriction:** This information applies to remote debugging between workstation platforms only. For information for debugging an OS/390 or AS/400 program from a workstation, see the online for help for the Distributed Debugger shipped with products that support OS/390 or AS/400.

Starting the Distributed Debugger user interface daemon

Start the Distributed Debugger user interface in daemon mode if you want the Distributed Debugger user interface to appear only after you have started a debug engine.

To start the Distributed Debugger user interface daemon, issue the following command at a command line prompt:

```
idebug -qdaemon -quiport=<port>
```

where `<port>`is the port number where you want the Distributed Debugger user interface daemon to listen for a debug engine.

AIX SOLARIS WIN When you start the debug engine that will connect to this daemon, the port specified with the `-quiport`option to the `irmtdbgc`command must be the same as the one specified with the `-quiport` option to the `idebug`command.

Attaching to a running process

When to attach

There are two main reasons for attaching the Debugger to a process or JVM:

- You anticipate a problem at a particular point in your program, and you do not want to step through the program or set breakpoints. In this situation, you can run your program, and during a program pause shortly before the anticipated failure (for example, while the program is waiting for keyboard input), you attach the Debugger. You can then provide the input, and debug from that point on.
- You are developing or maintaining a program that hangs sporadically, and you want to find out why it is hanging. In this situation, you can attach the Debugger, and look for infinite loops or other problems that might be causing your program to hang.

Note: You can attach the Debugger to an already running program or a running Java Virtual Machine (JVM) where an error or failure has occurred.

Attaching to a local running process

AIX WIN **Restriction:** Attaching to a local running process is only supported on AIX and Windows.

You can attach the debugger to a running process either by using the Attach dialog or from a command line by using the `-a` option of the `idebug`command. See the related topic below on when to attach to a running process.

To attach the debugger to a running process with the Attach dialog:

1. Select **File > Attach** to invoke the Attach dialog.
2. Select the **Compiled** tab.
3. Select the **Local** radio button.

4. Select the dominant language of the program.
5. Optionally, enter the full path name to the executable associated with the process you want to attach, in the **Process Path** field.
6. If you do not know the Process ID of the process you want to attach to, click **Get Process List**. The Process List dialog provides a list of the processes running on the local machine. Select a process in the **Process Path** field and click **OK** to close the Process List dialog.

   or

   If you know the Process ID, click the **Enter Process ID** radio button and enter the Process ID in the **Process ID** field.
7. Click **Use program profile** if you want to use this feature.
8. Click **Attach** to attach to the process and close the Attach dialog.

To attach the debugger to a running process from a command line, enter the following command:

AIX WIN
```
idebug -a<process_id>
```

where <process_id> is a valid process id on your system.

Important: Do not attach to operating system processes or to the debugger's own processes. Attaching to such processes can cause unpredictable results.

AIX The debugger detaches from the process on debugger exit. The **Terminate** button can be used to terminate an attached process. To detach without exiting the debugger, use the **Detach** button or select **Debug> Detach** from the menu bar.

You cannot restart a program that you have attached to.

Attaching to a remote running process

You can attach the debugger to a process running on a remote system either by using the Attach dialog or from a command line by using the -a option of the idebugcommand. See the related topic below on when to attach to a running process.

To attach the debugger to a running process from a command line:
1. On the remote system, start the debug engine using the irmtdbgccommand. If you specify the -qportoption, take note of it. You will need it later. The default port is 8000.
2. On the local system, enter the following command:
   ```
   idebug -qhost=<remote_host> [-qport=<host_port>][-qlang=<dominant
   language>] -a<process_id>
   ```

where <remote_host>is the the TCP/IP name or address of the remote system, and <process_id>is a valid process id on the remote system.

Important: Do not attach to operating system processes or to the debugger's own processes. Attaching to such processes can cause unpredictable results.

To attach the debugger to a running process on a remote system with the Attach dialog:
1. Select **File > Attach** or issue idebug -a to invoke the Attach dialog.
2. Select the **Compiled** tab.

3. Select the **Remote** radio button. Make sure you have already started the engine. Enter the host name and port number where the engine is listening.
4. Select the dominant language of the program.
5. Optionally, enter the full path name to the executable associated with the process you want to attach, in the **Process Path** field.
6. If you do not know the Process ID of the process you want to attach to, click **Get Process List**. The Process List dialog provides a list of the processes running on the machine entered in the **Process Path** field. Select a process from the list and click **OK** to close the Process List dialog.
   or
   If you know the Process ID, click the **Enter Process ID** radio button and enter the Process ID in the **Process ID** field.
7. Click **Use program profile** if you want to use this feature.
8. Click **Attach** to attach to the process and close the Attach dialog.

AIX SOLARIS The debugger detaches from the process on debugger exit. The **Terminate** button can be used to terminate an attached process. To detach without exiting the debugger, use the **Detach** button or select **Debug> Detach** from the menu bar.

You cannot restart a program that you have attached to.

Chapter 4. Working with breakpoints

Breakpoints

Breakpoints are temporary markers you place in your executable program to tell the Distributed Debugger to stop your program whenever execution reaches that point. For example, if a particular statement in your program is causing problems, you could set a breakpoint on the line containing the statement, then run your program. Execution stops at the breakpoint before the statement is executed. You can check the contents of variables, registers, storage, and the stack. You can then step over (execute) the statement to see how the problem arises or you can choose to skip the execution of the statement in question.

The Distributed Debugger supports the following types of breakpoints:

- **Line breakpoints** are triggered before the code at a particular line in a program is executed.
- C C++ **Function breakpoints** are triggered when the function they apply to is entered.
- **Storage change breakpoints** are triggered when storage at a specified address is changed. AIX Storage change breakpoints are not available when debugging programs running on AIX.
- **Load occurrence** breakpoints are triggered when a DLL is loaded.
- AIX WIN **Address breakpoints** are triggered before the disassembly instruction at a particular address is executed.

Setting breakpoints

Setting a line breakpoint

Line breakpoints can be set from either of the following:

- Source pane (page 15)
- Breakpoints menu (page 15).

To set a line breakpoint in the Source pane:

1. Ensure the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View**.
2. Make sure the appropriate line is visible in the Source pane by using the scroll bar or cursor keys to locate the line.
3. Do one of the following:
   - Double-click on the line number in the prefix area of the line.
   - Right-click on the line you want to set a breakpoint on, and select **Set Breakpoint** from the pop-up menu.

To set a line breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Line** from the menu bar.
2. Enter the name of the module or routine in which you want to set a breakpoint in the **Executable** or **Package** entry field in the Line Breakpoint dialog. If this module or routine is loaded, you can select it from the pulldown list in the **Executable** or **Package** entry field.

© Copyright IBM Corp. 1999, 2002

3. Choose or enter the object, class or source file that is associated with the module or routine specified in the **Executable** or **Package** entry field and contains the line where the breakpoint is to be set from the **Object** pulldown list.
4. Choose the source file containing the code for the object or class file from the **Source** pulldown list. (This step is optional if you have not selected to defer the breakpoint.)
5. Enter the line number within the source file where you want to place a breakpoint in the **Line** entry field
6. If the module or routine you entered in the **Executable** or **Package** entry field is not currently loaded, click on the **Defer breakpoint** check box.
7. Set any optional parameters that you want for the breakpoint.
8. Click **OK** to set the breakpoint and dismiss the Line Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Line Breakpoint dialog.

Setting an address breakpoint

AIX WIN **Restriction:** This is supported for AIX and Windows only.

You can set an address breakpoint from the Source pane, and from the Breakpoints menu.

To set an address breakpoint from the Source pane:

1. Ensure the Source pane is set to a disassembly or mixed view. To set the Source pane to a disassembly view, select **Source > Disassembly View**. To set the Source pane to a mixed view, select **Source > Mixed View**.
2. Make sure the appropriate line is visible in the pane by using the scroll bar or cursor keys to locate the line.
3. Double-click on the address in the prefix area of the line, or right-click on the Disassembly or Mixed view and select **Set Breakpoint** from the pop-up menu.

