

VisualAge[®] C++ Professional for AIX[®]



Debugging with the Distributed Debugger

Version 5.0

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page v.

Edition Notice

This edition applies to Version 5.0 of IBM® VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	v
Programming Interface Information	vii
Trademarks and Service Marks	vii
Industry Standards	viii
About This Book.	ix
Chapter 1. Distributed Debugger	1
Distributed Debugger: Overview	1
Distributed Debugger: Panes	1
Distributed Debugger: Monitors	2
Chapter 2. Establishing a debugging environment.	5
Setting environment variables for the debugger.	5
Chapter 3. Preparing a Program for Debugging	7
Writing a program for debugging	7
Optimized code debugging	7
Recursion and debugging	8
Compiling a program for debugging	8
Chapter 4. Starting the Debugger	9
Starting the debugger for local debugging	9
Starting the debugger for remote debugging.	9
Remote debugging	10
Starting the Distributed Debugger user interface daemon	10
Attaching to a Running Process.	11
When to attach	11
Attaching to a local running process	12
Attaching to a remote running process	12
Chapter 5. Working with Breakpoints	15
Breakpoints	15
Setting breakpoints	15
Setting a line breakpoint	15
Setting a function breakpoint	17
Setting an address breakpoint	18
Setting a storage change breakpoint	18
Setting a load occurrence breakpoint	19
Setting a deferred breakpoint	20
Setting multiple breakpoints.	20
Viewing set breakpoints	21
Modifying breakpoint properties	21
Enabling and disabling breakpoints	22
Deleting a breakpoint	23
Chapter 6. Controlling Program Execution	25
Running a program.	25
Stepping through a program.	25

Step commands	26
Skipping over sections of a program	27
Halting execution of a program.	28
Restarting a program	28

Chapter 7. Inspecting Variables, Registers, and Storage.	31
Adding a variable to the Expressions monitor	31
Viewing the contents of a variable.	32
Changing the contents of a variable	32
Viewing the contents of a register	33
Changing the contents of a register	33
Adding a register to the Expressions monitor	34
Viewing a location in storage	34
Changing the representation of storage contents	35
Changing the contents of a storage location.	35
Creating a new Storage monitor for an expression or register	35
Enabling and disabling a monitored expression	36
Enabling hover help for variables	37
Changing the representation of monitor contents	37

Appendix A. Debugger Commands	39
idebug command	39
irmtdbg command.	41

Appendix B. C/C++ expressions supported	43
C/C++ supported data types	43
C/C++ supported expression operands	43
C/C++ supported expression operators	44

Appendix C. Environment Variables	47
Other environment variables	47
INCLUDE workstation environment variable	47
LIBPATH environment variable.	47
DER_DBG_CASESENSITIVE workstation environment variable	48
DER_DBG_LOCAL_PATH environment variable	48
DER_DBG_NUMBEROFELEMENTS workstation environment variable	48
DER_DBG_OVERRIDE environment variable	48
DER_DBG_PATH environment variable	49
DER_DBG_TAB environment variable	49
DER_DBG_TABGRID environment variable	49

Appendix D. Remote debug limitations	51
---	-----------

Appendix E. UNIX® Call Handling During Debugging.	53
exec() handling	53
fork() handling	53
system() handling	53

Contacting IBM 55

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 document in other countries. 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 document. The furnishing of this document 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 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.

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.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

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 document 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 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:

® (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. ® Copyright IBM Corp. 1998, 2000. All rights reserved.

Programming Interface Information

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

General-use programming interface allow the customer 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 of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
AS/400
DB2
CICS
C Set ++
IBM
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architecture
TeamConnection
VisualAge
WebSphere
Workplace Shell

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Microsoft, Win32, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

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

UNIX is a registered trademark in the U.S. and other countries licensed exclusively through X/Open Company Limited.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered trademark of Ziff Communicatoins Company and is used by IBM Corporation under license

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

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899-1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159-1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945-1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

References to VisualAge or VisualAge C++ in this book should be interpreted as VisualAge C++, Version 5.0.

Chapter 1. Distributed Debugger

Distributed Debugger: Overview

The IBM 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 a 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. On Windows® or AIX, the types of panes and monitors available on a program page depend on the program you are debugging.

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

RELATED CONCEPTS

"Distributed Debugger: Panes"

"Distributed Debugger: Monitors" on page 2

"Remote debugging" on page 10

Distributed Debugger: Panes

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

Stacks pane

On Windows or AIX 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.

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, delete them, or add new ones.

Source pane

The Source pane provides a view of the source code for the program you are debugging. On Windows or AIX 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 from your workstation, either on a local or a network drive. On Windows and AIX if the source code file is not found, only a disassembled machine code view is available.

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 monitor panes. For more information on monitor panes, see the related topic below.

RELATED CONCEPTS

“Distributed Debugger: Monitors”

“Distributed Debugger: Overview” on page 1

Distributed Debugger: Monitors

Depending on the language you are debugging, the Distributed Debugger provides you with monitors to monitor various aspects of your program. The following monitors 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 to change the representation of values.

