# Debugging Core Files Using HP WDB

HP Part Number: 5992-4661
Published: September 2008
Edition: 2.0

# Table of Contents

About This Document....7
Intended Audience....7
Typographic Conventions....7
Related Information....8
Introduction....9
What Is a Core File?....9
Causes for a Core Dump....9
Common Signals That Cause Core Dumps....9
Using WDB to Debug Core Files....12
    Support for Invoking GDB Before a Program Aborts....13
    System Requirements for Core File Debugging....13
Commands For Core File Debugging....14
    Invoking WDB to Debug Core Files....14
    Setting the Path for the Relevant Shared Libraries....15
    Common Commands for Core File Debugging....15
What is a Symbol Table?....21
What is a Stripped Binary?....21
    Debugging Core Files Created by Stripped Binaries (When the Symbol Table is Available)....22
Debugging Core Files Created by Optimized or Stripped Binaries....22
    Limitations for Debugging Core Files Created by Optimized Binaries....22
    Limitations for Debugging Core Files Created by Binaries Compiled Without the `-g` Option....23
    Limitations for Debugging Core Files Created by Stripped Binaries....23
Forcing a Core Dump....29
    Saving the Core File to a Specific File Name....29
    Debugging a Core File Created by a Forced Core Dump....29
Debugging Core Files From a Different System....31
Displaying run time type information....32
Debugging PA-RISC Core Files on Integrity Systems....32
Avoiding Core File Corruption....34
    Avoiding Core File Corruption for Applications Running HP-UX 11i v1 and HP-UX 11i v2....34
    Avoiding Core File Corruption for Applications Running HP-UX 11i v3....35
Java Corefile Debugging Support....36
Summary....36
Examples Illustrating Core File Debugging....37
FAQ....55

# List of Tables

1 System Requirements for Core File Debugging........13
2 Commonly Used Commands for Core File Debugging........16
3 Supported Systems for PA-RISC Core File Debugging........31
4 Commands for Debugging a Core File From a Different System........32

# List of Examples

1 SIGBUS Causes a Core Dump....10
2 SIGSEGV Causes a Core Dump....11
3 SIGABRT Causes a Core Dump....12
4 Viewing Symbol Information by Using the nm Command....21
5 Debugging Core Files Created by Optimized Code, Stripped Binaries, and Code Compiled Without the `-g` Option....25
6 Debugging a Core File to Find the Values for Parameters of a Function When the Program is not Compiled with -g....37
7 Debugging a Core File to View Information on a Global Variable in a C program....45
8 Debugging a Core File Created by a Stripped Binary When the Symbol Table is Available....47
9 Debugging of a Core File Created by a Stripped Binary When the Symbol Table is Available from Another Program....49
10 Core File Debugging Session for a Stripped Binary When the Symbol Table is Available from Another Program....51

## About This Document

This document describes how to debug core files and analyze the process state of an application, using HP WDB.

## Intended Audience

This document is intended for C, Fortran, and C++ programmers who use HP WDB to debug core files generated by HP C, HP aC++, and Fortran90 compilers. Readers of this document must be familiar with the basic commands that HP WDB supports.

## Typographic Conventions

This document uses the following typographical conventions:

| | |
|---|---|
| `%`, `$`, or `#` | A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells. A number sign represents the superuser prompt. |
| *audit*(5) | A manpage. The manpage name is *audit*, and it is located in Section 5. |
| `Command` | A command name or qualified command phrase. |
| `Computer output` | Text displayed by the computer. |
| `ENVIRONMENT VARIABLE` | The name of an environment variable, for example, `PATH`. |
| **Key** | The name of a keyboard key. **Return** and **Enter** both refer to the same key. |
| *`Variable`* | The name of a placeholder in a command, function, or other syntax display that you replace with an actual value. |
| [] | The contents are optional in syntax. If the contents are a list separated by \|, you must choose one of the items. |
| {} | The contents are required in syntax. If the contents are a list separated by \|, you must choose one of the items. |
| ... | The preceding element can be repeated an arbitrary number of times. |
| \| | Separates items in a list of choices. |
| NOTE | A note contains additional information to emphasize or supplement important points of the main text. |

## Related Information

The following table lists the documentation available for HP WDB.

| Document | Location |
|---|---|
| *Debugging with GDB* | `/opt/langtools/wdb/doc/gdb.pdf` |
| *GDB Quick Reference Card* | `/opt/langtools/wdb/doc/refcard.pdf` |
| *Getting Started With HP WDB* | `/opt/langtools/wdb/doc/html/wdb/C/GDBtutorial.html` |
| *WDB GUI Online Help* | `/opt/langtools/wdb/doc/index.html` |
| HP WDB GUI Documentation | `/opt/langtools/wdb/doc/html/wdbgui/C/` |
| GDB manpage | *gdb*(1) |

For the most current HP WDB documentation, see the *HP WDB Technical Resources* website at:

http://www.hp.com/go/wdb

## Introduction

HP Wildebeest Debugger (WDB) is an HP-supported implementation of the open source debugger GDB. Apart from performing the normal debugging functions, it also enables you to debug core files and analyze the process state of an application.

HP WDB debugs core files that are created by source-level programs written in HP C, HP aC++, and Fortran 90 on Itanium®-based systems running HP-UX 11i v2 or HP-UX 11i v3, and HP 9000 systems running HP-UX 11i v1, HP-UX 11i v2, or HP-UX 11i v3 operating systems.

## What Is a Core File?

A core dump is an abnormal termination of a program. The most common types of programming errors that can cause a core dump include program aborts, memory violations, bus errors, and illegal instructions. When a core dump occurs during the execution of a program, the core file, `core`, is created in the working directory of the terminated process. This core file reflects the memory image of the terminated process. You can use the information in a core file to debug an abnormally-terminated program and analyze the causes for the core dump.

## Causes for a Core Dump

When a kernel encounters an un-handled signal, it creates a core file for that process. Alternately, the user can force a core dump to create a core file (through WDB or by using `gcore`).

The `file` command is the simplest method to analyze the cause of a core dump at the HP-UX prompt.

The `file` command displays the signal that triggered the core dump.

For example:

```
$ file core
core: core file from 'a' - received SIGBUS
```

This example illustrates that the program dumped a core after receiving the `SIGBUS` signal.

## Common Signals That Cause Core Dumps

Following are some signals that commonly cause core dumps:

- **`SIGBUS`**

  One reason why a`SIGBUS` signal is sent to a process is when the program attempts to load or store a data item at a non-aligned address.

  Example 1 illustrates a load or store operation of a data item at a non-aligned address.

**Example 1 SIGBUS Causes a Core Dump**

```
$ cat a.c
#include <stdio.h>

int main()
{
   char a[64], *b;
   int *i;

   b = a;
   b++;
   i = (int *)b;
   printf("%i", *i);
return 0;
}
$ aCC a.c -o a
$ ./a
Memory fault(core dump)
$file core
core: core file from 'a' - received SIGBUS
```

In this example, the program attempts to load a data item at a non-aligned address, which results in a `SIGBUS` signal.

The variable `a` is a local variable on the stack. The pointer `b` is set to point to the start of `a`. The pointer `b` is set to increment such that it does not point to a word aligned address. The value in pointer `b` is assigned to pointer `i`. When pointer `i` is de-referenced, a `SIGBUS` signal is encountered.

- **`SIGSEGV`**

  A `SIGSEGV` signal is sent to a program when a segmentation violation occurs. A segmentation violation occurs when a process attempts to access an address that is not in the currently allocated address space of the process.

  Example 2 illustrates how a `SIGSEGV` signal can cause a core dump.

**Example 2 SIGSEGV Causes a Core Dump**

```
$ cat a.c
int main()
{
   int *i, j;
   i = (int *)0x48000000;

   j = *i;
return 0;
}
$ aCC a.c -o a
$ ./a
Memory fault(core dump)
$ file core
core: core file from 'a' - received SIGSEGV
```

In this example, the program de-references a nonexistent pointer address, and this results in a `SIGSEGV` signal.

- **`SIGABRT`**

  A `SIGABRT` signal can be sent to a process in any of the following ways:

  — The process can send the abort signal, `SIGABRT`, by invoking the `abort(3)` function.

  — Another process or the user can invoke the `kill` command to send the `SIGABRT` signal.

  — As a result of calls to C++ `terminate()` function on various runtime library errors. Example 3 (page 12) illustrates a SIGABRT signal caused by a call to `terminate()`.

**Example 3 SIGABRT Causes a Core Dump**

```
$ cat gdb_throw_example.c
#include <stdio.h>
void foo(int i) {
   throw i;
}
int main() {
   foo(10);  // will not be caught
}
$ a.out
aCC runtime: Uncaught exception of type "int".
Abort(coredump)

Core was generated by `a.out'.
Program terminated with signal 6, Aborted.

(gdb) bt
#0  0x60000000c0349f50:0 in kill+0x30 () from /usr/lib/hpux32/libc.so.1
#1  0x60000000c0240e90:0 in raise+0x30 () from /usr/lib/hpux32/libc.so.1
#2  0x60000000c0304390:0 in abort+0x190 () from /usr/lib/hpux32/libc.so.1
#3  0x60000000c4744cb0:0 in std::terminate () at ../terminate.C:70
#4  0x60000000c476c550:0 in __cxa_throw () at ../NewExceptionHandling.C:610
#5  0x4000ad0:0 in foo () at gdb_throw_example.c:3
#6  0x4000ba0:0 in main () at gdb_throw_example.c:6
```

This example illustrates the core dump which is caused by a call to the C++ `terminate()` function.

