Debugging with DDD

User's Guide and Reference Manual
First Edition, for DDD Version 3.3.11
Last updated 15 January, 2004

Andreas Zeller

Debugging with DDD
User's Guide and Reference Manual

Distributed by
Free Software Foundation, Inc.
59 Temple Place – Suite 330
Boston, MA 02111-1307
USA

DDD and this manual are available via
the DDD WWW page (`http://www.gnu.org/software/ddd/`).

Send questions, comments, suggestions, etc. to `ddd@gnu.org`.
Send bug reports to `bug-ddd@gnu.org`.

Short Contents

Table of Contents

Summary of DDD

The purpose of a debugger such as DDD is to allow you to see what is going on "inside" another program while it executes—or what another program was doing at the moment it crashed.

DDD can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Technically speaking, DDD is a front-end to a command-line debugger (called *inferior debugger*, because it lies at the layer beneath DDD). DDD supports the following inferior debuggers:

- To debug *executable binaries*, you can use DDD with GDB, DBX, *Ladebug*, or XDB.
  - GDB, the GNU debugger, is the recommended inferior debugger for DDD. GDB supports native executables binaries originally written in C, C++, Java, Modula-2, Modula-3, Pascal, Chill, Ada, and FORTRAN. (see section "Using GDB with Different Languages" in *Debugging with* GDB, for information on language support in GDB.)
  - As an alternative to GDB, you can use DDD with the DBX debugger, as found on several UNIX systems. Most DBX incarnations offer fewer features than GDB, and some of the more advanced DBX features may not be supported by DDD. However, using DBX may be useful if GDB does not understand or fully support the debugging information as generated by your compiler.
  - As an alternative to GDB and DBX, you can use DDD with *Ladebug*, as found on Compaq and DEC systems. Ladebug offers fewer features than GDB, and some of the more advanced Ladebug features may not be supported by DDD. However, using Ladebug may be useful if GDB or DBX do not understand or fully support the debugging information as generated by your compiler.[1]
  - As another alternative to GDB, you can use DDD with the XDB debugger, as found on HP-UX systems.[2].
- To debug *Java byte code programs*, you can use DDD with JDB, the Java debugger, as of JDK 1.1 and later. (DDD has been tested with JDK 1.1 and JDK 1.2.)
- To debug *Python programs*, you can use DDD with PYDB, a Python debugger.
- To debug *Perl programs*, you can use DDD with the *Perl debugger*, as of Perl 5.003 and later.
- To debug *Bash programs*, you need a version Bash that supports extended debugging support. To get this enhanced version see `http://bashdb.sourceforge.net`. You will need version 2.05b-debugger-0.32 or later to work with DDD.

See Section 2.1.1 [Choosing an Inferior Debugger], page 15, for choosing the appropriate inferior debugger. See Chapter 1 [Sample Session], page 5, for getting a first impression of DDD.

---

[1] Within DDD (and this manual), Ladebug is considered a DBX variant. Hence, everything said for DBX also applies to Ladebug, unless stated otherwise.

[2] XDB will no longer be maintained in future DDD releases. Use a recent GDB version instead.

About this Manual

This manual comes in several formats:

- The *Info* format is used for browsing on character devices; it comes without pictures. You should have a local copy installed, which you can browse via Emacs, the stand-alone `info` program, or from DDD via '`Help ⇒ DDD Reference`'.

  The DDD source distribution '`ddd-3.3.11.tar.gz`' contains this manual as pre-formatted info files; you can also download them from the DDD WWW page (`http://www.gnu.org/software/ddd/`).
- The *PostScript* format is used for printing on paper; it comes with pictures as well.

  The DDD source distribution '`ddd-3.3.11.tar.gz`' contains this manual as pre-formatted PostScript file; you can also download it from the DDD WWW page (`http://www.gnu.org/software/ddd/`).
- The *PDF* format is used for printing on paper as well as for online browsing; it comes with pictures as well.

  The DDD source distribution '`ddd-3.3.11.tar.gz`' contains this manual as pre-formatted PDF file; you can also download it from the DDD WWW page (`http://www.gnu.org/software/ddd/`).
- The *HTML* format is used for browsing on bitmap devices; it includes several pictures. You can view it using a HTML browser, typically from a local copy.

  A pre-formatted HTML version of this manual comes in a separate DDD package '`ddd-3.3.11-html-manual.tar.gz`'; you can browse and download it via the DDD WWW page (`http://www.gnu.org/software/ddd/`).

The manual itself is written in TeXinfo format; its source code '`ddd.texi`' is contained in the DDD source distribution '`ddd-3.3.11.tar.gz`'.

The picture sources come in a separate package '`ddd-3.3.11-pics.tar.gz`'; you need this package only if you want to re-create the PostScript, HTML, or PDF versions.

Typographic conventions

⟨Ctrl+A⟩ The name for a key on the keyboard (or multiple keys pressed simultaneously)

`run` A sequence of characters to be typed on the keyboard.

'`~/.ddd/init`'
A file.

'`Help`' A graphical control element, such as a button or menu item.

'`File ⇒ Open Program`'
A sequence of menu items, starting at the top-level menu bar.

`argc - 1` Program code or debugger command.

'`-g`' A command-line option.

`$` System prompt.

`(gdb)` Debugger prompt.

_ Cursor position.

version A metasyntactic variable; something that stands for another piece of text.

definition A definition.

caution Emphasis.

A warning Strong emphasis.

DDD An acronym.

Here's an example. '***break*** *location*' is a typed command at the '`(gdb) `' prompt; the metasyntactic variable '*location*' would be replaced by the actual location. '_' is the cursor position after entering the command.

```
(gdb) break location
Breakpoint number at location
(gdb) _
```

Free software

DDD is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. DDD is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of DDD that they might get from you. The precise conditions are found in the GNU General Public License that comes with DDD; See Appendix G [License], page 187, for details.

The easiest way to get a copy of DDD is from someone else who has it. You need not ask for permission to do so, or tell any one else; just copy it.

Getting DDD

If you have access to the Internet, you can get the latest version of DDD from the anonymous FTP server '`ftp.gnu.org`' in the directory '`/gnu/ddd`'. This should contain the following files:

''`ddd-`*version*`.tar.gz`''
: The DDD source distribution. This should be all you need.

''`ddd-`*version*`-html-manual.tar.gz`''
: The DDD manual in HTML format. You need this only if you want to install a local copy of the DDD manual in HTML format.

''`ddd-`*version*`-pics.tar.gz`''
: Sources of images included in the DDD manual. You need this only if you want to recreate the DDD manual.

DDD can also be found at numerous other archive sites around the world; check the file '`ANNOUNCE`' in a DDD distribution for the latest known list.

Contributors to DDD

Dorothea Lütkehaus and Andreas Zeller were the original authors of DDD. Many others have contributed to its development. The files '`ChangeLog`' and '`THANKS`' in the DDD distribution approximates a blow-by-blow account.

History of DDD

The history of DDD is a story of code recycling. The oldest parts of DDD were written in 1990, when *Andreas Zeller* designed VSL, a box-based visual structure language for visualizing data and program structures. The VSL interpreter and the Box library became part of Andreas' Diploma Thesis, a graphical syntax editor based on the Programming System Generator PSG.

In 1992, the VSL and Box libraries were recycled for the NORA project. For NORA, an experimental inference-based software development tool set, Andreas wrote a graph editor (based on VSL and the Box libraries) and facilities for inter-process knowledge exchange. Based on these tools, *Dorothea Lütkehaus* (now *Dorothea Krabiell*) realized DDD as her Diploma Thesis, 1994.

The original DDD had no source window; this was added by Dorothea during the winter of 1994–1995. In the first quarter of 1995, finally, Andreas completed DDD by adding command and execution windows, extensions for DBX and remote debugging as well as configuration support for several architectures. Since then, Andreas has further maintained and extended DDD, based on the comments and suggestions of several DDD users around the world. See the comments in the DDD source for details.

Major DDD events:

April, 1995
: DDD 0.9: First DDD beta release.

May, 1995
: DDD 1.0: First public DDD release.

December, 1995
: DDD 1.4: Machine-level debugging, glyphs, Emacs integration.

October, 1996
: DDD 2.0: Color displays, XDB support, generic DBX support, command tool.

May, 1997
: DDD 2.1: Alias detection, button tips, status displays.

November, 1997
: DDD 2.2: Sessions, display shortcuts.

June, 1998
: DDD 3.0: Icon tool bar, Java support, JDB support.

December, 1998
: DDD 3.1: Data plotting, Perl support, Python support, Undo/Redo.

January, 2000
: DDD 3.2: New manual, Readline support, Ladebug support.

January, 2001
: DDD 3.3: Data themes, JDB 1.2 support, VxWorks support.