Tip: Enabling hover help 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)

On Windows or AIX the Locals pane helps you monitor all local variables in scope at the current execution point of your program. For multithreaded programs, each thread is listed and can be expanded to show the local variables in scope for each thread. 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 to change the representation of values.

Registers (Registers pane)

On Windows or AIX the Registers pane allows you to view and change the contents of processor registers for the threads in your program. Although threads may share the same set of registers, the operating system saves the register contents of each thread as the thread is suspended, and restores that thread's processor contents when the thread resumes. The registers are categorized, so you only need to expand the category of registers that you wish to view.

Storage (Storage pane and Storage Monitors pane)

Storage pane and Storage Monitors pane 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.

RELATED CONCEPTS

"Distributed Debugger: Overview" on page 1

"Distributed Debugger: Panes" on page 1

RELATED TASKS

"Enabling hover help for variables" on page 37

Chapter 2. Establishing a debugging environment

Setting environment variables for the debugger

The debugger user interface running on the workstation uses certain environment variables to determine the dominant language, the host files where the source files are found, and so on.

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

RELATED REFERENCES

> AIX **> WIN** "DER_DBG_CASESENSITIVE workstation environment variable" on page 48

> AIX **> WIN** "DER_DBG_LOCAL_PATH environment variable" on page 48

> AIX **> WIN** "DER_DBG_NUMBEROFELEMENTS workstation environment variable" on page 48

> AIX **> WIN** "DER_DBG_OVERRIDE environment variable" on page 48
"DER_DBG_PATH environment variable" on page 49

> AIX **> WIN** "DER_DBG_TAB environment variable" on page 49

> AIX **> WIN** "DER_DBG_TABGRID environment variable" on page 49
"INCLUDE workstation environment variable" on page 47

> AIX **> OS/2** "LIBPATH environment variable" on page 47
"Other environment variables" on page 47

Chapter 3. 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 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.

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

Optimized code debugging

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

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

Use the disassembly view or mixed view to see the source 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.

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.

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 in the object file. See your compiler reference documentation on how to compile your program with debug information.

RELATED TASKS

AIX **WIN** [“Optimized code debugging” on page 7](#)
[“Writing a program for debugging” on page 7](#)

Chapter 4. Starting the Dbugger

Starting the debugger for local debugging

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.

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.

RELATED REFERENCES

“`idebug` command” on page 39

Starting the debugger for remote debugging

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 daemon with the `irmtdbgc` command at a command line prompt. If you issue this command without any parameters, you will be prompted for required information in the Load Program dialog on the local system. For information on `irmtdbgc` command parameters, see the related topic below.
2. On the local system, start the debugger user interface with the `idebug` command using the remote debug parameters at a command line prompt. 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 `-qsession=multi` option of the `irmtdbgc` command.

RELATED REFERENCES

► **AIX** ► **WIN** “`irmtdbgc` command” on page 41
“`idebug` command” on page 39

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.

There are two types of remote debugging: *homogeneous* and *heterogeneous*. Homogeneous remote debugging occurs when both the local and remote system use the same operating system. Heterogeneous remote debugging is when the local and remote systems use different operating systems.

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.
- 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.

RELATED TASKS

AIX **WIN** "Starting the debugger for remote debugging" on page 9
"Starting the Distributed Debugger user interface daemon"

RELATED REFERENCES

"Appendix D. Remote debug limitations" on page 51

Starting the Distributed Debugger user interface daemon

Start the Distributed Debugger user interface in daemon mode if you want the 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 debugger user interface daemon to listen for a debug engine.

When you start the debug engine that will connect to this daemon, you must specify the same port number in the -qport option of the `irmtdbgc` command used to start the debug engine.

RELATED TASKS

> AIX > WIN “Starting the debugger for local debugging” on page 9

> AIX > WIN “Starting the debugger for remote debugging” on page 9

RELATED REFERENCES

“idebug command” on page 39

> AIX > WIN “irmtdbgc command” on page 41

Attaching to a Running Process

When to attach

> AIX > WIN **Restriction:** Attaching to a running process is only supported on AIX and Windows NT[®].

There are two main reasons for attaching the debugger:

- 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 at 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.

> AIX You can also use postmortem debugging to debug a core file containing information on the state of an application when it trapped. Postmortem debugging is available when debugging locally only.

RELATED TASKS

> AIX > WIN “Attaching to a local running process” on page 12

> AIX > WIN “Attaching to a remote running process” on page 12

Attaching to a local running process

You can attach the debugger to a running process either by using the Process List 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 running process with the Process List dialog:

1. Select **File > Attach Process** to invoke the Process List dialog.
2. Select the process you want to attach from the **Select Process** list.
3. If the **Process Path** field is enabled, enter the full path name to the executable associated with the process you want to attach.
4. Click **Attach**.

