

C/C++ Productivity Tools for OS/390



Distributed Debugger

Release 1.0

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 67.

First Edition (September 1999)

This edition applies to C/C++ Productivity Tools for OS/390 Release 1.0, program number 5655-B85 and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Consult the latest edition of the applicable system bibliography for current information on these products.

Order publications through your IBM representative or through the IBM branch office serving your locality.

© Copyright International Business Machines Corporation 1999. All rights reserved.

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

Contents

About this book.	v	Setting a storage change breakpoint	27
Who should read this book	v	Setting a load occurrence breakpoint	28
Conventions used in this book	v	Setting a deferred breakpoint.	29
Related information	v	Setting multiple breakpoints	30
How to send your comments.	vi	Viewing breakpoints	30
		Enabling and disabling breakpoints	31
		Deleting a breakpoint	32
Chapter 1. Distributed Debugger	1	Chapter 5. Controlling program execution	35
Distributed Debugger: Overview	1	Running a program	35
Distributed Debugger: Panes	2	Stepping through a program	36
Distributed Debugger: Monitors	3	Skipping over sections of a program	36
Distributed Debugger and Debug Tool	4	Selecting debugger recognized exceptions	37
Recursion and debugging	6	Terminating a debug session without exiting the debugger	38
Breakpoints	6		
Mapping Layouts	7	Chapter 6. Inspecting variables	39
UNIX call handling during debugging	9	Adding a variable or expression to the Monitors pane	39
exec() handling	9	Viewing the contents of a variable	39
fork() handling	9	Changing the contents of a variable	40
system() handling	9	Enabling hover help for variables	41
		Changing the representation of monitor contents.	41
Chapter 2. Preparing a program for debugging.	11	Chapter 7. Inspecting registers	43
Setting environment variables for the debugger	11	Viewing the contents of a register	43
Writing a program for debugging	11	Changing the contents of a register.	43
Compiling a program for debugging	12	Adding a register to the Monitors pane	44
Debugging a CICS application	12		
Debugging a DB2 program or stored procedure	14	Chapter 8. Inspecting storage	45
Debugging a Webserver application	16	Viewing a location in storage.	45
Debugging an IMS application	17	Changing the representation of storage contents.	46
Preparing your OS/390 C/C++ application for debugging.	19	Changing the contents of a storage location	46
		Adding a new Storage Monitor pane for an expression or register	46
Chapter 3. Starting the debugger for OS/390 programs	21	Chapter 9. Mapping storage	49
Starting the Distributed Debugger user interface daemon.	21	Mapping pointers, addresses and registers	49
Starting applications with Debug Tool in OS/390 batch	22	Defining a mapping layout	49
Starting applications with Debug Tool in OS/390 UNIX.	23	Enabling and disabling a monitored variable, expression	53
Chapter 4. Working with breakpoints	25		
Setting a line breakpoint	25		
Setting a function breakpoint.	26		

Chapter 10. Reference	55	Order of source file searching	61
idebug command	55	Exception levels	62
Step commands	56	Remote debug limitations	62
C/C++ expressions supported	57	Optional breakpoint parameters	63
C/C++ supported data types.	57	Program Profiles	64
C/C++ supported expression operands	57		
C/C++ supported expression operators	58	Notices	67
Environment variable	59	Trademarks and service marks	69
DER_DBG_PATH environment variable	59		
Troubleshooting	60		
Why the Distributed Debugger cannot			
find source files on the workstation . . .	60		

About this book

Distributed Debugger introduces you to the Distributed Debugger and provides information about how to debug an OS/390 C and C++ application.

Who should read this book

Distributed Debugger is intended for application programmers who want to debug C and C++ applications on OS/390 and want a workstation debugger to enhance their existing familiar host environment. For these users, this document introduces the Distributed Debugger and shows how to use it.

Conventions used in this book

The following conventions distinguish different text styles within this book:

plain	Window titles, folder names, icon names, and method names.
monospace	Programming examples, user input at the command line prompt or into an entry field, directory paths.
bold	Menu choices and menu names, labels for push buttons, check boxes, radio buttons, group-box controls, drop-down list boxes, combination-boxes, notebook tabs, and entry fields.
<i>italics</i>	Programming keywords and variables, and titles of documents.

Related information

For information on OS/390 C/C++ related features, news and Web sites, add this Web site to your browser's bookmark list:

<http://www.ibm.com/software/ad/c390>

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other C/C++ Productivity Tools documentation, send your comments by e-mail to torrcf@ca.ibm.com. Be sure to include the name of the book, the document number of the book, the version of C/C++ Productivity Tools, and, if applicable, the specific location of the information on which you are commenting (for example, a page number or a table number).

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. The IBM Debug Tool serves as the debug engine and is accessible through a network. For more information on how the Distributed Debugger user interface and Debug Tool work together, see the related topic below.

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.

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” on page 2
- “Distributed Debugger: Monitors” on page 3
- “Distributed Debugger and Debug Tool” on page 4

Distributed Debugger: Panes

The following panes are available in the Distributed Debugger user interface:

Stacks pane

The Stacks pane provides a view of the call stack for the current program thread 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.

When debugging a multithreaded application, the debugger serializes the events reported by the different active threads. For example, if your program runs as two tasks (Task A and Task B) and Task A has an event that needs to be reported (such as a breakpoint is encountered), the debugger will handle the request and begin operating on behalf of Task A. If, during that period, Task B has an event that needs to be reported again, the debugger will hold Task B until the event from Task A is complete (such as you telling the debugger to run or step after the breakpoint has been reached.) When Task A is complete the debugger releases its hold on Task B and handles the new event from Task B. The Distributed Debugger can only provide program state information and interact with the thread for which it is currently handling an event.

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, delete them, or add new ones.

Source pane

The Source pane provides a view of the source code for the program you are debugging. To view source code, the source code must be accessible from your workstation, either on a local or a network drive or accessible from OS/390.

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 view and modify 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 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)

The Locals pane allows you to monitor local variables for the current 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)

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

Storage (Storage pane and Storage 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, view, modify the contents of storage, and change the representation used to display storage.

The initial Storage pane shows the storage areas used by your program at its starting address. The starting address is the address at the entry to `main()` in the first load module.

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

Mapping (Mapping pane)

You can display the contents of a selected storage block according to a user-defined layout specified in an XML file.

RELATED CONCEPTS

“Distributed Debugger: Overview” on page 1

“Distributed Debugger: Panes” on page 2

RELATED TASKS

“Enabling hover help for variables” on page 41

Distributed Debugger and Debug Tool

The Distributed Debugger provides the client graphical user interface to the debug information provided by Debug Tool running on OS/390. The Distributed Debugger client is invoked on the workstation in a mode which causes it to wait for a TCP/IP connection from Debug Tool.

Your OS/390 application may be debugged with Debug Tool in any of these environments:

- TSO
- Batch
- OS/390 UNIX System Services (formerly known as OpenEdition MVS)
- CICS
- IMS
- DB/2
- Webserver

Debug Tool is the OS/390 engine of the Distributed Debugger. The Distributed Debugger client makes requests to Debug Tool and the Distributed Debugger client displays the results in the graphical windows. For example, to monitor a variable, the Distributed Debugger client asks Debug Tool for the value; Debug Tool responds with a value and the Distributed Debugger client displays the value in the Monitors pane.

To invoke Debug Tool on OS/390, you may need to specify the Language Environment« run-time option, TEST, when starting your application. For example, in an OS/390 batch environment, you can specify the TEST run-time option in the PARM parameter on the EXEC statement. For more information on the TEST run-time option, see *Debug Tool User's Guide and Reference*, SC09-2137.

Debug Tool depends on compiler generated symbol table information and debug hooks in your code such as:

- Function entry hook
- Function exit hook
- Line hook
- Statement hook

At least one of the compile units of your application must be compiled with the compile-time TEST option. The units compiled with the TEST option are the only ones which can be debugged.

For more information on Debug Tool, see *Debug Tool User's Guide and Reference*, SC09-2137.

RELATED TASKS

“Starting the Distributed Debugger user interface daemon” on page 21

“Starting applications with Debug Tool in OS/390 UNIX” on page 23

“Starting applications with Debug Tool in OS/390 batch” on page 22

“Debugging a DB2 program or stored procedure” on page 14

“Debugging a Webserver application” on page 16

“Debugging an IMS application” on page 17

“Debugging a CICS application” on page 12

RELATED REFERENCES

“Remote debug limitations” on page 62

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.

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 a particular function is entered.
- **Storage change breakpoints** are triggered when storage at a specified address is changed.
- **Load occurrence** breakpoints are triggered when a specified DLL is loaded.

You can set conditions on line breakpoints. When you run the program, execution stops at the breakpoint if the specified condition is true.