To set an address breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Address** from the menu bar.
2. Enter either the address where you want to set the breakpoint or an expression that evaluates to an address. The address must be entered in hexadecimal notation.
3. Set any optional parameters that you want for the breakpoint.
4. Click **OK** to set the breakpoint and dismiss the Address Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Address Breakpoint dialog.

Setting a function breakpoint

You can set function breakpoints from any of the following:

- Modules pane (page 16)
- Breakpoints menu (page 17)

To set a function breakpoint from the Modules pane:

1. Expand the list in the Modules pane until you see the function you want.
2. Right-click on that function and select **Set Function Breakpoint** from the pop-up menu.

To set a function breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Function** from the menu bar.
2. Enter the name of the executable which contains the function where you want to set a breakpoint in the **Executable** entry field in the Function Breakpoint dialog. This step is optional unless the executable is not loaded. If this executable is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. Optionally, choose or enter the object, class or source file for the executable specified in the **Executable** entry field from the **Object** pulldown list.
4. Enter the name of the function where the breakpoint is to be set in the **Function** entry field in the Function Breakpoint dialog. If this function is loaded, you can select it from the pulldown list in the **Function** entry field.
5. If the module or routine you entered in the **Executable** entry field is not currently loaded, click on the **Defer breakpoint** check box.
6. Set any optional parameters that you want for the breakpoint.
7. Click **OK** to set the breakpoint and dismiss the Function Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Function Breakpoint dialog.

Setting a storage change breakpoint

Restrictions: Storage change breakpoints are not supported on Solaris or when debugging AIX data types other than 64-bit.

Storage change breakpoints halt execution of your program whenever storage at a specific address is changed. For example, if a byte being watched contains X'40' and the program writes X'40' to that byte, the storage change breakpoint is not triggered. If the program writes X'41', the storage change breakpoint is triggered.

To set a storage change breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Storage Change** from the menu bar.
2. Enter an address or expression that evaluates to an address in the **Address or Expression** field.

   C C++ **Tip:** You can enter the address of a variable by specifying the variable name preceded by an ampersand (&).
3. Specify the number of bytes to be monitored in the **Bytes to Monitor** field.
4. Set any optional parameters that you want for the breakpoint.
5. Click **OK** to set the breakpoint and dismiss the Storage Change Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Storage Change Breakpoint dialog.

Caution: If you set a storage change breakpoint for any address that is on the call stack, be sure to remove the breakpoint before leaving the routine associated with it. Otherwise, when you return from the routine, the routine's stack frame will be removed from the stack, but the breakpoint will still be active. Any other routine that gets loaded on the stack will then contain the breakpoint.

Setting a load occurrence breakpoint

Load occurrence breakpoints halt execution of your program when the DLL or dynamically loaded module specified is loaded into memory. You can set load occurrence breakpoints from the Breakpoints menu.

To set a load occurrence breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Load Occurrence** from the menu bar.
2. Enter the name of the DLL or dynamically loaded module to set the breakpoint for.
3. Set any optional parameters that you want for the breakpoint.
4. Click **OK** to set the breakpoint and dismiss the Load Occurrence Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Load Occurrence Breakpoint dialog.

Setting a conditional breakpoint

When you set a breakpoint, you can specify the parameters or conditions for that breakpoint.

To set a conditional breakpoint:

1. Ensure the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View**.
2. Use the Breakpoints menu to select the type of breakpoint that you want to set.
3. On the Breakpoint dialog complete any or all optional parameters that you want as conditions for your breakpoint.
4. Click OK to set the conditional breakpoint and dismiss the dialog.

/a>

Setting a deferred breakpoint

A deferred breakpoint is a breakpoint set in a DLL, dynamically called routine, executable, or package that is not currently loaded. You can defer the following types of breakpoints:

- line breakpoints
- function breakpoints

To set a deferred breakpoint, click on the **Defer breakpoint** check box when setting one of the above types of breakpoints. When you restart a program, the breakpoints that were set in the previous debug session for classes not currently loaded will be set as deferred breakpoints. These deferred breakpoints will be enabled when the classes corresponding to the breakpoints are loaded.

Setting multiple breakpoints

You can set several breakpoints with the same optional parameters from any of the breakpoint dialogs.

To set multiple occurrences of a type of breakpoint:

1. Select the type of breakpoints you want to set from the **Breakpoints** menu.
2. From the Breakpoint dialog, enter the required information for the first breakpoint. Change any fields in the **Optional Parameters** section of the dialog, as desired.
3. Click on **Set**. The settings are saved for the current breakpoint.
4. For each additional breakpoint, change the information for the new breakpoint (for example, new line number, new function , or new address) and click on **Set**.
5. After you have set the last breakpoint, click on **Cancel** to dismiss the dialog.

Viewing set breakpoints

A list of breakpoints you have set is kept in the Breakpoints pane for the process you are debugging. This list is originally collapsed and can be expanded to show your installed breakpoints. The list of breakpoints is divided into the types of breakpoints you may have set. Expanding each type of breakpoint will provide you with a list of breakpoints for that type.

To view the list of breakpoints:

1. Click on the **Breakpoints** tab for the process or program you are debugging.
2. Expand or collapse the list of breakpoints to display the breakpoints you want to see.

To view the properties of a breakpoint, right-click on the desired breakpoint and select **Properties** from the pop-up menu.

Modifying breakpoint properties

You can change the following properties of a breakpoint:

- Which threads the breakpoint applies to.
- How often the debugger should skip the breakpoint (the frequency).
- Whether to stop on the breakpoint only when a given expression is true. Expressions can only be applied to the following breakpoints:
  - line breakpoints
  - function breakpoints
  - address breakpoints
- Whether to defer the breakpoint. Only the following breakpoints can be deferred:
  - line breakpoints
  - function breakpoints

You can also change the **Required parameters** fields for a breakpoint. Changing these fields results in the existing breakpoint being deleted and a new breakpoint being set.

To change a breakpoint's properties:

1. Click on the **Breakpoints** tab to bring the Breakpoints pane to the foreground.
2. In the Breakpoints pane, expand the list of breakpoints until you see the breakpoint you want to modify.
3. Right-click on the breakpoint you want to modify.
4. Select **Modify Breakpoint** from the pop-up menu. A Breakpoint dialog corresponding to the breakpoint type appears displaying the current settings for the breakpoint.
5. Change the breakpoint's properties in the Breakpoint dialog.

Enabling and disabling breakpoints

You can disable a breakpoint so that it does not stop execution, and then later enable it again. Information about the breakpoint (such as type, location, condition, and frequency) is saved by the Distributed Debugger when the breakpoint is disabled. Since this is not true when the breakpoint is deleted, the advantage of disabling a breakpoint instead of deleting it is that it is easier to enable a

breakpoint than to recreate it. Enabled breakpoints are indicated with a red dot ( ). Disabled breakpoints are indicated with a gray dot ( ).

You can enable or disable breakpoints from the Breakpoints pane. Also, you can enable or disable breakpoints from the Source pane.

To enable or disable a single breakpoint from the Breakpoints pane:

1. Click on the **Breakpoints** tab to bring the Breakpoints pane to the foreground.
2. In the Breakpoints pane, expand the list of breakpoints until you see the breakpoint you want to enable or disable.
3. Right-click on the breakpoint you want to enable or disable.
4. Select **Enable Breakpoint** or **Disable Breakpoint** from the pop-up menu.

To enable or disable a breakpoint from the Source pane:

1. Scroll to the line which contains the breakpoint you want to enable or disable.
2. Right-click on the line which contains the breakpoint you want to enable or disable.
3. Select **Enable Breakpoint** or **Disable Breakpoint** from the pop-up menu.

To enable all breakpoints, select **Breakpoints > Enable All Breakpoints** from the menu bar.

To disable all breakpoints, select **Breakpoints > Disable All Breakpoints** from the menu bar.

Deleting a breakpoint

You can delete single breakpoints from the Source pane and the Breakpoints pane. All breakpoints can be deleted at once from the Breakpoints menu. If you delete a breakpoint, all information on it is lost. If you do not want to lose your breakpoint information, but do not want the breakpoint to stop execution, disable the breakpoint instead. For information on disabling breakpoints, see the related topic below.