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

```
> AIX 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 process. Attaching to such processes can cause unpredictable results.

If you are currently debugging a process, that process is terminated when the new process is attached. When you exit the debugger, any attached process is also terminated.

You cannot restart a program that you have attached to.

RELATED CONCEPTS

> AIX > WIN "When to attach" on page 11

RELATED TASKS

> AIX > WIN "Attaching to a remote running process"

RELATED REFERENCES

"idebug command" on page 39

Attaching to a remote running process

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

You can attach the debugger to a process running on a remote system either by using the Process List 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 on a remote system with the Process List dialog:

1. On the remote system, start the debug engine using the `irmtdbgc` command. If you specify the `-qport` option, take note of it. You will need it later.
2. On the local system with the debugger user interface running, select **File > Attach Process** to invoke the Process List dialog.
3. In the Process List dialog, select **TCP/IP Connection**, then fill in the host name and port number for the remote system. If you specified the `-qport` option of the `irmtdbgc` command, the port number used in the `-qport` option must match the port number in Process List dialog.
4. Click **Refresh**.
5. Select the remote process you want to attach from the **Select Process** list.
6. If the **Process Path** field is enabled, enter the full path name to the executable associated with the process you want to attach.
7. Click **Attach**.

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

1. On the remote system, start the debug engine using the `irmtdbgc` command. If you specify the `-qport` option, take note of it. You will need it later.
2. On the local system, enter the following command:

```
> AIX idebug -qhost=<remote_host> [-qport=<host_port>] -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 process. Attaching to such processes can cause unpredictable results.

If you are currently debugging a process, that process is terminated when the new process is attached. When you exit the debugger, any attached process is also terminated.

You cannot restart a program that you have attached to.

RELATED CONCEPTS

> AIX **> WIN** "When to attach" on page 11

RELATED TASKS

> AIX **> WIN** "Attaching to a local running process" on page 12

RELATED REFERENCES

"idebug command" on page 39





> AIX **> WIN** "irmtdbgc command" on page 41

Chapter 5. 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.
-   **Function breakpoints** are triggered when the function they apply to is entered.
- **Load occurrence** breakpoints are triggered when a DLL is loaded.
-   **Address breakpoints** are triggered before the disassembly instruction at a particular address is executed.

You can set conditions on line breakpoints. When you run the program, execution stops at the breakpoint before the statement is executed if the breakpoint condition is met.



RELATED TASKS

“Setting a line breakpoint”

  “Setting a function breakpoint” on page 17

“Setting a storage change breakpoint” on page 18

“Setting a load occurrence breakpoint” on page 19

  “Setting an address breakpoint” on page 18

Setting a conditional breakpoint

Setting breakpoints

Setting a line breakpoint

You can set line breakpoints from the Source pane, the Source menu and the Breakpoints menu.

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 or in its margin.
- 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 Source menu:

1. Select **Source > Set Line Breakpoint** from the menu bar.
2. Enter the name of the module or routine you want to debug in the **Executable** entry field in the Line Breakpoint dialog. If this module or routine is loaded, you can select it from the pull-down list in the **Executable** entry field.
3. Choose or enter the object, class or source file you want to debug from the **Source** pull-down list.
4. Choose the source file containing the code for the object or class file from the **Include File** pull-down list. (This step is optional.)
5. Enter the line number within the source file you want to place a breakpoint on in the **Line** entry field.
6. If the module or routine you entered in the **Executable** 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.

To set a line breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Line** from the menu bar.
2. Enter the name of the executable you want to debug in the **Executable** entry field in the Line Breakpoint dialog. If this executable to be debugged is loaded, you can select it from the pull-down list in the **Executable** entry field.
3. Choose or enter the object, class or source file you want to debug from the **Source** pull-down list.
4. Choose the source file containing the code for the object or class file from the **Include File** pull-down list. (This step is optional.)
5. In the **Line** entry field, enter the source file line number for the desired location of your breakpoint.
6. If the module or routine you entered in the **Executable** 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.

RELATED CONCEPTS

“Breakpoints” on page 15

RELATED TASKS

“Setting multiple breakpoints” on page 20

“Modifying breakpoint properties” on page 21

“Enabling and disabling breakpoints” on page 22

“Deleting a breakpoint” on page 23

“Viewing set breakpoints” on page 21

Setting a function breakpoint

You can set function breakpoints from the Modules pane, the Source menu and the Breakpoints menu.

To set a function breakpoint from the Modules pane:

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

To set a function breakpoint from the Source menu:

1. Select **Source > Set Function Breakpoint** from the menu bar.
2. Enter the name of the function you want to debug 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.
3. If the executable or DLL containing the function you want to debug is not currently loaded, click on the **Defer breakpoint** check box.
4. Set any optional parameters that you want for the breakpoint.
5. 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.

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 you want to debug in the **Executable** entry field in the Function Breakpoint dialog. If this executable to be debugged is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. Choose or enter the object, class or source file you want to debug from the **Source** pulldown list.
4. Enter the name of the function you want to debug 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.