For more information about other common signals that can cause core dumps, see *signal*(5).

**NOTE:** The SIGKILL signal does not generate a core file.

## Using WDB to Debug Core Files

The core file debugging features in WDB enable you to analyze the cause of a core dump and analyze the process state of an application.

Core file debugging features in WDB are typically used under the following scenarios:

- The program dumps core as a result of programming errors.
- The program is forced to dump core by using the `dumpcore` command in WDB, or the `gcore` utility, that is available on HP-UX.

WDB can be used to debug the following kinds of core files:

- A core file is created by a program that is compiled without the `-g` option.
- A core file is created by a stripped executable.
- A core file is created by a program, and the source code for the program is available.

If you can reproduce the problem when running the program under WDB, it is easier to use a live debugging session in WDB to debug the program, instead of debugging the core file. However, the same debug information in the program can be used for core file debugging.

## Support for Invoking GDB Before a Program Aborts

WDB also provides the `-crashdebug` option to monitor the program execution and invoke the debugger when the program execution is about to abort. This option provides support for debugging a live process before the program aborts, instead of debugging the core file after the program aborts.

Once the debugger is invoked, you can debug the application by using the common debugger commands. You can examine the state of the process, make changes to the state, and continue program execution, force a core dump, or terminate execution.

It also enables you to control program execution under the debugger if the program is about to abort. You can load a new process or attach to a running process for monitoring.

To monitor a new process, enter the following command at the HP-UX prompt:

```
$ gdb -crashdebug [command][options]
```

To monitor and attach to a running process, enter the following command at the HP-UX prompt:

```
$ gdb -crashdebug -pid [pid]
```

## System Requirements for Core File Debugging

Table 1 lists the system requirements for debugging core files using WDB.

**Table 1 System Requirements for Core File Debugging**

| Requirement | Description |
|---|---|
| Operating System | HP–UX 11i v1, HP-UX 11i v2, or HP-UX 11i v3 on HP 9000 systems<br>HP-UX 11i v2 or HP-UX 11i v3 on Integrity systems |

## Commands For Core File Debugging

This section discusses the commands for debugging core files.

### Invoking WDB to Debug Core Files

To invoke the debugger on the core file, enter one of the following commands:

- At the HP-UX prompt:

  ```
  $ gdb a.out core
  ```

  or

  ```
  $ gdb a.out -c core
  ```

  or

  ```
  $ gdb -c core
  ```

- (Or) At the gdb prompt:

  ```
  (gdb) core core
  ```

(where `a.out` is the executable that dumped `core`.)

If the executable path is not provided, the debugger selects the invocation path of the process that generated the core file. The invocation path information is stored in the core file. If the invocation path is a relative path, you must specify the executable to debug the core file.

On invoking the core file debugging session, the debugger displays the following information (depending on the debug information available):

- The signal that caused the core file
- The cause of the un-handled signal
- The instruction at which this signal occurred
- The function name and the parameters of the function in which this instruction resides
- The source line information

The following example illustrates the output from the debugger on invoking a core file debugging session:

```
..
Core was generated by `a.out'.
Program terminated with signal 11, Segmentation fault.
SEGV_ACCERR - Invalid Permissions for object
#0  inline generate_core_dump () at a.c:11
11        printf ("Generated coredump\n");
(gdb) bt
#0  inline generate_core_dump () at a.c:11
#1  0x4000a00:0 in inline foo () at a.c:30
#2  0x40009b0:1 in main () at a.c:37
```

## Setting the Path for the Relevant Shared Libraries

The core files do not carry information about the exact version of shared libraries that were in use at the time of core dump. Analyzing a core file without the correct versions of shared libraries can produce misleading results. Hence, you must provide information about the relevant shared libraries before initiating a core file debugging session. All the required libraries must be copied to a temporary location on the system where you are debugging the core file (if it is different from the system where the core file was generated).

The executable and the core file inherently carry information about the list of shared libraries that were loaded at the instant of core dump. However, this list of shared libraries is referenced by pathnames (the invoked path of the shared libraries on the system where the core dump occurred).

If the shared libraries are located at a path that is different from the invoked path, you must provide WDB with the path for the shared libraries.

To associate the appropriate versions of the shared libraries with the core file, set the environment variable, `GDB_SHLIB_PATH`, as follows:

```
$ export GDB_SHLIB_PATH<path>
```

---

**NOTE:** You can use `packcore`, and `unpackcore` to pack, or unpack the core file along with the relevant executable and libraries in a single `tar` file, and debug the core file on a different system from the one on which the core file was invoked.

For more information on debugging a core file from a different system than the one on which the core file was created, see "Debugging Core Files From a Different System" (page 31)

---

## Common Commands for Core File Debugging

Table 2 lists the common commands for core file debugging.

## Table 2 Commonly Used Commands for Core File Debugging

| Debugging Feature | Command | Description |
|---|---|---|
| Invoking the core file | At the HP-UX prompt:<br>`$ gdb a.out core`<br>or<br>`$ gdb a.out -c core`<br>or<br>`$ gdb -c core` | Invokes the core file debugging feature in WDB.<br>If the executable path is not provided, the debugger selects the invocation path of the process that generated the core file. The invocation path information is stored in the core file. If the invocation path is a relative path, you must enter the executable while debugging the core file. |
| Viewing backtrace information | `backtrace [<-><count>]`<br>`where [<-><count>]` | Displays the backtrace information about the process that encountered the un-handled signal and the call chain (including inlined functions). The backtrace is displayed for the thread where the un-handled signal occurred.<br>All the stack frames are displayed if no arguments are specified. If *<COUNT>* is specified, it display the innermost *COUNT* frames. If a negative argument, *<-COUNT>*, is specified, it displays the outermost *COUNT* frames. |
| Traversing the stack | `up <number>`<br>`down <number>`<br>`frame <frame-number>` | The `up` and the `down` commands enable you to traverse (up or down) the call chain in the stack. You can traverse up to a specific number of frames in the stack if `<number>` is specified.<br>The `frame <frame-number>` command enables you to traverse the stack frame to the specified frame number, `<frame-number>`. This thread is marked with '`>`' in the `info thread` output, while the current selected thread is marked with a '`*`' symbol. |

**Table 2 Commonly Used Commands for Core File Debugging** *(continued)*

| Debugging Feature | Command | Description |
|---|---|---|
| Viewing thread information | `info thread`<br>`thread <thread-id>`<br>`thread apply <thread-id>[all] args`<br>`backtrace-other-thread` | The `info thread` command enables you to view the list of all the threads in the process at the time of core dump.<br>The `thread <thread-id>` command enables you to switch the thread view under the debugger from one thread to another. The thread that created the un-handled signal is the current thread when the core file is loaded in to the debugger.<br>The `thread apply` command allows you to apply a command to one or more threads. You can specify the numbers of the threads, where the command must be applied, with the command argument*`<thread-id>`*. the command argument*`<thread-id>`* is the internal GDB thread number, as shown in the first field of the`info threads` display. To apply a command to all threads, use `thread apply all` *`args`*.<br>The `backtrace-other-thread` command prints the backtrace of all stack frames for a thread with stack pointer `SP`, and program counter `PC`. This command is useful in cases where the debugger does not support a user thread package fully. |
| Printing global and local variables | `print </f><expr>` | Displays information about the global and local variables in the program.<br>The *`<expr>`* is an expression (in the source language). By default the value of *`<expr>`* is printed in a format appropriate to its data type. To change the display format, you can use the where *`</f>`* option, where *`f`* is a letter specifying the display format, *`[x|d|u|o|t|a|c|f]`*. |
| Printing the description of a data type | `ptype <typename>` | Prints the description of a data type,*`<typename>`*, where *`<typename>`* can be the name of a type, or it can have the form `class class-name`, `struct` *`<structtag>`*, `union` *`<union-tag>`* or `enum` *`<enum-tag>`* in the case of C code. |

**Table 2 Commonly Used Commands for Core File Debugging** *(continued)*

| Debugging Feature | Command | Description |
|---|---|---|
| Navigating the source code | `list [- \| <line-number> \|<function> \| <*address> ]` | Enables you to navigate the source code if it has been compiled with the `-g` option.<br>When no arguments are specified, it lists ten lines after or around the previous listing.<br>The `list -` command lists the ten lines before a previous ten-line listing.<br>The `list <line-number>` command lists the source code around the specified line in the current file. You can also specify the starting line number and the ending line number of the source code to be displayed (separated by a comma).<br>The `list <function>` command lists the source code around the beginning of the specified function.<br>The `list <*address>` command lists the source code around the line containing the specified address. |
| Disassembling the core file | `disassemble <address>`<br>`disassemble <func-name>`<br>`disassemble <address> - <address>` | Disassembles a specified section of the memory. The default disassembled memory is the function surrounding the `pc` of the selected frame.<br>If an address is specified, the function surrounding the specified address is disassembled.<br>If `<func-name>` is specified, the range of addresses for that function are disassembled.<br>If two addresses are specified, the function surrounding the specified address range is disassembled. |
| Examining memory | `x /[x\|s\|d] <address>` | Displays memory information of a specified address.<br>The `x / x <address>` command prints the contents of the specified memory address in hexadecimal format.<br>The `x / s <address>` command prints the contents of the address of a string.<br>The `x / d <address>` command prints the contents of the address in decimal format. |
| Viewing register information | `info registers` | Displays the contents of the registers at the time of core dump. |
| Viewing shared library information | `info shared` | Displays information about all the shared libraries that are loaded at the time of core dump. |