November, 2002
: DDD 3.3.2: Bash support.

March, 2003
: DDD 3.3.3: Better Bash support. Compiles using modern tools thanks to Daniel Schepler.

1 A Sample DDD Session

You can use this manual at your leisure to read all about DDD. However, a handful of features are enough to get started using the debugger. This chapter illustrates those features.

The sample program 'sample.c' (see Section 1.1 [Sample Program], page 14) exhibits the following bug. Normally, sample should sort and print its arguments numerically, as in the following example:

```
$ ./sample 8 7 5 4 1 3
1 3 4 5 7 8
$ _
```

However, with certain arguments, this goes wrong:

```
$ ./sample 8000 7000 5000 1000 4000
1000 1913 4000 5000 7000
$ _
```

Although the output is sorted and contains the right number of arguments, some arguments are missing and replaced by bogus numbers; here, 8000 is missing and replaced by 1913.[1]

Let us use DDD to see what is going on. First, you must compile 'sample.c' for debugging (see Section 4.1 [Compiling for Debugging], page 73), giving the '-g' flag while compiling:

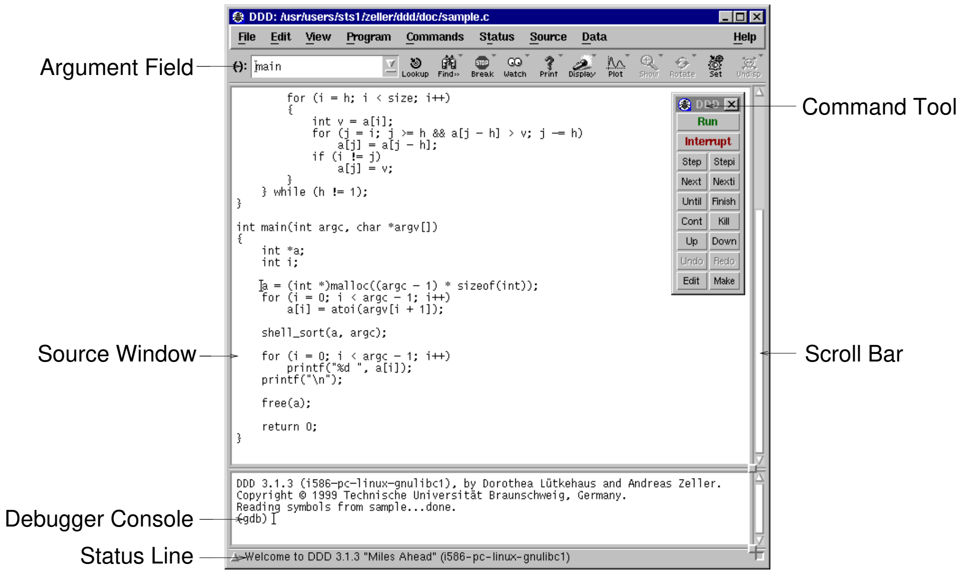
```
$ gcc -g -o sample sample.c
$ _
```

Now, you can invoke DDD (see Chapter 2 [Invocation], page 15) on the sample executable:

```
$ ddd sample
```

[1] Actual numbers and behavior on your system may vary.

After a few seconds, DDD comes up. The *Source Window* contains the source of your debugged program; use the *Scroll Bar* to scroll through the file.



Initial DDD Window

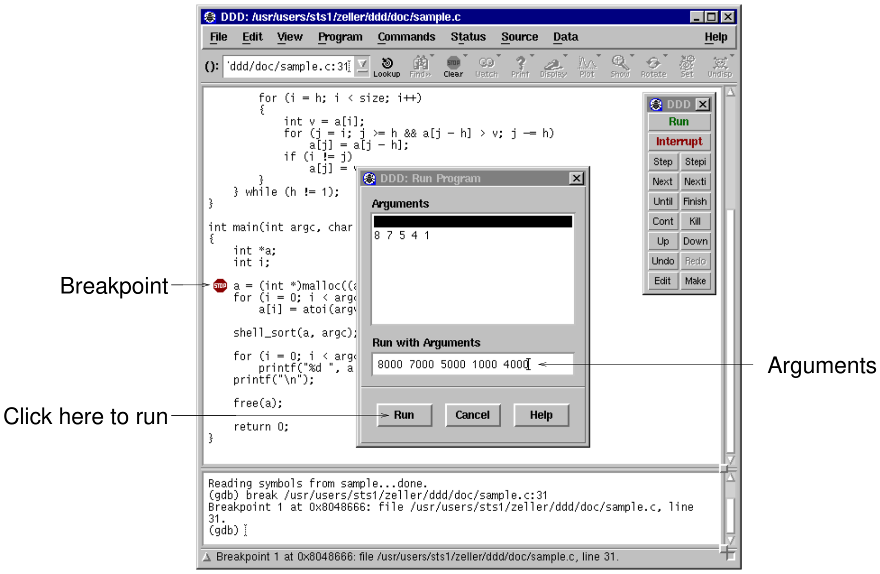
The *Debugger Console* (at the bottom) contains DDD version information as well as a GDB prompt.[1]

```
GNU DDD Version 3.3.11, by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001-2004 Universität des Saarlandes, Germany.
Reading symbols from sample...done.
(gdb) _
```

The first thing to do now is to place a *Breakpoint* (see Section 5.1 [Breakpoints], page 81), making `sample` stop at a location you are interested in. Click on the blank space left to the initialization of `a`. The *Argument field* '`():`' now contains the location ('`sample.c:31`'). Now, click on '`Break`' to create a breakpoint at the location in '`()`'. You see a little red stop sign appear in line 31.

[1] Re-invoke DDD with '`--gdb`', if you do not see a '`(gdb)`' prompt here (see Section 2.1.1 [Choosing an Inferior Debugger], page 15)

The next thing to do is to actually *execute* the program, such that you can examine its behavior (see Chapter 6 [Running], page 91). Select '`Program` ⇒ `Run`' to execute the program; the '`Run Program`' dialog appears.



Running the Program

In '`Run with Arguments`', you can now enter arguments for the `sample` program. Enter the arguments resulting in erroneous behavior here—that is, '`8000 7000 5000 1000 4000`'. Click on '`Run`' to start execution with the arguments you just entered.

`GDB` now starts `sample`. Execution stops after a few moments as the breakpoint is reached. This is reported in the debugger console.

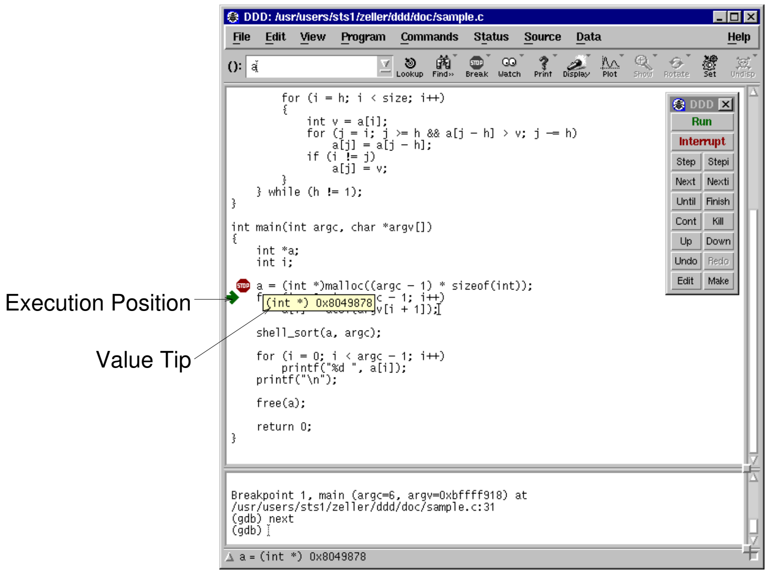
```
(gdb) break sample.c:31
Breakpoint 1 at 0x8048666: file sample.c, line 31.
(gdb) run 8000 7000 5000 1000 4000
Starting program: sample 8000 7000 5000 1000 4000

Breakpoint 1, main (argc=6, argv=0xbffff918) at sample.c:31
(gdb) _
```

The current execution line is indicated by a green arrow.

```
⇒ a = (int *)malloc((argc - 1) * sizeof(int));
```

You can now examine the variable values. To examine a simple variable, you can simply move the mouse pointer on its name and leave it there. After a second, a small window with the variable value pops up (see Section 7.1 [Value Tips], page 105). Try this with '`argc`' to see its value (`6`). The local variable '`a`' is not yet initialized; you'll probably see `0x0` or some other invalid pointer value.

To execute the current line, click on the '`Next`' button on the command tool. The arrow advances to the following line. Now, point again on '`a`' to see that the value has changed and that '`a`' has actually been initialized.