RELATED CONCEPTS

“Breakpoints” on page 15

RELATED TASKS

“Setting multiple breakpoints” on page 20

“Modifying breakpoint properties” on page 21

“Enabling and disabling breakpoints” on page 22

“Deleting a breakpoint” on page 23

“Viewing set breakpoints” on page 21

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, the Source menu, and from the Breakpoints menu.

To set an address breakpoint in 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 line number in the prefix area of the line.

To set an address breakpoint from the Source menu:

1. Select **Source > Set Address Breakpoint** 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.

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.

RELATED CONCEPTS

“Breakpoints” on page 15

RELATED TASKS

“Setting multiple breakpoints” on page 20

“Modifying breakpoint properties” on page 21

“Enabling and disabling breakpoints” on page 22

“Deleting a breakpoint” on page 23

“Viewing set breakpoints” on page 21


Setting a storage change breakpoint

Storage change breakpoints halt execution of your program whenever storage at a specific address is written to.

You can set storage change breakpoints from the Breakpoints menu.

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.

 **Tip:** You can enter the address of a variable by specifying the variable name preceded by an ampersand (&).

3. Select the **Bytes to Monitor** check box.
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 an 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.

RELATED CONCEPTS

"Breakpoints" on page 15

RELATED TASKS

"Setting multiple breakpoints" on page 20

"Modifying breakpoint properties" on page 21

"Enabling and disabling breakpoints" on page 22

"Deleting a breakpoint" on page 23

"Viewing set breakpoints" on page 21

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.

RELATED CONCEPTS

"Breakpoints" on page 15

RELATED TASKS

"Setting multiple breakpoints" on page 20

"Modifying breakpoint properties" on page 21

“Enabling and disabling breakpoints” on page 22
“Deleting a breakpoint” on page 23
“Viewing set breakpoints” on page 21

Setting a deferred breakpoint

A deferred breakpoint is a breakpoint set on a DLL or executable that is not currently loaded. You can defer the following types of breakpoints:

- line breakpoints
- function breakpoints
- method breakpoints

To set a deferred breakpoint, click on **Defer breakpoint** check box when setting your line breakpoint, your function, or your method breakpoint. Set any optional parameters that you want for the breakpoint.

RELATED CONCEPTS

“Breakpoints” on page 15

RELATED TASKS

“Setting multiple breakpoints”
“Modifying breakpoint properties” on page 21
“Enabling and disabling breakpoints” on page 22
“Deleting a breakpoint” on page 23
“Viewing set breakpoints” on page 21

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 either the **Source** menu or 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, the new line number, new function, new method, new entry point, or new address) and click on **Set**.
5. After you have set the last breakpoint, click on **Cancel** to dismiss the dialog.

RELATED CONCEPTS

“Breakpoints” on page 15

RELATED TASKS

“Modifying breakpoint properties” on page 21
“Enabling and disabling breakpoints” on page 22
“Deleting a breakpoint” on page 23
“Viewing set breakpoints” on page 21

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 (line, function, method, address, load occurrence, and storage change breakpoints). Expanding each type of breakpoint will provide you with a list of installed breakpoints for that type.

To view the list of installed breakpoints:

1. Click on the **Breakpoints** tab for the process 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 **Breakpoint Properties** from the pop-up menu.

RELATED CONCEPTS

“Breakpoints” on page 15

RELATED TASKS

“Setting multiple breakpoints” on page 20

“Modifying breakpoint properties”

“Enabling and disabling breakpoints” on page 22

“Deleting a breakpoint” on page 23

“Viewing set breakpoints”

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 line, function, method, procedure, or address breakpoints.
- Whether to defer the breakpoint. Only line, function, or method breakpoints can be deferred.

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. This brings forth a pop-up menu.
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.

RELATED CONCEPTS

“Breakpoints” on page 15

RELATED TASKS



“Setting multiple breakpoints” on page 20

“Enabling and disabling breakpoints”

“Deleting a breakpoint” on page 23

“Viewing set breakpoints” on page 21

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 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 line breakpoints from the Breakpoints pane. Also, you can enable or disable line 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. This brings forth a pop-up menu.
4. Select **Enable Breakpoint** or **Disable Breakpoint** from the pop-up menu.

To enable or disable a line breakpoint from 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. Scroll to the line where the breakpoint you want to enable or disable is by using the scroll bar or cursor.
3. Right-click on the line you want to enable or disable the breakpoint on. This brings forth a pop-up menu.
4. 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.

RELATED TASKS

“Deleting a breakpoint” on page 23


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 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 task below.

To delete a single breakpoint in the Source pane:

1. Locate the breakpoint you want to delete by using the scroll bar or cursor keys to locate the line where the breakpoint is set.
2. Do one of the following to delete the breakpoints:
 - 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. This brings forth a pop-up menu.
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**.

RELATED TASKS


“Enabling and disabling breakpoints” on page 22

Chapter 6. Controlling Program Execution

Running a program

You can have a program run until an active breakpoint is hit or you can have a program run to a specific line number.

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.

To run a program to a specific line number:

1. Ensure 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. 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.
3. 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.
 - Click on the line to select it, then press Ctrl+F10.

