QLogic PathScale™ Debugger
User Guide

Version 1.1.1

Information furnished in this manual is believed to be accurate and reliable. However, QLogic Corporation assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which may result from its use. QLogic Corporation reserves the right to change product specifications at any time without notice. Applications described in this document for any of these products are for illustrative purposes only. QLogic Corporation makes no representation nor warranty that such applications are suitable for the specified use without further testing or modification. QLogic Corporation assumes no responsibility for any errors that may appear in this document.

No part of this document may be copied nor reproduced by any means, nor translated nor transmitted to any magnetic medium without the express written consent of QLogic Corporation. In accordance with the terms of their valid PathScale agreements, customers are permitted to make electronic and paper copies of this document for their own exclusive use.

Linux is a registered trademark of Linus Torvalds.

QLA, QLogic, SANsurfer, the QLogic logo, PathScale, the PathScale logo, InfiniPath and EKOPath are registered trademarks of QLogic Corporation.

Red Hat and all Red Hat-based trademarks are trademarks or registered trademarks of Red Hat, Inc.

SuSE is a registered trademark of SuSE Linux AG.

All other brand and product names are trademarks or registered trademarks of their respective owners.

Document Revision History	
1.0 Beta	
1.0	
1.1	
1.1.1	
2-092004-07 , July 26, 2007	
Changes	Document Sections Affected
Remove references to qlogic.com	All
Update document number	All sections

Table of Contents

Figures

Tables

Section 1
Introduction

This User Guide describes how to use the QLogic PathScale™ debugger, PathDB. The PathDB debugger allows the developer to examine the state of a program by stopping it at points of interest, examining its state, changing the values of variables, and then resuming the execution of the program. This information can help the developer track down, identify, and fix bugs in the code.

The QLogic PathScale debugger will be referred to as PathDB or the PathScale Debugger in the rest of the document.

1.1
What's Familiar About PathDB?

The PathDB command structure is similar to that of many other debuggers and provides a large subset of the GDB commands.

With the PathDB debugger you can do a variety of things to examine the state of the program. The debugger can:

■ Debug programs in C, C++, and Fortran (77 and 90)
■ Run your program with arbitrary arguments
■ Set breakpoints to stop your program at a specific point
■ Examine variable values during program execution
■ Modify variable values at runtime

1.2
What's Different About PathDB?

The PathDB debugger offers several new and enhanced features that set it apart from other debuggers. They include:

■ Superior support for Fortran types and expressions

■ The ability to step backward through your code with the `rerun` command

■ Multiple process support (`pathdb` has the ability to follow both branches of a fork, for example)

■ Debugging support for Standard Template Libraries (pretty printing and subscripting operations)

■ Easier selection of breakpoint insertion points in C++

NOTE: While the PathDB debugger is likely to be compatible with the output from other compilers, the debugger is only supported for use with the QLogic PathScale Compiler Suite.

The PathDB debugger runs on Linux platforms using the AMD x86 and AMD64® family of chips, and Intel's ® EM64T families of chips (see section 2.1 for information on the supported platforms). You will need a 32-bit environment on a 64-bit machine to use pathdb to debug 32-bit programs.

1.3
How This Guide is Organized

The information in this guide is organized into these sections:

- Section 2 describes the system requirements for the PathDB debugger

- Section 3 explains how to install and run the PathDB debugger, including subscription management for the debugger

- Section 4 follows a sample debugging session

- Section 5 covers the most common commands for debugging

- Section 6 provides tips for using the debugger with different languages

- Section 7 covers advanced debugging

- Section 8 describes the command-line interface for the PathDB debugger

- Section 9 has tips and suggestions for troubleshooting

- Section 10 is a complete listing of the PathDB debugger commands

- Appendix A lists the expressions supported by the PathDB debugger

1.4
Conventions Used in this Document

These conventions are used throughout the PathDB documentation.

Convention	Meaning
command	Fixed-space font is used for literal items such as commands, files, routines, and pathnames.
variable	Italic typeface is used for variable names or concepts being defined.
user input	Bold, fixed-space font is used for literal items the user types in. Output is shown in non-bold, fixed-space font.
$	Indicates a shell command line prompt
#	Shell command line prompt as root
pathdb>	Debugger command line prompt
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
NOTE:	Indicates important information

1.5
Other Resources

The PathDB debugger includes online manual pages ("man pages") and an extensive Help system that describes the commands and options for `pathdb`. Type `man pathdb` to view the man page and the complete list of command line options available for use with the debugger. Basic information on the `pathdb` commands can be found in section 5 and in the online Help. The online Help may be invoked by typing `help` at the pathdb prompt:

```
pathdb> help
```

Please see the release notes with this release and the PathScale web site for further information about current releases and developer support for the PathScale debugger.

http://www.pathscale.com/support.html

Since the PathDB debugger supports many of the GDB debugger commands, you may also want to have a look at these books:

■ *Debugging with GDB: The GNU Source-Level Debugger* by Richard M. Stallman, Roland Pesch, Stan Shebs, et al., Free Software Foundation; 9th edition, 2002, ISBN 1-882114-88-4

■ *GCC: The Complete Reference* by Arthur Griffith, McGraw-Hill Osborne Media, 2002, ISBN 0-072224-05-3

Notes

Installation Prerequisites

The PathDB debugger is designed to work on the same platforms as the QLogic PathScale Compiler Suite.

2.1
System Requirements

The PathScale debugger has been tested on systems based on the AMD64 and EM64T families of chips, with one of the following Linux distributions installed. The supported x86_64-based systems are:

- Red Hat Enterprise Linux (RHEL) 4
- Fedora Core (FC) 3, FC4, FC5
- SUSE Linux Enterprise Server (SLES) 9, SLES 10
- openSUSE 10.0, 10.1
- SUSE Professional 9.3

It has not been tested on systems with less than 512 megabytes of memory. The PathDB debugger can debug either 32-bit or 64-bit code.

2.2
Required Libraries

There are some shared libraries that need to be installed. Both the 32-bit and the 64-bit versions of these libraries must be installed.

This example list comes from the SLES 9 distribution:

- `libncurses.so.5`
- `libstdc++.so.5`
- `libgcc_s.so.1`
- `libc.so.6`
- `libm.so.6`
- `libdl.so.2`

These libraries can be found in your Linux platform distribution.

NOTE: For a Fedora Core 3 installation of the debugger, you will have to manually install the `compat-libstdc++` library. It will not be installed by default.

- `compat-libstdc++.i386` (the 32-bit library)
- `compat-libstdc++.x86_64` (the 64-bit library)

2.3
PathDB Debugger Environment

You can run the PathDB debugger in any environment in which you can run the QLogic PathScale Compiler Suite. The debugger runs on IA32 (32-bit) and Linux x86_64 machines (32-bit/64-bit) machines with a supported Linux operating system. To run the PathDB debugger, you will need:

- A computer based on the x86, AMD64, or EM64T processor architecture
- 256 MB of RAM, 512 MB recommended
- A supported Linux distribution installed. See section 2.1 for more information.

NOTE: You will want to install either the IA32 (32-bit) RPM or the x86_64 (32-bit/64-bit) RPM for the debugger depending on the environment you will be using to debug.See section 3.1 for more information.

See section 4 for an example debugging session and information on how to get started debugging with the PathDB debugger.

2.4
Subscription Management

To run the PathDB debugger, QLogic PathScale subscription management must be installed and correctly configured for your system. See section 3.3 in this guide for information on subscription management for PathDB, and the *QLogic PathScale Compiler Suite and Subscription Manager Install Guide* for more information on subscription management.

Section 3
Installing PathDB

The PathDB debugger can be installed using RPMs or a non-root `tar` file. The RPMs and the non-root `tar` file for the PathDB debugger are distributed as part of the QLogic PathScale Compiler Suite. If you installed the Compiler Suite using an RPM install or the non-root `tar` file install, you automatically installed the PathDB debugger as well (both the 32-bit and 64-bit versions).

3.1
Installing the PathDB RPMs

If you want to install, or re-install the PathDB debugger RPMs, see the information in this section.

To install the debugger alone, you will find the PathDB debugger RPMs for your platform in the directory for that platform:

`/<distribution_dir>/<platform>/rpms`

For example, for the SLES 9 platform, you would find the `pathdb` RPMs here:

`/<distribution_dir>/SuSE9/rpms`

NOTE: There is an RPM for the 32-bit debugger (IA32) and one for the 64-bit debugger (Linux x86_64). Choose the appropriate one for your debugging environment. You can install one or both RPMs.

You must have root privileges to install the PathDB debugger RPMs. If you do not have root privileges, see section 3.2 for information on installing PathDB from a tar file.

Obtain root privileges, then install the `pathdb` RPM (for the 32-bit version) using this command:

`# `**`rpm -Uvh pathscale-pathdb-*i586.rpm`**

For the 64-bit version use this command:

`# `**`rpm -Uvh pathscale-pathdb-*x86_64.rpm`**

To install both the 32-bit and the 64-bit version, use this command:

`# `**`rpm -Uvh pathscale-pathdb*.rpm`**

You will see output similar to the following:

```
# rpm -Uvh pathscale-pathdb-*.rpm
Preparing...################################### [100%]
   1:pathscale-pathdb############################### [100%]
```

> **NOTE:** The system must also have the `pathscale-sub-client` RPM (for subscription management) installed in the same location as the debugger, or you must define the `PATHSCALE_SUBSCRIPTION_CLIENT` environment variable.

To install both the 32-bit and the 64-bit version, use this command:

```
# rpm -Uvh pathscale-pathdb*.rpm
```

> **NOTE:** If you do not have all of the required libraries, you may receive RPM errors while trying to install the debugger. Please contact the source of your Linux distribution to get these missing RPMs. Install them and then reattempt the debugger installation. If you still have problems with the installation, please contact Support at `support@pathscale.com`.

> **NOTE:** IMPORTANT: For a Fedora Core 3 installation of the debugger, you will have to manually install the `compat-libstdc++` library. It will not be installed by default:

- `compat-libstdc++.i386` (the 32-bit library)
- `compat-libstdc++.x86_64` (the 64-bit library)

3.2
Installing PathDB from a tar File

The non-root `tar` file images for the QLogic PathScale compiler environment, including the debugger, are available on the ISO image, in non-root `tar` file for each platform supported by the QLogic PathScale Compiler Suite, and on the physical CD in the `<distribution_dir>/<your_platform>/tar` directory. To unpack the tar file, use the command `tar xfj <filename>`. For verbose output (this will print a great deal of output), use `tar vxfj <filename>`. The tar file will be unpacked into the current directory, so change to the directory you want to install to before you unpack.

For example:

```
$ cd /mycompilers
$ tar xfj <path_to_tarfile>/ \
     pathscale-pathdb-2.3-90.328_suse9.0.tar.bz2
```

The `tar` file will be unpacked into a directory whose name looks like this:

```
pathscale-pathdb-2.3-90.328_suse9.0
```

To run the debugger, set up your shell's PATH so it contains the PATH to wherever you installed the debugger (`<install_directory>`). For instance, if you installed to the `/usr/local` directory, and renamed the directory `pathscale`, put `/usr/local/pathscale/bin` in your PATH. See section 4 in the *QLogic PathScale Compiler Suite and Subscription Manager Install Guide* for more information on setting your PATH and pointing to shared libraries.