**Table 2 Commonly Used Commands for Core File Debugging** *(continued)*

| Debugging Feature | Command | Description |
|---|---|---|
| Prints the target that is currently under the debugger | `info files`<br>`info target`<br>`help target` | The `info files` and `info target` commands print the current target, including the names of the executable and core dump files currently in use by GDB, and the files from which the symbols were loaded.<br>The command `help target` lists all possible targets rather than current ones. |
| Read symbol information from a file | `symbol-file` *`<filename>`* | Read symbol table information from file *`<filename>`*. The `symbol-file` command with no argument clears out GDB information on the symbol table of the program, and causes GDB to erase the contents of the convenience variables, the value history, and all breakpoints and auto-display expressions |
| Read additional symbol information | `add-symbol-file` *`<filename>`* *`<address>`*`[-s` *`<section>`* *`<sect-address>`* `-s` *`<section>`* *`<sect-address>`*`]` | Reads additional symbol table information from the file *`<filename>`* (when *`<filename>`* is dynamically loaded into the program that is running.<br>The *`<address>`* is the memory address at which the file is loaded. (GDB cannot detect this address, unless specified)<br>You can specify up to three addresses (the addresses of the text, data, and bss segments respectively). |
| Forcing a core dump | `dumpcore` *`<corefile-name>`* | Forces a core dump and creates a core image file for a process that is running under the debugger.<br>If the filename is specified, it saves the dumped core file in the file, `<corefile-name>`, instead of the default file, `core.<pid>` (where `pid` is the process ID number). |
| Packing the core file along with the associated shared libraries | `packcore` | Packs the core file along with the relevant executable and libraries in a single `tar` file for core file debugging on another system. |

**Table 2 Commonly Used Commands for Core File Debugging** *(continued)*

| Debugging Feature | Command | Description |
|---|---|---|
| Unpacking the core file along with the associated shared libraries | `unpackcore` | Unpacks the `tar` file that is generated by the `packcore` command so that the debugger can use the executable and shared libraries from this bundle, when debugging the core file on a different system from the one on which the core file was originally created. |
| Examine a core file which was previously created by `unpackcore` | `getcore` *`<packcore-directory>`* | Enables you to examine a `packcore` directory, which was previously created by `unpackcore`. It takes one optional argument, which is the name of the `packcore` directory. |

Example 6 (page 37) and Example 7 (page 45) illustrate the use of the common core file debugging commands.

## What is a Symbol Table?

A symbol table is a set of records that define the set of visible and important symbols in a program. These symbols are stored in the program. Each (unstripped) program has an associated symbol table.

The `nm` command displays the symbol information for a specified object file.

Example 4 illustrates how to view symbol information for an object file by using the `nm` command for an object file, on an HP 9000 system.

**Example 4 Viewing Symbol Information by Using the nm Command**

This example illustrates the use of the `nm` command to display the symbol information for the common linker and debugger symbols, `Cerrno` and `Cselectdraw`, on an HP 9000 system:

```
$ nm -x Cscreen_selection.o |grep Cerrno
Cerrno      |          |undef |data  |
$ nm -x Cscreen_selection.o |grep Cselectdraw
Cselectdraw |0x00001178|extern|entry |$CODE$
```

The output of the `nm` command illustrates that the symbol, `Cerrno` is an undefined data symbol and that the symbol, `Cselectdraw` is a function that is a code entry point.

Alternately, you can use `odump -slexport`, `elfdump -n`, or `dynsym -s`, instead of `nm` to view the symbol definition for stripped binaries.

The dynamic symbols are not removed with the `strip` command. The `strip -l` command only strips the line number tables.

## What is a Stripped Binary?

The `strip` command removes the symbol table, debug information, and line number information from the object file, including the archives. Thereafter, no symbolic debugging access is available for the stripped object file.

The `strip -l` strips only the line table information, and the symbolic debugging access continues to be available.

For more information on the `strip` command, see *strip*(1)

Stripped executables or shared libraries can also be built by using the `-s` compiler or linker option.

In HP 9000 systems, the `file` command displays whether an executable is stripped or not.

The following example illustrates the use of the `file` command before and after a `strip` operation:

```
$ file a
a: PA-RISC1.1 shared executable dynamically linked -not stripped
$ strip a
$ file a
a: PA-RISC1.1 shared executable dynamically linked
```

In Integrity systems, you must use the `nm` to display whether the binary is stripped or not. The output from the `nm` command displays '`no symbols`' for a stripped binary on Integrity systems.

### Debugging Core Files Created by Stripped Binaries (When the Symbol Table is Available)

You can debug a core file that is created by a stripped binary effectively, if the symbol table for the unstripped version of the program (before the program is stripped) is available.

Alternately, you can also debug the core file that is created by a stripped program if the symbol table is available from another program, which functionally uses the same symbols, but has a different link order.

Example 8 (page 47) illustrates the core file debugging for a stripped binary when the symbol table of the unstripped program is available.

Example 9 (page 49) illustrates the core file debugging for a stripped binary when the symbol table is available from another program, which uses the same symbols, but in a different link order.

## Debugging Core Files Created by Optimized or Stripped Binaries

All core file debugging features are available for unstripped binaries and shared libraries that are built using the `-g` option.

However, the following limitations apply for core files that are created by binaries that are compiled without the `-g` option, and for core files created by optimized (optimization level 2 or above) and stripped binaries.

### Limitations for Debugging Core Files Created by Optimized Binaries

The following limitations apply for core files that are created by optimized binaries (optimization level 2 or above) that are compiled with the `-g` option:

- Local variables and arguments in an optimized module are not displayed.
- The backtrace information displays the inlined functions. However, the line numbers are not displayed accurately at +O2 and higher levels of optimization.

For an illustration of these limitations, see "*Sample Debugging Session 1*" in Example 5 (page 25)

## Limitations for Debugging Core Files Created by Binaries Compiled Without the `-g` Option

The following limitations apply for core files that are created by binaries compiled without the `-g` option:

- Argument information in the stack traces is not displayed.
- Local variables and type information are not displayed.
- Inline frame information is not displayed. The source line information is not displayed for core files that are created by PA-RISC binaries.

  In the case of Integrity systems (Itanium-based binaries), the source line information is displayed for core files.

**NOTE:** In the case of core files that are created by Itanium-based binaries, the source line information is available, irrespective of whether the binary is compiled with the `-g` option, or not. To strip the line number information for Itanium-based binaries, you must use the `strip -l` command orr the `+nosrcpos` linker option.

For an illustration of these limitations, see “*Sample Debugging Session 2*” in Example 5 (page 25).

## Limitations for Debugging Core Files Created by Stripped Binaries

The following limitations apply for core files that are created by a stripped executable:

- Local variables and static variables in a stripped module are not displayed.
- Global variables and type information in a stripped module are not displayed. However, the debugger can access the global or local variables (within the scope of the variables) that are defined in other unstripped shared libraries, which are loaded in to the stripped executable.
- Argument information in the stack traces is not displayed.
- The static function names appearing in the stack traces are not displayed. The debugger may print random names instead of `<unknown_procedure>` while displaying these function names.
- In the case of core files created by PA-RISC binaries, the function names (static and non-static) appearing in the stack are not displayed.

**NOTE:**

- To avoid these limitations in debugging core files created by stripped binaries, you can use the original unstripped version of the executable, if it is available. For more information about debugging stripped binaries by using the symbol table from the unstripped version of the executable, see "Debugging Core Files Created by Stripped Binaries (When the Symbol Table is Available)" (page 22).

For an illustration of these limitations for core files created on an Integrity system, see "*Sample Debugging Session 3*" in Example 5 (page 25).

**Example 5 Debugging Core Files Created by Optimized Code, Stripped Binaries, and Code Compiled Without the `-g` Option**

## Sample Program

```
// a.c
// Generates coredump with 3 deep stack trace.

#include <stdio.h>

void
generate_core_dump ()
{
  int i = 0;
  *(int*)i = 10;
  printf ("Generated coredump\n");
  *(int*)i = 10;
}

void
foo (int arg_i)
{
  int local_j;
  if (arg_i == 10)
    {
      local_j = 5;
printf ("Hello World! Arg_i is 10 and
        local_j is %d\n",local_j);
    }
  else
    {
      local_j = 10;
      printf ("Hello World! Arg_i is not 10 and
              local_j is %d\n", local_j);
    }
  if (local_j == 5)
    generate_core_dump();
  printf ("Hello World\n");
}

int main()
{
 int local_i = 10;
 foo(local_i);
 return 0;
}
```

## Sample Debugging Session 1

### *Debugging a Core File Created by Optimized Code*

```
$ aCC -g -O a.c
$ /opt/langtools/bin/gdb ./a.out core
HP gdb for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
```

```
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest (based on GDB) is covered by the
GNU General Public License. Type "show copying" to see the conditions to
change it and/or distribute copies. Type "show warranty" for
warranty/support.
..
Core was generated by `a.out'.
Program terminated with signal 11, Segmentation fault.
SEGV_ACCERR - Invalid Permissions for object
#0  inline generate_core_dump () at a.c:11
11        printf ("Generated coredump\n");
(gdb) bt
#0  inline generate_core_dump () at a.c:11
#1  0x4000a00:0 in inline foo () at a.c:30
#2  0x40009b0:1 in main () at a.c:37
(gdb) up
#1  0x4000a00:0 in inline foo () at a.c:30
30          generate_core_dump();
(gdb) p local_j
No symbol "local_j" in current context.
```

The debugger cannot display information about the arguments and local variables because the program is compiled with the `-O` option (level 2 optimization). However, the debugger can display the inlined functions in the backtrace and provide the line number information. The line numbers may not be displayed accurately because the code is moved during optimization.