RELATED CONCEPTS

"Breakpoints" on page 15

RELATED TASKS

"Stepping through a program"

"Setting multiple breakpoints" on page 20

"Modifying breakpoint properties" on page 21

"Enabling and disabling breakpoints" on page 22


"Deleting a breakpoint" on page 23

"Viewing set breakpoints" on page 21


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, where applicable, disassembly instructions. You can issue step commands from the source pane that contains the current execution point. 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 F10.


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.

RELATED TASKS

“Running a program” on page 25





RELATED REFERENCES




“Step commands”

Step commands

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

The following types of step commands are available:

Step Command	Button	Shortcut	Description
Step Over		F10	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.

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

RELATED TASKS

“Stepping through a program” on page 25

Skipping over 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. Ensure the Source pane is set to source view. To set the Source pane to source view, select **Source > Source View** from the menu bar.
2. Make sure the line to jump to is visible in the Source pane by using the scroll bar or cursor keys to locate the line.
3. 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 function whose results are assigned to variables, or function that change the contents of variables passed by reference), or jump into the middle of a block such as a **for** loop.

RELATED TASKS

“Running a program” on page 25

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.

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.

RELATED CONCEPTS

“Step commands” on page 26


RELATED TASKS

“Running a program” on page 25



“Stepping through a program” on page 25

“Setting a line breakpoint” on page 15

Restarting a program



To start debugging your program from the beginning if your program is stopped, click on  in the toolbar or select **Debug > Restart** from the menu bar.

To start debugging your program again from the beginning, if your program is not stopped:

1. Issue a Halt command by clicking on  in the toolbar or selecting **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 on  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.

RELATED TASKS

  “Halting execution of a program” on page 28
“Running a program” on page 25

Chapter 7. Inspecting Variables, Registers, and Storage

Adding a variable to the Expressions monitor

From the Source pane or the Monitors menu, you can add variables and expressions to the Expressions monitor, so that you can keep track of how their contents change during program execution. You can add multiple variables and expressions to the Expressions monitor from the Monitors menu.

Local variables that are in scope can also be monitored in the Local Variables monitor. By default, all of the local variables in scope are available in the Locals Variables monitor.

To add a variable or expression to the Expressions monitor from 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** from the menu bar.
2. Highlight the variable or expression you want to monitor.
3. 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 Expressions monitor from the Monitors menu:

1. Select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the variable, expression, or register you want to monitor.
3. Select the **Program Monitor** check box.
4. Click **OK** to add the variable to the monitor and dismiss the dialog.

the Expressions monitor from the Monitors menu:

1. Select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the variable, expression, or register you want to monitor.
3. Click **Monitor** to add the variable to the monitor.
4. Repeat step 2 until you have added all the variables you want to monitor.
5. Click **Cancel** to dismiss the dialog.

 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.

RELATED REFERENCES

- “C/C++ supported data types” on page 43
- “C/C++ supported expression operands” on page 43
- “C/C++ supported expression operators” on page 44

Viewing the contents of a variable

You can view the contents of a variable in the Locals pane or the Monitors pane, if you have added the variable there.

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

1. Expand the thread in the Locals pane where the local variable you want to view appears.
2. If necessary, use the scroll bars or PageUp and PageDown keys to scroll the pane until the variable is visible.
3. If your variable is a class, struct or array, it can be expanded to show its individual elements.
4. If necessary, change the representation of the variable: right-click on the variable and select a representation from the **Monitor Representation** menu.

variable you have already added to the Monitors pane:

1. Use the scroll bars or PageUp and PageDown keys 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 necessary, change the representation of the variable: right-click on the variable and select a representation from the **Monitor Representation** menu.

in the Monitors pane instead of a value.

You can also view the contents of variables in the Source pane with hover help. To enable hover help, see the related task below.

RELATED TASKS

“Enabling hover help for variables” on page 37


“Adding a variable to the Expressions monitor” on page 31

“Viewing a location in storage” on page 34

Changing the contents of a variable

To change the contents of a variable in a Locals pane 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 value that is valid for the current representation of that variable or variable element. The debugger checks for a valid value.

 For example, if you want to change the contents of a character string and the current representation is a text string, you must enter a new string in double quotes, and the length of the string must not exceed the declared array size.

5. Press Enter to submit the change.

RELATED TASKS

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 necessary, use the scroll bars or PageUp and PageDown keys 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 or PageUp and PageDown keys to scroll the Monitors pane until the register is visible.
2. If necessary, change the representation of the register: right-click on the register and select a representation from the **Monitor Representation** menu.

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 necessary, change the representation of the register: right-click on the register and select a representation from the **Monitor Representation** menu.

RELATED TASKS

“Creating a new Storage monitor for an expression or register” on page 35

“Adding a register to the Expressions monitor” on page 34

Changing the contents of a register

To change the contents of a register in the Registers pane or the Monitors pane:

1. In the Registers pane or the Monitors pane, expand the entry where the register whose value you want to change is.
2. Scroll down 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. The debugger checks for a valid value. Depending on the language and platform, you may be able to change literal strings.
4. Press **Enter** to submit the change.