> ***NOTE:*** IMPORTANT: For a Fedora Core 3 installation of the debugger, you will
> have to manually install the `compat-libstdc++` library. It will not be
> installed by default:

- `compat-libstdc++.i386` (the 32-bit library)
- `compat-libstdc++.x86_64` (the 64-bit library)

3.3
Subscription Management for PathDB

To use PathDB, QLogic PathScale subscription management must be installed and
correctly configured for your system. If you are already using the QLogic PathScale
Compiler Suite environment, subscription management should be set up. See the
QLogic PathScale Subscription Management User Guide for information on
subscription management, and for more information on environment variables and
configuring the Subscription Manager.

> ***NOTE:*** If you have a nodelocked subscription, the debugger can only be used
> on that system.

When you try to debug a file (either from the command line or using the `file` or
`attach` commands), the debugger will try to find the QLogic PathScale subscription
management `subclient` program and invoke it. The debugger will look for the
`subclient` program in the directory which contains the debugger, or if the
debugger is in a directory ending in `/bin/`, it will look in
`/lib/<compiler_version>`.

The `subclient` program determines what language you are trying to debug and
assumes a lease for a compiler of the appropriate language. For example, if you
are debugging a program where the main was compiled with `pathf90`, `pathdb`
will pretend to be a Fortran compiler.

If the debugger is installed in a different location than the subscription client, you
will need to set the environment variable `PATHSCALE_SUBSCRIPTION_CLIENT`
so the debugger can find the `subclient` program.

To set this environment variable using the `bash` shell, type:

`$` **`export PATHSCALE_SUBSCRIPTION_CLIENT=<path_to_subclient_program>`**

To set it using the `csh` shell, type:

`$` **`setenv PATHSCALE_SUBSCRIPTION_CLIENT <path_to_subclient_program>`**

3.4
Running PathDB

Before you use the PathDB debugger, you should compile your program with the
`-g` option so that debugging information is included with the executable. If you do

not compile your programs with the `-g` option, limited debugging features will be available. See section 9.1 for more information on the `-g` option.

The name of the executable is `pathdb`. To start the program type:

```
$ pathdb
```

You will see output similar to this:

```
PathScale Debugger Version 1.1.1
Copyright (C) 2004,2005 PathScale Inc.
All Rights Reserved
```

See section 4 for an sample debugging session and the `pathdb` man page for the complete list of options available with the debugger.

3.5
Configuring PathDB

You can create a `.pathdbrc` file in which you can add commands to be executed on startup. This file can then be placed in your home directory.

Here is an example of a `.pathdbrc` file:

```
export PATH="/home/pathdb/pathscale:$PATH"
export PATHSCALE_SUBSCRIPTION_CLIENT=\
"/opt/pathscale/lib/2.3/subclient"
```

3.6
Removing PathDB

To uninstall the PathDB debugger RPMs you will need to have root privileges. If you have only one version of the debugger installed, and you want to uninstall it, do the following.

Using a `bash` shell, gain root privileges and type:

```
# rpm -e $(rpm -qa | grep 'pathscale-pathdb-')
```

Using a `csh` shell, gain root privileges and type:

```
# rpm -qa | grep 'pathscale-pathdb$' | xargs -r rpm -ev
```

This will uninstall the `pathdb` RPMs.

NOTE: This process will uninstall all copies of the debugger, even if multiple versions are installed.

You may need to remove the directory where the software was installed (for example `/opt/pathscale/bin/pathdb` or your `<install_directory>`) after removing the RPMs.

To remove a tar file installation, delete the entire install directory where you installed the debugger. For example, if you installed the debugger in `/opt/pathscale/bin/pathdb` type:

```
$ rm -rf /opt/pathscale/bin/pathdb
```

NOTE: The most recent version of PathDB is the best one to use for debugging. There is no reason to maintain earlier versions of the debugger.

Notes

Section 4
Sample Debugging Session

This section uses a sample debugging session to introduce some of the most common debugging commands and provide an overview of the debugging process. Section 5 covers these same commands in more detail, grouping them by functionality, and covering some more advanced topics like debugging threads and examining core dumps. Section 8 describes commands that you can use with `pathdb` from the command line. More specific information on using `pathdb` with different languages is covered in section 6.

Before you begin debugging, be sure that all of your programs have been compiled using the `-g` option, so that debugging information is included as part of your program. See section 9.1 if you have questions about using the `-g` option.

4.1
Getting Started

In this example, we will use a short Fortran program that sums the classic series used to calculate the mathematical constant "e". There is a subtle bug in this code that causes the result to be off by an unacceptable amount.

First we will run the program as is, then look through it using `pathdb`, and finally track down and fix the bug using the debugger.

Here is the code for the program EXP.

```
PROGRAM EXP                                    ! line 2
   INTEGER(KIND=4) :: I = 1
   REAL(KIND=8) :: U = 1.0
   REAL(KIND=8) :: ER, S, T

   ER = 1D-15                                  ! line 7
   S = 1.0D0
   T = 1.0D0

   DO
     T = AN(T,U,I)                             ! line 12
     IF (T.LT.ER) EXIT                         ! line 13

     S = S + T
     I = I + 1                                 ! line 16
   END DO

   T = ABS(S-EXP(U))

   print *, "Number of Steps = ", I
   print *, "Acceptable Error = ", Er
   print *, "Calculated Error = ", T

   CONTAINS
   REAL FUNCTION AN(T,X,N)                      ! line 26
     INTEGER(KIND=4) :: N
     REAL(KIND=8) :: T,X
     AN = T * (X/N)
   END FUNCTION

END PROGRAM                                     ! line 32
```

Compile the program with the -g option so the compiler will generate the debug information:

```
$ pathf90 -o -g EXP EXP.f
```

If we run the program, we see this output.

```
$ ./EXP
Number of Steps = 18
Acceptable Error = 1.00000000000000008E-15
Calculated Error = 6.68010313731315364E-9
```

The calculated error is greater than the acceptable error by several orders of magnitude. To see why this might be the case, start the debugger, specifying the program in the command line.

```
$ pathdb EXP
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc. All Rights Reserved
Reading symbols from /home/EXP...done
```

Using `pathdb`, we will step through the program and see what is going on. The first thing to do is add a breakpoint at `main`. To add this breakpoint, type:

```
pathdb> break main
Note: breakpoint set at the beginning of the FORTRAN program.
Breakpoint 1 at 0x400c03: file /home/EXP.f, line 7.
```

The output tells you that breakpoint 1 has been set in the file `EXP.f` at line 7 and `0x400c03` is the memory address of the breakpoint.

When you run the program, the debugger will stop at this first breakpoint.

```
pathdb> run
Starting program: /home/EXP
Breakpoint 1,  MAIN_ _ () at /home/EXP.f:7
7               ER = 1D-15
```

The program has been stopped just before executing line 7.

If you type `list`, `pathdb` will list the next ten lines of the program, starting from where you are currently paused. Notice the right arrow marking your current line.

```
pathdb> list
>7    ER = 1D-15
 8    S = 1.0D0
 9    T = 1.0D0
 10
 11   DO
 12   T = AN(T,U,I)
 13   IF (T.LT.ER) EXIT
 14
 15   S = S + T
 16   I = I + 1
pathdb>
```

If you type `list` again, `pathdb` will list the next ten lines.

4.2
Looking Through the Program

You can now proceed through the program, from where the debugger paused. Type `next` to go to the next executable line.

```
pathdb> next
8   S = 1.0D0
```

At line 8, the variable `S` is initialized to `1.0` as represented by a 64-bit real.

NOTE: When you press **Enter** without typing a new command, `pathdb` repeats the previous command.

Press **Enter** to proceed to the next executable line.

```
pathdb>
9      T = 1.0D0
pathdb>
12     T = AN(T,U,I)
```

If you wanted to, you could also type `n`, the alias for `next`.

NOTE: The PathDB debugger has a standard set of aliases for common commands, which you can add to or change. For `next`, type `n`; for `step`, type `s`; for `continue`, type `c`; for `breakpoint`, type `break` or `b`. Pressing the Enter key will repeat the last command you gave. See section 8.3 for more information on aliases.

4.2.1
Printing Types and Values

We can print the current values and types of some of the variables in the program.

```
pathdb> print T
$1 = 1
```

The value of `T` is `1`.

```
pathdb> ptype T
type = real(kind=8)
pathdb> print I
$2 = 1
pathdb> ptype I
type = integer(kind=4)
```

Type `next` to pause the program in front of line 13. Then print the value of the expression `T.LT.ER`.

```
pathdb> print T.LT.ER
$3 = .false.
```

Print the type of the expression `T.LT.ER` using `ptype`.

```
pathdb> ptype T.LT.ER
type = logical(kind=1)
```

Unlike most debuggers, `pathdb` has full support for Fortran types and Fortran expressions. This makes debugging in Fortran faster and much more intuitive.

4.2.2
The `rerun` Command

Let's step back through the program using the debugger. The `rerun` command keeps track of all of the commands, state changes, and value changes performed and reruns the program to '*ncommands*' before the current one. The default for `rerun` is 1, meaning it stops one command before the last issued one.

Currently we are at line 13. Using the `rerun` command, we will move back to line 12.

```
pathdb> rerun
Rerun the program? (y or n) y
```

The debugger displays each command up to this point.

```
break main
Note: breakpoint set at the beginning of the FORTRAN program.
Breakpoint 2 at 0x400c1d: file /home/exp.f, line 7.
run
Starting program: /home/EXP
Breakpoint 2, MAIN__ () at /home/exp.f:7
7    ER = 1D-15
next
8    S = 1.0D0
next
9    T = 1.0D0
next
12   T = AN(T,U,I)
print T
$4 = 1
print I
$5 = 1
next
13   IF (T.LT.ER) EXIT
```

Remember we typed `next` and then `print`. Type `list` and you will see that we are still at line 13. Type `rerun` again, type `list`, and now we are back at line 12 and ready to step into a function.

4.2.3
Stepping into Functions

Now type `step` to step into the function `T = AN(T,U,I)`.

```
pathdb> step
AN.in.EXP (t=1, x=1, n=1) at /home/EXP.f:29
29   AN = T * (X/N)
```

4.2.4
Backtrace

To view the function calls that brought you to your current position in the program, type `where`. This command displays, in reverse, the steps made through the program.

```
pathdb> where
=>#0 AN.in.EXP (t=1, x=1, n=1) at /home/EXP.f:29
#1 0x0000000000400c04 in MAIN_ _ ( ) at
    /home/EXP.f:12
#2 0x00000000004010d4 in _ _f90_main ( )
#3 0x00000000004010ac in main ( )
```

The debugger is currently in the function `AN` at line 29. This function was called from `MAIN_ _` on line 12. The lines below that are Fortran library calls made before entering the program block.

From here we can print the expression `T*(X/N)`.

```
pathdb> print T*(X/N)
$4 = 1
```

Let's go back to looking through the program.

```
pathdb> next
30    END FUNCTION
pathdb> next
0x0000000000400c17 in MAIN_ _ ( ) at
  /home/EXP. f:13
13    IF (T.LT.ER) EXIT
```

We've now exited the function and moved back to the beginning of the loop at line 13. You could also type `finish` to take you out of the function. We are now at the line containing the `if` statement.

4.3
Continuing On

We have gone through one iteration of the program's loop and calculated a new value for `T`.

```
pathdb> print T
$5 = 1
```

The first term in the series is `1.0`. This is correct. We'll follow the program through one more loop to check the next term.

```
pathdb> next
15    S = S + T
pathdb>
16    I = I + 1
pathdb>
17  END DO
pathdb>
      T = AN(T,U,I)
pathdb>
      IF (T.LT.ER) EXIT
pathdb> print T
$7 = 0.5
```

The next term in the series is `0.5`. This is correct.