RELATED TASKS

“Setting a line breakpoint” on page 25

► C ► C++ “Setting a function breakpoint” on page 26
“Setting a storage change breakpoint” on page 27
“Setting a load occurrence breakpoint” on page 28

RELATED REFERENCES

“Optional breakpoint parameters” on page 63

Mapping Layouts

You can display the contents of a storage block mapped according to a user-defined layout.

Selecting a storage block

You specify the starting address of the storage block either by selecting a pointer from the Monitors pane or the Locals pane, a register from the Registers pane, or an address from the Storage Pane or the Storage Monitors pane, or by typing it in an Add Layout dialog entry field. When selecting a pointer or a register, its current value is used as a base address of the storage block. This means that if later on the value stored in the pointer variable or in the register changes, the layout does not change. The size of the storage block is determined by the size of the selected layout.

Specifying layouts

Each predefined storage layout is stored in one XML file. The XML file format provides for describing either structures of predefined primitive type elements or nested layouts where a layout element can actually point to another storage layout file. The layout file also specifies the length of the storage block to be laid out. A list of available layout files is contained in a master layout file named UserViews.XML.

Layout definition and deployment

You need to create the XML layout files and the master layout file using a text editor. The Distributed Debugger allows for specifying the location of the master UserViews.XML file and of the layout files.



Mapping pane

When you map storage blocks, a Mapping pane appears. Initially this pane is empty. The Mapping pane displays a set of layout elements representing pairs


(storage block address, user defined layout) each containing a set of sub-elements representing pairs (Header, Value) according to the description in a user-defined layout file.

You can add layouts to this pane by either selecting the Map Storage action that is available for pointers from the Monitors pane or the Locals pane, registers from the Registers pane, or addresses from the Storage Pane or the Storage Monitors pane, or by using the Add Layout action from a pop-up menu on the Mapping Pane. When the Map Storage action is invoked, you are prompted to select a mapping layout from the list of available layouts specified in the master layout file: UserViews.XML.

If the specified storage block is protected or cannot be accessed the display values will be shown as a number of "?"s.

The layout elements may be expanded or contracted by clicking on  or  at the left of the layout element. Initially the layout element and any sub-elements representing nested layouts are not populated (no sub-elements are generated yet). The first time you expand a layout element it is populated according to the XML layout file. Populating the layout element means breaking the storage block into fragments corresponding to the layout elements specified in the XML. The values displayed for the layout sub-elements are formatted according to a default primitive type specified in the XML file.

Hover help for the layout sub-elements displays the offset in bytes from the base address represented as a HEX number.

You can take different actions for different objects from the Mapping pane. On the Mapping pane you can add a new layout element by specifying both the base address and the XML layout file. You must enter the base address as a HEX number and the name of the XML file. You can also expand all the layout elements and sub-elements. This action does not populate them if they are empty. You must click on  to populate a sub-element. You can collapse all the layout elements and sub-elements

At the layout element level you can update the mapped storage block's contents and refresh the layout element and its sub-elements. You can change the base address or the XML layout file for the selected layout element (the address must be specified as a HEX number) and refresh the layout element and its sub-elements. You can also remove the layout element and its sub-elements. Finally, you can select a sub-element searching for either the header of a sub-element or its offset. If an offset is specified and it does not match the offset of any sub-element the Find action will try to select the element with the closest starting offset.

At the layout sub-element level you can select the display value representation for the selected sub-element. You must select from a list of available representations which are specific to the type of the selected element. You can also edit the value of the selected layout sub-element.

UNIX call handling during debugging

exec() handling

When a process calls `exec()`, a new process is loaded and replaces the original process. The debugger will discard the original process and starts to debug the second process created by `exec()`.

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 starts another program using `system()` under the debugger the debugger will switch the debug process to the second program and load it

into another Source pane if it is debuggable. The first program is suspended and is resumed when the second program is completed.

RELATED REFERENCES

“exec() handling” on page 9

“fork() handling” on page 9

Chapter 2. Preparing a program for debugging

Setting environment variables for the debugger

The Distributed Debugger user interface running on the workstation uses the `DER_DBG_PATH` environment variable.

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

If you set the workstation `DER_DBG_PATH` environment variable, you must set it before starting the Distributed Debugger user interface daemon. If you set the OS/390 `DER_DBG_PATH` environment variable, you must set it before invoking the OS/390 program you wish to debug.

RELATED REFERENCES

“`DER_DBG_PATH` environment variable” on page 59

Writing a program for debugging

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

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

To debug programs, you must specify the compiler options that generate debug information. In some cases, you must specify additional options that enable the debug engine to work properly with your code. For information on these compiler options and for other information on preparing your application for debugging, see *Debug Tool User's Guide and Reference*, SC09-2137.

Compiling a program for debugging

In order to debug your program at the source code level, you need to compile your program with certain compiler options that instruct the compiler to generate symbolic information and debug hooks in the object file. See your compiler reference documentation on how to compile your program with debug information. For information on these compiler options and for other information on preparing your application for debugging, see *Debug Tool User's Guide and Reference*, SC09-2137.

RELATED TASKS

“Writing a program for debugging” on page 11

Debugging a CICS application

Preparing the CICS environment

- Refresh the CICS definitions for Debug Tool. You can find these definitions in the EQACCSO and EQACDCT members of Debug Tool's SEQASAMP data set.
- Update the JCL that starts CICS:
 - To include Debug Tool's SEQAMOD data set and the Language Environment runtime library (in the DFHRPL concatenation).
 - To include EQA00DYN from Debug Tool's SEQAMOD data set in the STEPLIB concatenation by either:
 - APF authorizing the SEQAMOD data set and adding the data set to the STEPLIB concatenation
 - Copying the EQA00DYN module from the SEQAMOD data set to a library already in the STEPLIB concatenation.
 - To ensure that no DD cards exist for CINSPIN, CINSPLS, or CINSPOP.

Setting up, configuring, and starting CICS TCP/IP

For details on TCP/IP implementation and configuration in a CICS subsystem, see *IP CICS Sockets Guide*, SC31-8518.

Setting compiler and run-time options

- Applications that you want to debug must be compiled with the TEST option
- To indicate the workstation where the Distributed Debugger client user interface daemon is running, do one of the following:
 - Use Debug Tool transaction DTCN to identify the workstation.

- Include a *#pragma runopts* directive in your application's source file. For example:

```
#pragma runopts(TEST(,,VADTCPIP&wkst_id%portid:*))
```

or

```
#pragma runopts(TEST(,, "VADTCPIP&9.11.22.33:*"))
```

If *wkst_id* is the numeric IP address, the subparameter must be put in quotes. *%portid* defaults to 8000, and if defined, must match the value that was specified when the Distributed Debugger client user interface daemon was started on the workstation.

Note: If you identify the workstation in your source file, you will need to recompile each time you change the workstation location. DTCN allows you to define the workstation location just prior to invoking the application. For details on using DTCN to specify debugging requirements, see *Debug Tool User's Guide and Reference*, SC09-2137. Also note that the `__ctest()` function is not supported in CICS.

Initiating a CICS transaction

When you debug your program, the actual program statements are shown in the Distributed Debugger as they are executing. To accomplish this, the data set input to the compiler is used. This might not be the original source, for example, if the program has been prepared by the CICS translator. The data set input to the compiler must be retained in a permanent data set.

To initiate a CICS transaction program, start the Distributed debugger user interface in daemon mode, specifying the port to be used in the OS/390 TEST run-time option. The daemon only needs to be started once. You can then conduct as many remote debug sessions as needed without having to restart the daemon. When the daemon dialog displays indicating that it is listening to a specified port where the daemon was started, run your CICS application. When Debug Tool establishes a connection with the debug daemon, the user interface will be started on the workstation, and debugging may proceed.

RELATED TASKS

"Debugging a DB2 program or stored procedure" on page 14

"Debugging a Webserver application" on page 16

"Debugging an IMS application" on page 17

"Starting applications with Debug Tool in OS/390 batch" on page 22

"Starting applications with Debug Tool in OS/390 UNIX" on page 23

"Starting the Distributed Debugger user interface daemon" on page 21

Debugging a DB2 program or stored procedure

You can debug programs that perform SQL calls to access DB2 tables or call DB2 stored procedures. The stored procedures may also be debugged, but they will appear as a process that is separate from the calling program.

Preparing the DB2 environment

Update the JCL that starts DB2:

- To include Debug Tool's SEQAMOD data set and the Language Environment runtime library. This applies to debugging a DB2 program that accesses tables and programs that call stored procedures.
- To debug stored procedures, the Debug Tool SEQAMOD data set must also be concatenated to the STEPLIB of the startup JCL for the stored procedure address space. Read RACF access to the Debug Tool library and the library that contains the stored procedure source must be given to DB2SYS, which owns the stored procedure process as it executes in the stored procedure address space.