RELATED TASKS

“Viewing the contents of a register”

“Adding a register to the Expressions monitor” on page 34

Adding a register to the Expressions monitor

You can add a register to the Expressions monitor 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 pane. To monitor all registers during program execution, use the Registers pane.

To add a register to the Monitors pane:

1. Click on the **Monitors** tab and do one of the following:
 - Select **Monitors > Monitor Expression** from the menu bar.
 - Press Shift+F9.
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 Expressions monitor:

1. Click on the **Expressions** tab and do one of the following:
 - 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 step 2 until you have added all the registers you want to monitor.
5. Click **Cancel** to dismiss the dialog.

RELATED TASKS

“Viewing the contents of a register” on page 33

“Creating a new Storage monitor for an expression or register” on page 35

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 necessary, 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 the view to jump to. This address can be an expression, for example &x.
 - Press Enter. The storage contents now seen in the Storage pane are centered around the address you just entered.

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

1. If necessary, 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 the view to jump to.
 - Press Enter. The storage locations now seen in the Storage pane are centered around the address you just entered.
4. Use the **Go to Address** button to return to the starting address of the Storage monitor pane.

RELATED TASKS

“Changing the representation of storage contents”

“Changing the contents of a storage location”

“Creating a new Storage monitor for an expression or register”

Changing the representation of storage contents

For each storage 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 pane 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 you are viewing from the **Content style** pulldown menu.

Select the number of columns shown in a Storage pane from the **Columns Per Line** pulldown menu.

Changing the contents of a storage location

To change the contents of a storage location in a Storage monitor:

1. Select the Storage monitor pane where you want to make the change.
2. Scroll down to the storage location you want to change.
3. Double-click on the value you want to change.
4. Enter a valid value for that storage location. The debugger checks for a valid value.

 For example, if you want to change the contents of storage, and the Content style setting for a particular Storage monitor is 32-bit integer, the value you enter must be a valid 32-bit integer, not a floating-point value or other value.

Creating a new Storage monitor for an expression or register

Tip: 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.

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

You may want to create a new Storage monitor pane for an expression or 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.

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

1. Highlight the register you want to create the new Storage monitor for.
2. Right-click on the highlighted expression and select **Add Storage Monitor** from the pop-up menu. A new Storage monitor pane will appear with the expression appearing in the monitor's tab.

To create a new Storage pane for an expression from the Monitors pane:

1. Click on the **Monitors** tab and do one of the following:
 - Select **Monitors > Monitor Expression** from the menu bar.
 - Press Shift+F9.
2. In the dialog, enter the expression or register that you want to monitor.
3. Select **Storage Monitor**.
4. Click **OK** to create the new Storage monitor.
5. A new Storage pane will appear with the expression or register appearing in the monitor's tab.

To create multiple new Storage pane from the Monitors pane:

1. Click on the **Monitors** tab and do one of the following:
 - Select **Monitors > Monitor Expression** from the menu bar.
 - Press Shift+F9.
2. In the dialog, enter the first expression or the name of the register that you want to monitor. Check the Registers pane to see a valid Registers name.
3. Select **Storage Monitor**.
4. Click **Monitor** to create the new Storage monitor pane for the expression or register entered.
5. Repeat step 2 until you have added all the storage locations or registers that you want to monitor.
6. Click **Cancel** to dismiss the dialog.

RELATED TASKS

“Changing the representation of storage contents” on page 35

“Changing the contents of a storage location” on page 35

“Viewing a location in storage” on page 34

Enabling and disabling a monitored expression

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 expressions from either the Program or Locals monitor pane.

To enable or disable a monitored expression:

1. Locate the expression you want to disable or enable in the Program or Locals pane.

2. Right-click on the expression you want to enable or disable.
3. Select **Enable** or **Disable** from the pop-up menu.

Enabling hover help for variables

Hover help for variables provides you with 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. This feature is enabled by default when you first start the debugger.

To enable hover help for variables, select **Source > Allow Tool Tip Evaluation** from the menu bar.

A check mark will appear next to the Allow Tool Tip Evaluation menu item to indicate that hover help for variables is enabled.

To enable hover help 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.

RELATED CONCEPTS

“Distributed Debugger: Monitors” on page 2

Changing the representation of monitor contents

You can change the representation of variables and expressions in the Monitors pane, Locals pane or Registers pane. 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 in the Monitors pane, Locals pane, or Registers pane:

1. Right-click on the variable or expression you want to change the representation of.
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 in the Monitors pane, Locals pane, or Registers pane:

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 > Default Monitor Representation**, where *program* is the name of a program loaded in the debugger you want to change the default representation for.
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 debugger to use when no program profile is available, click **Debugger Defaults**.
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. 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.