If you encounter issues while debugging inlined functions, you can use the `+d` compiler option to disable inlining, as follows:

```
$ aCC -g -O +d a.c
```

Examples on Integrity systems built without `-g` display significantly greater inlining and source line information than the same examples that are built on HP 9000 systems.

**Sample Debugging Session 2**

***Debugging a Core File Created by Code Compiled Without the `-g` Option***

```
$ aCC a.c

$ /opt/langtools/bin/gdb ./a.out core
HP gdb for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest (based on GDB) is covered by the
GNU General Public License. Type "show copying" to see the conditions to
change it and/or distribute copies. Type "show warranty" for
warranty/support.
..
Core was generated by `a.out'.
Program terminated with signal 11, Segmentation fault.
SEGV_ACCERR - Invalid Permissions for object
#0  0x40009b0:1 in generate_core_dump () at a.c:10
10        *(int*)i = 10;
(gdb) bt
#0  0x40009b0:1 in generate_core_dump () at a.c:10
#1  0x4000b40:0 in foo () at a.c:30
#2  0x4000bd0:0 in main () at a.c:37
(gdb) p local_j
No symbol "local_j" in current context.
(gdb)
```

In the case of core files created by PA-RISC - based binaries, the source line information is not available if the binary has not been compiled with the `-g` option. The information about the arguments and the local variables is not displayed.

In the case of core files created by Itanium-based binaries, the source line information is available, irrespective of whether the binary is compiled with the `-g` option, or not.

### Sample Debugging Session 3

#### *Debugging a Core File Created by a Stripped Binary When the Symbol Table is Not Available*

```
$ aCC -g a.c
$ strip a.out

$ /opt/langtools/bin/gdb ./a.out core
HP gdb for HP Itanium (32 or 64 bit) and target HP-UX 11.2x.
Copyright 1986 - 2001 Free Software Foundation, Inc.
Hewlett-Packard Wildebeest (based on GDB) is covered by the
GNU General Public License. Type "show copying" to see the conditions to
change it and/or distribute copies. Type "show warranty" for
warranty/support.
..
warning: Load module ./a.out has been stripped.
Debugging information is not available.

(no debugging symbols found)...
Core was generated by `a.out'.
Program terminated with signal 11, Segmentation fault.
SEGV_ACCERR - Invalid Permissions for object
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...#0  0x40009b0:1 in generate_core_dump+0x21
()
(gdb) bt
#0  0x40009b0:1 in generate_core_dump+0x21 ()
#1  0x4000b40:0 in foo+0x110 ()
#2  0x4000bd0:0 in main+0x20 ()
(gdb)
```

## Forcing a Core Dump

WDB enables you to force a core dump of a running process, and analyze the core file.

The `dumpcore` command forces a core dump and generates a core image file for a process that is running under the debugger. If no arguments are given, it saves the core image for the current debugged process in a file named `core.<pid>`, where `<pid>` is the process ID number.

Before debugging a forced core dump, you must enter the `set live-core 1` command at the `gdb` prompt. The `set live-core` command enables the debugging of a core file created by a forced core dump. Alternately, you can use the `--lcore` start-up option to debug a core file created by a forced core dump.

### Saving the Core File to a Specific File Name

You can specify a `<corefile-name>` as an option in the `dumpcore` command. This saves the dumped core file in the specified file, `<corefile-name>`, instead of `core.<pid>`.

To specify the filename as an option in the `dumpcore` command, enter the following command at the gdb prompt:

```
(gdb) dumpcore <corefile-name>
```

### Debugging a Core File Created by a Forced Core Dump

To debug a core file that is created by a forced core dump, complete the following steps:

1. To dump the core for a live process, enter the following command at the `gdb` prompt:

   ```
   (gdb) dumpcore
   ```

   For example:

   ```
   (gdb) run
   Starting program: sample
   Breakpoint 3, main () at sample.c:1010 b= foo(a);
   (gdb) dumpcore
   Dumping core to the core file core.24091
   (gdb)
   ```

2. To analyze the dumped core file, enter one of the following commands:
   - At gdb prompt:

     ```
     (gdb) core [core.<pid>|<corefile-name>]
     ```

     For example:

     ```
     (gdb) file sample
     Reading symbols from sample...done
     (gdb) set live-core 1
     (gdb) core core.24091
     Core was generated by 'sample'.
     ```

```
#0 main () at sample.c:1010 b = foo(a);
(gdb) backtrace
#0 main () at sample.c:10
(gdb)
```

(Or)

- At shell prompt:

  ```
  % gdb --lcore a.out [core.<pid>|<corefile-name>]
  ```

  For example:

  ```
  % ./gdb --lcore sample core.24091
  HP gdb  for PA-RISC (narrow), HP-UX 11.23. Copyright 1986 -
  2001 Free Software Foundation, Inc. Hewlett-Packard Wildebeest
  (based on GDB) is covered by the GNU General Public License.
  Type "show copying" to see the conditions to change it and/or
  distribute copies.
  Type "showwarranty" for warranty/support....

  Core was generated by 'sample'.
  #0 main() at sample.c:10
  (gdb)
  ```

## Debugging Core Files From a Different System

When debugging a core file, the debugger requires the exact versions of shared libraries and the executable that are associated with the core file.

Debugging a core file on a system other than the one on which it was originally produced is supported only under the following condition:

The correct system and user shared libraries are copied with the executable and core file to the other system, and the location of the shared libraries is defined by setting `GDB_SHLIB_PATH` or `GDB_SHLIB_ROOT` before debugging the core file.

For more information about these variables, see the *Debugging with GDB* manual available at:

http://www.hp.com/go/wdb

Table 3 lists the supported systems for debugging PA-RISC core files.

Core files produced by Integrity systems can be debugged on any Integrity system with an HP-UX version greater than or equal to the HP-UX version on the system where the core file was produced.

**Table 3 Supported Systems for PA-RISC Core File Debugging**

| Type of core files produced | Supported systems for debugging |
|---|---|
| Core files produced by 32–bit executables | Any PA-RISC 1.1 or PA-RISC 2.0 system with an HP-UX version greater than or equal to the HP-UX version on the system where the core file was produced. |
| Core files produced by 64–bit executables | Other PA-RISC 2.0 systems with HP-UX versions greater than or equal to the HP-UX version on the system where the core file was produced. |

Table 4 lists the commands for debugging a core file from a different system.

**Table 4 Commands for Debugging a Core File From a Different System**

| Command | Description |
|---|---|
| `packcore` | Packs the core file along with the relevant executable and libraries in a single `.tar` file. |
| `unpackcore` | Unpacks the `.tar` file generated by the `packcore` command. The debugger can use the executable and shared libraries from this bundle while debugging the core file on a system, which is different from the one on which the core file was originally created. |
| `getcore` | Enables you to examine a `packcore` directory, which was previously created by `unpackcore`. It takes one optional argument, which is the name of the`packcore` directory. |

To debug core files from a different system than the one on which the core file was created, complete the following steps:

1. Invoke WDB on the core file on the system where the core file was created.
2. Enter the `packcore` command to package the `.tar` file with the core file, the relevant libraries, and the relevant binaries.
3. Transfer the `.tar` file to the required system.
4. Enter the `unpackcore` command to unpack the `.tar` file on this system.
5. Start debugging the core file on this system.
6. If you exit from the debugging session, and must debug the same core file again, you can use the `getcore` command to examine the `packcore` directory, which was previously created by `unpackcore`. The `getcore` command accepts the name of the `packcore` directory as an argument.

## Displaying run time type information

HP WDB enables you to view the run time type information for C++ polymorphic objects. The following command displays the run time information for C++ polymorphic object.

```
info rtti <address>
```

The input to this command is address of the C++ polymorphic object. GDB displays de-mangled type name as output.

**NOTE:** This command is supported only on Integrity systems.

## Debugging PA-RISC Core Files on Integrity Systems

Using WDB, you can transparently debug PA-RISC programs running in compatibility mode under Aries on Integrity systems.

To debug a core file generated by a PA-RISC program on an Integrity system, complete the following steps:

1. Transfer the executable program, core file, and all shared libraries that are used by the PA-RISC application, to the target Integrity system.
2. Set the `GDB_SHLIB_PATH` environment variable to a colon-separated list of directory path names on the system where the transferred shared libraries reside.
3. Use WDB to examine the core file on the Integrity system.