Setting compiler and run-time options

- DB2 applications and stored procedures that you want to debug must be compiled with the TEST option.
- To indicate the workstation where the Distributed Debugger client user interface daemon is running, do one of the following:
 - In the application program or stored procedure source code (C/C++) , include a *#pragma runopts* directive. For example:
`#pragma runopts(TEST(,,VADTCPIP&wkst_id:*))`
or
`#pragma runopts(TEST(,, "VADTCPIP&9.11.22.33:*"))`
If *wkst_id* is the numeric IP address, the subparameter must be put in quotes. Since the *#pragma runopts* directive is imbedded into the source file, you will need to change the directive and recompile, any time the Distributed Debugger user interface will be run on a different workstation.
 - When submitting the DB2 program using the TSO RUN command set the PARMS parameter as follows:
`RUN PROG (MYPROG)
PLAN (MYPLAN)
LIB ('MY.TEST.LOAD')
PARMS ('TEST(,,VADTCPIP&wkst_id:*)/')`
If *wkst_id* is the numeric IP address, the subparameter must be put in quotes. The TEST option must be followed by the slash (/), to show that it is a run-time option and not a program parameter.

- For debugging the stored procedure itself, you may set the RUNOPTS column in the row for your stored procedure, in the system table SYSIBM.SYSPROCEDURES, to the following value:
`TEST(,,,VADTCPIP&wkst_id:*)`
 If *wkst_id* is the numeric IP address, the subparameter must be put in quotes. You must obtain an adequate DB2 privileged level to update this table. For more details on this table and stored procedures, see *DB2 for OS/390 Application Programming and SQL Guide*, SC26-8958.
 For the full syntax of the TEST run-time option for debugging, see *Debug Tool User's Guide and Reference*, SC09-2137.

Debugging a DB2 application program

When you debug your program, the actual program statements are shown in the Distributed Debugger as they are executing. To accomplish this, the data set input to the compiler is used. This might not be the original source, for example, if the program has been prepared by the CICS translator. The data set input to the compiler must be retained in a permanent data set.

To start debugging a DB2 application program, start the Distributed Debugger user interface in daemon mode, specifying the port to be used in the OS/390 TEST run-time option. The daemon only needs to be started once. You can then conduct as many remote debug sessions as needed without having to restart the daemon. When the daemon dialog displays indicating that it is listening to a specified port where the daemon was started, run your DB2 application.

When Debug Tool establishes a connection with the Distributed Debugger daemon, the user interface will be started on the workstation, and debugging may proceed. If the DB2 program calls a stored procedure, which has been set up for debugging as outlined above, step into the DB2 function call that invokes the stored procedure. A second debug program page appears, allowing you to debug the stored procedure itself. The debug session of the calling program will be suspended until the session for the stored procedure is complete.

RELATED TASKS

- “Debugging a CICS application” on page 12
- “Debugging a Webserver application” on page 16
- “Debugging an IMS application” on page 17
- “Starting applications with Debug Tool in OS/390 batch” on page 22
- “Starting applications with Debug Tool in OS/390 UNIX” on page 23
- “Starting the Distributed Debugger user interface daemon” on page 21

Debugging a Webserver application

You can debug Webserver applications as requested through a client Web browser.

Preparing the Webserver environment

Update the JCL or shell script which starts the Webserver to include Debug Tool's SEQAMOD data set in its STEPLIB.

Setting compiler options

Applications that you want to debug must be compiled with the appropriate TEST compiler option to include symbolic information for source-level debugging. Your application must be compiled as a dynamic link library (DLL).

Setting Webserver environment options

To allow Debug Tool to establish a connection with the Distributed Debugger user interface daemon running on a workstation, the network location of the workstation must be specified. This location is specified in the Webserver configuration file, for example, httpd.conf as follows:

```
debugtooladdr [workstation location] [debug port]
```

Note: The location of your workstation can be specified as a name for DNS resolution or as an IP address. The debug port number by default is 8000.

Debugging is done through the GWAPI debug Service directive. Each application you wish to debug must be specified as a dbgService directive. This directive has the same syntax as the Service directive. The dbgService directive specifies the debuggable entry point that the server calls during the service step. This in turn services the client's debug request. The syntax of this directive is as follows:

```
dbgService [web path]* [real path]:[exported entry point]*
```

[web path]

defines the virtual location of the application on the server as seen by the Web browser.

[real path]

is the actual location of the application on your web server.

[exported entry point]

defines the entry function of your debuggable application.

Debugging the Webserver application

When you debug your program, the actual program statements are shown in the Distributed Debugger as they are executing. To accomplish this, the data set input to the compiler is used. This might not be the original source, for example, if the program has been prepared by the CICS translator. The data set input to the compiler must be retained in a permanent data set.

To start debugging a Webserver application program, start the Distributed Debugger user interface in daemon mode on the workstation, specifying the port to be used in the OS/390 TEST run-time option. The daemon only needs to be started once. You can then conduct as many remote debug sessions as needed without having to restart the daemon. When the daemon dialog displays indicating that it is listening to a specified port where the daemon was started, run your Web application. When Debug Tool establishes a connection with the daemon, the user interface will be started on the workstation, and debugging may proceed.

RELATED TASKS

“Debugging a CICS application” on page 12

“Debugging a DB2 program or stored procedure” on page 14

“Debugging an IMS application”

“Starting applications with Debug Tool in OS/390 batch” on page 22

“Starting applications with Debug Tool in OS/390 UNIX” on page 23

“Starting the Distributed Debugger user interface daemon” on page 21

Debugging an IMS application

You can debug IMS DB programs submitted under MVS batch or IMS programs run under the control of IMS BTS (Batch Terminal Simulator).

Preparing the IMS environment

To debug IMS DB and TM applications:

1. With BTS, run in TSO in the foreground (IMS TM and DB)
2. With BTS, run BTS as a batch job (IMS TM and DB)
3. Without BTS, run as an IMS batch job (IMS DB only)

For BTS, you need to include the DT SEQAMOD data set in the STEPLIB or TASKLIB data set concatenations for BTS. For IMS batch, you need to put DT SEQAMOD in the STEPLIB concatenation, for example, in the STEPLIB concatenation for DLIBATCH.

Setting compiler and run-time options

- Applications that you want to debug must be compiled with the TEST compiler option .
- To indicate the workstation where the Distributed Debugger client user interface daemon is running, do one of the following:
 - include a *#pragma runopts* directive. For example:

```
#pragma runopts(TEST(,,VADTCPIP&wkst_id:*))
```

or

```
#pragma runopts(TEST(,,,"VADTCPIP&9.11.22.33:*))
```

 If the *wkst_id* is numeric, the IP address of the subparameter must be put in quotes. Since the *#pragma runopts* directive is imbedded into the source file, you will need to change the directive and recompile, any time the Distributed Debugger user interface will be run on a different workstation. When running BTS or an IMS batch program, you cannot code the TEST run-time option in the JCL as you would when debugging an OS/390 batch program. For the full syntax of the TEST run-time option for debugging, see *Debug Tool User's Guide and Reference*, SC09-2137.
 - include a run-time options module when linking the program to be debugged. The run-time options, including the TEST option, must be coded and assembled in a user-defined run-time option module. For example:

```
CEEUOPT CSECT
CEEUOPT AMODE ANY
CEEUOPT RMODE ANY
CEEXOPT TEST=(,,,"VADTCPIP&&9.11.22.33:*)
END
```

 For instructions on how to create the CEEUOPT run-time options module using the CEEXOPT macro, see *Debug Tool User's Guide and Reference*, SC09-2137.
 If you use the run-time options module method and you change the workstation where the daemon is running, you do not have to recompile the program (as with *#pragma runopts*), you need to assemble a new run-time options module and relink your program with the new run-time options module.

Debugging an IMS application program

When you debug your program, the actual program statements are shown in the Distributed Debugger as they are executing. To accomplish this, the data set input to the compiler is used. This might not be the original source, for example, if the program has been prepared by the CICS translator. The data set input to the compiler must be retained in a permanent data set.

To start debugging an IMS application program, start the Distributed debugger user interface in daemon mode, specifying the port to be used in the OS/390 TEST run-time option. The daemon only needs to be started once.

You can then conduct as many remote debug sessions as needed without having to restart the daemon. When the daemon dialog displays indicating that it is listening to a specified port where the daemon was started, run BTS or your IMS DB program. When Debug Tool establishes a connection with the daemon, the user interface will be started on the workstation, and debugging may proceed.