To delete a single breakpoint in the Source pane:

1. Scroll to the line which contains the breakpoint you want to delete.
2. Do one of the following to delete the breakpoint:
   - Double-click on the line number in the prefix area of the line to delete the breakpoint.
   - Right-click on the breakpoint and select **Delete Breakpoint** from the pop-up menu.

To delete a single breakpoint in the Breakpoints pane:

1. Click on the **Breakpoints** tab to bring the Breakpoints pane to the foreground.
2. In the Breakpoints pane, expand the list of breakpoints by clicking on the plus icons (⊞) until you see the breakpoint you want to delete.
3. Right-click on the breakpoint you want to delete.
4. Select **Delete Breakpoint** from the pop-up menu.

To delete all breakpoints, select **Breakpoints > Delete All Breakpoints** from the menu bar.

If you want to temporarily prevent all breakpoints from stopping execution, disable them instead by selecting **Breakpoints > Disable All Breakpoints**.

Chapter 5. Controlling program execution

Running a program

You can have a program run until one of the following occurs:

- end of program is reached
- an active breakpoint is hit
- a specific line number is reached
- an exception occurs.
- AIX SOLARIS a fork(), exec(), or system() call is reached.

If you select **Debug > Run**, the program will run until the end of the program is reached, an active breakpoint is hit, or an exception occurs if exception filtering is set to a level other than NONE.

To run a program until an active breakpoint is hit, do one of the following:

- Click the run button ( ).
- Select **Debug > Run** from the menu bar.
- Press F5.

If you select **Run to Location**, the program will run to the statement selected unless an active breakpoint is hit, an exception occurs or the end of the program is reached.

To run a program to a specific line number:

1. Make sure the line to run to is visible in the Source pane by using the scroll bar or cursor keys to locate the line.
2. Run the program to the line by doing one of the following:
   - Right-click on the line to bring up the pop-up menu, then select **Run To Location**.
   - Click on the line to select it, then select **Debug > Run To Location** from the menu bar.

Exception Handling

The Distributed Debugger allows you to investigate exceptions that occur while you are debugging your program.

You can choose the types of exception or the level of exception you want the debugger to recognize in the Exception Filter Preferences Setting field in Applications Preferences dialog box. The types of exception or level of exception you can select varies with the platform where you are running the program that you are debugging. For example, the exceptions the debugger can handle for a C++ program running on Windows NT are different from the exceptions the debugger can handle for a C++ program running on AIX.

When the debugger encounters an exception that matches one of the exceptions that are specified in the Exception Filter Preferences Settings dialog box, a dialog box opens to warn you an exception occurred. Also, the line where the exception occurred is highlighted in the Source pane.

© Copyright IBM Corp. 1999, 2002

After a program exception is encountered and the Application Exception Occurred dialog box is closed, the following actions are available:

Step exception
Step exception causes the debugger to step into the first registered exception handler (tracked by the operating system). Execution stops at the first executable line of code in the exception handler. If your application does not have a registered exception handler, the exception remains “unhandled” and the application may be terminated.

Run exception
Run exception causes the debugger to run the exception handler that is registered to handle the type of exception encountered. If your application does not have a registered exception handler, the exception remains “unhandled,” and the application may be terminated.

Examine/Retry exception

Examine/Retry exception discards the exception and allows you to investigate the cause of the exception and, if desired, retry program execution at the statement that triggered the exception. The debugger begins at this statement and attempts to continue.

Selecting debugger recognized exceptions

You can choose the type of exceptions the Distributed Debugger recognizes for processes you are debugging, so that stepping or execution will stop when one of the selected exceptions is thrown. By default, all unhandled exceptions are recognized by the Distributed Debugger.

To select the exceptions recognized by the Distributed Debugger:

1. Select **File > Preferences** from the menu bar.
2. Expand the **Debug** item in the left-hand window of the Application Preferences dialog.
3. Locate the process you want to set the exceptions recognized for.
4. Click on **Exception Filter Preferences Settings.**
5. Check the type of exceptions you want the Distributed Debugger to recognize or uncheck exceptions you want the Distributed Debugger to ignore.
6. Click **OK** to close the Application Preferences dialog.

To undo changes that you have made to the exception filter, click **Reset**.

To restore the exception filter preferences factory default settings, click **Default**.

To set your new exception filter preferences as the default for that programming language, check the **Debugger Defaults** box before clicking **OK**.

Stepping through a program

You can use **step commands** to step through your program a single statement at a time. The statements can be source code or disassembly instructions. For an explanation of the step commands, see the related topic below.

To execute a Step Over command, do one of the following:

- Click the step over button ( ) on the toolbar.
- Select **Debug > Step Over** from the menu bar.
- Press F8.

To execute a Step Into command, do one of the following:

- Click the step into button ( ) on the toolbar.
- Select **Debug > Step Into** from the menu bar.
- Press F11.

To execute a Step Debug command, do one of the following:

- Click the step debug button ( ) on the toolbar.
- Select **Debug > Step Debug** from the menu bar.
- Press F7.

To execute a Step Return command, do one of the following:

- Click the set return button ( ) on the toolbar.
- Select **Debug > Step Return** from the menu bar.
- Press Shift+F11.

Step commands

You can use **step commands** to step through your program a single line or, on AIX or Windows, one disassembly instruction at a time.

The following types of step commands are available:

Step Command	Button	Shortcut	Description
Step Over		F8	Executes the current line, without stopping in any functions or routines called within the line.
Step Into		F11	Executes the current line. If the current line contains a call to a function or routine, execution stops in the first line or disassembly instruction of the called function or routine. If the called function or routine was not compiled with debug information, the function or routine is shown in a disassembly view.

Step Command	Button	Shortcut	Description
Step Debug		F7	Executes the current line. Execution stops at the next line encountered for which debug information is available. This could be in the current function or routine, in the called function or routine, or in a function or routine called within the called function or routine.
Step Return		Shift+F11	Executes from the current execution point up to the line immediately following the line that called this function or routine. If you issue a Step Return command from the main entry point (in C++, the main() program), the program runs to completion.

Execution of your program may stop earlier than indicated in the step command descriptions, if the Distributed Debugger encounters a breakpoint or an exception occurs.

Tip: You can use combinations of step commands to step through multiple calls on a single line.

Skipping over sections of a program

Restriction: When debugging interpreted Java programs, you cannot skip sections of a program.

You can skip over sections of code to avoid executing certain statements or to move to a position that certain statements can be executed again.

To skip over a section of code:

1. Scroll to the line where there are statements that you want to avoid executing or you want to execute again.
2. Jump to the line by doing one of the following:
   - Right-click on the line and select **Jump to Location** from the pop-up menu.
   - Click on the line to select it, then select **Debug > Jump to Location** from the menu bar.

Using **Jump to Location** can cause unpredictable results if you jump outside the current function, jump over code that has side-effects (for example, calls to functions whose results are assigned to variables, or functions that change the contents of variables passed by reference), or jump into the middle of a block such as a **for** loop.

Halting execution of a program

Halting a program stops the execution of the program without terminating the execution of the program. It allows you to pause and examine the program's internal state and then continue execution without restarting the program.

To halt execution of a program that is currently running in the debugger, do one of the following:

- Click on .
- Select **Debug > Halt** from the menu bar.
- Press **Ctrl-F5**.

You may find that execution halts in a function other than the one you are debugging (for example, a system library routine). To run to the end of that routine and stop in your own code, do one of the following:

- Issue a Step Return command.
- If the previous technique results in the debugger displaying the message "Cannot determine return address", issue the Step Debug command until execution returns to your code
- If you know what line in your program will be the next to execute after the current function returns, go to the source pane containing that line, set a breakpoint on it, and issue the Run command.

Restarting a program

AIX WIN **Restriction:** Restarting a program is supported on AIX and Windows only. It is not supported if you have attached to a process or JVM.

To start debugging your program from the beginning when your program is stopped, do one of the following:

- Click in the toolbar.
- Select **Debug > Restart** from the menu bar.
- Press **Ctrl+Shift+F5**.