## Avoiding Core File Corruption

You can prevent overwriting of core files from a different process by setting the kernel to store the core file in a process-specific file name, `<core.pid>` (where `pid` is the process ID of the process that dumped the core).

### Avoiding Core File Corruption for Applications Running HP-UX 11i v1 and HP-UX 11i v2

To prevent overwriting of core files from different processes for applications running HP–UX 11i v1 or 11i v2 operating systems, you must set the kernel parameter *core_addpid* to *1*. The core file is stored in a file name, `<core.pid>` in the current directory. To store core files in a specific filesystem, you must switch to the required directory (using the `cd` command) and then run the required application.

To set the kernel parameter to prevent core file corruption, complete the following steps:

1. Create the following script, `corepid`, as a superuser of the system before running the application:

   Following is the `corepid` script for HP-UX 11i v1 systems:

   ```
   # cat <path>/corepid
   case $1 in
   on) echo "core_addpid/W 1\ncore_addpid?W 1" | adb -w -k /stand/vmunix /dev/kmem;;
   off) echo "core_addpid/W 0\ncore_addpid?W 0" | adb -w -k /stand/vmunix /dev/kmem;;
   stat) echo "core_addpid/D\ncore_addpid?D" | adb -w -k /stand/vmunix /dev/kmem;;
   *) echo "usage $0: on|off|stat";;
   esac
   ```

   Following is the `corepid` script for HP-UX 11i v2 systems:

   ```
   # cat <path>/corepid
   case $1 in
   on) echo "core_addpid/W 1\ncore_addpid?W 1" | adb -o -w /stand/vmunix /dev/kmem;;
   off) echo "core_addpid/W 0\ncore_addpid?W 0" | adb -o -w /stand/vmunix /dev/kmem;;
   stat) echo "core_addpid/D\ncore_addpid?D" | adb -o -w /stand/vmunix /dev/kmem;;
   *) echo "usage $0: on|off|stat";;
   esac
   ```

2. To enable or disable the feature to store the core file in a specific file, `core.pid`, run the script, `corepid`, with the following parameter:

   ```
   #<path>/corepid[on|off]
   ```

3. To view the current settings for this feature, run the `corepid`, with the following parameter:

   ```
   #<path>/corepid [stat]
   ```

   The following example illustrates how to use this script on HP-UX 11i v1:

   ```
   #cat /tmp/corepid
   case $1 in
   on) echo "core_addpid/W 1\ncore_addpid?W 1" | adb -w -k /stand/vmunix /dev/kmem;;
   off) echo "core_addpid/W 0\ncore_addpid?W 0" | adb -w -k /stand/vmunix /dev/kmem;;
   stat) echo "core_addpid/D\ncore_addpid?D" | adb -w -k /stand/vmunix /dev/kmem;;
   *) echo "usage $0: on|off|stat";;
   esac
   #/tmp/corepid stat
   core_addpid:
   core_addpid: 0
   core_addpid:
   core_addpid: 0
   #/tmp/corepid on
   core_addpid: 0 = 1
   core_addpid: 0 = 1
   #/tmp/corepid stat
   core_addpid:
   core_addpid: 1
   core_addpid:
   core_addpid: 1
   #/tmp/corepid off
   core_addpid: 1 = 0
   core_addpid: 1 = 0
   ```

## Avoiding Core File Corruption for Applications Running HP-UX 11i v3

To prevent overwriting of core files from different processes for applications running in HP-UX 11i v3, you can use the `coreadm` commandl.

The `coreadm` command enables you to specify the location and pattern for core files that are created by abnormally terminating processes. This command can also be used to specify the path for the core file placement. In addition, it can be used to specify the process specific pattern for the file name of the core file.

For example, to set the global core file settings to include the process-ID and the system name in the file name of the core, `<core.pid>` and to place the core file in the specified path, `<path>`, you can enter the following command as a superuser at the HP-UX prompt:

```
  # coreadm -e global -g  <path>/core.%p.%n
```

For more information about using the `coreadm` command to avoid core file corruption, see *coreadm*(1M)

**NOTE:** This feature (to prevent core file corruption) is not required for forced core-dumps. In the case of forced core-dumps, the core files are stored in a file name, `<core.pid>`, by default.

## Java Corefile Debugging Support

HP WDB shows stack traces of mixed Java, C, and C++ programs for java `corefile`. The `GDB_JAVA_UNWINDLIB` environment variable must be set to the path name of the Java unwind library.

Following are examples that illustrate the gdb command-line options for invoking gdb on a core file:

1. Invoke gdb on a core file generated when running a 32-bit Java application on an Integrity system with `/opt/java1.4/bin/java`:

   ```
   $ gdb /opt/java1.4/bin/IA64N/java core.java
   ```

2. Invoke gdb on a core file generated when running a 64-bit Java application on an Integrity system with `/opt/java1.4/bin/java -d64`:

   ```
   $ gdb /opt/java1.4/bin/IA64W/java core.java
   ```

3. Invoke gdb on a core file generated when running a 32-bit Java application on PA-RISC using `/opt/java1.4/bin/java`:

   ```
   $ gdb /opt/java1.4/bin/PA_RISC2.0/java core.java
   ```

4. Invoke gdb on a core file generated when running a 64-bit Java application on PA-RISC using `/opt/java1.4/bin/java`:

   ```
   $ gdb /opt/java1.4/bin/PA_RISC2.0W/java core.java
   ```

When debugging a core file, it is good practice to rename the file from core to another name to avoid accidentally overwriting it.

If the Java and system libraries used by the failed application reside in non-standard locations, then the `GDB_SHLIB_PATH` environment variable must be set to specify the location of the libraries.

## Summary

WDB enables you to debug a core file and analyze the cause for the core dump. It also enables you to force a core dump of an application and analyze the process state of the application. In addition, you can debug a core file on a system that is different from the system on which the core file was created.

## Examples Illustrating Core File Debugging

The examples in this section are for core files that are created on an HP 9000 system (PA-RISC). If the program is not compiled with `-g`, the line number information is not available in the case of core files on PA-RISC systems.

On the contrary, the source line number information is available for core files created by Itanium-based binaries, irrespective of whether the core file is compiled with the `-g` option, or not.

The following examples illustrate how to use the common core file debugging commands in WDB:

(The examples are based on core files created by PA-RISC 32–bit binaries)

**Example 6 Debugging a Core File to Find the Values for Parameters of a Function When the Program is not Compiled with -g**

**Sample Program**

The sample program used in this example has multiple functions. The `function_abort()` function in this program causes the application to abort. This example illustrates how to debug this core file and find values for the parameters of `function_abort()`.

Following is the code for the structures in `function_abort()`:

```
extern int function_abort(struct st_one *, int);

struct st_two {
   char *a;
   int b;
   float c;
   char *d;
};
struct st_one {
   int one;
   char *two;
   struct st_two *three;
   int *four;
   char *five;
};
.
.
.
```

**Sample Debugging Session**

1. Invoke WDB on the core file, as follows:

   ```
   $gdb example core
   HP gdb Copyright 1986 - 2001 Free Software Foundation, Inc.
   Hewlett-Packard Wildebeest )Wildebeest is free software, covered
   by the GNU General Public License, and you are welcome to change it
   and/or distribute copies of it under certain conditions.
   Type "show copying" to see the conditions. There is
   absolutely no warranty for Wildebeest.
   Type "show warranty" for details.

   ..(no debugging symbols found)...
   Core was generated by `example'.
   Program terminated with signal 6, Aborted.
   warning: The shared libraries were not privately mapped; setting a
   breakpoint in a shared library will not work until you rerun the program.
   (no debugging symbols found)...(no debugging symbols found)...
   (no debugging symbols found)...#0 0xc01082b8 in kill () from /usr/lib/libc.2
   ```

2. View the backtrace information to analyze the flow of routines in the program that resulted in a program abort. The following backtrace information of the core file illustrates that `function_abort()` invoked the abort library call in `libc` to kill the process:

   ```
   (gdb) bt
   #0 0xc01082b8 in kill () from /usr/lib/libc.2
   #1 0xc00a52e8 in raise () from /usr/lib/libc.2
   #2 0xc00e5c8c in abort_C () from /usr/lib/libc.2
   #3 0xc00e5ce4 in abort () from /usr/lib/libc.2
   #4 0x2420 in function_abort () from /home/u492893/example/./example
   #5 0x23d0 in function_b () from /home/u492893/example/./example
   #6 0x23a0 in function_a () from /home/u492893/example/./example
   #7 0x2370 in main () from /home/u492893/example/./example
   ```

   This program is not compiled with the debug option. However, if parts of the program are compiled using the debug option, you can view information about the source file that contains this code, the line number of the code where the program crashed, and all the function parameters. You can also list the source code for the parts of the program that are compiled using the debug option.