4.3.1

Setting a Conditional Breakpoint

We suspect that our bug occurs near the end of the series, so we would like to skip to the last iteration. To do this we can set a *conditional* breakpoint. A conditional breakpoint differs from a normal breakpoint in that the debugger only stops when a particular condition is met. First we set the breakpoint at line 12.

```
pathdb> break 12
Breakpoint 2 at 0x400c17: file /home/EXP.f, line 12.
```

Notice that this is "Breakpoint 2." Now we condition breakpoint 2 on the value of `I`. We want the breakpoint to become active when `I` equals `18` and the program is about to add the last term of the series.

```
pathdb> condition 2 (I.EQ.18)
```

Now continue through the program by typing `continue`.

```
pathdb> continue
Continuing.
Breakpoint 2, 0x0000000000400c44 in MAIN_ _ ( ) at /home/EXP.f:12
12    T = AN(T,U,I)
```

The program stops when the condition is met.

4.4

Hunting Down the Bug

Now we call the function to examine the next term in the series.

```
pathdb> call AN(T,U,I)
$6 = 1.561920814e-16
```

This is smaller than the acceptable tolerance, and contrary to our suspicions, appears to be correct. Type `next` to proceed through the last iteration.

```
pathdb> next
13  IF (T.LT.ER) EXIT
pathdb> print T
$7 = 1.5619208138030493162e-16
```

Notice that the last digits of the term have changed. This looks like our bug. To examine the type of variable returned by AN, type ptype AN.

```
pathdb> ptype AN
type = real(kind=4) function AN(t, x, n)
real(kind=8) :: t
real(kind=8) :: x
integer(kind=4) :: n
```

While all of the other variables are KIND=8, the function AN does not explicitly declare its type and so it is implicitly declared KIND=4. This causes the returned result to be truncated and makes an accurate summation impossible.

4.5
The Solution to the Bug

To ensure that the value returned by AN has the necessary precision, we explicitly declare the return KIND=8. Here's the original line:

```
REAL FUNCTION AN(T,X,N)          ! line 26
```

We'll change line 26 to read:

```
REAL(KIND=8) FUNCTION AN(T,X,N)  ! line 26
```

Now re-run the program and see what happens.

```
pathdb> ./EXP
Number of Steps = 18
Acceptable Error = 1.00000000000000008E-15
Calculated Error = 4.44089209850062616E-16
Program exited normally.
pathdb>
```

4.6
Finishing Up

If you are running the program from within the debugger, you can type quit to exit the pathdb program. If you are running the program, and stopped at a breakpoint, type continue until the program runs through to the end and completes.

```
pathdb> continue
Program exited normally.
pathdb> quit
$
```

Section 5

PathDB Commands

This section covers the most common commands you will use with PathDB. The commands are grouped by functionality.

5.1
Invoking PathDB

There are various ways of invoking PathDB. The next sections provide three examples of ways to start a debugging session

5.1.1
Debugging a Running Process

If you want to debug a process that is already running, start `pathdb` with this command:

```
$ pathdb <program_name> <process_id>
```

An example of this would be debugging the process. Start `pathdb` by typing:

```
$ pathdb a.out 29911
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc.
All Rights Reserved
Attaching to process 29911.
program is /home/a.out
Reading symbols from /home/a.out...done
#0 0x0000000000400569 in main () at /home/test.c:5
5    while(c);
```

You could also start `pathdb` by attaching to a running process with this command:

```
$ pathdb
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc. All Rights Reserved
pathdb> attach <process_id>
```

Here is an example of using the attach command:

```
pathdb> attach 29911
Attaching to process 29911.
program is /home/a.out
Reading symbols from /home/a.out...done
#0 0x000000000040056c in main () at /home/test.c:5
5     while(c);
```

Attaching to a running process leaves the program running. You may now access the program exactly as if it were started from within `pathdb`. When you quit `pathdb`, it detaches from the process and leaves it running.

5.1.2
Debugging a core File

If your program has crashed and created a corefile, you can debug the corefile using `pathdb`. To do this, use this command:

$ **`pathdb <program_name> core.8546`**

For example, to debug a corefile from a file named `a.out`, type:

```
$ pathdb a.out core.8546
Reading symbols from /a.out...done
Program terminated with signal SIGSEGV, Segmentation violation.
Reading symbols from /opt/pathscale/lib/2.3/ libpscrt.so.1...done
Reading symbols from /lib64/tls/libc.so.6. . .done Reading symbols
from /lib64/ld-linux-x86-64.so.2...done
#0 0x0000000000400569 in main () at /home/test.c:5
5     return *a;
```

Since the program has exited, you cannot continue program execution nor can you modify any variable values. However, you can view the values of all variables immediately before the crash. You can also move up the stack frame to view variables from previous calls. See section 5.5 for details.

5.1.3
Starting a Debugging Session

To start a debugging session using the PathDB debugger, use this command:

$ **`pathdb <filename>`**

For example, type:

```
$ pathdb EXP
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc. All Rights Reserved
Reading symbols from /home/EXP...done
pathdb>
```

5.1.3.1
The `run` Command

After `pathdb` reads in the symbols for the executable, type `run` to run the program.

```
pathdb> run
```

After the first call to the `run` command, the arguments are stored by the debugger so that subsequent calls will reuse those arguments.

For example, to start the command `echo` with the arguments `hello` and `world`, use these commands:

```
$ pathdb /bin/echo
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc.
All Rights Reserved
Reading symbols from /bin/echo...(no symbol information)...done
pathdb> run hello world
Starting program: /bin/echo hello world
hello world
Program exited normally.
pathdb> run
Starting program: /bin/echo hello world
hello world
Program exited normally.
pathdb>
```

NOTE: You can show and modify the arguments passed to the program using the `show args` and `set args` commands.

5.2
Some Common Commands

Here are some commands you will find useful.

NOTE: When you press **Enter** or **Return** without typing a command, `pathdb` repeats the previous command. Type `history` to see a list of your recent commands.

`backtrace [levels]`

> Show the stack backtrace for the specified number of levels (default: all levels). See section 5.5.

`break <location>`

> Set a breakpoint at the specified location. See section 5.6.

`call <expression>`

> Call a function in the program being debugged. Record the value and print it if the type is not void. See section 5.9.2.

`continue [signal-number]`

> Continue execution of the program being debugged. See section 5.6.

`delete <breakpoint-number>`

> Purpose:Delete breakpoints, watchpoints, or displays. See section 5.7.5.

`file <file>`

> Change the file being debugged.

`list [location [,location]]`

> List the source files at the specified locations. See section 5.4.

`next [number]`

> Step the program being debugged by a number of lines, stepping over called functions. See section 5.7.2.

`print [/format] [expression]`

> Print the value of the expression and put result in debugger variable.See section 5.8

`ptype <expression>`

> Print the type of the expression. See section 5.8.2.

`quit`

> Quit the debugger.

`run [arguments]`

> Run the program being debugged. See section 5.1.3.1.

`step [number]`

> Step the program being debugged by a number of lines, stepping into called functions. If no number is specified, then step one line. See section 5.7.1.

Other commands discussed in this section include:

- `finish` (see section 5.7.3)
- `info` (see section 5.7.5)
- `jump` (see section 5.9.4)
- `rerun` (see section 5.7.4)
- `return` (see section 5.9.3)
- `set` (see section 5.9.1)
- `show` (see section 5.10.2)
- `tbreak` (see section 5.6.2)
- `watch` (see section 5.6.3)

See section 10 for a complete listing of the PathDB commands and more details about each one.

5.3
PathDB Debugger Help

The PathDB debugger includes a command-line Help system. Type `help` at the command line to access this information.

```
pathdb> help
This is the Help system for the PathScale Debugger (pathdb). Type
'help' followed by a topic or command name. Help is available
for the following topics:
1. Breakpoints - information on breakpoints
2. Running - how to run the program
3. Locations - how to specify locations used by some commands.
4. Commands - information on all the commands available
5. Formats - formats for the print function
6. Expressions - expression support
7. Parameters - debugger parameters
Note that the first letter is uppercase, to distinguish the topic
from a command of the same name.
```

The Help system includes a description of the commands available for use with `pathdb` and information, arguments, and tips for using each command.

```
pathdb> help set
Help for set
Command syntax: set var value
Set the value of control parameters or program variables.
Sub commands available are:
args -- Set the arguments to be passed to program being debugged.
variable -- Set the value of a program variable.
```

A basic description of the `pathdb` commands is also included in .

5.4
Viewing Source Code

You can track down bugs by stepping through your program and watching as it executes. One way to do this is by setting breakpoints at each of the main functions in your program and examining the data at each of these points.

You can view source code using the `list` command to print the code around the current line:

```
pat hdb> list
>6          ER = 1D-15
 7          S = 1.0D0
 8          T = 1.0D0
 9
10          DO
11          T = AN(T,1D0,I)
12          IF (T.LT.ER) EXIT
13
14          S = S + T
15          I = I + 1
```

The list command can also take a line number as an argument.

NOTE: The list command will "scroll" the output. If you issue the command
multiple times consecutively you will see the next lines of your source
code. To "scroll" upwards, use the command `list -`.

5.5
Stackframes

When a function is called from within your program then the local variables from
the previous function are pushed onto the stack. The region of memory where these
variables are stored are referred to as a *frame* or sometimes a *stackframe*. You can
use stackframe to list the series of calls by which you arrived at your current location,
specifically by using the backtrace command.

The command syntax for backtrace is:

```
pathdb> backtrace [levels]
```

You can specify the number of levels to print, or all levels are printed by default.
The where command is an alias for the backtrace command.

For each frame (starting at main), the debugger prints the value of the program
counter, the name of the function, the function arguments and their values, and
finally the source file and line number. The current frame is printed first and is
numbered #0.

The following is an example of a backtrace command taken from the Fortran
debugging example in section 4:

```
pathdb> backtrace
=>#0  0x0000000000400c10 in MAIN__ () at /home/pathdb/exp.f:12
#1    0x0000000000401134 in __f90_main ()
#2    0x000000000040110c in main ()
```
The current frame is marked with =>. You can use the up and down commands to
change the current frame and view local variables from previous functions.

The `up` command will move the stackframe to the previously called function and make available all of the local variables from that function. For example, to pull out of the function `AN` in order to view local variables in `MAIN_ _`, use these commands:

```
pathdb> up
pathdb> up
0x0000000000401134 in __f90_main ()
pathdb> print S
%4 = 263300036608
```

To move back down into the function `AN`, use the `down` command.

```
pathdb> down
0x0000000000400c10 in MAIN__ () at /home/pathdb/exp.f:12
12              T = AN(T,U,I)
pathdb> print X
```

You may also move up or down a fixed number of levels by giving the levels as an argument or jump to a particular frame using the `frame` command.

```
pathdb> down 2
0x0000000000400c10 in MAIN__ () at /home/pathdb/exp.f:12
12        T = AN(T,U,I)
```

NOTE: If you intend to use stackframes, you should take care not to compile your program with the `-fomit-framepointer` flag. This flag causes the compiler to stop maintaining stackframes, thereby freeing a register for other use. However, it makes movement through the stackframes impossible.

5.6
Setting Breakpoints

Breakpoints allow you to stop the execution of your program at points where you want more information about what is happening. The debugger will stop the process at any point set by you. These points can be set at line numbers, at functions, and at static addresses. You can also set breakpoints in multiple source files and in multiple compilation units (e.g. shared libraries).