To start debugging your program again from the beginning when your program is running:

1. Issue a Halt command by doing one of the following:
   - Click in the toolbar.
   - Press **Ctrl+F5**.
   - Select **Debug > Halt** from the menu bar, if the program is currently executing within the debugger.
2. Set a breakpoint at the location you want to run to, if it is not the beginning of your program and you have not already set a breakpoint there.
3. Click in the toolbar or select **Debug > Restart** from the menu bar.

If the previous run of your program produced side effects such as the creation of an output file and the program logic will be changed by the existence of such files from a previous debug session, you may want to erase these files before restarting.

Chapter 6. Inspecting data

Inspecting variables

Adding a variable or expression to the Monitors pane

If you want to keep track of the contents of variables and expressions during program execution, add them to the Monitors pane. You can add variables and expressions to the Monitors pane from the Source Pane or the Monitors menu.

Local variables that are in scope can also be monitored in the Locals pane. By default, all local variables in scope are added to the Locals pane.

To add a variable or expression to the Monitors pane from the Source pane:
1. Ensure that the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View** from the menu bar.
2. Do one of the following:
   - Ensure that **Add to program monitor** in the **Preferences > Debug** menu is selected and then double-click the variable or expression you want to monitor.
   - Double-click or highlight the variable or expression you want to monitor and then right-click on the highlighted variable, and select **Add to Program Monitor** from the pop-up menu.

To add a variable or expression to the Monitors pane from the Monitors menu:
1. Select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the variable or expression you want to monitor.
3. Select the **Program monitor** radio button.
4. Click **OK** to add the variable or expression to the monitor and dismiss the dialog.

To add multiple variables or expressions to the Monitors pane from the Monitors menu:
1. Select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the variable or expression you want to monitor.
3. Select the **Program monitor** radio button.
4. Click **Monitor** to add the variable to the monitor.
5. Repeat steps 2 to 4 until you have added all the variables or expressions you want to monitor.
6. Click **Cancel** to close the dialog.

Viewing the contents of a variable or expression

You can view the contents of a variable or expression in the Locals pane or the Monitors pane, if you have added the variable there. By default, all local variables in scope for the selected stack entry are added to the Locals pane. Call stack entries can be viewed or selected in the Stacks pane, and they include threads and, depending on the language you are debugging, functions, methods, procedures, or entry points.

© Copyright IBM Corp. 1999, 2002

To view the contents of a variable or expression in the Locals pane:

1. If the variable that you want to view is associated with a function, method, procedure, or entry point call of a particular thread, go to the Stacks pane and choose the appropriate call stack entry. This will update the Locals pane with the value of the local variables of that particular call.

   Selecting a thread in the Stacks pane is equivalent to selecting the top or first entry for that thread (for example, the first function, method, procedure, or entry point call). If the debug engine does not support locals monitoring for secondary entries (entries other than the top entry), then the Locals pane thread nodes for those secondary entries displays ″Locals not available″ when they are selected.
2. Expand the thread in the Locals pane where the local variable you want to view appears.
3. If necessary, use the scroll bars to scroll the pane until the variable is visible.
4. If your variable is a class, struct or array, it can be expanded to show its individual elements.
5. If desired, change the representation of the variable: right-click on the variable and select **Representation** from the pop-up menu. Then select the desired representation from the Monitor Representation dialog box.

To view the contents of a variable or expression you have already added to the Monitors pane:

1. Use the scroll bars to scroll the pane until the variable is visible.
2. If your variable is a class, struct or array, it can be expanded to show its individual elements.
3. If desired, change the representation of the variable: right-click on the variable and select **Representation** from the pop-up menu. Then select the desired representation from the Monitor Representation dialog box.

If a variable or expression is not in scope, a message displays in the Monitors pane instead of a value.

You can also view the contents of variables in the Source pane with Tool-tip evaluation. To enable Tool-tip evaluation, see the related topic below.

Changing the contents of a variable

To change the contents of a variable in the Locals or Monitors pane:

1. Expand the monitor containing the variable whose value you want to modify.
2. If your variable is a class, struct or array, expand it to show its individual elements.
3. Scroll down to the variable you want to change and do one of the following:
   - Double-click on the variable or variable element.
   - Right-click on the variable and select **Edit** from the pop-up menu.
4. Enter a new value for the variable or variable element.

Inspecting registers

Viewing the contents of a register

You can view the contents of a register from the Registers pane, the Monitors pane if you have added the register there, or a Storage Monitor pane if you have added the register there.

To view the contents of a register in the Registers pane:
1. Expand the thread for which you want to view the registers.
2. Expand the register category that contains the register you want to view.
3. If desired, use the scroll to scroll the pane until the register is visible.

To view the contents of a register you have already added to the Monitors pane:
1. If necessary, use the scroll bars to scroll the Monitors pane until the register is visible.
2. If desired, change the representation of the register: right-click on the register and select **Representation** from the pop-up menu. Then select the desired representation from the Monitor Representation dialog box.

To view the contents of a register you have already added to a Storage pane:
1. If necessary, use the scroll bars or PageUp and PageDown keys to scroll the Storage pane until the register is visible.
2. If desired, change the representation of the register: right-click on the register and select **Representation** from the pop-up menu. Then select the desired representation from the Monitor Representation dialog box.

Changing the contents of a register

To change the contents of a register in the Registers pane, Monitors pane or Storage Monitor pane:
1. In the Registers pane, or Monitors pane expand the entry which contains the register whose value you to want to change.
2. Scroll to the register you want to change and do one of the following:
   - Double-click on the register.
   - Select **Edit** from the pop-up menu.
3. Enter a value that is valid for the current representation of that register.
4. Press **Enter** to submit the change. The Distributed Debugger checks for a valid value.

Adding a register to the Monitors pane

You can add a register to the Monitors pane if you want to monitor only a few registers during the execution of your program. Registers can also be monitored in the Registers pane and Storage Monitor panes. To monitor all registers during program execution, use the Registers pane.

To add a register to the Monitors pane:
1. Do one of the following:
   - Select **Source > Monitor Expression** from the menu bar.
   - Press Shift+F9.
   - Click on the Monitors tab and select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the name of the register you want to monitor. Check the Registers pane to see a valid Registers name.
3. Select **Program Monitor**.
4. Click **OK** to add the register to the Expressions monitor and dismiss the dialog.

To add multiple registers to the Monitors pane:
1. Click on the **Monitors** tab and do one of the following:

- Select **Source > Monitor Expression** from the menu bar.
- Click on the Monitors tab and select **Monitors > Monitor Expression** from the menu bar.
- Press Shift+F9.

2. In the dialog, enter the register you want to monitor.
3. Click **Monitor** to add the register to the monitor.
4. Repeat steps 2 and 3 until you have added all the registers you want to monitor.
5. Click **Cancel** to dismiss the dialog.

Inspecting storage

Viewing a location in storage

You can view the contents of storage from the Storage pane or from a new Storage Monitor pane that you have created.

To view the contents of storage from the Storage pane:

1. If desired, change the representation of the storage contents in the Storage pane.
2. If necessary, use the scroll bar in the Storage pane to view storage locations above or below the starting address of the Storage pane.
3. You can jump directly to an address in the Storage pane by doing the following:
   - Double-click on any address field in the Storage pane.
   - Enter the address you want to view. This address can be an expression, for example &x.
   - Press Enter. The storage contents now shown in the Storage pane are centered around the address you just entered.

To view the contents of storage from a Storage Monitor pane that you have created:

1. If desired, change the representation of the storage contents in the Storage Monitor pane.
2. If necessary, use the scroll bar in the Storage Monitor pane to view storage locations above or below the starting address of the Storage Monitor pane.
3. Use the **Go to Address** button to return to the starting address of the Storage Monitor pane.

C C++ To view the *contents* of a C or C++ variable (such as an integer) in a Storage monitor, precede the variable with an ampersand (&), or select a pointer that points to that variable. For example, given the following C or C++ source code:

```
int i=10;
int* p=&i;
```

You can monitor the storage for the variable `i`by entering either `&i` or `p`in the Monitor expression dialog, then selecting the **Storage monitor** radio button in that dialog.