3. Traverse the stack to view the call chain.

   Following are some of the basic commands for traversing the stack:

   - To traverse (up or down) the call chain, enter the `up` or `down` command as in the following example:

     ```
     (gdb) up
     #1 0xc00a52e8 in raise () from /usr/lib/libc.2
     ```

     To execute the previous command at gdb prompt, press **Enter** at the gdb prompt, as in the following example:

     ```
     (gdb)
     #2 0xc00e5c8c in abort_C () from /usr/lib/libc.2
     (gdb)
     #3 0xc00e5ce4 in abort () from /usr/lib/libc.2
     ```

```
(gdb)
#4 0x2420 in function_abort () from /home/u492893/example/./example
```

- You can also directly traverse the stack by entering the number of frames as an option in the `up` or `down` command, as follows:

```
(gdb) up 4
#4 0x2420 in function_abort () from /home/u492893/example/./example
(gdb) down 4
#0 0xc01082b8 in kill () from /usr/lib/libc.2
```

- To traverse the stack by using the frame number, enter the `frame` command, as in the following example:

```
(gdb) frame 4
#4 0x2420 in function_abort () from /home/u492893/example/./example
```

4. Disassemble the required calling function.

   To view information about the function prototype and the definition of the structures in the prototype, you must disassemble the required function. This displays the location of the stored function parameters if the function has stored the parameters.

   The following example illustrates the disassembly of `function_abort`:

```
(gdb) disassemble function_abort
Dump of assembler code for function function_abort:
0x23f4 <function_abort>: stw %rp,-0x14(%sr0,%sp)
0x23f8 <function_abort+4>: ldo 0x40(%sp),%sp
0x23fc <function_abort+8>: stw %r26,-0x64(%sr0,%sp)
0x2400 <function_abort+12>: stw %r25,-0x68(%sr0,%sp)
0x2404 <function_abort+16>: ldw -0x64(%sr0,%sp),%r20
0x2408 <function_abort+20>: ldw 8(%sr0,%r20),%r21
0x240c <function_abort+24>: ldi 0x63,%r22
0x2410 <function_abort+28>: stw %r22,4(%sr0,%r21)
0x2414 <function_abort+32>: ldil L'0x2000,%r31
0x2418 <function_abort+36>: be,l 0x3dc(%sr4,%r31)
0x241c <function_abort+40>: copy %r31,%rp
0x2420 <function_abort+44>: ldw -0x54(%sr0,%sp),%rp
0x2424 <function_abort+48>: bv %r0(%rp)
0x2428 <function_abort+52>: ldo -0x40(%sp),%sp
End of assembler dump.
```

   If the parameters are not stored on the stack, the task of reading the core file is similar to reading a kernel crash dump. In such cases, you must analyze the routines that are invoked before the required function and check if the parameters are passed up the stack by these routines. You must also check if these routines have saved the address or the value on the stack.

5. Analyze the assembler dump from the disassembly output.

   The first four arguments for a function are passed through registers for PA-RISC 32–bit binaries. However, the stack is not updated using these values. The invoked function saves the arguments to the stack, if required. If the function parameters are not passed up the stack, the value of the parameters are not available when you debug the core file.

You can analyze the following lines from the assembler dump to view information about the function parameters:

```
0x23fc <function_abort+8>: stw %r26,-0x64(%sr0,%sp)
0x2400 <function_abort+12>: stw %r25,-0x68(%sr0,%sp)
```

These lines provide information about the location of the function parameters in the stack. This calling convention for the function parameters is defined by the PA-RISC runtime architecture.

In the case of Itanium architecture, the arguments are normally passed through the stacked general registers, `gr32`, and `gr33`.

For example:

```
r26 == arg0, r25==arg1, r24==arg2, r23==arg3
```

The convenience variable, `$sp`, stores the stack pointer in WDB. The disassembly output for the function displays the addresses of the arguments that are relative to the stack pointer. Hence, `arg0` is stored at `$sp-0x64` on the stack and `arg1` is stored at `$sp-0x68`.

6. Examine the contents of the required memory location.

   The `x` command enables you to examine the contents at a specified memory location.

   You can use the `x` command to view the contents of `arg0` at `$sp-0x64`, and `arg1` at `$sp-0x68`.

   For example:

```
(gdb) x/x $sp-0x68
0x7f7e6738: 0x00000020
(gdb) x/x $sp-0x64
0x7f7e673c: 0x7f7e6688
```

7. To determine the value of the variables, you must analyze the contents of the required memory location.

   In this example, the value of `arg1` is an integer, and hence this value is `32(0x20)`.

   The argument, `arg0`, is a pointer to a structure. To arrive at the value of `arg0`, the offsets for the variables in the structure must be determined manually.

   Following are the offsets for the variables in the structure `st_one` (in case of the PA-RISC 32–bit binary):

   ```
   struct st_one {
      int one;              +0x0
      char *two;            +0x4
      struct st_two *three; +0x8
      int *four;            +0xC
      char *five;           +0x10
   };
   ```

   Following are two methods to determine the values in the structure:

   **NOTE:** If the debug information is available, these offsets can be displayed by using the `ptype -v` *`struct st_one`* command.

   - **Method 1:**

     In this method, the memory location of the fields in the structure `st_one` are calculated by determining the offsets for each field relative to the address location of `arg0`. The contents of the calculated address locations are displayed by using the `x` command.

     The following debugging session illustrates how to determine the values in the structure, `st_one`:

     — To display the `char*` value at the second field in the structure, enter the following command at the gdb prompt:

     ```
     (gdb) x/x 0x7f7e6688+0x4
     0x7f7e668c: 0x40001140
     ```

     To display the `string` value at the displayed address, enter the following command at the gdb prompt:

     ```
     (gdb) x/s 0x40001140
     0x40001140 <__d_trap_fptr+292>: "NOT!"
     ```

     — To display the `int*` value in the fourth field, enter the following command at the gdb prompt:

     ```
     (gdb) x/x 0x7f7e6688+0xc
     0x7f7e6694: 0x7f7e6688
     ```

     To display the `int` value at this address, enter the following command at the gdb prompt:

```
(gdb) x/x 0x7f7e6688
0x7f7e6688: 0x00000011
```

- To display the `char*` value at the last field in the structure, enter the following command at the gdb prompt:

  ```
  (gdb) x/x 0x7f7e6688+0x10
  0x7f7e6698: 0x40001120
  ```

  To display the `char` value at this address, enter the following command at the gdb prompt:

  ```
  (gdb) x/s 0x40001120
  0x40001120 <__d_trap_fptr+260>: "The meaning"
  ```

- To display the address of the `st_two` structure, enter the following command at the gdb prompt:

  ```
  (gdb) x/x 0x7f7e6688+0x8
  0x7f7e6690: 0x7f7e669c
  ```

Similarly, the offsets for the following structure `st_two` are calculated:

```
struct st_two {
   char *a;         +0x0
   int b;           +0x4
   float c;         +0x8
   char *d;         +0xC
};
```

The values of the variables in the structure `st_two` are determined by using these offsets, as follows:

— To examine the first word of the structure `st_two`, enter the following command at the gdb prompt:

```
(gdb) x/x 0x7f7e669c
0x7f7e669c: 0x40001130
```

To display the `string` value at this address, enter the following command at the gdb prompt:

```
gdb) x/s 0x40001130
0x40001130 <__d_trap_fptr+276>: "of life"
```

— To examine the second word of the structure `st_two`, enter the following command at the gdb prompt:

```
(gdb) x/x 0x7f7e669c+0x4
0x7f7e66a0: 0x00000063
```

— To display the `float` value at the third word, enter the following command at the gdb prompt:

```
(gdb) x/f 0x7f7e669c+0x8
0x7f7e66a4: 19.2099991
```

— To display the `char*` value at the fourth word, enter the following command at the gdb prompt:

```
(gdb) x/x 0x7f7e669c+0xc
0x7f7e66a8: 0x40001138
```

To display the `string` value at the displayed address, enter the following command at the gdb prompt:

```
(gdb) x/s 0x40001138
0x40001138 <__d_trap_fptr+284>: "is 42"
```

— To display the address of the structure, enter the following command at the gdb prompt:

```
(gdb) x/x $sp-0x64
0x7f7e673c: 0x7f7e6688
```

To display the `int` value at the start of the structure, enter the following command at the gdb prompt:

```
(gdb) x/x 0x7f7e6688
0x7f7e6688: 0x00000011
```

- **Method 2:**

  In this method, you can use the gdb convenience variables to store and manipulate memory addresses. You can use the `show conv` command to view the current values of the convenience variables. The nomenclature of all convenience variables is such that they start with the `$` symbol. The following debugging session illustrates this method:

  — To set `$my_arg0` as the value pointed by `$sp-0x64`, enter the following command at the gdb prompt:

  ```
  (gdb) set $my_arg0=*($sp-0x64)
  ```

  To examine the contents of `$my_arg0`, enter the following command at the gdb prompt:

  ```
  (gdb) x/x $my_arg0
  0x7f7e6688: 0x00000011
  ```

  — To display the `string` value at `$my_arg0+4`, enter the following command at the gdb prompt:

  ```
  (gdb) x/s *($my_arg0+4)
  0x40001140 <__d_trap_fptr+292>: "NOT!"
  ```

  — To store the value pointed by `$my_arg0+0x8` in `$xtra` , enter the following command at the gdb prompt:

  ```
  (gdb) set $xtra=*($my_arg0+0x8)
  ```

  — To display the `int` value pointed to by `$my_arg0+0xc` , enter the following command at the gdb prompt:

  ```
  (gdb) x/x *($my_arg0+0xc)
  0x7f7e6688: 0x00000011
  ```

  — To display the `string` value pointed to by `$my_arg0+0x10`, enter the following command at the gdb prompt:

  ```
  (gdb) x/s *($my_arg0+0x10)
  0x40001120 <__d_trap_fptr+260>: "The meaning"
  ```

  — To display the `string` value pointed to by `$xtra`, enter the following command at the gdb prompt:

  ```
  (gdb) x/s *($xtra)
  0x40001130 <__d_trap_fptr+276>: "of life"
  ```

  — To display the `int` value at `$xtra+0x4`, enter the following command at the gdb prompt:

  ```
  (gdb) x/x $xtra+0x4
  0x7f7e66a0: 0x00000063
  ```

  — To display the `float` value at `$xtra+0x8`, enter the following command at the gdb prompt:

```
(gdb) x/f $xtra+0x8
0x7f7e66a4: 19.2099991
```

— To display the `string` value pointed to by `$xtra+0xC`, enter the following command at the gdb prompt:

```
(gdb) x/s *($xtra+0xC)
0x40001138 <__d_trap_fptr+284>: "is 42"
```

**Example 7 Debugging a Core File to View Information on a Global Variable in a C program**

In this example, the address, `&global_vars`, of `global_vars` is required for debugging. If the required structure is a pointer, the address of the structure is not required. The address of the structure is cast to (`char*`) so that any increments to this address will be 1 byte.