When you set a breakpoint, the debugger will stop just before executing the code at that point.

Here are some examples of setting breakpoints.

Table 5-1. Set Breakpoint Example

Command	What it Does
`break 125`	Creates a breakpoint at line number `125` of the current file
`break <function>`	Creates a breakpoint at the first line of the `<function>`

Table 5-1. Set Breakpoint Example (Continued)

Command	What it Does
`break t.c:127`	Creates a breakpoint at line number `127` of the program `t.c`
`break * 0x504030`	Creates a breakpoint at the address `*0x504030` within the program

If you type `break` without any arguments, `pathdb` will list all of the currently set breakpoints. Type `continue` to restart the execution of your code.

pathdb> **continue**

5.6.1
Setting Conditional Breakpoints

You can set a conditional breakpoint where the debugger will stop only when is a certain condition is present.

Set a conditional breakpoint by typing:

pathdb> **break ... if <condition>**

This will evaluate the expression `<condition>` each time the breakpoint is reached in the program. If the condition evaluates to `true` then execution is stopped.

You can also set a condition on an existing breakpoint with the `condition` command. For example, to set a condition on `breakpoint 3`, you would type:

pathdb> **condition 3 (a<2)**

5.6.2
Setting Temporary Breakpoints

You can also set a breakpoint that will be removed after the first time it is hit. Set a temporary breakpoint by typing:

pathdb> **tbreak <location>**

You can also "advance" to a particular location. Type:

pathdb> **tbreak <location>; continue**

For example, to continue until line 17 of the file `foo.f` type:

pathdb> **advance foo.f:17**

5.6.3
Setting Watchpoints

Watchpoints can be used to monitor changes in variable's value. The syntax is:

pathdb> **watch <expression>**

There are two kinds of watchpoints: software watchpoints and hardware watchpoints. To set a software watchpoint, use the `watch` command with an expression.

For example:

```
pathdb> watch a*b
```

This expression will be evaluated at every step, and the debugger will stop execution when the value changes. Since this requires the expression to be evaluated, the watchpoint will be removed when the terms in the expression become invalid (for example if the expression uses local variables and control leaves the function).

To set a watchpoint at a particular address in memory, you can use a hardware watchpoint. Hardware watchpoint have the advantage of being much faster than software watchpoints. Since they only specify an address in memory, they have no scope and are never removed.

For example, to stop execution when the value of the data at the address `0xffffe3` is modified, use the command:

```
pathdb> watch *0xffffe3
```

5.7
Resuming Program Execution

5.7.1
The `step` Command

Type `step` to step into a function.

```
pathdb> step
```

5.7.2
The `next` Command

Type `next` to go to the next executable line.

```
pathdb> next
```

5.7.3
The `finish` Command

The `finish` command continues the execution of your program until the current function returns to its caller.

```
pathdb> finish
```

The value returned by the function (if it is not void) will be printed and saved to the value history.

See the following commands for more information:

- **`run`** Run the program being debugged.

- **`return`** Return from the current function with the value specified.

- **`continue`** Continue execution of the program being debugged.

5.7.4
The **`rerun`** Command - Step Backwards

Looking through code you may want to step backwards, or replay the commands you have issued. The `rerun` command reruns the program in the debugger, issuing all the commands typed up until `number` (of commands) before the last one. The default for `number` is 1, meaning that the program will run to the point one command before the last command you issued.

This is an effective 'step backwards' through your program. The command syntax for `rerun` is:

```
pathdb> rerun [number]
```

You can change the 'step backwards' with the *number* argument.

```
pathdb> rerun 4
```

This will return you to a point in your program four commands before the last command.

5.7.5
The **`delete`** Command

The `delete` command removes a breakpoint, watchpoint, or display from the list of active items. The command takes the item number as an argument, or removes all breakpoints if no argument is given. The item number is first displayed when it is initially set with the `break`, `watch`, or `display` command. You can view existing items using the `info` command.

For example, the command `info break` shows all existing breakpoints with their ID number in the left-most column.

```
pathdb> info break
Num Type       Disp Enb Address              What
1 breakpoint keep y   0x00000000004006a6 in main at
   /home/examples/types.cpp: 24
      breakpoint already hit 1 time
2 breakpoint keep y   0x000000000040076e in Planet: :get_orbit()
   at /home/examples/types.cpp: 6
3 breakpoint keep y   0x000000000040076e in Planet: :get_orbit()
   at /home/examples/types.cpp: 6
```

Now delete the third breakpoint using `delete` and the breakpoint number.

```
pathdb> delete 3
```

Use `info break` to see the list of breakpoints currently set in the program.

```
pathdb> info break
Num Type         Disp Enb Address            What
1   breakpoint keep y    0x00000000004006a6 in main at
     /home/examples/types .cpp: 24
          breakpoint already hit 1 time
2   breakpointkeep y     0x000000000040076e in Planet::get_orbit()
     at /home/examples/types .cpp: 6
```

5.8
Printing Program Variables

5.8.1
Print Values

When the program execution has stopped at a particular point, you can view the current values of the variable using the `print` command:

```
pathdb> print <expression>
```

For example:

```
pathdb> print T
$3 = 1.561920814e-16
```

NOTE: The condition on the breakpoint will be removed if you exit the current function and the expression becomes invalid.

5.8.2
Expressions

The PathDB debugger has extensive support for expressions in both C (and C++) and Fortran 90. The expression parser is context sensitive and thus depends on which language the current function is written in.

Both C/C++ and Fortran expressions support type casts. The syntax of the type is language dependent. The following are examples of type casts in C and Fortran:

```
C:
print (const int *)foo
Fortran:
print (integer(kind=8), pointer)bar
```

To print the value of an expression use `ptype`:

```
pathdb> ptype T type = real (kind=8)
```

See Appendix A for more discussion of expressions.

5.9
Modifying Program Execution

One way to track down a bug in code is to modify the values of data as the code executes. The PathDB debugger provides several ways to do this.

5.9.1
Using the `set` Command

Using the `set` command you can set value of control parameters or program variables in the program you are debugging.

NOTE: If the `set` command is used to set the value of an unknown parameter, it will try to set the value of a variable in the program you are debugging. See section 5.9.4 and section 5.10.3 for information on setting variables in a program.

Examples of setting parameter values:

```
set elements 300 set
confirm off
set prompt (pathdb)
```

5.9.2
The `call` Command

The `call` command is used to call a function in the program being debugged, record the value in the value history, and print it if the type is not void.

The `call` command has the syntax:

```
pathdb> call <expression>
```

This command will cause the program to execute the given expression. Typically, the expression will be a function call. The program being debugged executes the expression and if the result is non-void, the debugger prints the result and inserts it into the value history.

The number of arguments passed to the function must match those defined for the function. If the function being called is one of a set of C++ overloads, the types of the arguments will be used to determine which overload to call.

5.9.3
The `return` Command

The `return` command causes the current function to return, possibly with a value. The value is specified by the expression argument and must match the return type of the function.

The command syntax for `return` is:

```
pathdb> return <expression>
```

See the following commands for more information:

- **`finish`** Continue execution until the current function returns

5.9.4
The `jump` Command

The `jump` command allows you to jump to a specific location in your program, skipping over the code between the specified location and the current location. The location may be specified as either a line number or as an address within the program.

For example, to jump to the program address `0xca6db`, use this command:

```
pathdb> jump *0xca6db
```

5.10
User Configurable Parameters

5.10.1
Controlling Parameters

Parameters are variables that modify the behavior of the debugger. They are modified using the `set` command and displayed using the `show` command. The debugger has default parameters that are set initially. The `show` command will display the current parameters for `pathdb`.

See section 5.10.3 for more discussion of the parameters `pathdb` supports.

5.10.2
Using the `show` Command

Use the `show` command to show the current value of control parameters. If `show` is invoked with no arguments, the values for all parameters are listed. The `show` command can also be invoked with the name of a particular parameter. Some examples of `show` commands are:

```
pathdb> show
annotate: Current annotation level is 0
args: Argument list passed to program is ""
can-use-hw-watchpoints: Support for hardware watchpoints is
enabled
confirm: Confirmation of dangerous commands is enabled
endian: Endianness of the host platform is auto (currently little
endian)
follow-fork-mode: Process to follow after a fork event is parent
height: Height of the display screen is 24
history filename: The file to save command history is
"~/.pathdb_history" ...
pathdb> show print elements
Number of array elements to print is 100
pathdb> show confirm
confirm: Confirmation of dangerous commands is enabled
pathdb> show prompt
Prompt used for command interface is "pathdb> "
```

5.10.3
Supported Parameters

Most parameters that the PathDB debugger supports are similar to the parameters of the same name used by GDB. There are a few new ones as well They are given in the following table.

Table 5-2. Supported Parameters

Parameter	Definition	Default
annotate	Set annotation level.	0
args	Set list of arguments to be passed to program.	0
can-use-hw-watchpoints	Support for hardware watchpoints.	on
confirm	Confirmation of dangerous commands.	on
endian	Set the endianness of the host platform.	auto
follow-fork-mode	Process to follow after a fork event. This parameter can take additional value of "both," causing the debugger to follow both parent and child.	parent
height	Height of the display screen in lines.	42

Table 5-2. Supported Parameters (Continued)

Parameter	Definition	Default
`history filename`	Filename to save command history.	`~/.pathdb_history`
`history save`	Saving history to file on exit.	`on`
`history size`	Number of commands stored in history.	`1000`
`language`	Current program source language.	`auto`
`listsize`	Number of lines to display.	`10`
`multi-process`	Support for multiprocessing. When on this parameter is on, the debugger will track multiple processes.	`off`
`pagination`	Paging of printed output.	`on`
`print address`	Printing address of reference type.	`on`
`print array`	Pretty print of arrays.	`on`
`print elements`	Number of array elements to print.	`100`
`print null-stop`	Printing char arrays as strings.	`off`
`print object`	Printing of dynamically typed objects.	`on`
`print pretty`	Pretty print of structures.	`on`
`print repeats`	Number of printed repeats allowed.	`10`
`print sevenbit-strings`	Printing 8-bit characters in octal.	`on`
`print sigdigits`	Number of significant digits to use.	`10`
`print simplify-template`	Printing simplified templates. When on, standard templates are simplified for ease of reading.	`on`
`print static-members`	Printing of C++ static members.	`on`
`print std-list`	Pretty printing of C++ STL lists. When on, the contents of `std::list` objects are printed.	`on`

Table 5-2. Supported Parameters (Continued)

Parameter	Definition	Default
`print std-map`	Pretty printing of C++ STL maps. When on, the contents of `std::map` objects are printed.	`on`
`print std-string`	Pretty printing of C++ STL strings. When on, the contents of `std::string` objects are printed.	`on`
`print std-vector`	Pretty print of C++ STL vectors. When on, the contents of `std::vector` objects are printed.	`on`
`print union`	Printing of union members.	`on`
`prompt`	Prompt used for command interface by `pathdb`.	`pathdb>`
`shell-mode`	Executing unknown commands with shell. With the default behavior, any unrecognized commands are sent to the shell. When on, all unknown commands are passed directly to the shell.	`off`
`std-step`	Stepping over C++ STL operators.	`on`
`stop-on-solib-events`	Catch shared library events, eg. loading and unloading.	`0 or off`
`verbose`	Verbose output.	`off`
`width`	Width of the display screen.	`screen-width`

Section 6
Language-Specific Debugging

The PathDB debugger can be used to debug programs in C, C++, or Fortran. This section contains some examples and some tips for debugging in specific languages.