Changing the representation of storage contents

For each Storage or Storage Monitor pane you have, you can change the representation of the storage and the number of columns shown in each pane.

These settings affect only the Storage or Storage Monitor pane you are viewing, so you can have multiple Storage Monitor panes with different settings.

- Select the representation of storage for the Storage pane or Storage Monitor pane you are viewing from the **Content style** pulldown list. The **Content Style** pulldown list is at the bottom of the pane.
- Select the number of columns shown in a Storage pane or Storage Monitor pane from the **Columns Per Line** pulldown list. The **Columns Per Line** pulldown list is at the bottom of the pane.

Changing the contents of a storage location

To change the contents of a storage location in a Storage pane or Storage Monitor pane:

1. Select the Storage pane or Storage Monitor pane where you want to make the change.
2. Scroll down to the storage location you want to change.
3. Right click and select **Edit** or double-click on the value you want to change.
4. Enter a valid value for that storage location.
5. Press **Enter** to submit the change. The Distributed Debugger checks for a valid value.

Adding a new Storage Monitor pane for an expression or register

Registers can also be monitored in the Registers pane and the Monitors pane. To monitor all registers during program execution, use the Registers pane.

You may wish to add a new Storage Monitor pane for an expression or a register if you want to monitor specific locations in storage or only a few registers during the execution. To monitor all locations in storage during program execution, use the Storage pane.

You can also monitor all storage locations in a storage monitor, but it follows a specific variable, expression, or register as it changes. **Warning:** If there is a variable in scope which has the same name as the register that you are trying to use, the variable will be used first.

To add a new Storage Monitor pane for an expression or register from the Registers pane:

1. Highlight the expression or register for which you want to add a new Storage Monitor pane.
2. Right-click the highlighted expression or register and select **Add to Storage Monitor** from the pop-up menu. A new Storage Monitor pane will appear with the expression or register appearing as the monitor's tab title. The current value of the expression or register will be highlighted.

To add a new Storage Monitor pane for an expression or register from the Monitors pane:

1. Do one of the following:

- Select **Source > Monitor Expression** from the menu bar.
- Press Shift+F9.
- Click on the Monitors tab and select **Monitors > Monitor Expression** from the menu bar.

2. In the dialog, enter the expression or register that you want to monitor.
3. Select the **Storage Monitor** radio button.
4. Click **OK** to add the new Storage Monitor pane.
5. A new Storage Monitor pane will appear with the expression or register appearing in the monitor's tab.

To add multiple new Storage Monitor panes from the Monitors pane:

1. Do one of the following:
   - Select **Source > Monitor Expression** from the menu bar.
   - Press Shift+F9.
   - Click on the Monitors tab and select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the first expression or the name of the register that you want to monitor.
3. Select the **Storage Monitor** radio button.
4. Click **Monitor** to add the new Storage Monitor pane for the expression or register entered.
5. Repeat steps 2 to 4 until you have added all the storage locations or registers that you want to monitor.
6. Click **Cancel** to close the dialog.

Tip: Check the Registers pane to see a valid register name.

Heap errors

AIX WIN **Restriction:** This information applies to debugging C or C++ programs on AIX, Windows, and OS/2 only.

Heap errors can occur when your code inadvertently overwrites control information that the memory management functions use to control heap usage. Each block of allocated storage within a heap consists of a data area, which starts at the address returned by the allocating function, as well as a control area adjacent to the data area, which is needed by the memory management functions to free the storage properly when you deallocate the storage. If you overwrite a control structure in the heap (for example, by writing to elements outside the allocated bounds of an array, or by copying a string into too small a block of allocated storage), the control information is corrupted and may cause incorrect program behavior even if the data areas of other allocated blocks are not overwritten.

You should consider the following points when you are trying to locate heap errors:

Finding heap errors outside the debugger

To detect heap errors, you can compile your program to use the heap-checking versions of memory management functions. When you run a program compiled with this option, each call to a memory management function causes a heap check to be performed on the default heap. This heap check involves checking the control

structures for each allocated block of storage within the heap, and ensuring that none was overwritten. If an error is encountered, the program terminates and information is written to standard error including the address where heap corruption occurred, the source file and line number at which a valid heap state was last detected, and the source file and line number at which the memory error was detected.

Heap checking for default and other heaps

Heap checking is only enabled for the default heap used by each executable. If the debug versions of the memory management functions do not report heap corruption and you still suspect a problem, you may be using additional heaps and corrupting them. You can debug usage of nondefault heaps by adding calls to the **_uheapchk** C Library function to your source code. See your compiler documentation for more information.

Pinpointing heap errors within the debugger

You can pinpoint the cause of a heap error from within the debugger, provided the heap causing the error is known to be the default heap, by continually narrowing down the gap between the last line at which the heap was valid, and the first line at which corruption occurred. From within the Source pane, use a combination of run commands, step commands, line and function breakpoints, and the **Perform Heap Check on Stop** setting on the **Debug** menu, to narrow the scope of your search.

Perform heap check on stop may expose other coding errors

For semantically incorrect programs, **Perform Heap Check on Stop** is intrusive in that it may cause different results where a program is incorrectly accessing data on the stack. This is because **Perform Heap Check on Stop** causes the process and thread being debugged to call a heap check function each time execution stops, and this heap check function affects the safe area of the stack by overwriting part of that area with its stack frame. For example, if a called function returns the address of a local variable, that local variable's contents are accessible from the calling function, and does not change, as long as the stack frame used by the called function is not overwritten by a subsequent call. However, if you issue a Step return command from the called function while **Perform Heap Check on Stop** is enabled, the heap checking function is called immediately on return from the called function, and the storage pointed to by the returned pointer may have been overwritten by the stack frame of the heap checking function.

Perform heap check on stop affects performance

Heap checking within the debugger has a high overhead cost for step commands, because the heap is checked after each step. If you are stepping through large sections of code, or frequently stopping at breakpoints, and you find debug performance too slow, try turning on **Perform Heap Check on Stop** only in those areas you suspect are causing heap errors.

Notes on Perform Heap Check on Stop:

- For the Perform Heap Check on Stop choice to work, you have to compile your application to use the heap-checking versions of memory management functions. See your compiler reference for more information.

- If you enable the Check Heap on Stop choice and run your application to termination, and the application contains a heap error, the heap check is not made. To check the heap just before termination, set a breakpoint on the last line of your application.
- If you are debugging a multiple thread program and a thread stops while running in compiler memory management code that is holding a memory semaphore, the heap check is not performed.
- If the stopping thread is running in 16-bit code, the heap check is not performed.

Enabling and disabling a monitored variable, expression or register

You can disable the monitoring of a variable, expression or register. The advantage of disabling a monitored expression instead of deleting it is that it is easier to enable a monitored expression than to recreate it.

You can enable or disable monitored variables, expressions or registers from either the Monitor pane or Locals pane.

To enable or disable a monitored expression:

1. Locate the variable, expression or register you want to disable or enable in the Monitors pane or Locals pane.
2. Right-click on the variable, expression or register you want to enable or disable.
3. Select **Enable** or **Disable** from the pop-up menu.

Enabling tool tip evaluation for variables

Tool tip evaluation of variables ("hover help for variables") provides you with a quick way to view the value of a variable in the Source pane. When you point at a variable in the Source pane, the variable is evaluated and its value displays in a manner similar to a tool tip. This feature is enabled by default when you first start the debugger.

To enable tool tip evaluation of a variable, select **Source > Allow Tool Tip Evaluation** from the menu bar.

A check mark appears next to the Allow Tool Tip Evaluation menu item when tool tip evaluation for variables is enabled.

To enable tool tip evaluation for variables as the default:

1. Select **File > Preferences** from the main menu.
2. Select **Debug** from the list of preferences to set.
3. Select **Allow Tool Tip Evaluation** from the **Debugger Defaults** section.
4. Click **OK** to enable the tool tip monitor and dismiss the dialog.