RELATED TASKS

- “Debugging a CICS application” on page 12
- “Debugging a DB2 program or stored procedure” on page 14
- “Debugging a Webserver application” on page 16
- “Starting applications with Debug Tool in OS/390 batch” on page 22
- “Starting applications with Debug Tool in OS/390 UNIX” on page 23
- “Starting the Distributed Debugger user interface daemon” on page 21

Preparing your OS/390 C/C++ application for debugging

To be able to debug your application you must:

- Compile your application program with the TEST option.
- If you intend to specify the Language Environment runtime TEST option using the *#pragma runopts* directive in your source code, use the following syntax:

```
#pragma runopts("TEST(,,VADTCPIP&wkst_id%portid:*)")
```

where:

wkst_id

is the symbolic name of the workstation id or the numeric IP address of the workstation where Distributed Debugger client is waiting.

%portid

is the TCP/IP port number which is optional and defaults to 8000; if specified, the value must match the port number that was entered when Distributed Debugger client user interface daemon was started on the workstation.

To begin debugging at a specific location within a program, you can use the `__ctest()` or `ctest()` function. Place one of these functions at the location in the program where you want to begin debugging. If you use this approach to debug a program, you need to specify the NOTEST run-time option with the workstation IP address. For example, you can specify the option as `#pragma runopts("NOTEST(,,VADTCPIP&wkst_id%portid:*)")`. The `__ctest()` function is not supported in CICS. For more information on the `ctest()` function, see *Debug Tool User's Guide and Reference*, SC09-2137.

RELATED TASKS

- “Starting applications with Debug Tool in OS/390 batch” on page 22
- “Starting applications with Debug Tool in OS/390 UNIX” on page 23
- “Debugging a CICS application” on page 12
- “Debugging a Webserver application” on page 16
- “Debugging a DB2 program or stored procedure” on page 14
- “Debugging an IMS application” on page 17

Chapter 3. Starting the debugger for OS/390 programs

Starting the Distributed Debugger user interface daemon

To debug an application running on OS/390, you must start the Distributed Debugger user interface daemon first. The daemon only needs to be started once. You can then conduct as many remote debug sessions as needed without having to restart the daemon. The Distributed Debugger user interface will appear only after a remote debug session has connected to it from OS/390.

The application you want to debug runs on the host and causes Language Environment to load Debug Tool. Debug Tool connects to the Distributed Debugger user interface daemon running on your workstation and a program pane will display. However, although the TCP/IP port number is optional and defaults to 8000, if specified, the value on the TEST option must match the port number that was entered when the Distributed Debugger client user interface daemon was started on the workstation.

You can start the Distributed Debugger daemon in any of the following ways:

- By double-clicking on its icon from a program folder. If you start the Distributed Debugger from an icon you must make sure the properties for the icon include the `-qdaemon` and `-quiport` options.
- By selecting **IBM C and C++ Productivity Tools for OS 390 > IBM Distributed Debugger** from the Windows **Start** menu. The port number is set to 8000. Make sure the port parameter of the TEST run-time option matches this already set port number.
- By entering the Distributed Debugger command `idebug -qdaemon -quiport=<port>` on a Productivity Tools command line.

<port> The port number where you want the Distributed Debugger user interface daemon to listen for Debug Tool. For the port number in this example, use 8000. This is the port number used by default in the port parameter of the TEST run-time option which causes the Language Environment to load Debug Tool. The same port number must be used by the Distributed Debugger user interface daemon and Debug Tool.

RELATED CONCEPTS

“Distributed Debugger and Debug Tool” on page 4

RELATED TASKS

“Starting applications with Debug Tool in OS/390 UNIX” on page 23
“Starting applications with Debug Tool in OS/390 batch”

RELATED REFERENCES

“idebug command” on page 55

Starting applications with Debug Tool in OS/390 batch

You must set your STEPLIB to point to Debug Tool SEQAMOD data set, the Language Environment runtime library, and the library which contains the application you intend to debug. You also need to specify the runtime TEST option. If you did not code *#pragma runopts* in your program, you need to specify the runtime TEST option through the PARM parameter on the EXEC JCL statement. One way of doing this is to create a JCL script as shown below.

In the sample JCL below, the following names have been used:

- MYPROG is the executable file for the program to be debugged.
- USERID.PROJ1.LOAD is the load library where the executable file resides.
- *wkst_id* is the numeric IP address of your workstation or the TCP/IP name of your workstation.
- *%portid* is the TCP/IP port number which is optional and defaults to 8000; if specified, the value must match the port number that was entered when Distributed Debugger client user interface daemon was started on the workstation.

The example JCL below illustrates how you can invoke your host application (MYPROG) so that Debug Tool will connect with your workstation (*wkst_id*) for a debug session.

```
//RUN EXEC PGM=MYPROG,  
// PARM='TEST(,,,VADTCPIP&wkst_id%portid:*)/'  
//STEPLIB DD DISP=SHR,DSN=USERID.PROJ1.LOAD /*Application executable library */  
// DD DISP=SHR,DSN=EQAW.V1R2M0.SEQAMOD /*Debug Tool library*/  
// DD DISP=SHR,DSN=SYSID.CEE.SCEERUN /*LE runtime library */  
// DD DISP=SHR,DSN=SYSID.CBC.SCLBDLL /*LE runtime class library */  
//SYSTCPD DD DISP=SHR,DSN=TCPIP.PROFILE(TCPDATA) /*TCP/IP data file */
```

Note: If your OS/390 batch environment is not using the default TCP/IP data set named TCPIP.TCPIP.DATA, you need to specify the name of your environment’s TCP/IP data set on the SYSTCPD DD statement. If your OS/390 batch environment is using the default data set name, you can omit the SYSTCPD DD statement.

You can also use the workstation IP address instead of your workstation's TCP/IP name, for example:

```
//          PARM='TEST(,,VADTCPIP&9.22.34.46:*)/' /* C/C++ program parm syntax */
```

In this statement, *%portid* is not specified and the default *%portid* of 8000 is used by Debug Tool.

RELATED CONCEPTS

"Distributed Debugger and Debug Tool" on page 4

Starting applications with Debug Tool in OS/390 UNIX

You must set your STEPLIB to point to Debug Tool SEQAMOD data set, the Language Environment runtime library, and the library which contains the application you intend to debug, if you intend to run the application from PDS. You also need to specify the runtime TEST option. If you did not code *#pragma runopts* in your program, you need to specify the runtime TEST option through the system variable `_CEE_RUNOPTS`. One way of doing this is to create a shell script, for example, `dbg` in OS/390 UNIX System Services with the following statements:

```
#dbg - shell script to debug program
export STEPLIB=USERID.PROJ1.LOAD:\
EQAW.V1R2M0.SEQAMOD:\
SYSID.CEE.SCEERUN:\
SYSID.CBC.SCLBDLL
export _CEE_RUNOPTS="TEST(,,VADTCPIP&wkst_id%portid:*)";
"$@"
```

where:

wkst_id

is the numeric IP address of your workstation or the TCP/IP name of your workstation.

%portid

is the TCP/IP port number which is optional and defaults to 8000; if specified, the value must match the port number value that was entered when Distributed Debugger user interface daemon was started on the workstation.

To set up your environment and debug myprog, use the shell script `dbg` and execute as follows:

```
>dbg myprog
```

The application being debugged (myprog) runs on the host and causes Language Environment to load Debug Tool. Debug Tool connects to the Distributed Debugger user interface daemon running on your workstation and a program pane will display.

If your program is an OS/390 PDS or PDSE member, you need to:

- Create an OS/390 UNIX System Services file which has the same name as the OS/390 load module (PDS/PDSE member), for example, touch myprog10.
- Make the OS/390 UNIX file executable by setting the sticky bit on (for example, `chmod 1700 myprog10`).
- Concatenate your load library containing the program by exporting the STEPLIB environment variable. See the dbg shell script example provided above.
- Set up the environment and debug the program by using the dbg shell script. For example, `dbg myprog10`.

For example, if the program you want to debug is `USERID.PROJ1.LOAD(MYPROG10)`, use the following commands:

```
>touch /u/userid/myprog10
>chmod 1700 /u/userid/myprog10
>dbg myprog10
```

You can export STEPLIB and _CEE_RUNOPTS with the values shown in the above dbg shell script either by issuing the export command in the current shell or by setting these environment variables in your .profile file. However, doing so causes the variable values to be used for all commands and applications run in the current shell. For example, with the _CEE_RUNOPTS TEST option specified, if you issue an ls command, the system attempts to debug the ls command itself. To prevent this situation, use a shell script such as dbg described above. For more information on shell scripts, see *OS/390 UNIX System Services User's Guide*, SC28-1891.

RELATED CONCEPTS

“Distributed Debugger and Debug Tool” on page 4

Chapter 4. Working with 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. Make sure the appropriate line is visible in the Source pane by using the scroll bar or cursor keys to locate the line.
2. Do one of the following:
 - Double-click on the line number in the prefix area of the line.
 - Right-click on the line you want to set a breakpoint on, and select **Set Breakpoint** from the pop-up menu.