6.1
Debugging C/C++

In C, all operators are atomic, so you can't step into them. With an atomic function, `next` and `step` behave the same. In C++ you can `step` into operators, functions, constructors, and destructors with pathdb in C++ programs. You can use `catch` to catch exceptions being thrown.

Structures and arrays are automatically displayed with pretty print. Variables with STL types are also displayed with pretty print.

See section 6.1.3 for more information.

You can set breakpoints at functions (with very long names) and use tab completion to complete the function name for you.

6.1.1
Navigating Classes

Here is an example program in C++, followed by a debugging session. First the program (`types.cpp`):

```
#include <stdlib.h>
#include <stdio.h>
class Planet {
public:
   int get_orbit() {return 85;}
   virtual void spin() {printf("fast\n");}
};
class Venus : public Planet {
public:
   int get_orbit() {return 27;}
   virtual void spin() { printf ("medium\n");}
};
class Mars : public Planet {
public: int get_orbit() {return 800;}
   virtual void spin() {printf("slow\n");}
};
int main()
{
   Planet moon;
   Mars redPlanet;
   Venus morningStar;
   moon.get_orbit();
   redPlanet .get_orbit();
   morningStar.get_orbit();
   moon.spin();
   redPlanet. spin();
   return 0;
}
```

And here is that same program being debugged:

```
$ pathdb a.out
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc. All Rights Reserved
Reading symbols from /home/a.out...done
pathdb> break main
Breakpoint 1 at 0x40075a: file /home/test.cc, line 24.
pathdb> run
Starting program: /home/a.out
Breakpoint 1, main () at /home/test.cc:26
24   Planet moon;
pathdb> next
25   Mars redPlanet();
```

The `ptype` command will show all class information.

```
pathdb> ptype moon
type = class Planet {
  public:
  void ** _vptr.Planet;
  int get_orbit();
  void spin();
}
```

Member functions can be called using either the call or the `print` commands:

```
pathdb> print moon.get_orbit()
$1 = 85
```

The `ptype` and `print` commands will also show derived classes, with their inherited members.

```
pathdb> print redPlanet
$2 = {
  Planet = {
    _vptr.Planet = NULL }
  }
```

The expression handler has full support for derived members, including making function calls.

```
pathdb> print redPlanet.get_orbit ()
$2 = 800
```

Skip down through the next lines of the example.

```
pathdb> next
26    Venus morningStar;
pathdb> next
28    moon.get_orbit();
```

The `ptype` command shows virtual functions.

```
pathdb> ptype morningStar
type = class Venus : public Planet {
  public:
    int get_orbit();
    void spin();
}
```

The expression handler understands virtual functions.

```
pathdb> print moon.get_orbit()

$3 = 85
```

Skip down a bit over calls.

```
pathdb> next
29     redPlanet.get_orbit();
```

6.1.2
C++ Basenames

If the function name given to the command is ambiguous, a menu of alternative functions will be presented. This is particularly useful for C++ member functions.

6.1.3
Standard Template Libraries

The PathDB debugger understands standard template libraries (STL) for lists, vectors, arrays, and maps. The `pathdb` output uses pretty print to display these types in an understandable form.

We can take this program (`test.cpp`) written in C++ and compile it using `pathCC -g`.

```
#include <stdio.h>
#include <string>
#include <vector>
#include <map>
#include <list>
using namespace std;
int main() {
  string name ;
  name = "foo" ;
  list<int> l ;
  l.push_back (1000) ;
  l.push_back (2000) ;
  l.push_back (3000) ;
  vector<int> vec ;
  vec.push_back (1) ;
  vec.push_back (2) ;
  vec.push_back (3) ;
  map<string, int> m ;
  m[ "dave"] = 40 ;
  m[ "sandra"] = 38 ;
  m[ "elliott"] = 6 ;
  m[ "lucy"] = 3 ;
  return 0;
}
```

When we run this program in `pathdb`, lists, vectors, maps, and string names are displayed in a format that is clearer and easier to understand.

```
$ pathdb test.cpp
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc. All Rights Reserved
Reading symbols from /home/a.out. . .done
pathdb> break main
Breakpoint 1 at 0x400bdd: file /home/test.cpp, line 9.
pathdb> run
Starting program: /home/a.out
Breakpoint 1, main () at /home/test.cpp:9
pathdb> print l
$1 = {1000, 2000, 3000}
pathdb> print vec
$2 = {1, 2, 3}
pathdb> print m
$3 = {"dave" = 40, "sandra" = 38, "elliott" = 6, "lucy" = 3}
pathdb> print name
$4 = "foo"
```

You can use the following parameters to control these options:

■ **print std-list** Pretty printing of C++ STL lists is enabled
■ **print std-map** Pretty printing of C++ STL maps is enabled
■ **print std-string** Pretty printing of C++ STL strings is enabled
■ **print std-vector** Pretty printing of C++ STL vectors is enabled

6.1.4
Using **step** with STL

When debugging C++ code, stepping through the code for the Standard Template Library can be time consuming. You might not want to debug this code, only the original code you have written. The PathDB debugger includes an option (`step-std`) that enables you to step over the STL code when you are debugging your program. The option set to on by default, but can also set to `off`, so that you can debug STL code.

For this example we'll use the program `stl.cpp`:

```
#include <stdio.h>
#include <stdlib.h>
#include <list>
using namespace std;
void foo(int a)
{
   printf ("Number: %d\n", a);
}
int main()
{
   vector<int> a;
   a.push_back(1);

   foo (a[0]);
}
```

Here is example output using with the default setting of `step-std=on`, followed by output showing the results from `setting step-std=off`.

```
$ pathdb a.out
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc.
All Rights Reserved
Reading symbols from /home/a.out...done
pathdb> break main
Breakpoint 1 at 0x40095a: file /home/test.cc, line 13.
pathdb> run
Starting program: /home/a.out
Breakpoint 1, main () at /home/test.cc:13
13     vector<int> a;
pathdb> next
14     a.push_back(1);
pathdb> next
15     foo(a[0]);
pathdb> step
foo (a=1) at /home/test.cc:6
6      printf ("hello");
```

Now let's run the program again, this time setting `step-std` to `off`. This time the debugger will step into the STL code.

```
pathdb> run
Starting program: /home/a.out
Breakpoint 1, main () at /home/test.cc:13
13  vector<int> a;
pathdb> set step-std 0
pathdb> next
14  a.push_back(1);
pathdb>
15  foo(a[0]);
pathdb> step
std::vector<int, std: :allocator<int> >: :operator[ ]
   (this=0x7fbfffedb0, __n=0) at /opt/pathscale/lib/gcc-lib/
   x86_64-pathscale-linux/3.3.1/include
   /c++/bits/stl_vector.h: 501
501 operator[ ] (size_type __n)
   { return *(begin() + __n); }
pathdb> step
std::vector<int, std: :allocator<int> >: :begin
   (this=0x7fbfffedb0) at /opt/pathscale/lib/gcc-lib/
   x86_64-pathscale-linux/3.3.1/include
   /c++/bits/stl_vector.h:355
355 begin() { return iterator (_M_start); }
pathdb> step
__gnu_cxx: :__normal_iterator<int*, std: :vector<int,
   std: :allocator<int> > >: :__normal_iterator
   (this=0x7fbfffecd0, __i=@0x7fbfffedb0) at
   /opt/pathscale/lib/gcc-lib/x86_64-pathscale-linux
   /3.3.1/include/c++/bits/stl_iterator.h:593
593 __normal_iterator (const _Iterator& __i)
   : _M_current(__i) { }
pathdb> step
std::iterator<std: :random_access_iterator_tag, int,
   long, int*, int&>::iterator (this=0x7fbfffecd0) at
   /opt/pathscale/lib/gcc-lib/x86_64-pathscale-linux/3.3.1
   /include /c++/bits/stl_iterator.h: 593
593 __normal_iterator (const _Iterator& __i)
   : _M_current(__i) { }
pathdb> step
__gnu_cxx: :__normal_iterator<int*, std: :vector<int,
   std: :allocator<int> > >: :operator+ (this=0x7fbfffed40,
   __n=@0x7fbfffed38) at /opt/pathscale/lib/gcc-lib/
   x86_64-pathscale-linux/3.3.1/include
   /c++/bits/stl_iterator.h: 631
631 { return __normal_iterator(_M_current + __n); }
pathdb> step
__gnu_cxx: :__normal_iterator<int*, std::vector<int,
   std: :allocator<int> > >: :__normal_iterator
```

```
        (this=0x7fbfffecd0, __i=@0x7fbfffece0) at
        /opt/pathscale/lib/gcc-lib/x86_64-pathscale-linux
        /3.3.1/include /c++/bits/stl_iterator.h:593
593 __normal_iterator (const _Iterator& __i)
    : _M_current(__i) { }
pathdb> step
std::iterator<std: :random_access_iterator_tag, int,
    long, int*,int&>::iterator (this=0x7fbfffecd0) at
/opt/pathscale/lib/gcc-lib/x86_64-pathscale-linux /
    3.3.1/include/c++/bits/stl_iterator.h:593
593 __normal_iterator (const _Iterator& __i)
    : _M_current(__i) { }
pathdb> step
__gnu_cxx: :__normal_iterator<int*, std: :vector<int,
    std::allocator<int> > >::operator* (this=0x7fbfffed10) at
    /opt/pathscale/lib/gcc-lib/x86_64-pathscale-linux/3.3.1/
    include /c++/bits/stl_iterator.h: 602
602 operator* () const { return *_M_current; }
pathdb> step
foo (a=1) at /home/pathdb_clean/other/test.cc:6
6 printf ("hello");
```

6.2
Debugging Fortran

Section 4 is a sample debugging session in Fortran. When you set your first breakpoint to main, it is automatically set to stop at MAIN_ _.

```
pathdb> break main
Note: breakpoint set at the beginning of the FORTRAN program.
Breakpoint 1 at 0x400c03: file /home/EXP.f, line 7.
```

You can also set a breakpoint at a subroutine of a module in Fortran.

```
pathdb> break func.in.moda
```

Arrays and structures are printed in a readable form and all Fortran types are recognized.

```
pathdb> ptype T.LT.ER
type = logical(kind=1)
```

See the debugging example in section 4 for more details.

6.2.1
Expressions in Fortran

Most Fortran expressions are supported, and there is support for some Fortran intrinsics. This is the list of currently supported intrinsics:

- KIND
- LEN

- SIZE
- ALLOCATED
- ASSOCIATED
- ADDR

See appendix A.2 for more information about expressions in Fortran.

6.2.2
Calling Member Functions in Fortran

You can call member functions in Fortran with the `call` command.

```
pathdb> call AN(T,U,I)
$1 = 1.561920814e-16
```

See the debugging example in section 4 for more examples.

Notes

Section 7
Advanced Debugging

The PathDB debugger allows you to do some advanced debugging. With `pathdb` you can debug multithreaded programs and look though stackframes.

7.1
Debugging Multi-threaded Programs

There are two main commands that deal with debugging threads; `thread` and `info threads` .The `thread` command allows you to change threads and the `info threads` command tells you what threads currently exist.

The syntax for `thread` is:

`pathdb>` **`thread <thread-number>`**

This will switch to another thread (`thread-number`), thus setting the current thread.