Viewing Values in DDD

To examine the individual values of the '`a`' array, enter '`a[0]`' in the argument field (you can clear it beforehand by clicking on '`():`') and then click on the '`Print`' button. This prints the current value of '`()`' in the debugger console (see Section 7.2 [Printing Values], page 106). In our case, you'll get

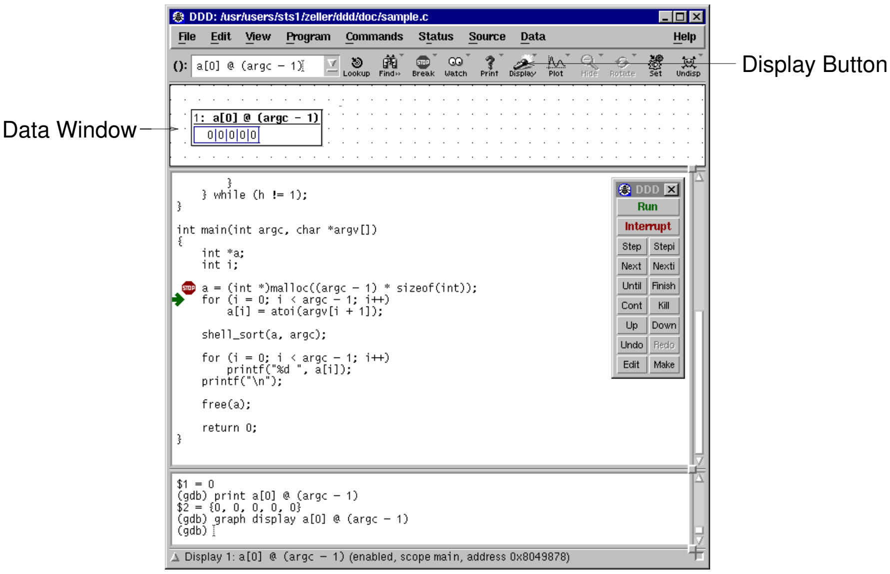
```
(gdb) print a[0]
$1 = 0
(gdb) _
```

or some other value (note that '`a`' has only been allocated, but the contents have not yet been initialized).

To see all members of '`a`' at once, you must use a special GDB operator. Since '`a`' has been allocated dynamically, GDB does not know its size; you must specify it explicitly using the '`@`' operator (see Section 7.3.2.1 [Array Slices], page 117). Enter '`a[0]@(argc - 1)`' in the argument field and click on the '`Print`' button. You get the first `argc - 1` elements of '`a`', or

```
(gdb) print a[0]@(argc - 1)
$2 = {0, 0, 0, 0, 0}
(gdb) _
```

Rather than using ‘Print’ at each stop to see the current value of ‘a’, you can also *display* ‘a’, such that its is automatically displayed. With ‘a[0]@(argc - 1)’ still being shown in the argument field, click on ‘Display’. The contents of ‘a’ are now shown in a new window, the *Data Window*. Click on ‘Rotate’ to rotate the array horizontally.



Data Window

Now comes the assignment of ‘a’’s members:

```
⇒  for (i = 0; i < argc - 1; i++)
       a[i] = atoi(argv[i + 1]);
```

You can now click on ‘Next’ and ‘Next’ again to see how the individual members of ‘a’ are being assigned. Changed members are highlighted.

To resume execution of the loop, use the ‘Until’ button. This makes GDB execute the program until a line greater than the current is reached. Click on ‘Until’ until you end at the call of ‘shell_sort’ in

```
⇒  shell_sort(a, argc);
```

At this point, ‘a’’s contents should be ‘8000 7000 5000 1000 4000’. Click again on ‘Next’ to step over the call to ‘shell_sort’. DDD ends in

```
⇒  for (i = 0; i < argc - 1; i++)
       printf("%d ", a[i]);
```

and you see that after ‘shell_sort’ has finished, the contents of ‘a’ are ‘1000, 1913, 4000, 5000, 7000’—that is, ‘shell_sort’ has somehow garbled the contents of ‘a’.

To find out what has happened, execute the program once again. This time, you do not skip through the initialization, but jump directly into the ‘shell_sort’ call. Delete the old breakpoint by selecting it and clicking on ‘Clear’. Then, create a new breakpoint in line 35 before the call to ‘shell_sort’. To execute the program once again, select ‘Program ⇒ Run Again’.

Once more, DDD ends up before the call to ‘shell_sort’:

```
⇒  shell_sort(a, argc);
```

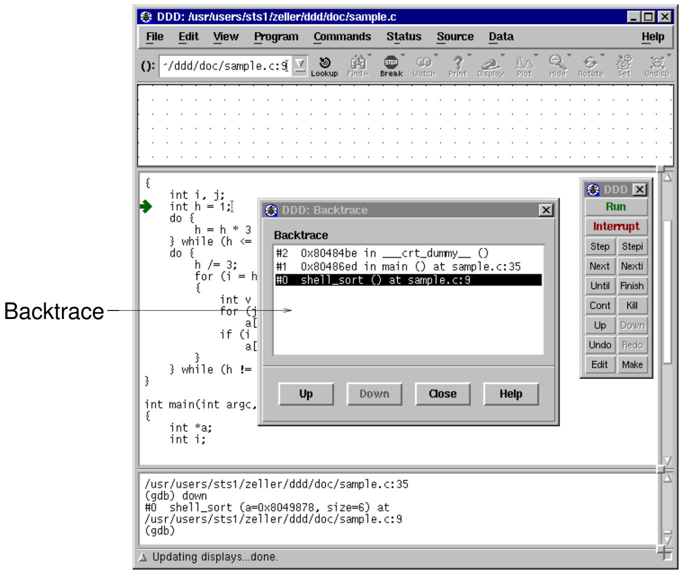
This time, you want to examine closer what '`shell_sort`' is doing. Click on '`Step`' to step into the call to '`shell_sort`'. This leaves your program in the first executable line, or

```
⇒ int h = 1;
```

while the debugger console tells us the function just entered:

```
(gdb) step
shell_sort (a=0x8049878, size=6) at sample.c:9
(gdb) _
```

This output that shows the function where '`sample`' is now suspended (and its arguments) is called a *stack frame display*. It shows a summary of the stack. You can use '`Status` ⇒ `Backtrace`' to see where you are in the stack as a whole; selecting a line (or clicking on '`Up`' and '`Down`') will let you move through the stack. Note how the '`a`' display disappears when its frame is left.



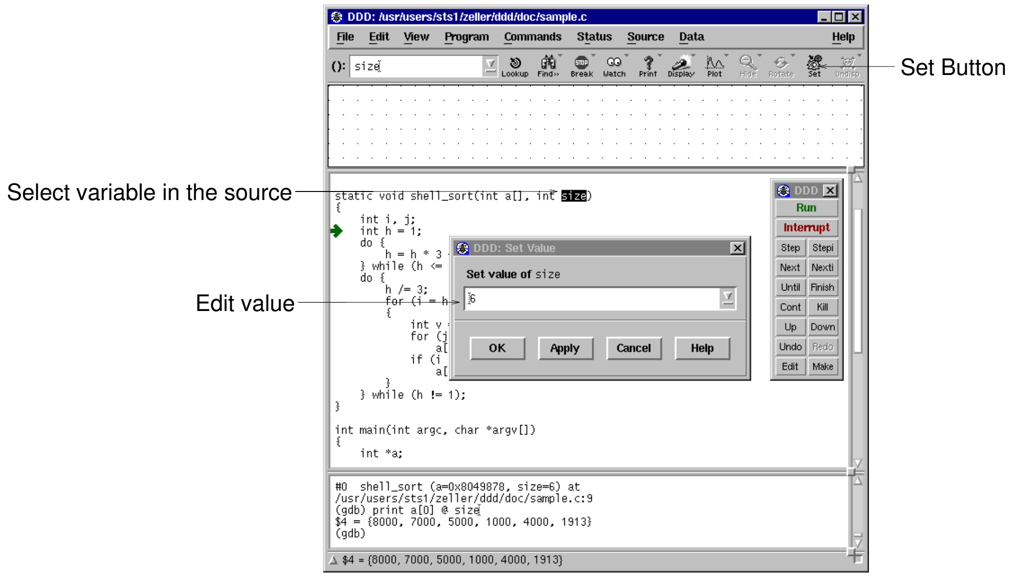
The DDD Backtrace

Let us now check whether '`shell_sort`''s arguments are correct. After returning to the lowest frame, enter '`a[0]@size`' in the argument field and click on '`Print`':

```
(gdb) print a[0] @ size
$4 = {8000, 7000, 5000, 1000, 4000, 1913}
(gdb) _
```

Surprise! Where does this additional value `1913` come from? The answer is simple: The array size as passed in '`size`' to '`shell_sort`' is *too large by one*—`1913` is a bogus value which happens to reside in memory after '`a`'. And this last value is being sorted in as well.