To set a line breakpoint from a menu:

1. Select **Source > Set Line Breakpoint** from the menu bar or **Select Breakpoints > Set Line** from the menu bar.
2. Enter the name of the module or routine in which you want to set a breakpoint in the **Executable** entry field in the Line Breakpoint dialog. If this module or routine is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. In the **Source** entry field, enter the object, class or source file that is associated with the module or routine specified in the **Executable** entry field and contains the line where the breakpoint is to be set. If the module or routine specified in the **Executable** entry field is loaded, you can select the file from the **Source** pulldown list.
4. In the **Source** entry field, enter the source file containing the code for the object or class file. If the module or routine specified in the **Executable** entry field is loaded, you can select the file from the **Include File** pulldown list. (This step is optional if you have not selected to defer the breakpoint.)
5. Enter the line number within the source file where you want to place a breakpoint in the **Line** entry field.
6. If the module or routine you entered in the **Executable** entry field is not currently loaded, click on the **Defer breakpoint** check box.
7. Set any optional parameters that you want for the breakpoint.

- Click **OK** to set the breakpoint and close the Line Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without closing the Line Breakpoint dialog.

RELATED CONCEPTS

“Breakpoints” on page 6

RELATED TASKS

- “Setting a function breakpoint”
- “Setting a storage change breakpoint” on page 27
- “Setting a load occurrence breakpoint” on page 28
- “Setting a deferred breakpoint” on page 29
- “Setting multiple breakpoints” on page 30
- “Enabling and disabling breakpoints” on page 31
- “Deleting a breakpoint” on page 32
- “Viewing breakpoints” on page 30

RELATED REFERENCES

“Optional breakpoint parameters” on page 63

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:

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

To set a function breakpoint from a menu:

- Select **Source > Set Function Breakpoint** from the menu bar or **Select Breakpoints > Set Function** from the menu bar.
- Enter the name of the executable which contains the function where you want to set a breakpoint in the **Executable** entry field in the Function Breakpoint dialog. If this executable is loaded, you can select it from the pulldown list in the **Executable** entry field.
- In the **Source** entry field, enter the object, class or source file that is associated with the module or routine specified in the **Executable** entry field and contains the function where the breakpoint is to be set. If the module or routine specified in the **Executable** entry field is loaded, you can select the file from the **Source** pulldown list.

4. In the **Function** entry field, enter the name of function where the breakpoint is to be set. If the function specified in the **Executable** entry field is loaded, you can select the file from the **Function** pulldown list. (This step is optional if you have not selected to defer the breakpoint.)
5. If the executable containing the function you want to debug 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 close the Function Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without closing the Function Breakpoint dialog.

RELATED CONCEPTS

“Breakpoints” on page 6

RELATED TASKS

“Setting a line breakpoint” on page 25

“Setting a storage change breakpoint”

“Setting a load occurrence breakpoint” on page 28

“Setting a deferred breakpoint” on page 29

“Setting multiple breakpoints” on page 30

“Enabling and disabling breakpoints” on page 31

“Deleting a breakpoint” on page 32

“Viewing breakpoints” on page 30

RELATED REFERENCES

“Optional breakpoint parameters” on page 63

Setting a storage change breakpoint

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

To set a storage change breakpoint from the Breakpoints menu:

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



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

3. Specify the number of bytes to be monitored in the **Bytes to Monitor** field.

4. Set any optional parameters that you want for the breakpoint.
5. Click **OK** to set the breakpoint and close the Storage Change Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without closing the Storage Change Breakpoint dialog.

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

RELATED CONCEPTS

"Breakpoints" on page 6

RELATED TASKS

"Setting a line breakpoint" on page 25

"Setting a function breakpoint" on page 26

"Setting a load occurrence breakpoint"

"Setting a deferred breakpoint" on page 29

"Setting multiple breakpoints" on page 30

"Enabling and disabling breakpoints" on page 31

"Deleting a breakpoint" on page 32

"Viewing breakpoints" on page 30

RELATED REFERENCES

"Optional breakpoint parameters" on page 63

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 close the Load Occurrence Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without closing the Load Occurrence Breakpoint dialog.

RELATED CONCEPTS

“Breakpoints” on page 6

RELATED TASKS

“Setting a line breakpoint” on page 25

“Setting a function breakpoint” on page 26

“Setting a storage change breakpoint” on page 27

“Setting a deferred breakpoint”

“Setting multiple breakpoints” on page 30

“Enabling and disabling breakpoints” on page 31

“Deleting a breakpoint” on page 32

“Viewing breakpoints” on page 30

RELATED REFERENCES

“Optional breakpoint parameters” on page 63

Setting a deferred breakpoint

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

- line breakpoints
- function breakpoints

To set a deferred breakpoint, click on the **Defer breakpoint** check box when setting one of the above types of breakpoints.

RELATED CONCEPTS

“Breakpoints” on page 6

RELATED TASKS

“Setting a line breakpoint” on page 25 “Setting a function breakpoint” on page 26

“Setting a storage change breakpoint” on page 27

“Setting a load occurrence breakpoint” on page 28

“Setting multiple breakpoints” on page 30

“Enabling and disabling breakpoints” on page 31

“Deleting a breakpoint” on page 32

“Viewing breakpoints” on page 30

RELATED REFERENCES

“Optional breakpoint parameters” on page 63

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, new line number or function) and click on **Set**.
5. After you have set the last breakpoint, click on **Cancel** to close the dialog.

RELATED CONCEPTS

“Breakpoints” on page 6

RELATED TASKS

“Setting a line breakpoint” on page 25

“Setting a function breakpoint” on page 26

“Setting a storage change breakpoint” on page 27

“Setting a load occurrence breakpoint” on page 28

“Setting a deferred breakpoint” on page 29

“Enabling and disabling breakpoints” on page 31

“Deleting a breakpoint” on page 32

“Viewing breakpoints”

RELATED REFERENCES

“Optional breakpoint parameters” on page 63

Viewing breakpoints

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

To view the list of breakpoints:

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

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

RELATED CONCEPTS

“Breakpoints” on page 6

RELATED TASKS

“Setting a line breakpoint” on page 25

“Setting a function breakpoint” on page 26

“Setting a storage change breakpoint” on page 27

“Setting a load occurrence breakpoint” on page 28

“Setting a deferred breakpoint” on page 29

“Setting multiple breakpoints” on page 30



“Enabling and disabling breakpoints”

“Deleting a breakpoint” on page 32

RELATED REFERENCES

“Optional breakpoint parameters” on page 63

Enabling and disabling breakpoints

You can disable a breakpoint so that it does not stop execution and then later enable it again. Information about the breakpoint (such as type, location, condition, and frequency) is saved by the Distributed Debugger when the breakpoint is disabled. When you enable a breakpoint, the Distributed Debugger restores the saved information. When you delete a breakpoint, however, the Distributed Debugger does not save any information about the breakpoint. If you decide to reinstate the breakpoint, you must add the breakpoint and reenter the information. Enabled breakpoints are indicated with a red dot (). Disabled breakpoints are indicated with a gray dot ().

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

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

1. Click on the **Breakpoints** tab to bring the Breakpoints pane to the foreground.

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

To enable or disable a breakpoint from the Source pane:

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

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

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

RELATED TASKS

“Deleting a breakpoint”

Deleting a breakpoint


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

To delete a single breakpoint in the Source pane:

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

To delete a single breakpoint in the Breakpoints pane:

1. Click on the **Breakpoints** tab to bring the Breakpoints pane to the foreground.

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

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

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

RELATED TASKS

“Enabling and disabling breakpoints” on page 31


Chapter 5. Controlling program execution

Running a program

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

- end of program is reached
- an active breakpoint is hit
- a specific line number is reached
- an exception occurs.

To run a program until an active breakpoint is encountered, end of program is reached, or an exception occurs, do one of the following:

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

To run a program until a specific statement is encountered, end of program is reached, or an exception occurs, do the following:

1. Make sure the line to run to is visible in the Source pane by using the scroll bar or cursor keys to locate the line.
2. Run the program to the line by doing one of the following:
 - Right-click on the line to bring up the pop-up menu, then select **Run To Location**.
 - Click on the line to select it, then select **Debug > Run To Location** from the menu bar.
 - Click on the line to select it, then press F10.

Note: A reported exception is determined by the exception level specified in the Distributed Debugger exception filtering dialog.

RELATED CONCEPTS

“Breakpoints” on page 6

RELATED TASKS

“Stepping through a program” on page 36

“Setting multiple breakpoints” on page 30

“Enabling and disabling breakpoints” on page 31

“Deleting a breakpoint” on page 32

“Viewing breakpoints” on page 30

“Selecting debugger recognized exceptions” on page 37

RELATED REFERENCES

“Exception levels” on page 62


Stepping through a program

You can use step commands to step through your program a single statement at a time. 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 Return command, do one of the following:

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

RELATED TASKS

“Running a program” on page 35

RELATED REFERENCES

“Step commands” on page 56

Skipping over sections of a program

You can skip over sections of code to avoid executing certain statements or to move to a position so that certain statements can be executed again. If you attempt to jump to a line which is outside the block containing the current execution point, a message is displayed and the jump is not permitted.