The syntax for `info threads` is:

`pathdb>` **`info threads`**

The currently existing threads will be listed. Use these reference numbers to change threads with the `thread` command.

7.2
Debugging Multi-process Programs

The PathDB debugger has the ability to follow a fork in a program's process. When it forks into two processes, the debugger can follow the parent, the child, or both processes. By default, `pathdb` will stay with the parent process. The child fork is handled as a separate process. You can use the `process condition` command to attach to the child fork.

The syntax for the `fork` command is:

`pathdb>` **`set follow-fork-mode [parent|child|ask|both]`**

Forks will occur mainly with C and C++ programs. Debugging OpenMP code is not yet supported.

7.3
Handling Signals

Three commands deal with handling signals; `handle`, `info signals`, and `continue [signal-number]`.

7.3.1
The `handle` Command

Syntax for handle:

`handle <signal|all> <action>`

Action can include `[no]pass|[no]stop|[no]print|[no]ignore...`

The `handle` command specifies what to do with signals when they are raised by the program being debugged. When signals are received by the program being debugged, they are passed up to the debugger. How the debugger responds is configurable.

For each signal, the choice is:

1. Print the reception of the signal to the screen (`print`)

2. Pass the signal back to the program being debugged (`pass`)

3. Stop the debugger and provide a prompt (`stop`)

Each signal has a name. See section 7.3.2 for information on how to find the name of a signal. This name should be used to identify to which signal the command will apply to.

You can also use a signal number or the text `all` to specify all signals. You can then specify what to do with the signal using the options. More than one option can be used on the same command.

For example:

```
pathdb> handle SIGINT pass noprint
Signal    Stop    Print   Pass to program    Description
SIGINT    Yes     No      Yes                Interrupt
```

7.3.2
The `info signals` Command

The syntax for `info signals` is:

```
pathdb> info signals
```

This command will show the current signal handling settings. Here is a sample listing from `info signals`:

```
pathdb> info signals
Signal        Stop    Print   Pass    Description
SIGHUP        Yes     Yes     Yes     Hangup
SIGINT        Yes     Yes     No      Interrupt
SIGQUIT       Yes     Yes     Yes     Quit
SIGILL        Yes     Yes     Yes     Illegal instruction
SIGTRAP       Yes     Yes     No      Trace trap
SIGABRT       Yes     Yes     Yes     Abort
```

```
SIGBUS       Yes    Yes    Yes    BUS error
SIGFPE       Yes    Yes    Yes    Floating-point exception
SIGKILL      Yes    Yes    Yes    Kill, unblockable
SIGUSR1      Yes    Yes    Yes    User-defined signal 1
SIGSEGV      Yes    Yes    Yes    Segmentation violation
SIGUSR2      Yes    Yes    Yes    User-defined signal 2
SIGPIPE      Yes    Yes    Yes    Broken pipe
SIGALRM      No     No     Yes    Alarm clock
SIGTERM      Yes    Yes    Yes    Termination
SIGSTKFLT    Yes    Yes    Yes    Stack fault
SIGCHLD      No     No     Yes    Child status has changed
SIGCONT      Yes    Yes    Yes    Continue
SIGSTOP      Yes    Yes    Yes    Stop, unblockable
SIGTSTP      Yes    Yes    Yes    Keyboard stop
SIGTTIN      Yes    Yes    Yes    Background read from tty
SIGTTOU      Yes    Yes    Yes    Background write to tty
SIGURG       No     No     Yes    Urgent condition on socket
SIGXCPU      Yes    Yes    Yes    CPU limit exceeded
SIGXFSZ      Yes    Yes    Yes    File size limit exceeded
SIGVTALRM    No     No     Yes    Virtual alarm clock
SIGPROF      No     No     Yes    Profiling alarm clock
SIGWINCH     No     No     Yes    Window size change
SIGIO        No     No     Yes    I/O now possible
SIGPWR       Yes    Yes    Yes    Power failure restart
```

7.3.3
The `continue` Command

After you have set the debugger to handle signals, you may want to continue your program with a particular signal number. The syntax for using `continue` with signals is:

pathdb> **continue [*signal-number*]**

Notes

Command Interface

This section covers using the command line with `pathdb`. Using the command line with `pathdb` can simplify the debugging process.

8.1
PathDB Defaults

The PathDB debugger is initially set to default parameters for debugging. See section 5.10.1 for the list of default parameters. You can change these defaults by creating a `.pathdbrc` file. This file can include any commands to be executed when `pathdb` starts up and/or changes to the `pathdb` defaults. See section 3.5 for an example `.pathdbrc` file.

8.2
Use of the Command Line

There are a variety of ways you can use the command line to speed up and simplify your debugging. This section covers a few of these features.

8.2.1
Tab Completion

The PathDB debugger has context-sensitive tab completion. If you are typing a file name after a `run` command, it will display the list of possible file names. If you are typing a variable name after a `print` command, `pathdb` will display the possible variable names. And if you are typing the name for a symbol or function after typing `break`, `pathdb` will display the list of possible function or variable names.

From the command line, if you type `run`, begin to type the file name and then press the tab key, `pathdb` will complete the file name for you, if there is one in its current scope matching what you have typed. If you press the tab key again, `pathdb` will display a list of all of the possible file names. See table 8-1 for other tab completion options.

Table 8-1. Tab Completion Options

Command	Possible Completions
run	file names within `pathdb` scope
print	variable names within scope
break	symbols and functions within scope

History

If you type `history` at the command line, pathdb will display a list of the commands that you have typed.

```
pathdb> history
1 break main
2 run
3 next
4 ptype moon
5 print moon.get_orbit()
6 print redPlanet
7 history
pathdb>
```

Menus

The PathDB debugger will present you with a menu of options when there are duplicate possibilities for completion. For instance, if you have a number of C++ methods with the same or very similar names and you want to set a breakpoint at one of them, type `break` and part of the method name.

```
pathdb> break evaluate
Choose one of the following functions:
0. cancel
1. CallExpression::evaluate
2. ArrayExpression::evaluate
a. all
```

Choose the method you want to break at from the list.

Defining Commands

From the command line you can define a new command, usually an aggregate of other commands, that you use often. The syntax for doing this is:

```
pathdb> define <command>
```

For instance, you might have a common group of commands you use often, such as `break main`, `run`, and `next`; so you might define `foo` like this:

```
pathdb> define foo
Type commands for definition of "foo". End with a line saying just
"end".
>break main
>run
>next
>end
```

You might want to document the `foo` command to remember what it does:

```
pathdb> document foo
Type documentation for "foo". End with a line saying just "end".
>Use foo to set up debugging.
>end
```

Here is an example. The test program looks like this:

```
1
2
3 int main()
4 {
5    int a = 1;
6    int b = 2;
7    int c = 3;
8    return 0;
9 }
```

Here is the example:

```
$ pathdb
PathScale Debugger, Version 1.1.1
Copyright (C) 2004-2005 Pathscale Inc.
All Rights Reserved
pathdb> file a.out
Reading symbols from /home/a.out...done
pathdb> break 8
Breakpoint 1 at 0x400632: file /home/test.c, line 8.
pathdb> commands 1
Type commands for when breakpoint 1 is hit,
one per line. End with a line saying just "end".
>print a
>print b
>print c
> end
pathdb> run
Starting program: /home/a.out
Breakpoint 1, 0x0000000000400632 in main () at /home/test.c:8
8 return 0;
$1 = 1
$2 = 2
$3 = 3
```

8.3
Setting Aliases

The PathDB debugger has a set of pre-defined aliases for common commands. Typing `break` (and longer commands like `continue`) may get tiresome after awhile, so the aliases come in handy.

Use the alias for `break` by typing:

```
pathdb> b main
```

The pre-defined aliases for common commands are:

```
b      break
bt     backtrace
c      continue
core   target core
d      delete
exit   quit
f      frame
i      info
n      next
ni     nexti
p      print
r      run
s      step
si     stepi
where  backtrace
```

You can type either the full command, or its (usually shorter) alias; `next` or `n`, `continue` or `c`. You can add to or modify these aliases by typing:

```
pathdb> alias <new_alias> <existing_alias>
```

For example, <*new_alias*> will become the new alias for <*existing_alias*>. If the new alias name might be ambiguous, `pathdb` with issue a warning.

8.3.1
Ambiguous Commands

If you type any command that might be ambiguous (have more than one possible meaning) `pathdb` will issue a warning. For example,

```
pathdb> co
Ambiguous command "co": continue, condition, commands.
```

The `co` command could mean `continue`, `condition`, or `commands`. You might create an alias for one or more of these commands if you use them frequently. Here's another example of an ambiguous command:

```
pathdb> st
Ambiguous command "st": step, stepi, stop.
```

8.4
Using PathDB with emacs

With this release of the PathDB debugger, `emacs` support is now available. You can run the debugger and step through files using an `emacs` interface.

8.4.1
Installing GUD for Use With emacs

With the latest release, the PathDB debugger now supports the GUD interface to `emacs`. This will handle all key bindings, command completion, and allow you to edit commands from the `emacs` window. This is the same interface that `gdb` uses to interact with `emacs`.

To use `emacs` with the PathDB debugger, you will need to load the routines distributed with the QLogic PathScale Compiler Suite into your personal `emacs` configuration.

To do this, add these two lines to your configuration file (`~/.emacs`):

```
;; Add support for pathdb debugging
(load-file "/opt/pathscale/share/pathdb/emacs.el")
```

8.4.2
Starting Up the `emacs` Interface

1. Start an emacs session.

   $ **emacs**
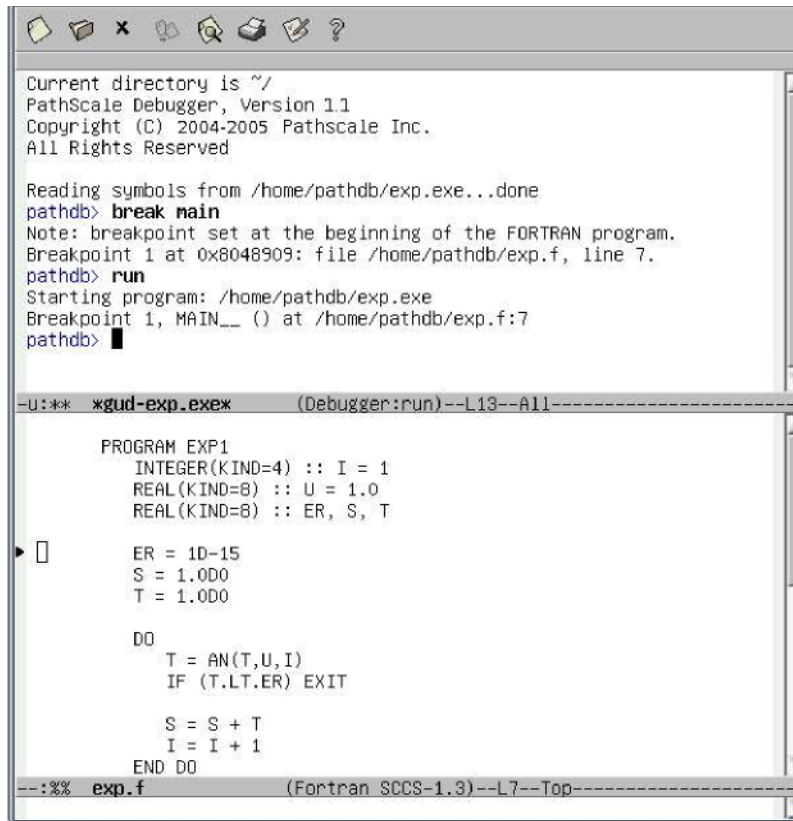
2. Start the PathScale debugger from within `emacs`.