To see whether this is actually the problem cause, you can now assign the correct value to '`size`' (see Section 7.3.3 [Assignment], page 119). Select '`size`' in the source code and click on '`Set`'. A dialog pops up where you can edit the variable value.

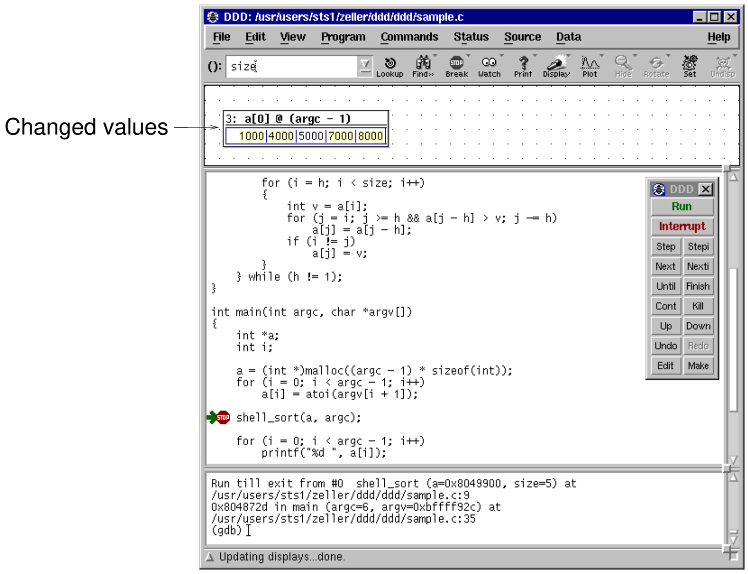


Setting a Value

Change the value of '`size`' to 5 and click on '`OK`'. Then, click on '`Finish`' to resume execution of the '`shell_sort`' function:

```
(gdb) set variable size = 5
(gdb) finish
Run till exit from #0  shell_sort (a=0x8049878, size=5) at sample.c:9
0x80486ed in main (argc=6, argv=0xbffff918) at sample.c:35
(gdb) _
```

Success! The ‘a’ display now contains the correct values ‘1000, 4000, 5000, 7000, 8000’.



Changed Values after Setting

You can verify that these values are actually printed to standard output by further executing the program. Click on ‘Cont’ to continue execution.

```
(gdb) cont
1000 4000 5000 7000 8000

Program exited normally.
(gdb) _
```

The message ‘Program exited normally.’ is from GDB; it indicates that the sample program has finished executing.

Having found the problem cause, you can now fix the source code. Click on ‘Edit’ to edit ‘sample.c’, and change the line

```
shell_sort(a, argc);
```

to the correct invocation

```
shell_sort(a, argc - 1);
```

You can now recompile sample

```
$ gcc -g -o sample sample.c
$ _
```

and verify (via ‘Program ⇒ Run Again’) that sample works fine now.

```
(gdb) run
'sample' has changed; re-reading symbols.
Reading in symbols...done.
Starting program: sample 8000 7000 5000 1000 4000
```

```
1000 4000 5000 7000 8000

Program exited normally.
(gdb) _
```

All is done; the program works fine now. You can end this DDD session with 'Program ⇒ Exit' or ***Ctrl***+***Q***.

1.1 Sample Program

Here's the source 'sample.c' of the sample program.

```
/* sample.c -- Sample C program to be debugged with DDD
 */

#include <stdio.h>
#include <stdlib.h>

static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

2 Getting In and Out of DDD

This chapter discusses how to start DDD, and how to get out of it. The essentials are:

- Type '`ddd`' to start DDD (see Section 2.1 [Invoking], page 15).
- Use '`File` ⇒ `Exit`' or ***`Ctrl`+`Q`*** to exit (see Section 2.2 [Quitting], page 28).

2.1 Invoking DDD

Normally, you can run DDD by invoking the program `ddd`.

You can also run DDD with a variety of arguments and options, to specify more of your debugging environment at the outset.

The most usual way to start DDD is with one argument, specifying an executable program:

`ddd` *program*

If you use GDB, DBX, Ladebug, or XDB as inferior debuggers, you can also start with both an executable program and a core file specified:

`ddd` *program core*

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

`ddd` *program* `1234`

would attach DDD to process `1234` (unless you also have a file named '`1234`'; DDD does check for a core file first).

You can further control DDD by invoking it with specific *options*. To get a list of DDD options, invoke DDD as

```
ddd --help
```

Most important are the options to specify the inferior debugger (see Section 2.1.1 [Choosing an Inferior Debugger], page 15), but you can also customize several aspects of DDD upon invocation (see Section 2.1.2 [Options], page 16).

DDD also understands the usual X options such as '`-display`' or '`-geometry`'. See Section 2.1.3 [X Options], page 24, for details.

All arguments and options that are not understood by DDD are passed to the inferior debugger; See Section 2.1.4 [Inferior Debugger Options], page 24, for a survey. To pass an option to the inferior debugger that conflicts with an X option, or with a DDD option listed here, use the '`--debugger`' option (see Section 2.1.2 [Options], page 16).

2.1.1 Choosing an Inferior Debugger

The most frequently required options are those to choose a specific inferior debugger.

Normally, the inferior debugger is determined by the program to analyze:

- If the program requires a specific interpreter, such as Java, Python, Perl or Bash, then you should use a JDB, PYDB, Perl, or Bash inferior debugger.

  Use

  `ddd --jdb` *program*

  `ddd --pydb` *program*

```
ddd --perl program
ddd --bash program
ddd --interpreter='path-to-debugger-bash --debugger' program
```

to run DDD with JDB, PYDB, Perl, or Bash as an inferior debugger.

- If the program is an executable binary, you should use GDB, DBX, Ladebug, or XDB. In general, GDB (or its HP variant, WDB) provides the most functionality of these debuggers.

  Use

  ```
  ddd --gdb program
  ddd --wdb program
  ddd --dbx program
  ddd --ladebug program
  ddd --xdb program
  ```

  to run DDD with GDB, WDB, DBX, Ladebug, or XDB as inferior debugger.

If you invoke DDD without any of these options, but give a *program* to analyze, then DDD will automatically determine the inferior debugger:

- If *program* is a Python program, a Perl script, or a Java class, DDD will invoke the appropriate debugger.
- If *program* is an executable binary, DDD will invoke its default debugger for executables (usually GDB).

See Section 2.5 [Customizing Debugger Interaction], page 34, for more details on determining the inferior debugger.

2.1.2 DDD Options

You can further control how DDD starts up using the following options. All options may be abbreviated, as long as they are unambiguous; single dashes '-' instead of double dashes '--' may also be used. Almost all options control a specific DDD resource or resource class (see Section 3.6 [Customizing], page 60).

'`--attach-windows`'
: Attach the source and data windows to the debugger console, creating one single big DDD window. This is the default setting.

  Giving this option is equivalent to setting the DDD '`Separate`' resource class to '`off`'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'`--attach-source-window`'
: Attach only the source window to the debugger console.

  Giving this option is equivalent to setting the DDD '`separateSourceWindow`' resource to '`off`'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'`--attach-data-window`'
: Attach only the source window to the debugger console.

  Giving this option is equivalent to setting the DDD '`separateDataWindow`' resource to '`off`'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'--automatic-debugger'
: Determine the inferior debugger automatically from the given arguments.

  Giving this option is equivalent to setting the DDD 'autoDebugger' resource to 'on'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--button-tips'
: Enable button tips.

  Giving this option is equivalent to setting the DDD 'buttonTips' resource to 'on'. See Section 3.6.2 [Customizing Help], page 61, for details.

'--configuration'
: Print the DDD configuration settings on standard output and exit.

  Giving this option is equivalent to setting the DDD 'showConfiguration' resource to 'on'. See Section B.5 [Diagnostics], page 170, for details.

'--check-configuration'
: Check the DDD environment (in particular, the X configuration), report any possible problem causes and exit.

  Giving this option is equivalent to setting the DDD 'checkConfiguration' resource to 'on'. See Section B.5 [Diagnostics], page 170, for details.

'--data-window'
: Open the data window upon start-up.

  Giving this option is equivalent to setting the DDD 'openDataWindow' resource to 'on'. See Section 3.6.4.4 [Toggling Windows], page 70, for details.

'--dbx'
: Run DBX as inferior debugger.

  Giving this option is equivalent to setting the DDD 'debugger' resource to 'dbx'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--debugger *name*'
: Invoke the inferior debugger *name*. This is useful if you have several debugger versions around, or if the inferior debugger cannot be invoked under its usual name (i.e. gdb, wdb, dbx, xdb, jdb, pydb, or perl).

  This option can also be used to pass options to the inferior debugger that would otherwise conflict with DDD options. For instance, to pass the option '-d *directory*' to XDB, use:

  ```
  ddd --debugger "xdb -d directory"
  ```

  If you use the '--debugger' option, be sure that the type of inferior debugger is specified as well. That is, use one of the options '--gdb', '--dbx', '--xdb', '--jdb', '--pydb', or '--perl' (unless the default setting works fine).

  Giving this option is equivalent to setting the DDD 'debuggerCommand' resource to *name*. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--debugger-console'
: Open the debugger console upon start-up.

  Giving this option is equivalent to setting the DDD 'openDebuggerConsole' resource to 'on'. See Section 3.6.4.4 [Toggling Windows], page 70, for details.

'`--disassemble`'
: Disassemble the source code. See also the '`--no-disassemble`' option, below.

  Giving this option is equivalent to setting the DDD '`disassemble`' resource to '`on`'. See Section 4.4 [Customizing Source], page 77, for details.

'`--exec-window`'
: Run the debugged program in a specially created execution window. This is useful for programs that have special terminal requirements not provided by the debugger window, as raw keyboard processing or terminal control sequences. See Section 6.2 [Using the Execution Window], page 93, for details.

  Giving this option is equivalent to setting the DDD '`separateExecWindow`' resource to '`on`'. See Section 6.2.1 [Customizing the Execution Window], page 94, for details.

'`--font` *fontname*'
'`-fn` *fontname*'
: Use *fontname* as default font.

  Giving this option is equivalent to setting the DDD '`defaultFont`' resource to '`fontname`'. See Section 3.6.4.3 [Customizing Fonts], page 67, for details.

'`--fonts`'
: Show the font definitions used by DDD on standard output.

  Giving this option is equivalent to setting the DDD '`showFonts`' resource to '`on`'. See Section B.5 [Diagnostics], page 170, for details.

'`--fontsize` *size*'
: Set the default font size to *size* (in 1/10 points). To make DDD use 12-point fonts, say '`--fontsize 120`'.

  Giving this option is equivalent to setting the DDD '`FontSize`' resource class to '`size`'. See Section 3.6.4.3 [Customizing Fonts], page 67, for details.

'`--fullname`'
'`-f`'
: Enable the TTY interface, taking additional debugger commands from standard input and forwarding debugger output on standard output. Current positions are issued in GDB '`-fullname`' format suitable for debugger front-ends. By default, both the debugger console and source window are disabled. See Section 10.2 [TTY mode], page 150, for a discussion.

  Giving this option is equivalent to setting the DDD '`TTYMode`' resource class to '`on`'. See Section 10.2 [TTY mode], page 150, for details.

'`--gdb`'
: Run GDB as inferior debugger.

  Giving this option is equivalent to setting the DDD '`debugger`' resource to '`gdb`'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'`--glyphs`'
: Display the current execution position and breakpoints as glyphs. See also the '`--no-glyphs`' option, below.

  Giving this option is equivalent to setting the DDD '`displayGlyphs`' resource to '`on`'. See Section 4.4 [Customizing Source], page 77, for details.

'--help'
'-h'
'-?' Give a list of frequently used options. Show options of the inferior debugger as well.

Giving this option is equivalent to setting the DDD 'showInvocation' resource to 'on'. See Section B.5 [Diagnostics], page 170, for details.

'--host *hostname*'
'--host *username*@*hostname*'
Invoke the inferior debugger directly on the remote host *hostname*. If *username* is given and the '--login' option is not used, use *username* as remote user name. See Section 2.4.2 [Remote Debugger], page 32, for details.

Giving this option is equivalent to setting the DDD 'debuggerHost' resource to *hostname*. See Section 2.4.2 [Remote Debugger], page 32, for details.

'--jdb' Run JDB as inferior debugger.

Giving this option is equivalent to setting the DDD 'debugger' resource to 'gdb'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--ladebug'
Run Ladebug as inferior debugger.

Giving this option is equivalent to setting the DDD 'debugger' resource to 'ladebug'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--lesstif-hacks'
Equivalent to '--lesstif-version 999'. Deprecated.

Giving this option is equivalent to setting the DDD 'lessTifVersion' resource to 999. See Section C.8 [LessTif], page 179, for details.

'--lesstif-version *version*'
Enable some hacks to make DDD run properly with LessTif. See Section C.8 [LessTif], page 179, for a discussion.

Giving this option is equivalent to setting the DDD 'lessTifVersion' resource to *version*. See Section C.8 [LessTif], page 179, for details.

'--license'
Print the DDD license on standard output and exit.

Giving this option is equivalent to setting the DDD 'showLicense' resource to *on*. See Section B.5 [Diagnostics], page 170, for details.

'--login *username*'
'-l *username*'
Use *username* as remote user name. See Section 2.4.2 [Remote Debugger], page 32, for details.

Giving this option is equivalent to setting the DDD 'debuggerHostLogin' resource to *username*. See Section 2.4.2 [Remote Debugger], page 32, for details.

'--maintenance'
Enable the top-level 'Maintenance' menu with options for debugging DDD. See Section 3.1.9 [Maintenance Menu], page 49, for details.

Giving this option is equivalent to setting the DDD 'maintenance' resource to *on*. See Section 3.1.9 [Maintenance Menu], page 49, for details.

'--manual'
: Print the DDD manual on standard output and exit.

Giving this option is equivalent to setting the DDD 'showManual' resource to *on*. See Section B.5 [Diagnostics], page 170, for details.

'--news'
: Print the DDD news on standard output and exit.

Giving this option is equivalent to setting the DDD 'showNews' resource to *on*. See Section B.5 [Diagnostics], page 170, for details.

'--no-button-tips'
: Disable button tips.

Giving this option is equivalent to setting the DDD 'buttonTips' resource to 'off'. See Section 3.6.2 [Customizing Help], page 61, for details.

'--no-data-window'
: Do not open the data window upon start-up.

Giving this option is equivalent to setting the DDD 'openDataWindow' resource to 'off'. See Section 3.6.4.4 [Toggling Windows], page 70, for details.

'--no-debugger-console'
: Do not open the debugger console upon start-up.

Giving this option is equivalent to setting the DDD 'openDebuggerConsole' resource to 'off'. See Section 3.6.4.4 [Toggling Windows], page 70, for details.

'--no-disassemble'
: Do not disassemble the source code.

Giving this option is equivalent to setting the DDD 'disassemble' resource to 'off'. See Section 4.4 [Customizing Source], page 77, for details.

'--no-exec-window'
: Do not run the debugged program in a specially created execution window; use the debugger console instead. Useful for programs that have little terminal input/output, or for remote debugging. See Section 6.2 [Using the Execution Window], page 93, for details.

Giving this option is equivalent to setting the DDD 'separateExecWindow' resource to 'off'. See Section 6.2.1 [Customizing the Execution Window], page 94, for details.

'--no-glyphs'
: Do not use glyphs; display the current execution position and breakpoints as text characters.

Giving this option is equivalent to setting the DDD 'displayGlyphs' resource to 'off'. See Section 4.4 [Customizing Source], page 77, for details.

'--no-lesstif-hacks'
: Equivalent to '--lesstif-version 1000'. Deprecated.

Giving this option is equivalent to setting the DDD 'lessTifVersion' resource to 1000. See Section C.8 [LessTif], page 179, for details.

'--no-maintenance'
: Do not enable the top-level 'Maintenance' menu with options for debugging DDD. This is the default. See Section 3.1.9 [Maintenance Menu], page 49, for details.

  Giving this option is equivalent to setting the DDD 'maintenance' resource to *off*. See Section 3.1.9 [Maintenance Menu], page 49, for details.

'--no-source-window'
: Do not open the source window upon start-up.

  Giving this option is equivalent to setting the DDD 'openSourceWindow' resource to 'off'. See Section 3.6.4.4 [Toggling Windows], page 70, for details.

'--no-value-tips'
: Disable value tips.

  Giving this option is equivalent to setting the DDD 'valueTips' resource to 'off'. See Section 7.1 [Value Tips], page 105, for details.

'--nw'
: Do not use the X window interface. Start the inferior debugger on the local host.

'--perl'
: Run Perl as inferior debugger.

  Giving this option is equivalent to setting the DDD 'debugger' resource to 'perl'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--pydb'
: Run PYDB as inferior debugger.

  Giving this option is equivalent to setting the DDD 'debugger' resource to 'pydb'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--panned-graph-editor'
: Use an Athena panner to scroll the data window. Most people prefer panners on scroll bars, since panners allow two-dimensional scrolling. However, the panner is off by default, since some M*tif implementations do not work well with Athena widgets. See Section 7.3.5.5 [Display Resources], page 128, for details; see also '--scrolled-graph-editor', below.

  Giving this option is equivalent to setting the DDD 'pannedGraphEditor' resource to 'on'. See Section 7.3.5.5 [Display Resources], page 128, for details.

'--play-log *log-file*'
: Recapitulate a previous DDD session.