To skip over a section of code:

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

RELATED TASKS

“Running a program” on page 35

Selecting debugger recognized exceptions

You can select the level of exceptions the Distributed Debugger recognizes for processes you are debugging, so that stepping or execution will stop when an exception occurs that matches the specified level. By default, all unhandled exceptions are recognized by the Distributed Debugger.

To specify the level of exceptions to be recognized by the Distributed Debugger:

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

Note: If no level is selected, the Distributed Debugger defaults to the value specified in the TEST run-time option. If more than one level is selected, the one with the greater scope is used. For example, if TEST(NONE) and TEST(ALL) are selected, the TEST(ALL) selection is used.

To cancel your exception filter preferences settings, click **Reset**.

To set your exception filter preference to the default settings, click **Default**.

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

RELATED REFERENCES

“Exception levels” on page 62

Terminating a debug session without exiting the debugger

To terminate the execution of a program that is currently running in the debugger and not exit the debugger, do one of the following:

- Click on .
- Select **Debug > Terminate** from the menu bar.

RELATED TASKS

“Running a program” on page 35

“Starting applications with Debug Tool in OS/390 batch” on page 22

“Starting applications with Debug Tool in OS/390 UNIX” on page 23

Chapter 6. Inspecting variables

Adding a variable or expression to the Monitors pane

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

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

To add a variable or expression to the Monitors pane from the Source pane:

1. Highlight the variable or expression you want to monitor.
2. Right-click on the highlighted variable, and select **Add to Program Monitor** from the pop-up menu.

To add a variable or expression to the Monitors pane from the Monitors menu:

1. Select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the variable or expression you want to monitor.
3. Select the **Program monitor** radio button.
4. Click **OK** to add the variable or expression to the monitor and close the dialog. Alternatively, use the **Monitor** button to add the variable or expression to the monitor without closing the dialog.

RELATED REFERENCES

“C/C++ supported data types” on page 57

“C/C++ supported expression operands” on page 57

“C/C++ supported expression operators” on page 58

Viewing the contents of a variable

You can view the contents of a variable or expression in the Locals pane or the Monitors pane, if you have added the variable there. By default, all local variables in scope are added to the Locals pane.

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

1. Expand the thread in the Locals pane where the local variable you want to view appears.

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

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

1. If your variable's type is a class, struct or array, it can be expanded to show its individual elements.
2. If desired, change the representation of the variable: right-click on the variable and select a representation from the **Monitor Representation** menu.

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

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

RELATED TASKS

“Enabling hover help for variables” on page 41

“Adding a variable or expression to the Monitors pane” on page 39

“Viewing a location in storage” on page 45

“Mapping pointers, addresses and registers” on page 49

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's type is a class, struct or array, expand it to show its individual elements.
3. Scroll 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.
5. Press Enter to submit the change.

RELATED TASKS

“Adding a variable or expression to the Monitors pane” on page 39

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 disabled 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 close the dialog.

RELATED CONCEPTS

“Distributed Debugger: Monitors” on page 3

Changing the representation of monitor contents

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

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 close the Monitor Representation dialog.

To change the default representation of variables or expressions:

1. Select **File > Preferences** from the main menu bar. The Application Preferences dialog appears.
2. In the left-hand pane of the Application Preferences dialog, expand **Debug > *program* > Default Monitor Representation**, where *program* is the name of a program loaded in the Distributed 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 Distributed Debugger, click **Debugger Defaults**.
5. Click **OK** to change the default representations and close the Application Preferences dialog.

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

Chapter 7. Inspecting registers

Viewing the contents of a register

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

To view the contents of a register in the Registers pane:

1. Expand the thread for which you want to view the registers.
2. Expand the register category that contains the register you want to view.
3. If desired, 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, scroll the Monitors pane until the register is visible.
2. If desired, change the representation of the register: right-click on the register and select 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, scroll the Storage pane until the register is visible.
2. If desired, change the representation of the register: right-click on the register and select a representation from the **Monitor Representation** menu.

RELATED TASKS

“Adding a new Storage Monitor pane for an expression or register” on page 46

“Adding a register to the Monitors pane” on page 44

“Mapping pointers, addresses and registers” on page 49

Changing the contents of a register

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

1. In the Registers pane or Monitors pane, expand the entry which contains the register whose value you want to modify.
2. Scroll to the register you want to change and do one of the following:

- Double-click on the register.
 - Right-click on the register and select **Edit** from the pop-up menu.
3. Enter a value that is valid for the current representation of that register. Changes do not take effect until the application regains control, for example, if you issue a step or run command. If you change the contents of a register more than once before returning control to the application, the last value entered will be used. Also changes to the IAR are ignored. You can enter a new value but the application will not use it.
 4. Press Enter to submit the change.

RELATED TASKS

“Viewing the contents of a register” on page 43

“Adding a register to the Monitors pane”

Adding a register to the Monitors pane

You can add a register to the Monitors pane if you want to monitor only a few registers during the execution of your program. Registers can also be monitored in the Registers pane and Storage Monitor 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.
3. Select **Program Monitor**.
4. Click **OK** to add the register to the Monitors pane and close the dialog. Alternatively, use the **Monitor** button to add the register to the monitor without closing the dialog.

Tip: Check the Registers pane to see the valid registers names.

RELATED TASKS

“Viewing the contents of a register” on page 43

“Adding a new Storage Monitor pane for an expression or register” on page 46

Chapter 8. Inspecting storage

Viewing a location in storage



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

To view the contents of storage from the Storage pane:

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

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

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

  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 TASKS

- “Changing the representation of storage contents”
- “Changing the contents of a storage location”
- “Adding a new Storage Monitor pane for an expression or register”
- “Mapping pointers, addresses and registers” on page 49

Changing the representation of storage contents

You can change the representation of the storage and the number of columns shown in the Storage monitor or Storage Monitor panes.

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

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

Changing the contents of a storage location

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

1. Select the Storage pane or Storage Monitor pane where you want to make the change.
2. Scroll down to the storage location you want to change.
3. Double-click on the value you want to change or right-click on the value and select **Edit** from the pop-up menu.
4. Enter a valid value for that storage location.

Note: The left most column is not editable. Performing the above steps in the far left column will cause the Storage pane to scroll and the Storage Monitor pane to change.

Adding a new Storage Monitor pane for an expression or register

Note: Registers and expressions can also be monitored using the Monitors pane. If you want to view all registers at once, use the Registers pane.

If you want to monitor specific locations in storage or a few registers during program execution, you can add a new Storage Monitor pane for an expression or register.

Warning: If there is a variable in scope which has the same name as the register that you are trying to use, the variable will be used.

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

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

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

1. Select the variable from the source. Right-click and from the pop-up menu select **Add to Storage Monitor** or 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 the **Storage Monitor** radio button.
4. Click **OK** to add the new Storage Monitor pane. Alternatively, use the **Monitor** button to add the expression or register to the Storage Monitor pane without close the dialog.
5. A new Storage Monitor pane will appear with the expression or register appearing in the monitor's tab.

Tip: Check the Registers pane to see the valid registers names.

RELATED TASKS

“Changing the representation of storage contents” on page 46

“Changing the contents of a storage location” on page 46

“Viewing a location in storage” on page 45

Chapter 9. Mapping storage

Mapping pointers, addresses and registers

You can map pointers from the Monitors or Locals pane. You can map addresses from a Storage pane or Storage Monitor pane. You can map registers from the Registers pane.

These elements are mapped according to user-defined layout. The layouts are defined in XML. For more information on defining a mapping layout in XML, see the related topic below.

To map a pointer, address or register:

1. Highlight the pointer, address or register you want to map.
2. Right-click on the highlighted pointer, address or register, and select **Map Storage** from the pop-up menu.
3. Select a desired storage mapping from the list.
4. Click **OK** to add the storage layout of the item you selected to the Mapping pane and close the dialog.

RELATED CONCEPTS

“Mapping Layouts” on page 7

RELATED TASKS

“Defining a mapping layout”

Defining a mapping layout

Defining a mapping layout is a two step process. In the first step you create the layout XML file. In the second step you add it to the master UserViews.XML file.