The program in this example uses the global structure, `global_vars`.

Following is the global structure, `global_vars`:

```
struct gvals {
   char *program;       +0x0
   int arg_count;       +0x4
   char *first_arg;     +0x8
   char *path;          +0xC
   int secret;          +0x10
};
struct gvals global_vars;
```

### Sample Debugging Session

1. To store the address of `global_vars` in the convenience variable, `$glob`, enter the following command at the gdb prompt:
   ```
   (gdb) set $glob= (char*)&global_vars
   ```
2. To display the `string` value pointed to by `$glob+0x0`, enter the following command at the gdb prompt:
   ```
   (gdb) x /s *($glob+0x4)
   0x7f7e6000: "./example1"
   ```
3. To display the `int` value at `$glob+0x4`, enter the following command at the gdb prompt:
   ```
   (gdb) x/x $glob+0x1
   0x40001184 <global_vars+4>: 0x00000001
   ```
4. To display the `string` value pointed to by `$glob+0x8`, enter the following command at the gdb prompt:
   ```
   (gdb) x/s *($glob+0x8)
   0x0: Error accessing memory address 0x0: Invalid argument.
   ```

This indicates that the variable is a null pointer.

5. To display the string value pointed to by `$glob+0xC`, enter the following command at the gdb prompt:

```
(gdb) x/s *($glob+0xC)
0x7f7e62f9: "/opt/softbench/bin:/usr/bin:/opt/user/bin:
/opt/ansic/bin:/usr/ccs/bin:/usr/contrib/bin:/opt/net/bin:
/opt/fc/bin:/opt/fcms/bin:/opt/upgrade/bin:/opt/pd/bin:/usr/bin/X11:
/usr/contrib/bin/X11:/o"...
```

6. To display the `int` value at `$glob+0x10` in hexadecimal format, enter the following command at the gdb prompt:

```
gdb) x/x $glob+0x10
0x40001190 <global_vars+16>: 0x0001b669
```

7. To display the `int` value at `$glob+0x10` in decimal format, enter the following command:

```
((gdb) x/d $glob+0x10
0x40001190 <global_vars+16>: 112233
```

**Example 8 Debugging a Core File Created by a Stripped Binary When the Symbol Table is Available**

### Sample Program

The program in this example has the following global structure, `global_vars`:

```
struct gvals {
   char *program;      +0x0
   int arg_count;      +0x4
   char *first_arg;    +0x8
   char *path;         +0xC
   int secret;         +0x10
};
struct gvals global_vars;
```

### Sample Debugging Session

This sample debugging session illustrates how to debug a core file that is created by the stripped binary of this program.

```
$ nm -x example | grep global_var
global_vars |0x40001180|extern|data |$BSS$
$ strip example
$ ./example
Abort(core dump)
$ gdb example core
HP gdb
...
..(no debugging symbols found)...
Core was generated by `example'.
Program terminated with signal 6, Aborted.
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...#0 0xc01082b8 in kill () from /usr/lib/libc.2
(gdb) bt
#0 0xc01082b8 in kill () from /usr/lib/libc.2
#1 0xc00a52e8 in raise () from /usr/lib/libc.2
#2 0xc00e5c8c in abort_C () from /usr/lib/libc.2
#3 0xc00e5ce4 in abort () from /usr/lib/libc.2
#4 0x2394 in <unknown_procedure> () from /home/u492893/examples/./example
```

The addresss obtained from the output from `nm` command works only for the main module of the binary. In the case of the shared libraries, the relocation offset (due to the relocation of the addresses) must be applied to the addresses that displayed as output for the `nm`command.

The debugger does not provide the function names for stripped binaries, only the program counter (`PC`) is displayed.

However, you can use the symbol information from the unstripped program for debugging. The symbol table is displayed as output from the `nm` command for the unstripped program.

In this example, the address of the global variable `global_vars` (`0x40001180`) is displayed as output from the `nm` command, and this address is used for debugging the core file.

```
(gdb) set $glob=0x40001180
(gdb) x/s *($glob+0x0)
0x7f7e6000: "./example"
(gdb) x/x $glob+0x4
0x40001184: 0x00000001
(gdb) x/s *($glob+0x8)
0x0: Error accessing memory address 0x0: Invalid argument.
(gdb) x/s *($glob+0xc)
0x7f7e62f9: "/opt/softbench/bin:/usr/bin:/opt/butthead/bin:/opt/an
sic/bin:/usr/ccs/bin:/usr/contrib/bin:/opt/nettladm/bin:/opt/fc/bi
n:/opt/fcms/bin:/opt/upgrade/bin:/opt/pd/bin:/usr/bin/X11:/usr/con
trib/bin/X11:/o"...
(gdb) x/d $glob+0x10
0x40001190: 112233
```

### Example 9 Debugging of a Core File Created by a Stripped Binary When the Symbol Table is Available from Another Program

In this example, three copies of a program (program `a1`, program `a2`, and program `a3` ) are compiled and linked with a different order.

For example:

Program `a1` is stripped.

Program `a2` is an unstripped copy of program `a1`.

Program `a3` is functionally the same as program `a1`. However, the code and the symbols are in a different link order.

Using the symbol information from `a3` to debug the core file generated by `a1` does not provide reliable symbol information as illustrated in the following example:

```
$ aCC main.c a.c b.c -o a1
$ aCC b.c main.c a.c -o a3
$ cp a1 a2
$ strip a1
$ ./a1
Abort(core dump)
$ gdb a1 core
HP gdb
...  (Some output dropped)
Core was generated by `a1'.
Program terminated with signal 6, Aborted.
warning: Unable to find __dld_flags symbol in object file.
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...#0 0xc01f2740 in kill () from /usr/lib/libc.2
```

The backtrace of `a1` does not display information about the routines, because the program is stripped.

The symbol information of `a2` can be used to analyze the backtrace from `a1`.

The symbol information from `a3` does not provide reliable results, because the link order is different. Unless the program `a3` has a similar link order, the symbol information is not reliable for debugging the core file created by `a1`. The version of the compiler and the compiler options used can also alter the reliability of this approach.

The following example shows the backtrace for `a1`, `a2`, and `a3`.

```
(gdb) bt
#0 0xc01f2740 in kill () from /usr/lib/libc.2
#1 0xc018fc94 in raise () from /usr/lib/libc.2
#2 0xc01d00dc in abort_C () from /usr/lib/libc.2
#3 0xc01d0134 in abort () from /usr/lib/libc.2
#4 0x2498 in <unknown_procedure> () from /home/shane/test/./a1
#5 0x2430 in <unknown_procedure> () from /home/shane/test/./a1
(gdb) symbol a2
Reading symbols from a2...(no debugging symbols found)...done.
(gdb) bt
#0 0xc01f2740 in kill () from /usr/lib/libc.2
#1 0xc018fc94 in raise () from /usr/lib/libc.2
```

```
#2 0xc01d00dc in abort_C () from /usr/lib/libc.2
#3 0xc01d0134 in abort () from /usr/lib/libc.2
#4 0x2498 in b () from /home/shane/test/./a1
#5 0x2430 in main () from /home/shane/test/./a1
(gdb) symbol a3
Reading symbols from a3...(no debugging symbols found)...done.
(gdb) bt
#0 0xc01f2740 in kill () from /usr/lib/libc.2
#1 0xc018fc94 in raise () from /usr/lib/libc.2
#2 0xc01d00dc in abort_C () from /usr/lib/libc.2
#3 0xc01d0134 in abort () from /usr/lib/libc.2
#4 0x2498 in a () from /home/shane/test/./a1
#5 0xc018fc94 in raise () from /usr/lib/libc.2
```

If the program is built with debug support, you can use the symbol information from this program to debug the stripped version of the program.

**Example 10 Core File Debugging Session for a Stripped Binary When the Symbol Table is Available from Another Program**

This example is similar to Example 9 (page 49). This example illustrates the debugging of a stripped binary when the symbol table is available from another program that uses the same symbols.