Note: Tool tip evaluation uses spaces and punctuation to parse source and, therefore, will only evaluate identifiers and not expressions. If you wish to evaluate more complicated expressions with tool tip, highlight the entire expression in the Source view and then position the mouse pointer to hover over the highlighted expression and see the evaluation.

Changing the representation of monitor contents

You can change the representation of variables and expressions in the Monitors or Locals panes. You can change the representation for existing entries or the default representation for future entries in the Applications Preferences dialog.

To change the representation of a variable or expression:

1. Right-click on the variable or expression for which you want to change the representation.
2. Select **Representation** from the pop-up menu. The Monitor Representation dialog appears.
3. Select the representation you want from the list of available representations.
4. Click **OK** to change the representation and dismiss the Monitor Representation dialog.

To change the default representation of variables or expressions:

1. Select **File > Preferences** from the main menu bar. The Application Preferences dialog appears.
2. In the left-hand pane of the Application Preferences dialog, go to **Debug >** ***program*** **>** ***language*****: Default Monitor Representation**, where *program* is the name of a program loaded in the Distributed Debugger you want to change the default representation for and *language* is the language the program you are debugging was written in.
3. Change the representations for variable types by clicking on the representation associated with a variable type and selecting a representation from the list.
4. If you want these representations to become the default for the Distributed Debugger to use when no program profile is available, select the **Debugger Defaults** checkbox. If you want to restore the Debugger factory default settings, click on the **Default** push button.
5. Click **OK** to change the default representations and dismiss the Application Preferences dialog.

The default representations of variables and expressions in programs you have previously debugged will not be affected by these changes.

Appendix A. Distributed debugger commands

idebug command

AIX WIN The `idebug`command starts both the Distributed Debugger interface and the debug engine when debugging a program locally. When debugging remotely, it is used to connect to a debug engine daemon on a remote system or to start the Debugger user interface as a daemon on your local system.

AIX WIN The `idebug`command has the following syntax for AIX or Windows:

```
idebug [idebug_options] [local_debug_parameters | remote_debug_parameters |
ui_daemon_parameters] [--] [program_name [program_parameters]]
```

This page contains discussions of the following categories of `idebug`parameters:

- “Basic idebug parameters”
- “Local debug parameters” on page 40
- “Remote debug parameters” on page 41
- “Parameters to use the Debugger user interface as a daemon” on page 41
- “Program name parameters” on page 42

Warning: Temporary Level 3 Header

Warning: Temporary Level 4 Header

Basic idebug parameters: The `idebug_options` are zero or more of the following:

Option	Purpose
-a `process_id`	Attach to the already running process `process_id`.
-a	Invoke the Attach dialog.
-s	This option can only be used in conjunction with `-a`. Prevents the Debugger from stopping in the first debuggable statement in the program. Program execution only stops when the first set breakpoint is encountered. This option requires that the program you want to debug has program profile information available. If no program profile information is available, or you specify the `-p-` option, the program will run to completion.
-h or -?	Display help for the `idebug` command.

© Copyright IBM Corp. 1999, 2002

Option	Purpose
-i	Start the Debugger in the system initialization code that precedes the call to the main entry point for the program. C++ This can be useful if you need to debug the constructors for static class objects.
-p+	Use program profile information. The Debugger will restore the monitors, fonts, and breakpoints that were associated with your program during its last debug session. If you are debugging the program for the first time, the Debugger windows start up with their default appearance, and no breakpoints are set. Any changes you make to the monitors and breakpoints are saved. **Note:** If you add or delete lines in your source file, recompile it, and then debug the program again with a saved program profile, line breakpoints may no longer match the code they were initially set for because line breakpoint information is saved by line number, not by the content of the line. If the Debugger has saved a profile of the breakpoint and monitor settings from a previous debug session for this program, the profile is used to restore those settings. This is the default setting for the Debugger.
-p-	Do not use program profile information. The Debugger ignores any program profile information, and the Debugger windows start up with their default appearance, and no breakpoints are initially set.
-qquiet	Suppresses the splash screen when the Debugger starts. Also suppresses the Debugger daemon waiting dialog when used with the `-qdaemon` and `-quiport` options.
-qnewsession	Starts a new debug session rather than reusing an existing session.
-qterminate	Closes any running Debugger user interfaces or user interface daemons.

Local debug parameters: Use the `local_debug_parameters`when you want to start debugging a program on your local system. If a parameter is not specified in the command, the default is assumed.

The `local_debug_parameters` are:

Parameter	Description
AIX WIN -qlang	-qlang=*dominant_language* Specifies the dominant programming language to use for debugging. The valid values are shown in the table below: **Value** Use to when debugging: **c** C programs **cpp** C++ programs

Remote debug parameters: Use the `remote_debug_parameters`when you want to connect to a debug engine daemon on a remote system. If a parameter is not specified in the command, the default is assumed.

The `remote_debug_parameters` are:

Parameter	Description
AIX WIN SOLARIS HP-UX -qhost	-qhost=*remote_host* Required parameter that specifies the TCP/IP name or address of the machine where the debug engine is running.
AIX WIN -qport	-qport=*host_port* Specifies the port number on the machine where the debug engine is running. The default port is 8000. This port number must match the port number used in the `-qport` parameter of the `irmtdbgc` command.
-qlang	-qlang=*dominant_language* Specifies the dominant programming language to use for debugging. The valid values are shown in the table below: **Value** Use when debugging: **c** C programs **cpp** C++ programs

Parameters to use the Debugger user interface as a daemon: The `ui_daemon_parameters`are used when starting the Distributed Debugger user interface as a daemon. When running as a daemon, the Distributed Debugger user interface listens on a specific port number for a debug engine. Once a connection is made, the Distributed Debugger user interface appears and you can begin debugging your program. The `ui_daemon_parameters` are:

Parameter	Description
-qdaemon	Tells the Distributed Debugger user interface to run as a daemon. You must use the `-quiport` option when specifying `-qdaemon`. If this option is not specified, the Distributed Debugger does not listen for connections from Debugger engines.

Parameter	Description
-qlang	-qlang=*dominant_language* Specifies the dominant programming language to use for debugging. The valid values are shown in the table below: **Value** — **Use when debugging:** **c** — C programs **cpp** — C++ programs
-quiport	-quiport=*port* Specifies the port numbers where the Distributed Debugger user interface daemon should listen for a debug engine. You can specify a single port or multiple ports. When specifying multiple ports, `<port>`must be a comma-delimited list of port numbers. This option is required when using the `-qdaemon`option. There is no default port number. C++ One of the port numbers specified here must be used as the port number for the `-quiport`parameter of the `irmtdbgc` command.
-qterminate	Closes any running Debugger user interfaces or user interface daemons.

Program name parameters: Use the "**--**" parameter to separate Debugger options and parameters from the program name and parameters. Use this option if your program name or parameters include forward slashes ("/") or dashes ("-"). If you do not use this option, anything preceded by a slash or a dash will be interpreted as a Debugger option.

If you do not specify `program_name`when issuing the `idebug`command, the Debugger will prompt you for the required information in the Load Program dialog.

irmtdbgc command

AIX SOLARIS WIN **Restriction:** This is supported on AIX, Solaris, and Windows only.

Requirement: You must have the debug engine installed on the remote system in order to use this command. Check the install documentation for instructions on how to install the debug engine on a remote system. This command can also be used on the local system.

The `irmtdbgc`command starts the debug engine on the remote system. If the debug engine detects a Debugger user interface daemon, then you can start debugging your program immediately. If no Debugger user interface daemon is detected, the

debug engine will run as a daemon until you start the Debugger user interface on the local system with the idebug command.

Note: The Debugger can operate in the remote manner when the UI and engine are on the same machine.