  ```
  ddd --play-log log-file
  ```

  invokes DDD as inferior debugger, simulating the inferior debugger given in *log-file* (see below). This is useful for debugging DDD.

  Giving this option is equivalent to setting the DDD 'playLog' resource to 'on'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--PLAY *log-file*'
: Simulate an inferior debugger. *log-file* is a '~/.ddd/log' file as generated by some previous DDD session (see Section B.5.1 [Logging], page 170). When a command is entered, scan *log-file* for this command and re-issue the logged reply; if the command is not found, do nothing. This is used by the '--play' option.

'`--rhost` *hostname*'
'`--rhost` *username*`@`*hostname*'
: Run the inferior debugger interactively on the remote host *hostname*. If *username* is given and the '`--login`' option is not used, use *username* as remote user name. See Section 2.4.2 [Remote Debugger], page 32, for details.

  Giving this option is equivalent to setting the DDD '`debuggerRHost`' resource to *hostname*. See Section 2.4.2 [Remote Debugger], page 32, for details.

'`--scrolled-graph-editor`'
: Use M*tif scroll bars to scroll the data window. This is the default in most DDD configurations. See Section 7.3.5.5 [Display Resources], page 128, for details; see also '`--panned-graph-editor`', above.

  Giving this option is equivalent to setting the DDD '`pannedGraphEditor`' resource to '`off`'. See Section 7.3.5.5 [Display Resources], page 128, for details.

'`--separate-windows`'
'`--separate`'
: Separate the console, source and data windows. See also the '`--attach`' options, above.

  Giving this option is equivalent to setting the DDD '`Separate`' resource class to '`off`'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'`--session` *session*'
: Load *session* upon start-up. See Section 2.3.2 [Resuming Sessions], page 30, for details.

  Giving this option is equivalent to setting the DDD '`session`' resource to *session*. See Section 2.3.2 [Resuming Sessions], page 30, for details.

'`--source-window`'
: Open the source window upon start-up.

  Giving this option is equivalent to setting the DDD '`openSourceWindow`' resource to '`on`'. See Section 3.6.4.4 [Toggling Windows], page 70, for details.

'`--status-at-bottom`'
: Place the status line at the bottom of the source window.

  Giving this option is equivalent to setting the DDD '`statusAtBottom`' resource to '`on`'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'`--status-at-top`'
: Place the status line at the top of the source window.

  Giving this option is equivalent to setting the DDD '`statusAtBottom`' resource to '`off`'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'`--sync-debugger`'
: Do not process X events while the debugger is busy. This may result in slightly better performance on single-processor systems.

  Giving this option is equivalent to setting the DDD '`synchronousDebugger`' resource to '`on`'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--toolbars-at-bottom'
: Place the toolbars at the bottom of the respective window.

  Giving this option is equivalent to setting the DDD 'toolbarsAtBottom' resource to 'on'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'--toolbars-at-top'
: Place the toolbars at the top of the respective window.

  Giving this option is equivalent to setting the DDD 'toolbarsAtBottom' resource to 'off'. See Section 3.6.4.2 [Window Layout], page 65, for details.

'--trace'
: Show the interaction between DDD and the inferior debugger on standard error. This is useful for debugging DDD. If '--trace' is not specified, this information is written into '~/.ddd/log' ('~' stands for your home directory), such that you can also do a post-mortem debugging. See Section B.5.1 [Logging], page 170, for details about logging.

  Giving this option is equivalent to setting the DDD 'trace' resource to *on*. See Section B.5 [Diagnostics], page 170, for details.

'--tty'
'-t'
: Enable TTY interface, taking additional debugger commands from standard input and forwarding debugger output on standard output. Current positions are issued in a format readable for humans. By default, the debugger console is disabled.

  Giving this option is equivalent to setting the DDD 'ttyMode' resource to 'on'. See Section 10.2 [TTY mode], page 150, for details.

'--value-tips'
: Enable value tips.

  Giving this option is equivalent to setting the DDD 'valueTips' resource to 'on'. See Section 7.1 [Value Tips], page 105, for details.

'--version'
'-v'
: Print the DDD version on standard output and exit.

  Giving this option is equivalent to setting the DDD 'showVersion' resource to 'on'. See Section B.5 [Diagnostics], page 170, for details.

'--vsl-library *library*'
: Load the VSL library *library* instead of using the DDD built-in library. This is useful for customizing display shapes and fonts.

  Giving this option is equivalent to setting the DDD 'vslLibrary' resource to *library*. See Section 7.3.5.6 [VSL Resources], page 128, for details.

'--vsl-path *path*'
: Search VSL libraries in *path* (a colon-separated directory list).

  Giving this option is equivalent to setting the DDD 'vslPath' resource to *path*. See Section 7.3.5.6 [VSL Resources], page 128, for details.

'--vsl-help'
: Show a list of further options controlling the VSL interpreter. These options are intended for debugging purposes and are subject to change without further notice.

'--wdb' Run WDB as inferior debugger.

Giving this option is equivalent to setting the DDD 'debugger' resource to 'wdb'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

'--xdb' Run XDB as inferior debugger.

Giving this option is equivalent to setting the DDD 'debugger' resource to 'xdb'. See Section 2.5 [Customizing Debugger Interaction], page 34, for details.

2.1.3 X Options

DDD also understands the following X options. Note that these options only take a single dash '-'.

'-display *display*'
: Use the X server *display*. By default, *display* is taken from the DISPLAY environment variable.

'-geometry *geometry*'
: Specify the initial size and location of the debugger console.

'-iconic'
: Start DDD iconified.

'-name *name*'
: Give DDD the name *name*.

'-selectionTimeout *timeout*'
: Specify the timeout in milliseconds within which two communicating applications must respond to one another for a selection request.

'-title *name*'
: Give the DDD window the title *name*.

'-xrm *resourcestring*'
: Specify a resource name and value to override any defaults.

2.1.4 Inferior Debugger Options

All options that DDD does not recognize are passed to the inferior debugger. This section lists the most useful options of the different inferior debuggers supported by DDD. In case these options do not work as expected, please lookup the appropriate reference.

2.1.4.1 GDB Options

These GDB options are useful when using DDD with GDB as inferior debugger. Single dashes '-' instead of double dashes '--' may also be used.

'-b *baudrate*'
: Set serial port baud rate used for remote debugging.

'--cd *dir*' Change current directory to *dir*.

'--command *file*'
: Execute GDB commands from *file*.

'`--core` *corefile*'
Analyze the core dump *corefile*.

'`--directory` *dir*'
'`-d` *dir*' Add *directory* to the path to search for source files.

'`--exec` *execfile*'
Use *execfile* as the executable.

'`--mapped`'
Use mapped symbol files if supported on this system.

'`--nx`'

'`-n`' Do not read '`.gdbinit`' file.

'`--readnow`'
Fully read symbol files on first access.

'`--se` *file*'
Use *file* as symbol file and executable file.

'`--symbols` *symfile*'
Read symbols from *symfile*.

See section "Invoking GDB" in *Debugging with GDB*, for further options that can be used with GDB.

2.1.4.2 DBX and Ladebug Options

DBX variants differ widely in their options, so we cannot give a list here. Check out the *dbx(1)* and *ladebug(1)* manual pages.

2.1.4.3 XDB Options

These XDB options are useful when using DDD with XDB as inferior debugger.

'`-d` *dir*' Specify *dir* as an alternate directory where source files are located.

'`-P` *process-id*'
Specify the process ID of an existing process the user wants to debug.

'`-l` *library*'
Pre-load information about the shared library *library*. '`-l ALL`' means always pre-load shared library information.

'`-S` *num*' Set the size of the string cache to *num* bytes (default is 1024, which is also the minimum).

'`-s`' Enable debugging of shared libraries.

Further options can be found in the *xdb(1)* manual page.

2.1.4.4 JDB Options

JDB as of JDK 1.2

The following JDB options are useful when using DDD with JDB (from JDK 1.2) as inferior debugger.

'`-attach` *address*'
: attach to a running virtual machine (VM) at *address* using standard connector

'`-listen` *address*'
: wait for a running VM to connect at *address* using standard connector

'`-listenany`'
: wait for a running VM to connect at any available address using standard connector

'`-launch`'
: launch VM immediately instead of waiting for '`run`' command

These JDB options are forwarded to the debuggee:

'`-verbose[:class|gc|jni]`'
'`-v`' Turn on verbose mode.

'`-D`*name*`=`*value*'
: Set the system property *name* to *value*.

'`-classpath` *path*'
: List directories in which to look for classes. *path* is a list of directories separated by colons.

'`-X` *option*'
: Non-standard target VM option

JDB as of JDK 1.1

The following JDB options are useful when using DDD with JDB (from JDK 1.1) as inferior debugger.

'`-host` *hostname*'
: host machine of interpreter to attach to

'`-password` *psswd*'
: password of interpreter to attach to (from '`-debug`')

These JDB options are forwarded to the debuggee:

'`-verbose`'
'`-v`' Turn on verbose mode.

'`-debug`' Enable remote Java debugging,

'`-noasyncgc`'
: Don't allow asynchronous garbage collection.

'`-verbosegc`'
: Print a message when garbage collection occurs.

'-noclassgc'
: Disable class garbage collection.

'-checksource'
'-cs'
: Check if source is newer when loading classes.

'-ss *number*'
: Set the maximum native stack size for any thread.

'-oss *number*'
: Set the maximum Java stack size for any thread.

'-ms *number*'
: Set the initial Java heap size.

'-mx *number*'
: Set the maximum Java heap size.

'-D*name*=*value*'
: Set the system property *name* to *value*.

'-classpath *path*'
: List directories in which to look for classes. *path* is a list of directories separated by colons.

'-prof'
'-prof:*file*'
: Output profiling data to './java.prof'. If *file* is given, write the data to './*file*'.

'-verify'
: Verify all classes when read in.

'-verifyremote'
: Verify classes read in over the network (default).

'-noverify'
: Do not verify any class.

'-dbgtrace'
: Print info for debugging JDB.

Further options can be found in the JDB documentation.

2.1.4.5 PYDB Options

For a list of useful PYDB options, check out the PYDB documentation.

2.1.4.6 Perl Options

The most important Perl option to use with DDD is '-w'; it enables several important warnings. For further options, see the *perlrun(1)* manual page.

2.1.4.7 Bash Options

If you have the proper bash installed, the option needed to specify debugging support is '--debugger'. (If your bash doesn't understand this option you need to pick up a version of bash that does from http://bashdb.sourceforge.net.)

2.1.5 Multiple DDD Instances

If you have multiple DDD instances running, they share common preferences and history files. This means that changes applied to one instance may get lost when being overwritten by the other instance. DDD has two means to protect you against unwanted losses. The first means is an automatic reloading of changed options, controlled by the following resource (see Section 3.6 [Customizing], page 60):

checkOptions (class CheckOptions) Resource