   $ **Meta-x pathdb**

3. Type the name of the program you want to debug. Run `pathdb` (like this):
   pathdb myprogram f

When you run the debugger from within `emacs`, you will see a window similar to Figure 8.1, which shows `pathdb` running under `emacs`.

Figure 8-1. PathDB with GUD emacs

> *NOTE:* The previous interface is still available, but it will require a different command to use.

The previous interface is a "proper" terminal, meaning that you can interact with it as you would a terminal (for example using command history). The disadvantages might be that it is unfamiliar and that you cannot copy and paste source code within the GUI.

To use that interface, type:

```
$ Meta-X pathdb-term
```

8.4.3
Quitting the `emacs` interface

To exit from `emacs` while using the debugger, quit pathdb first and then exit `emacs` the way you normally would.

```
pathdb> quit
$ Ctrl-x Ctrl-c
```

Section 9

Troubleshooting

This section covers specific examples of issues you may encounter while using the PathDB debugger.

NOTE: Contact support at `support@pathscale.com`. Also see the PathScale web site for the latest information on releases.

www.pathscale.com/support.html

9.1
Debugger Problems

9.1.1
Problem: Debugging information incomplete

If you compile your code without the `-g` option (or only compile part of your program with the `-g` option), the code will not have all the debugging information necessary for `pathdb` to work correctly.

To check to see if your file has been compiled with the `-g` option, type this command:

```
$ readelf -w [filename] | grep debug_info
```

Simple example output might look like this:

```
$ readelf -w t | grep debug_info
Offset into .debug_info:  0
Offset into .debug_info:  924
Offset into .debug_info:  9d3
Offset into .debug_info:  203b
Offset into .debug_info section:     136
Size of area in .debug_info section: 2204
Offset into .debug_info section:     2526
Size of area in .debug_info section: 5725
The section .debug_info contains:
```

If you do not see information similar to this, re-compile all of your source code using the `-g` option.

9.1.2
Problem: Installation Overwrites 32-bit Version

When you install the PathDB debugger on the SuSE 9 platform, you get a warning and the 32-bit version of the debugger doesn't seem to get installed.

```
# rpm -Uvh --force /opt/pathscale/SuSE9-2.3/rpms
warning: package pathscale-pathdb = 2.3-89.328_suse9.0_psc was
   already added,replacing with pathscale-pathdb <=
   2.3-90.328_suse9.0_psc
Preparing...    ######################################### [100%]
1:pathscale-base ###################################v#### [  5%]
2:pathscale-base-x86_64 ################################# [10%]
3:pathscale-gnu-devel ############################### [15%]
4:pathscale-libgcc ################################### [20%]
5:pathscale-libstdc++ ##############################v#### [25%]
6:pathscale-compilers-lib##########################v### [30%]
7:pathscale-sub-client ################################# [35%]
8:pathscale-binutils ################################# [40%]
9:pathscale-gcc ################################### [45%]
10:pathscale-compilers-com########################### [50%]
11:pathscale-gcc-c++ ############################### [55%]
12:pathscale-binutils-docs######################### [60%]
13:pathscale-c++ ################################### [65%]
14:pathscale-c ####################################### [70%]
15:pathscale-compilers-doc######################### [75%]
16:pathscale-compilers-lib######################### [80%]
17:pathscale-f90 ################################### [85%]
18:pathscale-pathdb ################################# [90%]
19:pathscale-pathdb-devel ######################### [95%]
20:pathscale-gnu-devel-doc######################### [100%]
```

9.1.3
Problem: Symbolic Link Not Created for Non-default Installation

When you install the PathDB debugger in a non-default location (not
`/opt/pathscale`) the PathDB symbolic link to the install directory is not created
correctly.

There is a script in the PathDB RPM that attempts to make a symbolic link in the
`<prefix>/bin` directory, like this:

```
lrwxrwxrwx 1 root root 17 Sept 8 11:45 pathdb -> pathdb-x86_64-2.3
-rwxr-xr-x 1 root root 4191194 Sept 7 17:40 pathdb-i386-2.3
-rwxr-xr-x 1 root root 4422838 Sept 7 17:55 pathdb-x86_64-2.3
```

The script is looking in `/opt/pathscale/bin`, instead of the install directory for
PathDB. As a temporary workaround, you need to manually create the link like this:

```
# cd <prefix>/bin # ln -s pathdb-x86_64-2.3 pathdb
```

Alternatively, if you want to use the 32-bit debugger by default instead, use:

```
# cd <prefix>/bin # ln -s pathdb-i386-2.3 pathdb
```

QLOGIC

9.2
Subscription Manager Problems

To use the PathDB debugger, QLogic PathScale subscription management must be installed and correctly configured for your system. If you have already installed the QLogic PathScale Compiler Suite environment, subscription management should be set up.

See the *QLogic PathScale Compiler Suite and Subscription Manager Install Guide* for information on subscription management and environment variables, and configuring the Subscription Manager.

9.2.1
Problem: Subscription Manager Not Found

The QLogic PathScale subscription management must be installed and correctly configured to use `pathdb`. When you try to debug a file, the debugger will try to find the QLogic PathScale subscription management `subclient` program and invoke it. If it can't find this program, you get a message similar to this and the program will quit:

Unable to obtain license – could not locate the license client program

NOTE: The Subscription Manager server is only needed if you have a floating subscription, not if you have a nodelocked subscription. The subscription management subclient is automatically installed with the compiler suite or the debugger. See the *QLogic PathScale Compiler Suite and Subscription Manager Install Guide* for more information on QLogic PathScale subscription management.

To help determine more about the problem with the subscription management, use the `-subverbose` option with `pathdb`. Your output might look something like this:

```
$ pathdb EXP -subverbose
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale Inc.
All Rights Reserved
Reading symbols from /home/EXP...done
looking for /opt/pathscale/lib/2.1/subclient
Attempting to get license for language: FORTRAN90
Subscription client: No PATHSCALE_SUBSCRIPTION_DAEMON variable
  is present
Subscription client: looking for pscsubscriptionserver in DNS
Subscription client: using CNAME server: pscsubscriptionserver
Subscription client: pscsubscriptionserver resolved to
  192.168.3.64
Subscription client: attempting to contact subscription server at
  pscsubscriptionserver: 7143
Subscription client: subscription granted by server
pathdb>
```

See the *QLogic PathScale Subscription Management User Guide* for more
information about QLogic PathScale subscription management.

9.2.2
Problem: Subscription Manager Not Working

The QLogic PathScale subscription management subclient (and server, if you have
a floating subscription) and the PathDB debugger must be from the same compiler
release. For example, you cannot use QLogic PathScale subscription management
from the 1.4 compiler release with the PathDB debugger from the 2.1 compiler
release. The debugger and subscription management must be part of the same
compiler release to function properly.

NOTE: The compiler release or version numbers and the PathDB debugger
        release or version numbers are not the same.

9.2.3
Problem: Nodelocked Subscriptions and PathDB

If you have a nodelocked subscription, then pathdb can only be used on the system
for which the subscription is valid. To use the debugger on other systems, you will
need a floating subscription. See section 2.4 for more information on subscription
management for the debugger.

Section 10
Complete Command List

This list of commands can also be found in the Help system for the PathDB debugger. Type `help` followed by a topic or command name.

pathdb> **help *<topic>***

Help is available for the following topics:

1. Breakpoints - information on breakpoints

2. Running - how to run the program

3. Locations - how to specify locations used by some commands.

4. Commands - information on all the commands available

5. Formats - formats for the print function

6. Expressions - expression support

7. Parameters - debugger parameters

Note that the first letter is uppercase, to distinguish the topic from a command with the same name.

For example, to get help on PathDB breakpoints, you would type:

```
pathdb> help break
Help for break
Command syntax: break location
Set a breakpoint at the specified location.
Set a breakpoint at a location. The location is specified by:
1.  A * followed by an address
2.  A line number in the current file
3.  A filename and line number, specified by filename:lineno
4.  A function name
The result will be a breakpoint inserted at the given address in
the program. If the function name given to the command is
ambiguous, a menu of alternative fu
For example: pathdb> b evaluate
---Type <return> to continue, or q <return> to quit---
```

10.1
Command Options Syntax

The syntax for the PathDB debugger command options is:

`[ ]`   - Indicates optional arguments

< >   - Indicates required arguments

...   - Indicates that multilple arguments can be used

`/format`- Specifies the format for the output. See section 10.2 for details on format.

For example, look at the `print` option:

print *[/format] <expression>*
> Print the value of the expression and store the result in debugger variable.

With this option, you can specify the format for the output (this is optional) and you must specifiy the expression you want to print. See section 10.2 for details on format.

10.1.1
PathDB Commands

These are the PathDB commands, in alphabetical order:

`advance <location>`
> Continue execution until the specified location is reached.

`alias [<name>[<value>]]`
> With no args, show all aliases. With one argument, show the named alias. With two arguments, set the named alias to the named value.

`attach <pid>`
> Attach to a running process.

`awatch <location>`
> Create a watchpoint for a read or write of the specified location.

`backtrace [levels]`
> Show the stack backtrace for the specified number of levels (default: all levels).

`break <location>`
> Set a breakpoint at the specified location.

`breakpoints`
> Show all breakpoints, watchpoints or catchpoints.

`call [/format] <expression>`
> Call a function in the program being debugged. Record the value and print it if the type is not void. See section 10.2 for details on format.

catch

> Set catchpoint at C++ expressions, exec calls, fork calls, or shared library calls.

catch catch

> Catch C++ exceptions catches.

catch exec

> Catch a call to exec.

catch fork

> Catch a call to fork.

catch throw

> Catch a C++ exception throw.

catch vfork

> Catch a call to vfork.

cd *<dir>*

> Change the current working directory to that specified. No arg means to change to the home directory.

clear *[location]*

> Clear breakpoints at the specified location (line or address). If no location is specified, it clears the breakpoint at the current locations.

commands *<breakpoint-number>*

> Specify a set of commands to be executed when the breakpoint is activated. If no breakpoint in specified, the last one set is targeted.

condition *<breakpoint-number> [condition]*

> Set or clear the condition on the specified breakpoint.

continue *[signal-number]*

> Continue execution of the program being debugged. If a signal number is specified, continue executing by sending specified signal.

define

> Define a new command as a sequence of existing commands.

delete *[breakpoint-number...]*

> Delete breakpoints, watchpoints or displays.

`delete breakpoints`
> Delete all breakpoints.

`detach`
> Detach from a running process that is currently being controlled by the debugger

`dir <directory>`
> Set the search directory for source file searches.

`disable [breakpoint-number...]`
> Disable the numbered breakpoints.

`disassemble [start] [end]`
> Disassemble the object code in the specified address range. If no start or stop address is specified, the current function is disassembled.

`display [/format] <expression>`
> Print the value of the expression every time the debugger stops the program being debugged. See section 10.2 for details on format.

`down [levels]`
> Move the current stack frame down the stack by the specified number of levels. If the number of levels is unspecified, then the stack frame is moved down one level.

`enable [breakpoint-number...]`
> Enable or set the disposition on a set of breakpoints.

`enable delete [args]`
> Set the disposition so that the breakpoints will be deleted when hit.

`enable display [args]`
> Enable the named displays.

`enable once [args]`
> Set the disposition so that the breakpoints will be disabled when hit.