The irmtdbgc command has the following syntax:

```
irmtdbgc [irmtdbgc_parameters] [--] [program_name [program_parameters]]
```

where irmtdbgc_parameters are:

Parameter	Description
-qport=<port>	Specifies the TCP/IP port used for the connection. If you do not use the default port, specify the same port number you use here in the -qportparameter of the idebug command. The default port is 8000. **Note:** Do not use this parameter when connecting to a Debugger user interface daemon. You must use the -quiport option.
-quiport=<ui_daemon_port>	Specifies the TCP/IP port used for connecting to a Debugger user interface daemon listening on another machine. This port number must match the port number used in the -quiport parameter of the idebug command.
-qhost=<ui_daemon_host>	Specifies the TCP/IP name or address of the machine where the Debugger user interface daemon is listening.
-qdaemon	Tells the debug engine to run as a daemon. The debug engine re-initializes itself and waits for a new connection when the program you are debugging runs to completion or is terminated manually. The debug engine must be terminated manually on the remote system. Without this option, the debug engine terminates when the program you are debugging runs to completion or is terminated manually.

Use the "--" parameter to separate irmtdbgcparameters from the program name and parameters. Use this option if your program name or parameters include forward slashes ("/") or dashes ("-"). If you do not use this option, anything preceded by a slash or a dash will be interpreted as a Debugger option.

If you do not specify program_name when issuing the irmtdbgccommand, the Debugger will prompt you for the required information in the Load Program dialog of the Debugger user interface.

Appendix B. Optional breakpoint parameters

Optional breakpoint parameters are used to control the behavior of breakpoints. You can set the following parameters when you set a breakpoint:

Optional breakpoint parameter	Description	Type of breakpoint supported
Threads	This parameter is supported by all breakpoint types.	
Frequency	This parameter is supported by all breakpoint types.	
Expression	You can enter an expression into this field. The execution of the program stops at the breakpoint only if the condition specified in this field tests true. For example, if you are debugging a C++ program you could type the following: `(i==1) || (j==k) && (k!=5)`	Line, function or method
Defer	Select this check box if you want to set a breakpoint in a program module that is not currently loaded. If you enter an incorrect source, file, function, or program unit, the debugger will not be able to activate the breakpoint when the program is loaded, and the breakpoint will remain in the deferred state. **Restriction:** You cannot set a deferred breakpoint in a preloaded DLL or dynamically called routine, but you can set one in a program that has some preloaded DLLs or dynamically called routines and some dynamically loaded ones.	Line, function or method

© Copyright IBM Corp. 1999, 2002

Appendix C. Step commands

You can use **step commands** to step through your program a single line or, on AIX or Windows, one disassembly instruction at a time.

The following types of step commands are available:

Step Command	Button	Shortcut	Description
Step Over		F8	Executes the current line, without stopping in any functions or routines called within the line.
Step Into		F11	Executes the current line. If the current line contains a call to a function or routine, execution stops in the first line or disassembly instruction of the called function or routine. If the called function or routine was not compiled with debug information, the function or routine is shown in a disassembly view.
Step Debug		F7	Executes the current line. Execution stops at the next line encountered for which debug information is available. This could be in the current function or routine, in the called function or routine, or in a function or routine called within the called function or routine.

© Copyright IBM Corp. 1999, 2002

Step Command	Button	Shortcut	Description
Step Return		Shift+F11	Executes from the current execution point up to the line immediately following the line that called this function or routine. If you issue a Step Return command from the main entry point (in C++, the main() program), the program runs to completion.

Execution of your program may stop earlier than indicated in the step command descriptions, if the Distributed Debugger encounters a breakpoint or an exception occurs.

Tip: You can use combinations of step commands to step through multiple calls on a single line.

Appendix D. Environment variables

PATH environment variable

The PATH environment variable is used to locate the debugger executable and the executable programs to be debugged, as well as any other executables being run on the workstation.

INCLUDE environment variable

The INCLUDE environment variable is used by the debugger to locate include files on the workstation.

The environment variable does not apply to languages that do not support include files.

LIBPATH environment variable

AIX OS/2 **Restriction:** This is supported on AIX and OS/2 only.

The LIBPATH environment variable tells the debug engine where to look for debugger DLLs on the workstation.

DER_DBG_CASESENSITIVE environment variable

The DER_DBG_CASESENSITIVE environment variable, if set to a non-null value (for example, "yes", 1, "true", etc.) tells the debugger to compare part names and module names on a case-sensitive basis. By default the debugger converts all names to uppercase for comparison purposes. Note that this does not affect filesystem accesses which are operating system dependent and not affected by DER_DBG_CASESENSITIVE.

DER_DBG_PATH environment variable

The DER_DBG_PATH environment variable is used by the debug engine, the debugger user interface, or both to locate debug source files on your client workstation that are not stored in the same location as the executable being debugged. For example, if your debug executable is stored in F:\BUILDS\SANDDUNE\TEST but your source code is stored in F:\SOURCE and F:\SOURCE\INCLUDE, you should set your DER_DBG_PATH variable as follows:

```
set DER_DBG_PATH=F:\SOURCE:F:\SOURCE\INCLUDE
```

You can set the DER_DBG_PATH environment variable on both client and server systems. The search for source files starts on the server first.

Note: When using the DER_DBG_PATH environment variable with the Distributed Debugger user interface or engine daemon, be sure to set the variable before starting the daemon.

© Copyright IBM Corp. 1999, 2002

DER_DBG_TAB environment variable

The DER_DBG_TAB environment variable affects how the debugger expands tab characters in a source or mixed view within a Source pane. The value for this variable is an integer, indicating the number of spaces to convert a tab character into. Unlike DER_DBG_TABGRID, DER_DBG_TAB does not cause the debugger to place tabbed information in specific columns; it simply results in each tab in the displayed files being converted to the indicated number of spaces.

Note: If DER_DBG_TABGRID has been set to a nonzero value, the setting of DER_DBG_TAB has no effect.

DER_DBG_TABGRID environment variable

The DER_DBG_TABGRID environment variable affects how the debugger uses tab characters to align tabs to columns in a source or mixed view within a Source pane. The value of this variable is an integer indicating the starting position and frequency of the tab. For example, if you set DER_DBG_TABGRID=6, the debugger sets tab stops at 6, 12, 18, 24, and so on. If DER_DBG_TABGRID is set to a nonzero value, the setting of DER_DBG_TAB has no effect.

DER_DBG_DEEP_STEP_DEBUG

The environment variable DER_DBG_DEEP_STEP_DEBUG allows you to bias the step-debug behavior to favor either performance or function. The default step-debug behavior favors better stepping performance by having the debugger step over calls made to shared libraries that do not have any debug information. This improves performance by ignoring libraries that are unlikely to lead to debuggable code.

If the value of this environment variable is set to 1, step-debug will follow such calls. This allows step-debug to reach callbacks from libraries that have no debug information, at the cost of somewhat degraded stepping performance.

For example:

Modules A and C contain debug information, B does not.

line	module A	line	module B	line	module C
100	call B()				
		5	call C()		
				500	do some stuff
				510	return
		6	do some stuff		
		20	return		
101	do some stuff				

A default step-debug executed at line 100 in module A will cause the debugger to stop next at line 101 in module A. If the DER_DBG_DEEP_STEP_DEBUG environment variable is set, then the debugger will stop at line 500 in module C.

Appendix E. Postmortem debugging

Errors during UNIX workstation postmortem debugging

You may experience two kinds of problems while debugging core files:
- Errors that prevent you from debugging the core file
- Unusual debugger behavior.

Postmortem debugging on AIX

Postmortem debugging is intended to help you isolate the causes of unanticipated traps or unhandled exceptions, in programs that are already in production or widespread use. There are three stages to the postmortem debugging process:
1. You compile the program with debug information, and ship the resulting object code.
2. When the end user experiences a trap or unanticipated exception, AIX automatically creates a core dump file with the name **core**, in the current directory (provided that directory is writable).
3. You debug the core file instead of a live object code. Because the core file contains information about the state of the application at the time of the trap or exception, and is not a live object file, only a subset of debugger features are available. For example, you can view memory and register contents, but you cannot step or run the program or set breakpoints.

Debugging Dump Files

AIX OS/2 **Note:** Postmortem debugging is available on AIX and OS/2 only.

To start debugging a dump file, from the command shell type:

```
idebug [path]dumpfilename
```

where `dumpfilename`is the name of the dump file you want to debug. You can also enter the dump file name from the Startup dialog, by starting the debugger without parameters, or by choosing **File->Startup** from the Source or Session Control windows.