The example below defines the layout for the following C language structure:

```
typedef struct {  
    char char_val;  
    unsigned short ushort_val;  
    short short_val;  
    unsigned long ulong_val;  
    long long_val;  
    char string_val[32];  
} _test;
```

Creating the layout XML file

The XML file format is defined in the layout.dtd document type definition (DTD) file as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!ELEMENT LAYOUT (FIELD)+>
  <!ATTLIST LAYOUT Header CDATA #REQUIRED length CDATA #REQUIRED >
  <!ELEMENT FIELD (FIELD)*>
  <!ATTLIST FIELD
    Header CDATA #REQUIRED
    Type
      (16_BIT_INT|16_BIT_UINT|16_BIT_HINT|32_BIT_INT|32_BIT_UINT|32_BIT_HINT|32_BIT_FLOAT|
       64_BIT_INT|64_BIT_FLOAT|CHARACTER|HEX|ASCII|EBCDIC|STRUCTURE|PADDING|BIT|BITMASK|MAP)
    #REQUIRED
    length CDATA #REQUIRED
    layout CDATA #IMPLIED>
```

This means the the XML layout file would specify first a header (title) and the total length of the layout followed by a list of sub-elements (FIELD) described by a header (name), length and primitive type which is used to determine the default representation of that sub-element.

There are also special sub-element types:

- STRUCTURE introduces a nested structure; the sub-element has no value
- PADDING used to define a block of bytes that does not need to be specifically laid out
- BITMASK used to define a bitmasked sub-element. Its sub-elements represent bits or groups or bits defined by the BIT type.

The XML file describing the _test structure and conforming to this format is:

```
<?xml version="1.0"?>
<!DOCTYPE LAYOUT SYSTEM "Layout.dtd" >
  <LAYOUT Header = "A Layout" length="17">
    <FIELD Header = "char_val" Type = "CHARACTER" length = "1" > </FIELD>
    <FIELD Header = "ushort_val" Type = "16_BIT_UINT" length = "2" > </FIELD>
    <FIELD Header = "short_val" Type = "16_BIT_INT" length = "2" > </FIELD>
    <FIELD Header = "ulong_val" Type = "32_BIT_UINT" length = "4" > </FIELD>
    <FIELD Header = "long_val" Type = "32_BIT_INT" length = "4" > </FIELD>
    <FIELD Header = "string_val" Type = "ASCII" length = "32"> </FIELD>
  </LAYOUT>
```

Note: The <!DOCTYPE LAYOUT SYSTEM "Layout.dtd" > line specifies which DTD file is used to parse the contents of this file. If the Layout.DTD file is situated in a different location then the full path to that location must be specified.

Defining padding fields

If you decide to ignore the `long_val` field but want to show the `string_val` type in the layout, the XML file will look like:

```
<FIELD Header = "short_val" Type = "16_BIT_INT" length = "2" > </FIELD>
<FIELD Header = "ulong_val" Type = "32_BIT_UINT" length = "4" > </FIELD>
<FIELD Header = "" Type = "PADDING" length = "4" > </FIELD>
<FIELD Header = "string_val" Type = "ASCII" length = "32"> </FIELD>
```

The initial purpose for defining of the PADDING sub-elements is to deal with byte aligned structures but it can also be used to skip a data area that does not need to be detailed in the layout.

Defining structures

The following piece of XML shows the usage of STRUCTURE fields for mapping nested structures. A structure top element does not have an associated value and it can be expanded to show its sub-elements. While the length of the STRUCTURE field is added to the total size of the XML layout, the included field sizes are intended for display only. For example, the following structure would only mean 344 bytes out of the total layout size.

```
<FIELD Header = "MACHINE CHECK LOG OUT AREA" Type = "STRUCTURE" length = "344" >
  <FIELD Header="reserved" Type="HEX" length="16"></FIELD>
  <FIELD Header="FLCSID" Type="HEX" length="4"></FIELD>
  <FIELD Header="FLCIOFP" Type="HEX" length="4"></FIELD>
  <FIELD Header="reserved" Type="HEX" length="20"></FIELD>
  <FIELD Header="FLCESAR" Type="HEX" length="4"></FIELD>
  <FIELD Header="FLCCTSA" Type="HEX" length="8"></FIELD>
  <FIELD Header="FLCCCSA" Type="HEX" length="8"></FIELD>
  <FIELD Header="FLCMCIC" Type="HEX" length="8"></FIELD>
  <FIELD Header="reserved" Type="HEX" length="8"></FIELD>
  <FIELD Header="FLCFSA" Type="HEX" length="4"></FIELD>
  <FIELD Header="reserved" Type="HEX" length="4"></FIELD>
  <FIELD Header="FLCFLA" Type="HEX" length="16"></FIELD>
  <FIELD Header="FLCRV110" Type="HEX" length="16"></FIELD>
  <FIELD Header="FLCARSAV" Type="STRUCTURE" length="64">
    <FIELD Header="AR0" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR1" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR2" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR3" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR4" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR5" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR6" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR7" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR8" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR9" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR10" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR11" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR12" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR13" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR14" Type="HEX" length="4"></FIELD>
    <FIELD Header="AR15" Type="HEX" length="4"></FIELD>
  </FIELD>
</FIELD>
```

```

<FIELD Header="FLCFPSAV" Type="HEX" length="32"></FIELD>
<FIELD Header="" Type="PADDING" length="64"></FIELD>
<FIELD Header="" Type="PADDING" length="64"></FIELD>
</FIELD>

```

Defining bitmask fields

The following XML piece is a sample for describing BITMASK fields. The length of the BITMASK is specified in bytes and it contains a set of BIT fields for which the length is specified in bits. The offset shown for the BIT fields is a bit offset within the BITMASK field. While the length of the bitmask field is added to the total size of the XML layout, the individual BIT field sizes are intended for display only.

```

<FIELD Header="byte_field" Type="BITMASK" length="2">
  <FIELD Header="hi_byte" Type="BIT" length="8"></FIELD>
  <FIELD Header="lo_byte" Type="BIT" length="8"></FIELD>
</FIELD>

```

Defining nested layouts

The MAP field type together with the optional layout field, let you describe nested layouts as in the following OS/390 DSA layout example:

```

<?xml version="1.0"?>
<!DOCTYPE LAYOUT SYSTEM "Layout.dtd" >
<LAYOUT Header = "DSA" length="72">
<FIELD Header = "FLAGS" Type = "HEX" length = "2"></FIELD>
<FIELD Header = "junk" Type = "HEX" length = "2"></FIELD>
<FIELD Header = "Back Chain" Type = "MAP" length = "4" layout="dsa.xml" ></FIELD>
<FIELD Header = "Forward Chain" Type = "MAP" length = "4" layout="dsa.xml" ></FIELD>
<FIELD Header = "R14" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R15" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R0" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R1" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R2" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R3" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R4" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R5" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R6" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R7" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R8" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R9" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R10" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R11" Type = "HEX" length = "4"></FIELD>
<FIELD Header = "R12" Type = "HEX" length = "4"></FIELD>
</LAYOUT>

```

This well-formed XML layout is stored in a file called DSA.XML. Since you know that fields 3 and 4 contain pointers to different DSA structures you add two nested layout definitions.

Note: The actual storage mapping for that layout is actually executed when you expand the layout element for the first time in order to prevent recursive layout expansions.

Adding the new layout file to the list of available layouts

Once the XML layout file is ready it must be made visible to Distributed Debugger through the UserViews.XML file. The DTD for this file follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!ELEMENT CLASSES (CLASS)+>
  <!ATTLIST CLASSES Total CDATA #REQUIRED>
  <!ELEMENT CLASS EMPTY >
  <!ATTLIST CLASS Name CDATA #REQUIRED ActionLabel CDATA #REQUIRED>
```

It defines a list of XML layout file and label pairs. The XML layout file is the full path to the required layout file while the label is used when you select a storage mapping.

If you want to add the TestLayout.XML file to the UserViews.XML file, the final result looks like:

```
<?xml version="1.0"?>
<!DOCTYPE CLASSES SYSTEM "Classes.dtd" >
<CLASSES Total="2">
<CLASS Name = "OtherLayout.XML" ActionLabel = "other layout"></CLASS>
<CLASS Name = "TestLayout.XML" ActionLabel = "_test C structure"></CLASS>
</CLASSES>
```

Note: Specifying the .XML at the end of the XML layout file name is required.

Once the XML file is added to the UserViews.XML it will be available from the Map Storage dialog.

Enabling and disabling a monitored variable, 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 variables, expressions or registers from either the Monitor pane or Locals pane.

To enable or disable a monitored expression, variable or register:

1. Locate the variable, expression or register you want to disable or enable in the Monitors pane or Locals pane.
2. Right-click on the variable, expression or register you want to enable or disable.