The programs, `example.c` and `example2.c`, have the same symbol table.

**Sample Program 1**

```
$ cat example.c
#include <stdio.h>
#include <stdlib.h>

struct st_two {
    char *a;
    int b;
    float c;
    char *d;
};

struct st_one {
    int one;
    char *two;
    struct st_two *three;
    int *four;
    char *five;
};

extern int function_a(struct st_one *, int);
extern int function_b(int, struct st_one *);
extern int function_abort(struct st_one *, int);

main()
{
    char *temp1="The meaning";
    char *temp2="of life";
    char *temp3="is 42";
    char *temp4="NOT!";
    struct st_one one;
    struct st_two two;

    one.one=17;
    one.two=temp4;
    one.three=&two;
    one.four=&one.one;
    one.five=temp1;
    two.a=temp2;
    two.b=42;
    two.c=19.21;
    two.d=temp3;
    function_a(&one, 32);
}

int function_a(struct st_one *a, int b)
{
    function_b(b, a);
}

int function_b(int a, struct st_one *b)
{
    function_abort(b, a);
}

int function_abort(struct st_one *a, int b)
{
    a->three->b=99;
    abort();
}
```

**Sample Program 2**

```
$ cat example2.c
struct st_two {
    char *a;
    int b;
    float c;
    char *d;
};

struct st_one {
    int one;
    char *two;
    struct st_two *three;
    int *four;
    char *five;
};

main()
{

}
```

**Sample Debugging Session**

In this example,`example.c` is compiled and stripped. The program,`example2.c`, is compiled with the `-g` option. The symbol table from `example2` is used to debug the core file that is created by the stripped executable,`example`, as illustrated in the following debugging session:

```
$ aCC -g example2.c -o example2
$ aCC -g example.c -o example
$ strip example
$ ./example
Abort(coredump)
$ gdb example core
HP gdb
...
Core was generated by `example'.
Program terminated with signal 6, Aborted.
warning: The shared libraries were not privately mapped; setting a
breakpoint in a shared library will not work until you rerun the program.
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...#0 0xc01082b8 in kill () from /usr/lib/libc.2
(gdb) bt
#0 0xc01082b8 in kill () from /usr/lib/libc.2
#1 0xc00a52e8 in raise () from /usr/lib/libc.2
#2 0xc00e5c8c in abort_C () from /usr/lib/libc.2
#3 0xc00e5ce4 in abort () from /usr/lib/libc.2
#4 0x2428 in function_abort () from /home/u492893/example/temp/./example
#5 0x23d8 in function_b () from /home/u492893/example/temp/./example
#6 0x23a8 in function_a () from /home/u492893/example/temp/./example
#7 0x2378 in main () from /home/u492893/example/temp/./example
(gdb) up
#1 0xc00a52e8 in raise () from /usr/lib/libc.2
(gdb)
#2 0xc00e5c8c in abort_C () from /usr/lib/libc.2
(gdb)
#3 0xc00e5ce4 in abort () from /usr/lib/libc.2
(gdb)
#4 0x2428 in function_abort () from /home/u492893/example/temp/./example
(gdb) x/x $sp
0x7f7e67b0:
0x00000001
```

The stack is traversed such that the stack pointer reflects the address of the required function.

The symbol table from `example2` is loaded for debugging the core file generated by `example`, as follows:

```
(gdb) symbol example2
Reading symbols from example2...done.
(gdb) x/x $sp
0x7f7e68f0:
0x7f7d27c0
```

The stack pointer value in `$sp` is changed when the new symbol table is loaded. Hence, you must keep track of the earlier values of `$sp` manually.

When a new symbol table is loaded, the values stored in the gdb convenience variables are changed. Hence you cannot store the value of `$sp` in the gdb convenience variables.

After the new symbol table is loaded, you can use the gdb features which were not previously available for the stripped executable.

You can use the print command to print the values of the structure `struct st_one` , as follows:

```
(gdb) p *(struct st_one *)(*(0x7f7e67b0-100))
$1 = {one = 17, two = 0x40001140 "NOT!", three = 0x7f7e66ac,
    four = 0x7f7e6698, five = 0x40001120 "The meaning"}
(gdb) x/x (0x7f7e67b0-100)
0x7f7e674c:
0x7f7e6698
(gdb) x/x (*(0x7f7e67b0-100))
0x7f7e6698:
0x00000011
```

From previous disassembling we know that the pointer to the `struct st_one` was stored at `-0x64 (-100)` on from `$sp` in the `function_abort` routine. However, the pointer to the structure is stored on the stack.

Taking the offset from the stack pointer into account, we must deference the pointer twice to get from the stack pointer to the structure, and subsequently dereference this structure to print all the members of this structure.

```
(gdb) p *(struct st_two *)(0x7f7e66ac)
$2 = {a = 0x40001130 "of life", b = 99, c = 19.2099991, d = 0x40001138 "is 42"}
(gdb) p *(struct st_two *)($1.three)
$3 = {a = 0x40001130 "of life", b = 99, c = 19.2099991, d = 0x40001138 "is 42"}
```

```
(gdb) bt
#0 0xc01082b8 in kill () from /usr/lib/libc.2
#1 0xc00a52e8 in raise () from /usr/lib/libc.2
#2 0xc00e5c8c in abort_C () from /usr/lib/libc.2
#3 0xc00e5ce4 in abort () from /usr/lib/libc.2
#4 0x2428 in sigismember () from /home/u492893/example/temp/./example
#5 0x23d8 in main () at example2.c:18
#6 0x23a8 in lwp_setprivate () from /home/u492893/example/temp/./example
#7 0xc00a52e8 in raise () from /usr/lib/libc.2
#8 0x40001140 in __d_trap_fptr ()
#9 0xc00a52e8 in raise () from /usr/lib/libc.2
Cannot access memory at address 0xffffffad.
(gdb) symbol example
Load new symbol table from "example"? (y or n) y
Reading symbols from example...
(no debugging symbols found)...done.
(gdb) bt
#0 0xc01082b8 in kill () from /usr/lib/libc.2
#1 0xc00a52e8 in raise () from /usr/lib/libc.2
#2 0xc00e5c8c in abort_C () from /usr/lib/libc.2
#3 0xc00e5ce4 in abort () from /usr/lib/libc.2
#4 0x2428 in function_abort () from /home/u492893/example/temp/./example
#5 0x23d8 in function_b () from /home/u492893/example/temp/./example
#6 0x23a8 in function_a () from /home/u492893/example/temp/./example
#7 0x2378 in main () from /home/u492893/example/temp/./example
```

## FAQ

1 Are shared memory segments dumped in the core file by default?

No.

However, the shared memory segments can be dumped into the core file if you set the following kernel symbols:

- *core_addshmem_read*

  This kernel symbol controls if shared memory segments that are mapped read-only into a process are dumped in a core file. To view the current value of the tunable and change it , enter the following commands:

```
# echo 'core_addshmem_read/X'|adb -k /stand/vmunix /dev/kmem
core_addpid:
core_addpid: 0
# echo 'core_addshmem_read/W 1'|adb -k -w /stand/vmunix /dev/kmem
core_addpid: 0 = 1
```

- *core_addshmem_write*

  This kernel symbol controls if shared memory segments that are mapped read/write into a process are dumped in a core file. To view the current value of the tunable and change it , enter the following commands:

```
# echo 'core_addshmem_write/X'|adb -k /stand/vmunix /dev/kmem
core_addpid:
core_addpid: 0
# echo 'core_addshmem_write/W 1'|adb -k -w /stand/vmunix /dev/kmem
core_addpid: 0 = 1
```

**NOTE:**

- HP does not provide complete support for these kernel symbols.
- The large size of the shared memory segments must be taken into account before using these kernel symbols.

2 How do I check if the system is enabled for creating core files with sizes greater than 2 GB? How do I enable the system for creating core files with sizes greater than 2 GB?

To check if the system is enabled for creating core files with sizes greater than 2 GB, enter the following command:

```
fsadm <filesystem>
```

If this command displays “`largefiles`”, the system is enabled for creating core files greater than 2 GB.

To enable a system for creating core files greater than 2 GB, enter the following command:

```
    fsadm -o largefiles <filesystem>
```

3 How do I verify if a core file is truncated?

To verify if a core file is truncated, enter the following command:

```
elfdump -o -S core
```

If the core file is not truncated or corrupted, the output of the `elfdump -o -S core` is as follows:

```
core:
                *** Program Header ***
Type     Offset           Vaddr            FSize            Memsz
CoreVer  0000000000003028 0000000000000000 0000000000000004 0000000000000004
CoreKern 000000000000302c 0000000000000000 000000000000003c 000000000000003c
CoreComm 0000000000003068 0000000000000000 000000000000000a 000000000000000a
CoreProc 0000000000003078 0000000000000000 000000000000be00 000000000000be00
CoreLoad 000000000000ee78 6000000000000000 0000000005160000 0000000005160000
CoreMMF  000000000516ee78 9fffffffdd6f4000 0000000000004000 0000000000004000
.
.
.
CoreStck 000000001725fe78 9fffffffef7ff000 0000000000009000 0000000000009000
CoreStck 0000000017268e78 9fffffffffec0000 0000000000140000 0000000000140000
```