Once you have started debugging a dump file, you can use a subset of debugger features to examine registers, storage, and code for the executable that caused the trap. You can access all the debugger monitors (registers, call stack, storage, local variables, program, private and popup), but you cannot change the contents of items in these monitors (for example, registers in the registers monitor).

The following debugger commands are disabled, and icons for them are hidden or greyed out, when you are debugging a dump file:
- Halt command
- Step commands
- Commands to set, clear, disable, enable, or delete breakpoints.

The Run command is available, but its only effect is to cause a redisplay of the exception that caused the trap.

© Copyright IBM Corp. 1999, 2002

Appendix F. Limitations

Remote debug limitations

Remote debugging imposes the following limitations:

- **Browse** only displays the file system on the local system. The file system on the remote system cannot be displayed.

Limitations during postmortem debugging

Once you have started debugging a dump file, you can use a subset of debugger features to examine registers, storage (if available), and code for the executable that caused the trap. You can access only the following debugger monitors:

- Registers monitor
- Storage monitor
- Call Stack monitor
- Local Variables monitor

You cannot change the contents of items in these monitors (for example, registers in the registers monitor). Where the core file does not contain appropriate information, question marks or messages such as "expression evaluation failed" may appear in place of data. For example, if stack information was not saved, local variable values may not be displayed, even though the names of those variables are.

The following debugger commands are disabled, and icons for them are hidden or greyed out, when you are debugging a dump file:

- Run and stop commands
- Step commands
- Commands to set, clear, disable, enable, or delete breakpoints.

Unusual debugger behavior

AIX If the debugger behaves in unexpected ways when you are debugging core files, it is usually an indication that the core file is missing necessary information, or that modules cannot be located.

There are limitations during postmortem debugging regarding what subset of debugging features and actions is normally available for core files. Missing core file information and missing modules can further limit the debugger in the following ways:

- The contents of local variables cannot be displayed or used in expression evaluation. This indicates that the program stack is absent from the core file.
- No entries appear in the Call Stack monitor. This also indicates that the program stack is absent from the core file. If some entries appear in the Call Stack monitor but others are missing, this indicates that the process call stack has an entry for a function contained within a module that is not available. This situation causes the call stack to be truncated in the monitor; that is, only the entries in the stack up to that entry are shown.

© Copyright IBM Corp. 1999, 2002

- Threads are missing. This indicates that the current execution location of a thread is contained in a module that is not available. The debugger handles this situation as if the thread never existed.
- You cannot use a variable of static storage duration in an expression (for example, in the **Monitor Expression** dialog). This indicates that data segments are missing from the core file.
- Memory in the storage window shows “??????” instead of the expected contents. This indicates that the core file was missing those memory locations (for the stack, data segments, and memory-mapped regions), or that the debugger could not locate necessary modules (for code segment regions).

Appendix G. Program profiles

AIX If you are debugging on AIX, user profiles are stored in the workstation, in $HOME/.DbgProf.

Using program profiles means that the Distributed Debugger will restore debugger fonts, monitor settings, and breakpoints for your program from the last time you debugged the program. If you are debugging the program for the first time, the debugger windows start up with their default appearance, and no breakpoints are set.

When you use program profiles, any changes you make to the exception filter settings, monitor settings, and breakpoints are saved. If the debugger has saved a profile containing information on window, breakpoint, and monitor settings from a previous debug session for a program, the profile is used to restore those settings. To delete them you can use the Application Preferences dialog.

Attention: If you add or delete lines in your source file, recompile it, and then debug the program again with a saved program profile, line breakpoints may no longer match the code they were initially set for because line breakpoint information is saved by line number, not by the content of the line.

© Copyright IBM Corp. 1999, 2002

Appendix H. C/C++ expressions supported

C/C++ supported data types

You can monitor an expression that includes a cast to any of the following types:

- 8-bit signed char
- 8-bit unsigned char
- 16-bit signed integer
- 16-bit unsigned integer
- 32-bit signed integer
- 32-bit unsigned integer
- 64-bit signed integer
- 64-bit unsigned integer
- 32-bit floating-point
- 64-bit floating-point
- pointers
- complex type
- user-defined types

These data types include **int**, **short**, **char**, and so on.

C/C++ supported expression operands

You can monitor an expression that uses the following types of operands only:

Operand	Definition
Variable	A variable used in your program.
Constant	The constant can be one of the following types: • Fixed-point or floating-point constant within the ranges supported by the system the program you are debugging is running on. • A string constant, enclosed in double quotation marks (for example, `"mystring"`) • A character constant, enclosed in single quote marks (for example, ’`x`’)
AIX SOLARIS OS/2 WIN Register	Any of the processor registers that can be displayed in the Registers Monitor. In the case of conflicting names, program variable names take precedence over register names. For conversions that are done automatically when the registers display in mixed-mode expressions, general-purpose registers are treated as unsigned arithmetic items with a length appropriate to the register. For example, on Intel platforms EAX is 32-bits, AX is 16-bits, and AL is 8-bits.

© Copyright IBM Corp. 1999, 2002

If you monitor an enumerated variable, a comment appears to the right of the value. If the value of the variable matches one of the enumerated types, the comment contains the name of the first enumerated type that matches the value of the variable. If the length of the enumerated name does not fit in the monitor, the contents appear as an empty entry field.

The comment (empty or not) lets you distinguish between a valid enumerated value and an invalid value. An invalid value does not have a comment to its right.

You *cannot* update an enumerated variable by entering an enumerated type. You must enter a value or expression. If the value is a valid enumerated value, the comment to the right of it is updated.

You cannot look at macros that have been defined using the `#define` preprocessor directive.

C/C++ supported expression operators

You can monitor an expression that uses the following operators only:

Operator	**Coded as**
C++ Global scope resolution	::a
C++ Class or namespace scope resolution	a::b
Subscripting	a[b]
Member selection	a.b or a->b
Size	`sizeof` *a* or `sizeof` (*type*)
Logical not	!a
Ones complement	~a
Unary minus	-a
Unary plus	+a
Dereference	*a
Type cast	(type) a
Multiply	a * b
Divide	a / b
Modulo	a % b
Add	a + b
Subtract	a - b
Left shift	a << b
Right shift	a >> b
Less than	a < b
Greater than	a > b
Less than or equal to	a <= b
Greater than or equal to	a >= b
Equal	a == b
Not equal	a != b
Bitwise AND	a & b
Bitwise OR	a | b

Operator	Coded as
Bitwise exclusive OR	*a* ^ *b*
Logical AND	*a* && *b*
Logical OR	*a* || *b*

C/C++ compiler options on workstation UNIX platforms

Compile your C/C++ programs with the -g option (to generate debugging information) if you want to be able to debug your program at the source code statement level. You should also consider using the following options:

Option	Purpose
-qnoopt	Compiles your program with optimization off. This is the default. (Some optimizations reorder the execution sequence of your program, while others may eliminate expressions whose result is never used. You may find it confusing to debug a program compiled with optimization, because statements may execute in a nonsequential fashion or not at all.)
-Q!	Compiles your program with inlining off. This is the default.

Note: If you use the -bstabcmpct linker option when compiling programs that use DirectToSom or C++ namespaces, you should only specify a value of 1 (the default) or 0 (for example, -bstabcmpct:0) for versions of the program you intend to debug. If you specify -bstabcmpct:2 you will not be able to debug such programs.

Notices

Note to U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this Documentation in other countries. (In this notice, Documentation includes the product, in whole or in part, (including but not limited to source code files and object code files) and its associated materials whether in electronic or print form.) Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this Documentation. The furnishing of this Documentation does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

© Copyright IBM Corp. 1999, 2002

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this Documentation and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (C) Copyright IBM Corp. 1997, 2002. All rights reserved.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both:

- AFS
- DB2
- DB2 Extenders
- DB2 Universal Database
- CICS
- IBM
- IMS
- OS/390
- OS/400
- VisualAge
- WebSphere
- WorkPad

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

ActiveX, Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

UNIX is a registered trademark of The Open Group

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

(c) Copyright IBM Corp. 2000, 2002. All Rights Reserved.