`env`
> Show the values of all the environment variables.

file *[file]*

> Change the file being debugged. If no file is specified, the current file is unloaded.

finish

> Continue execution until the current function returns.

frame *[frame-number]*

> Select the specified frame number or show the current frame. If no frame is specified, information about the current frame is printed.

handle [signal |all] [no]pass |[no]stop |[no]print |[no]ignore...]

> Specify how signals are to be handled when raised by the program being debugged. If the signal-number is "all" then all signals are handled in the specified manner. See section 7 for more information on signals.

hbreak *[location]*

> Set a hardware-assisted breakpoint at the specified location. If no location is given, the current location is used.

history

> Show all the commands typed by the user.

if *<expression>*

> Execute a sequence of commands if the expression evaluates to non-zero. The expression must be on the same line as the if command. The commands to execute are entered, one per line, terminated by end on its own.

ignore *[breakpoint-number] [n]*

> Set the ignore-count for the specified breakpoint. If no breakpoint is specified, the last breakpoint set is used.

info *<name>*

> Provide information on the named entity.

info address *<symbol>*

> Show where the specified symbol is stored.

info all-registers

> Show the current values of all registers.

info args

> Show the values of all the arguments to the current function.

info catch

> Show the exceptions that can be caught within the current frame.

info copying

> Show the copyright information for the debugger.

info display

> Show the expressions printed when the program hits a breakpoint or otherwise stops execution.

info line

> Show information about the current line.

info locals

> Show the names and values of all the variables local to the current function.

info proc

> Show information about the running process.

info program

> Show the status of the program being debugged.

info registers

> Show the values of the machine registers.

info scope *<location>*

> Purpose: List the variables in the named scope.

info signals

> Show how the debugger handles signals caught by the debugged program.

info source

> Show information about the current source file.

info sources

> Show the source files in the program.

info stack

> Show stack backtrace of current position.

info symbol <*address*>

> Purpose: Show what symbol is at the specified address.

info threads

> List all the currently running threads.

info variables

> List all global and static variables.

info watchpoints

> Show all breakpoints, watchpoints, and catchpoints.

kill

> Kill the program being debugged.

list [*line-number*]

> List the source files at the specified locations.

locals

> Show the names and values of all the variables local to the current function.

memdump <*address*>

> Dump the address specified in hex and ASCII.

next *[number]*

> Step the program being debugged by a number of lines, stepping over called functions. If no number is specified, then step one line.

nexti *[number]*

> Step the program being debugged by a number of instructions, stepping over call instructions. If no number is specified, then step one instruction.

output *[/format]* <*expression*>

> Print the value of the expression with no line feed and do not insert the value into a debugger variable. See section 10.2 for details on format.

print *[/format]* <*expression*>

> Print the value of the expression and store the result in debugger variable. See section 10.2 for details on format.

process *[process-number]*

        Switch to another process. If no process is specified, print the current process identifier.

processes

        List all the processes being debugged.

ptype *<expression>*

        Print the type of the expression.

pwd

        Print the current working directory.

quit

        Quit the debugger.

rerun *[number]*

        Rerun the program, issuing all commands typed up until 'ncommands' before the last one. This is an effective 'step backwards.' Rerun keeps track of all of the commands, state changes, and value changes and reruns the program to 'ncommands' before the current one. Default is 1.

return *[expression]*

        Return from the current function with the value specified.

run *[args]*

        Run the program being debugged with specified arguments.

rwatch *<location>*

        Create a watchpoint for a read of the specified location.

set *[var] [value]*

        Set the value of control parameters or program variables. See section 5.9.1 and section 5.10 for more information on the set command.

set args *[args]*

        Set the arguments to be passed to the program being debugged on the next run.

set variable *[var = value]*

        Set the value of the named variable to the specified value in the program being debugged.

setenv *[name [value]]*

> With no args, print the environment variables. With one or two args, set the named environment variable to the given value.

shell *[command-args]*

> Execute the rest of the line as a shell command. With no args, spawn an interactive shell. The shell spawned is that specified in the `$SHELL` environment variable (default `/bin/sh`)

show

> Show the current parameters set for `pathdb`.

show version

> Show the current version of `pathdb`.

source *<file>*

> Execute debugger commands from the specified file

step *[number]*

> Step the program being debugged by a number of lines, stepping into called functions. If no number is specified, then step one line.

stepi *[number]*

> Step the program being debugged by a number of instructions, following call instructions. If no number is specified, then step one instruction.

target core *<file>*

> Attach to a core file as a target.

target child

> Use a UNIX child process as a target.

target exec *<file>*

> Use an executable file as a target (same as file command).

tbreak *<location>*

> Set a temporary breakpoint at the specified location.

thbreak *<location>*

> Set a temporary hardware-assisted breakpoint at the specified location.

thread *[thread-number]*
> Switch to another thread, thus setting the current thread. If no thread is specified, show information about the current thread.

unalias *<alias>*
> Delete the specified alias.

undisplay *[display-number]*
> Remove the specified display. If no display is specified, remove all displays.

until *[location]*
> Single step until the location is reached, or until the function returns. If no location is specified, then step one line.

up *[levels]*
> Move the current stack frame down the stack by the specified number of levels. If the number of levels is unspecified, move the stack up one level.

watch *<location>*
> Create a watchpoint for a change in the value of the specified location.

whatis *<expression>*
> Print the type of the expression.

while *<expression>*
> Execute a sequence of commands while the expression evaluates to non-zero. The expression must be on the same line as the while command. The commands to execute are entered, one per line, terminated by 'end' on its own.

x [/*format*] *<address>*
> Examine memory at the specified address, using the format specified. See section 10.2 for details on format.

10.2
Formats

The PathDB debugger has a number of options for the format of output from the call, display, output, print, and x commands. These commands can take a format specifier as their first argument. If this is present, it must be prefixed by a '/' character.

The syntax of the format specifier is: `/` `[count]` `[code]` `[size]` (no spaces are allowed).

The `count` specifier is a decimal number specifying the number of elements to which this format applies.

The `code` specifies the output format for the elements. The `size` specifies the size of each element. The `code` is specified by a single lowercase letter. The allowed codes are:

1. `'o'` - octal
2. `'x'` - hexadecimal
3. `'d'` - decimal
4. `'u'` - unsigned decimal
5. `'t'` - binary
6. `'f'` - floating point
7. `'a'` - address
8. `'c'` - character
9. `'s'` - string
10. `'i'` - instruction disassembly

The size is also specified by a single lowercase letter:

1. `'b'` - byte (8 bits)
2. `'h'` - half word (16 bits)
3. `'w'` - word (32 bits)
4. `'g'` - giant word (64 bits)

10.2.1
Format Examples

Here is an example of using format with the following source file:

```
int main()
{
   int a[ ] = {1,2,3};
   return 0;
}
```

Compile the program with the `-g` option, and then run `pathdb` with the executable file.

```
$ pathdb a.out
PathScale Debugger Version 1.1.1
Copyright (C) 2004, 2005 PathScale
Inc. All Rights Reserved
Reading symbols from /home/a.out...done
pathdb> break main
Breakpoint 1 at 0x40055a: file /home/test.c, line 4.
pathdb> run
Starting program: /home/a.out Breakpoint 1, main () at
/home/test.c:4
4    int a[] = {1,2,3};
pathdb> next
5    return 0;
```

The format string has three parts. The first part specifies the number of elements to print, the second part the size of each element, and the third string the display format. All parts are optional.

By default `print` will display integers in decimal.

```
pathdb> print a[2]
$1 = 3
```

However, if you wanted to display the integer in hexadecimal you could add an 'x' to the format

```
pathdb> print /x a[2]
$2 = 0x3
```

If you wanted to display in octal, you could add an 'o' to the format string.

```
pathdb> print /o a[2]
$3 = 03
```

The format string is also used extensively with the x command. This command reads some number of bytes from memory and displays the value. Since values are read as 'raw' data, you should use the format string to specify how to interpret the data. When the result is printed, the address will be displayed on the left, and the value of the bytes on the right.

For example, to read the data from the address in the variable 'a' and display the result as a 32-bit integer (the 'w' part) in decimal (the 'd' part) use this command:

```
pathdb> x /dw a
0x7fbfffec50:    1
```

The x command supports taking the address from an arbitrary expression, so this also works with:

```
pathdb> x /dw a+1
0x7fbfffec54:    2
```

The x command also supports printing multiple elements at once using the count flag. This command prints the first element of the array 'a' in hexadecimal. Since

the default is one, this is actually what would have printed even without the count flag.

```
pathdb> x /1xw a
0x7fbfffec50:   0x00000001
```

This command prints the first two elements in hexadecimal:

```
pathdb> x /3xw a
0x7fbfffec50:   0x00000001 0x00000002
```

And now all three are printed in decimal:

```
pathdb> x /3dw a
0x7fbfffec50:   1 2 3
```

Expressions

The PathDB debugger recognizes and works with a wide range of expression operators for Fortran, C, and C++. This appendix lists the priority order of those operators, with tips about syntax and usage.

Expression Operators in C/C++

The syntax `'identifier @ line-number'` allows a way to distinguish among variables with the same name in nested lexical scopes. All identifiers are case-sensitive.

Numbers are in the standard C syntax (prefix `0` for octal, `0x` for hex). No suffixes are supported at present.

The postfix `'( )'` operator allows functions to be called in the program being debugged. The array subscripting operator `'[ ]'` allows a range to be specified using a `':'` to separate the lower and upper bounds of the range (e.g. `print foo[3:5]`).

In C/C++ mode, the following expression operators are available from within pathdb (listed in priority order):

- ::
- () [:] ++ – . -< (postfix operators)
- - + * &! ~ ++ – sizeof cast (unary operators)
- */%
- +-
- <<>>
- <<=>>=
- == !=
- &
- ^
- |
- &&
- ||
- ?:
- =+=-=*=/=%=&= |=^=<<=>>=
- ,

The highest priority is the ʼprimaryʼ expression which consists of:

- (expression)
- identifier [@line-number]

- number (integer or floating point)
- "string" or 'string'
- 'character'
- $regname or $debug_var
- {vector literal, ...}
- type specification

Expression Operators in Fortran

In Fortran mode, the expression operators are (in order of priority):

- % (member selector)
- () (array, function call or intrinsic)

- cast
- **
- */
- + - (unary)
- + - (binary)
- //
- .eq. .ne. .lt. .le. .gt. .ge.
- .not.
- .and.
- .or.
- .negv. .eqv.
- =+=-=*=/=%=&= |=^=<<=>>;=
- ,

NOTE:　Some of the operators available in Fortran mode are not strictly Fortran operators, but are useful for debugger functionality.

The highest priority is the 'primary expression', which is one of:

- (expression)
- intrinsic
- identifier [@line-number]
- $regname
- $debug_var
- number (in Fortran syntax)
- 'string' or "string"
- 'character' • .true. or .false.
- {vector literal, ...}

Numbers are in both C and Fortran syntax, so both `0x` and `' Z` can be used to specify a hex number. All identifiers are case-blind.

The instrinsics supported are:

```
KIND ()
LOC ()
ALLOCATED ()
ASSOCIATED ()
UBOUND ()
LBOUND ()
LEN ()
SIZE ()
ADDR ()
```

Both C/C++ and Fortran expressions support type casts. The syntax of the type is language dependent. The following are examples of type casts in C and Fortran:

```
print (const int *)foo - C
print (integer(kind=8), pointer)bar - Fortran
```

Notes

Index