3. Select **Enable** or **Disable** from the pop-up menu.

Chapter 10. Reference

idebug command

The idebug command starts the Distributed Debugger user interface daemon which waits for Debug Tool on OS/390 to connect to it. The idebug command has the following syntax:

```
idebug [ui_daemon_parameters]
```




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	<p>Tells the Distributed Debugger user interface to run as a daemon. You must use the -quiport option when specifying -qdaemon.</p> <p>This is a required parameter.</p>
-quiport=<port>	<p>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 use a comma to delimit the port numbers.</p> <p>This option is required when using the -qdaemon option.</p> <p>One of the port numbers specified here must be used as the port number in the port parameter of the TEST run-time option.</p>
-qterminate	<p>Closes the Distributed Debugger user interface daemon.</p>
-qremotesource=<path>	<p>Specifies the location of source files on the OS/390 system where the OS/390 application will be running and debugged. Locations can include PDS names, sequential file names, HFS path names or both sequential file names and HFS path names.</p> <p>If more than one name is specified, the names must be separated by a semicolon. For example, -qremotesource=USER1.PROJ.C;/u/user1/dir2;</p>

Step commands

You can use step commands to step through your program a single line.

The following types of step commands are available:

Step Command	Button	Shortcut	Description
Step Over		F10	Executes the current statement, without stopping in any functions or routines called within the statement.
Step Into		F11	Executes the current statement. Execution stops at the next statement 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 statement 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 an active 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 36

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
- 80-bit floating-point
- Pointers

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.

Operand	Definition
Constant	<p>The constant can be one of the following types:</p> <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')

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>

Operator	Coded as
Unary minus	$-a$
Unary plus	$+a$
Dereference	$*a$
Type cast	$(type) a$
Multiply	$a * b$
Divide	a / b
Modulo	$a \% b$
Add	$a + b$
Subtract	$a - b$
Left shift	$a << b$
Right shift	$a >> b$
Less than	$a < b$
Greater than	$a > b$
Less than or equal to	$a <= b$
Greater than or equal to	$a >= b$
Equal	$a == b$
Not equal	$a != b$
Bitwise AND	$a \& b$
Bitwise OR	$a b$
Bitwise exclusive OR	$a \wedge b$
Logical AND	$a \&\& b$
Logical OR	$a b$

Environment variable

DER_DBG_PATH environment variable

The DER_DBG_PATH workstation environment variable is used to locate debug source files on your client workstation. If 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
```

There is also an OS/390 DER_DBG_PATH environment variable which you can set to locate source files on your OS/390 system. If your source code is stored in /u/userid/source and /u/userid/source/include, you should set your OS/390 DER_DBG_PATH variable as follows:

```
set DER_DBG_PATH=/u/userid/source:/u/userid/source/include
```

If the source file is a PDS member, you may specify a PDS name in the OS/390 DER_DBG_PATH variable and the member will be searched for in that PDS.

Note: If you set the workstation DER_DBG_PATH environment variable, you must set it before starting the Distributed Debugger user interface daemon. If you set the OS/390 DER_DBG_PATH environment variable, you must set it before invoking the OS/390 program you wish to debug.

RELATED TASKS

“Setting environment variables for the debugger” on page 11

RELATED REFERENCES

“Order of source file searching” on page 61

Troubleshooting

Why the Distributed Debugger cannot find source files on the workstation

You may not be able to open the source for an object, even though the code was compiled with debug information, if the Distributed Debugger cannot find the source code for it. When you start debugging such a program, or when execution lands in a part of the program that was compiled with debug information but the Distributed Debugger cannot find the source code for it, the Distributed Debugger opens a Source Filename dialog. In this dialog you can enter the name of the file (HFS file, sequential data set, or PDS and member name) containing the source code.

Another reason the source code might not display at your workstation is that you expected the Distributed Debugger to look for source files on the workstation, but have not downloaded the required source files, or have not told the Distributed Debugger where to find them.

If you want the Distributed Debugger to search for source files on the workstation, make sure that:

- You have downloaded the necessary source files or mounted the necessary file system as a Windows NT drive using a remote file system such as NFS.
- The DER_DBG_PATH environment variable on the workstation contains the paths of all directories where such source files are stored. Ensure that you have separated entries with semicolons. For example, if you have files in F:\PRIVATE\SOURCE and G:\BUILD\SRC on your workstation, you

should set DER_DBG_PATH to:
F:\PRIVATE\SOURCE;G:\BUILD\SRC

Note: If you set the workstation DER_DBG_PATH environment variable, you must set it before starting the Distributed Debugger user interface daemon. If you set the OS/390 DER_DBG_PATH environment variable, you must set it before invoking the OS/390 program you wish to debug.

RELATED REFERENCES

“DER_DBG_PATH environment variable” on page 59

“Order of source file searching”

Order of source file searching

Debug Tool searches for C and C++ source files in the following locations:

1. Debug Tool will look for the source in the location specified at compile time which is in the object code.
2. If not found, Debug Tool will use the path set by the DER_DBG_PATH environment variable on OS/390.
3. If still not found, Debug Tool will use the locations specified by the Distributed Debugger -qremotesource option.
4. If still not found, Debug Tool will request that the Distributed Debugger look in the path set by the DER_DBG_PATH environment variable on the workstation.
5. Finally, if still not found, the Distributed Debugger will look in any paths previously set by you in this debug session through the Source Search Path dialog.

Note: If the source file cannot be located in any of the OS/390 locations, the OS/390 file name is mapped as follows before proceeding with a search on the workstation:

- A sequential data set name such as USER.PROJECT.SRC1.C is mapped to .\user\project\src1.c
- A PDS member name such as USER.PROJECT.CPP(PART1) is mapped to .\user\project\part1.cpp
- A UNIX name is mapped such as /user/Project/Part1.cpp is mapped to .\user\Project\Part1.cpp

If the source file cannot be found on the workstation a Source Filename dialog opens requesting the path name for the source file. The path name you enter is searched following the order described above.

RELATED REFERENCES

“DER_DBG_PATH environment variable” on page 59

Exception levels

The following OS/390 exception levels can be selected:

Test Level	Description
ALL (or blank)	Specifies that the occurrence of an attention interrupt, termination of your program (either normally or through an ABEND), or any program or Language Environment condition of Severity 1 and above causes the Distributed Debugger to gain control.
ERROR	<p>Specifies that only the following conditions cause the Distributed Debugger to gain control.</p> <p>For C/C++:</p> <ul style="list-style-type: none">• An attention interrupt• Program termination• A predefined Language Environment condition of Severity 2 or above• Any C/C++ condition other than SIGUSR1, SIGUSR2, SIGINT or SIGTERM. <p>Language Environment conditions are described in the <i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942.</p>
NONE	Specifies that no condition causes Distributed Debugger to gain control.

RELATED TASKS

“Selecting debugger recognized exceptions” on page 37

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.

RELATED CONCEPTS

“Distributed Debugger and Debug Tool” on page 4

Optional breakpoint parameters

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

Optional breakpoint parameter	Description	Type of breakpoint supported
Threads	This selection list lets you choose what threads to set the breakpoint in. To select a thread ID from the list, highlight the thread where you want to set the breakpoint. This list is available only on platforms that support multithreaded programs.	This parameter is supported by all breakpoint types.
Frequency	<p>Use the Frequency controls to tell the debugger when to stop on a breakpoint and when to skip it. The debugger keeps track of how many times each breakpoint is encountered. The fields in this section tell the debugger on which encounter of a breakpoint the debugger should first stop, how often it should stop, and on which encounter the debugger should no longer stop.</p> <p>The following parameters are used to set the breakpoint frequency:</p> <p>From Enter the first breakpoint encounter you want the debugger to stop on. For example, if you want the debugger to skip over the breakpoint the first five times it is encountered, enter "6".</p> <p>To Enter the last breakpoint encounter you want the debugger to stop on. For example, if you want it to start ignoring the breakpoint after the 20th encounter, enter "20". To have it always stop on the breakpoint, enter "Infinity".</p> <p>Every Enter the frequency with which you want the debugger to stop on this breakpoint. For example, if you want it to stop on only one out of every four it encounters, enter "4".</p>	This parameter is supported by all breakpoint types.

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

RELATED REFERENCES

“C/C++ supported data types” on page 57

“C/C++ supported expression operands” on page 57

“C/C++ supported expression operators” on page 58

Program Profiles

Using program profiles means that the Distributed 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.

When you use program profiles any changes you make to the windows and breakpoints are saved. Program profiles are the default setting for the Distributed Debugger. To delete them you can use the Application Preferences dialog.

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.

Notices

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
Intellectual Property and Licensing
IBM Corporation
North Castle Drive, MD-NC 119
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 DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

be incorporated in new editions of the document. 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 Ave E
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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

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

CICS	IMS
CICS/ESA	Language Environment
CICS/MVS	MVS/ESA
CICS/VSE	OS/390
DB2	S/390
IBM	

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and/or other countries licensed exclusively through X/Open Company Limited.

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP, AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARDS TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Other company, product, and service names may be trademarks or service marks of others.