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]]
```

The idebug_options are zero or more of the following:

Option	Purpose
-a process_id	Attach to the already running process process_id.
-h or -?	Display help for the idebug command.
-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 window sizes, positions, fonts, 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. Any changes you make to the windows 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 containing information on window, 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.
-quiet	Suppresses the splash screen when the debugger starts.

Option	Purpose
-s	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.

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						
-qlang= <dominant_language>	Specifies the dominant language to use for debugging. Valid values for <dominant_language> are:						
	<table border="1"> <thead> <tr> <th>Value</th> <th>Use to when debugging:</th> </tr> </thead> <tbody> <tr> <td>c</td> <td>C programs</td> </tr> <tr> <td>cpp</td> <td>C++ programs</td> </tr> </tbody> </table>	Value	Use to when debugging:	c	C programs	cpp	C++ programs
Value	Use to when debugging:						
c	C programs						
cpp	C++ programs						

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						
-qhost=<remote_host>	Specifies the TCP/IP name or address of the machine where the debug engine is running. This parameter is required when debugging remotely.						
-qlang= <dominant_language>	Specifies the dominant language to use for debugging. These are the valid values for <dominant_language>:						
	<table border="1"> <thead> <tr> <th>Value</th> <th>Use when debugging:</th> </tr> </thead> <tbody> <tr> <td>c</td> <td>C programs</td> </tr> <tr> <td>cpp</td> <td>C++ programs</td> </tr> </tbody> </table>	Value	Use when debugging:	c	C programs	cpp	C++ programs
Value	Use when debugging:						
c	C programs						
cpp	C++ programs						
-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 <code>irmtdbg</code> command.						



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 will run locally, not as a daemon.						
-qlang= <dominant_language>	Specifies the dominant language to use for debugging. These are the valid values for <dominant_language>:						
	<table border="1"> <thead> <tr> <th>Value</th> <th>Use when debugging:</th> </tr> </thead> <tbody> <tr> <td>c</td> <td>C programs</td> </tr> <tr> <td>cpp</td> <td>C++ programs</td> </tr> </tbody> </table>	Value	Use when debugging:	c	C programs	cpp	C++ programs
Value	Use when debugging:						
c	C programs						
cpp	C++ programs						
-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. On AIX or Windows there is no default port number.						
-qterminate	Closes any running Debugger user interface daemons before starting a new user interface daemon.						

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.

RELATED TASKS

  “Starting the debugger for local debugging” on page 9

  “Starting the debugger for remote debugging” on page 9

irmtdbgc command

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.

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.

The irmtdbgc command has the following syntax:

```
irmtdbg [irmtdbg_parameters] [-] [program_name [program_parameters]]
```

where `irmtdbg_parameters` are:

Parameter	Description
<code>-qprotocol=<protocol></code>	Specifies the communications protocol to use. Only TCP/IP is supported. This is the default protocol.
<code>-qport=<port></code>	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 <code>-qport</code> parameter of the <code>idebug</code> command. The default port is 8000. Restriction: Do not use this parameter when connecting to a debugger user interface daemon. You must use the <code>-quiport</code> option.
<code>-quiport=<ui_daemon_port></code>	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 <code>-quiport</code> parameter of the <code>idebug</code> command.
<code>-qhost=<ui_daemon_host></code>	Specifies the TCP/IP name or address of the machine where the debugger user interface daemon is listening.
<code>-qsession=single multi</code>	Specifies whether to support single session debugging or multiple session debugging. The default is <code>single</code> . In single session debugging, the debug engine terminates when the program you are debugging runs to completion or is terminated manually. In multiple session debugging, 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.

Use the “-” parameter to separate `irmtdbg` 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 `irmtdbg` command, the debugger will prompt you for the required information in the Load Program dialog of the debugger user interface.

Tip: If this command gives you an error, you may have an older version of the debug engine on your remote system. Install a new version of the debug engine, if available. If no newer version of the debug engine is available, try using the `irmtdbg` command.

RELATED REFERENCES

“`idebug` command” on page 39

Appendix B. 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
- 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: <ul style="list-style-type: none">• 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')
 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.

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

Operator	Coded as
Bitwise OR	$a \mid b$
Bitwise exclusive OR	$a \hat{\ } b$
Logical AND	$a \ \&\& \ b$
Logical OR	$a \ \mid\mid \ b$

Appendix C. Environment Variables

Other environment variables

> AIX > WIN **Restriction:** This information applies to debugging programs on AIX and Windows only.

The debugger also uses the following standard environment variables on the Windows or OS/2[®] workstation. These variables all contain one or more directory names separated by semicolons:

PATH

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. On Windows platforms the PATH environment variable is also used to locate DLLs.

> OS/2 DPATH

The DPATH environment variable is used to locate message files, which the debugger needs to display messages and the text of menus and dialogs.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

RELATED REFERENCES

> AIX > OS/2 “LIBPATH environment variable”

INCLUDE workstation 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.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

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 workstation 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.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

DER_DBG_LOCAL_PATH environment variable

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

> AIX > WIN The DER_DBG_LOCAL_PATH environment variable is used to locate executables and DLLs on the system where you are debugging your program.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

DER_DBG_NUMBEROFELEMENTS workstation environment variable

The DER_DBG_NUMBEROFELEMENTS environment variable can be set to an integer value to tell the debugger the maximum number of elements to display for an array, structure, or object in a Program or Storage monitor.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

DER_DBG_OVERRIDE environment variable

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

The DER_DBG_OVERRIDE environment variable takes precedence over DER_DBG_PATH. If you set your DER_DBG_PATH variable in your system settings, but you want to temporarily add another path that takes precedence over DER_DBG_PATH, set DER_DBG_OVERRIDE. To restore DER_DBG_PATH as the path used to locate executables and DLLs, clear DER_DBG_OVERRIDE. You can clear DER_DBG_OVERRIDE using the following command:

```
set DER_DBG_OVERRIDE=
```

RELATED TASKS

“Setting environment variables for the debugger” on page 5

RELATED REFERENCES

“DER_DBG_PATH environment variable” on page 49

DER_DBG_PATH environment variable

The DER_DBG_PATH environment variable is used 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: The DER_DBG_PATH environment variable must be set on both the client and server systems before starting the Distributed Debugger user interface daemon.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

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.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

RELATED REFERENCES

“DER_DBG_TABGRID environment variable”

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.

RELATED TASKS

“Setting environment variables for the debugger” on page 5

RELATED REFERENCES

“DER_DBG_TAB environment variable” on page 49

Appendix D. 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.

Appendix E. 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 similar to the Load Program dialog, which allows you to choose whether to debug program initialization and whether to use a program profile. The name of the new program is shown, but unlike in the Load Program dialog, 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.

If you on AIX or Windows checked **Use program profile** or used the `p+` option of the `idebug` command and the Distributed Debugger finds enabled breakpoints in the profile, these breakpoints are set when you start debugging the child process.

RELATED REFERENCES

"fork() handling"

"system() handling"

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. Breakpoints set in the process you did not choose are ignored, and the page pertaining to that process is closed. Execution stops at the next source code statement in the program that contains debugging information.

If the process you did *not* choose performs an `exec()`, a new Distributed Debugger page will open for the new child process.

RELATED REFERENCES

"exec() handling"

"system() handling"

system() handling

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. On AIX 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. The `/bin/sh` process issues a `fork()` call. Again, the Distributed Debugger stops and brings up a Fork action dialog.
8. 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.
9. The new child 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. On AIX 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()`.
10. Click **OK**. The Distributed Debugger stops at `main()`. From here you can continue debugging.

RELATED REFERENCES

- “`exec()` handling” on page 53
- “`fork()` handling” on page 53

Contacting IBM

We're delighted to offer a solid cornerstone for your application development: IBM's comprehensive support services. Whether you are an occasional user with a simple question, a power user with many complex technical questions, or someone who requires application design assistance and consulting, IBM Support can meet your needs.

Comments on This Help

Please let us know about any errors or omissions in this online help or in the hardcopy Getting Started book, or our PDF documents. Send your e-mail to: torrcf@ca.ibm.com

Fee Support

Developers on the VisualAge C++ for AIX Service and Support team handle everything from simple how-to's to complex technical problems. Solutions may take the form of a brief explanation, a workaround, a fix to the current product, or a fix to the next release.

<http://www.ibm.com/support/> describes IBM Support Offerings on all platforms, worldwide.

<http://www.ibm.com/rs6000/support/> describes support offerings on the RS/6000® platform, in your country. It also indicates whether your country provides support electronically over the Internet in addition to telephone support.

<http://www.lotus.com/passport> describes the IBM and Lotus® Passport Advantage™ contracting option.

The IBM Software Support Handbook, accessible from <http://www.ibm.com/software/support>, also lists worldwide support contacts.

Phone numbers for information on Support Line offerings are:

- **United States:** 1-888-426-4343 (IBM Global Services), option 3 as of December 1999. Should this number change, IBM general information at 1-800-IBM-4YOU (1-800-426-4968) can route you to the appropriate group.
- **Canada:** 1-800-465-9600, option 3 as of December 1999. Should this number change, you can also contact IBM general information at 1-800-IBM-4YOU (1-800-426-4968).

Please call 1-800-266-8720 in the U.S. and Canada for information on Passport Advantage offerings.

- Elsewhere, please contact your local IBM office.

If you contact support, please have the following information available:

- The product name
- The product version
- The hardware configuration and software (product names and versions you are using)
- What happened and what you were doing when the problem occurred

- Whether you tried to solve the problem and how
- The exact wording of any messages displayed

Consulting Services

VisualAge and WebSphere™ Product Affinity Services Group is a core group of technical specialists from the IBM development labs that created the IBM VisualAge and WebSphere products. With access to a network of IBM product area experts, IBM and industry business partners, and some of the best resources in the industry, we can put the optimal team in place to meet the challenge of absorbing new technology. Our goal is to enable organizational success with VisualAge and WebSphere — ensuring that our products are used effectively within your development team.

For more information, visit <http://www.ibm.com/software/ad/vaws-services/> or contact the Product Affinity Services Team at:

AIM_SERVICES@us.ibm.com