Every *n* seconds, where *n* is the value of this resource, DDD checks whether the options file has changed. Default is `30`, which means that every 30 seconds, DDD checks for the options file. Setting this resource to `0` disables checking for changed option files.

Normally, automatic reloading of options should already suffice. If you need stronger protection, DDD also provides a warning against multiple instances. This warning is disabled by default, If you want to be warned about multiple DDD invocations sharing the same preferences and history files, enable '`Edit` ⇒ `Preferences` ⇒ `Warn if Multiple DDD Instances are Running`'.

This setting is tied to the following resource (see Section 3.6 [Customizing], page 60):

warnIfLocked (class WarnIfLocked) Resource

Whether to warn if multiple DDD instances are running ('`on`') or not ('`off`', default).

2.1.6 X warnings

If you are bothered by X warnings, you can suppress them by setting '`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Suppress X warnings`'.

This setting is tied to the following resource (see Section 3.6 [Customizing], page 60):

suppressWarnings (class SuppressWarnings) Resource

If '`on`', X warnings are suppressed. This is sometimes useful for executables that were built on a machine with a different X or M*tif configuration. By default, this is '`off`'.

2.2 Quitting DDD

To exit DDD, select '`File` ⇒ `Exit`'. You may also type the `quit` command at the debugger prompt or press ⟨Ctrl+Q⟩. GDB and XDB also accept the `q` command or an end-of-file character (usually ⟨Ctrl+D⟩). Closing the last DDD window will also exit DDD.

An interrupt (⟨ESC⟩ or '`Interrupt`') does not exit from DDD, but rather terminates the action of any debugger command that is in progress and returns to the debugger command level. It is safe to type the interrupt character at any time because the debugger does not allow it to take effect until a time when it is safe.

In case an ordinary interrupt does not succeed, you can also use an abort (⟨Ctrl+\⟩ or '`Abort`'), which sends a `SIGABRT` signal to the inferior debugger. Use this in emergencies only; the inferior debugger may be left inconsistent or even exit after a `SIGABRT` signal.

As a last resort (if DDD hangs, for example), you may also interrupt DDD itself using an interrupt signal (`SIGINT`). This can be done by typing the interrupt character (usually ⟨Ctrl+C⟩) in the shell

DDD was started from, or by using the UNIX '`kill`' command. An interrupt signal interrupts any DDD action; the inferior debugger is interrupted as well. Since this interrupt signal can result in internal inconsistencies, use this as a last resort in emergencies only; save your work as soon as possible and restart DDD.

2.3 Persistent Sessions

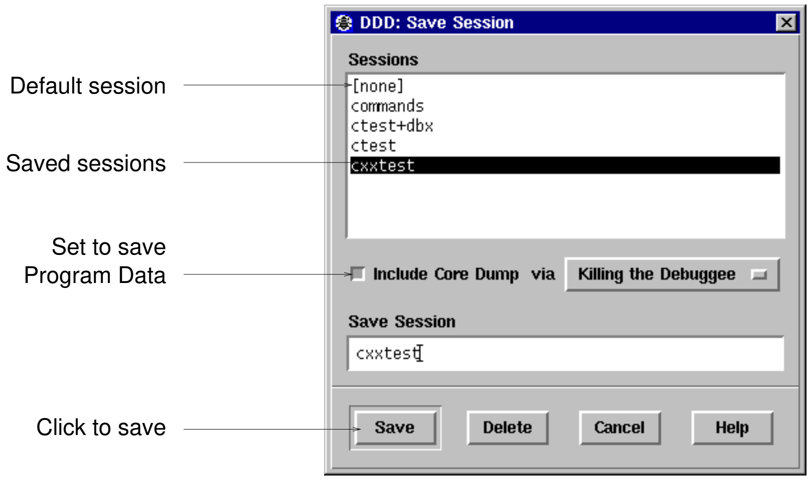
If you want to interrupt your current DDD session, you can save the entire the entire DDD state as *session* on disk and resume later.

2.3.1 Saving Sessions

To save a session, select '`File` ⇒ `Save Session As`'. You will be asked for a symbolic session name *session*.

If your program is running (see Chapter 6 [Running], page 91), or if you have opened a core file (see Section 4.2.2 [Opening Core Dumps], page 74), DDD can also include a core file in the session such that the debuggee data will be restored when re-opening it. To get a core file, DDD typically must *kill* the debuggee. This means that you cannot resume program execution after saving a session. Depending on your architecture, other options for getting a core file may also be available.

Including a core dump is necessary for restoring memory contents and the current execution position. To include a core dump, enable '`Include Core Dump`'.



Saving a Session

After clicking on '`Save`', the session is saved in '`~/.ddd/sessions/`*session*'.

Here's a list of the items whose state is saved in a session:

- The state of the debugged program, as a core file.[1]

[1] Only if a core file is included.

- All breakpoints and watchpoints (see Chapter 5 [Stopping], page 81).
- All signal settings (see Section 6.10 [Signals], page 102).
- All displays (see Section 7.3 [Displaying Values], page 107).[2]
- All DDD options (see Section 3.6.1.3 [Saving Options], page 61).
- All debugger settings (see Section 3.6.5 [Debugger Settings], page 71).
- All user-defined buttons (see Section 10.4 [Defining Buttons], page 151).
- All user-defined commands (see Section 10.5 [Defining Commands], page 154).
- The positions and sizes of DDD windows.
- The command history (see Section 10.1.2 [Command History], page 148).

After saving the current state as a session, the session becomes *active*. This means that DDD state will be saved as session defaults:

- User options will be saved in '`~/.ddd/sessions/`*session*`/init`' instead of '`~/.ddd/init`'. See Section 3.6.1.3 [Saving Options], page 61, for details.
- The DDD command history will be saved in '`~/.ddd/sessions/`*session*`/history`' instead of '`~/.ddd/history`'. See Section 10.1.2 [Command History], page 148, for details.

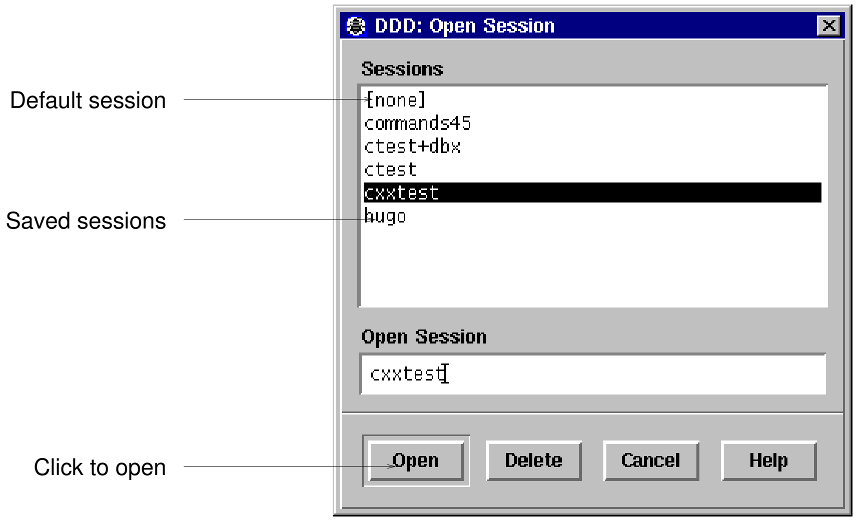
To make the current session inactive, open the *default session* named '`[None]`'. See Section 2.3.2 [Resuming Sessions], page 30, for details on opening sessions.

2.3.2 Resuming Sessions

To resume a previously saved session, select '`File` ⇒ `Open Session`' and choose a session name from the list. After clicking on '`Open`', the entire DDD state will be restored from the given session.

The session named '`[None]`' is the *default session* which is active when starting DDD. To save options for default sessions, choose the default session before exiting DDD. See Section 3.6.1.3 [Saving Options], page 61, for details.

[2] If a core file is *not* to be included in the session, DDD data displays are saved as *deferred*; that is, they will be restored as soon as program execution reaches the scope in which they were created. See Section 7.3.1.1 [Creating Single Displays], page 107, for details.



Opening a Session

If a the restored session includes a core dump, the program being debugged will be in the same state at the time the session was saved; in particular, you can examine the program data. However, you will not be able to resume program execution since the process and its environment (open files, resources, etc.) no longer exist. However, you can restart the program, re-using the restored breakpoints and data displays.

Opening sessions also restores command definitions, buttons, display shortcuts and the source tab width. This way, you can maintain a different set of definitions for each session.

You can also specify a session to open when starting DDD. To invoke DDD with a session *session*, use

`ddd --session` *session*

There is also a shortcut that opens the session *session* and invokes the inferior debugger on an executable named *session* (in case *session* cannot be opened):

`ddd =`*session*

There is no need to give further command-line options when restarting a session, as they will be overridden by the options saved in the session.

You can also use an X session manager such as `xsm` to save and restore DDD sessions.[3] When being shut down by a session manager, DDD saves its state under the name specified by the session manager; resuming the X session makes DDD reload its saved state.

2.3.3 Deleting Sessions

To delete sessions that are no longer needed, select '`File` ⇒ `Open Session`' or '`File` ⇒ `Save Session`'. Select the sessions you want to delete and click on '`Delete`'.

The default session '`[None]`' cannot be deleted.

---

[3] Requires X11R6 or later.

2.3.4 Customizing Sessions

You can change the place where DDD saves its sessions by setting the environment variable `DDD_SESSIONS` to the name of a directory. Default is '`~/.ddd/sessions/`'.

Where applicable, DDD supports a `gcore` command to obtain core files of the running program. You can enter its path via '`Edit` ⇒ `Preferences` ⇒ `Helpers` ⇒ `Get Core File`'. Leave the value empty if you have no `gcore` or similar command.

This setting is tied to the following resource (see Section 3.6 [Customizing], page 60):

getCoreCommand (class GetCoreCommand) Resource

A command to get a core dump of a running process (typically, `gcore`) '`@FILE@`' is replaced by the base name of the file to create; '`@PID@`' is replaced by the process id. The output must be written to '`@FILE@.@PID@`'.

Leave the value empty if you have no `gcore` or similar command.

2.4 Remote Debugging

You can have each of DDD, the inferior debugger, and the debugged program run on different machines.

2.4.1 Running DDD on a Remote Host

You can run DDD on a remote host, using your current host as X display. On the remote host, invoke DDD as

`ddd -display` *display*

where *display* is the name of the X server to connect to (for instance, '*hostname*`:0.0`', where *hostname* is your host).

Instead of specifying '`-display` *display*', you can also set the `DISPLAY` environment variable to *display*.

2.4.2 Using DDD with a Remote Inferior Debugger

In order to run the inferior debugger on a remote host, you need '`remsh`' (called '`rsh`' on BSD systems) access on the remote host.

To run the debugger on a remote host *hostname*, invoke DDD as

`ddd --host` *hostname remote-program*

If your remote *username* differs from the local username, use

`ddd --host` *hostname* `--login` *username remote-program*

or

`ddd --host` *username*`@`*hostname remote-program*

instead.

There are a few *caveats* in remote mode:

- The remote debugger is started in your remote home directory. Hence, you must specify an absolute path name for *remote-program* (or a path name relative to your remote home directory). Same applies to remote core files. Also, be sure to specify a remote process id when debugging a running program.
- The remote debugger is started non-interactively. Some DBX versions have trouble with this. If you do not get a prompt from the remote debugger, use the '`--rhost`' option instead of '`--host`'. This will invoke the remote debugger via an interactive shell on the remote host, which may lead to better results.

  Note: using '`--rhost`', DDD invokes the inferior debugger as soon as a shell prompt appears. The first output on the remote host ending in a space character or '>' and not followed by a newline is assumed to be a shell prompt. If necessary, adjust your shell prompt on the remote host.
- To run the remote program, DDD invokes an '`xterm`' terminal emulator on the remote host, giving your current '`DISPLAY`' environment variable as address. If the remote host cannot invoke '`xterm`', or does not have access to your X display, start DDD with the '`--no-exec-window`' option. The program input/output will then go through the DDD debugger console.
- In remote mode, all sources are loaded from the remote host; file dialogs scan remote directories. This may result in somewhat slower operation than normal.
- To help you find problems due to remote execution, run DDD with the '`--trace`' option. This prints the shell commands issued by DDD on standard error.

See Section 2.4.2.1 [Customizing Remote Debugging], page 33, for customizing remote mode.

2.4.2.1 Customizing Remote Debugging

When having the inferior debugger run on a remote host (see Section 2.4 [Remote Debugging], page 32), all commands to access the inferior debugger as well as its files must be run remotely. This is controlled by the following resources (see Section 3.6 [Customizing], page 60):

rshCommand (class RshCommand) Resource

The remote shell command to invoke TTY-based commands on remote hosts. Usually, `remsh`, `rsh`, `ssh`, or `on`.

listCoreCommand (class listCoreCommand) Resource

The command to list all core files on the remote host. The string '`@MASK@`' is replaced by a file filter. The default setting is:

```
Ddd*listCoreCommand: \
file @MASK@ | grep '.*:.*core.*' | cut -d: -f1
```

listDirCommand (class listDirCommand) Resource

The command to list all directories on the remote host. The string '`@MASK@`' is replaced by a file filter. The default setting is:

```
Ddd*listDirCommand: \
file @MASK@ | grep '.*:.*directory.*' | cut -d: -f1
```

listExecCommand (class listExecCommand) Resource

The command to list all executable files on the remote host. The string '`@MASK@`' is replaced by a file filter. The default setting is:

```
Ddd*listExecCommand: \
file @MASK@ | grep '.*:.*exec.*' \
  | grep -v  '.*:.*script.*' \
  | cut -d: -f1 | grep -v '.*\.o$'
```

listSourceCommand (class listSourceCommand) Resource

The command to list all source files on the remote host. The string '`@MASK@`' is replaced by a file filter. The default setting is:

```
Ddd*listSourceCommand: \
file @MASK@ | grep '.*:.*text.*' | cut -d: -f1
```

2.4.3 Debugging a Remote Program

The GDB debugger allows you to run the *debugged program* on a remote machine (called *remote target*), while GDB runs on the local machine.

See section "Remote Debugging" in *Debugging with* GDB, for details. Basically, the following steps are required:

- Transfer the executable to the remote target.
- Start `gdbserver` on the remote target.
- Start DDD using GDB on the local machine, and load the same executable using the GDB `file` command.
- Attach to the remote '`gdbserver`' using the GDB `target remote` command.

The local '`.gdbinit`' file is useful for setting up directory search paths, etc.

Of course, you can also combine DDD remote mode and GDB remote mode, running DDD, GDB, and the debugged program each on a different machine.

2.5 Customizing Interaction with the Inferior Debugger

These settings control the interaction of DDD with its inferior debugger.

2.5.1 Invoking an Inferior Debugger

To choose the default inferior debugger, select '`Edit` ⇒ `Preferences` ⇒ `Startup` ⇒ `Debugger Type`'. You can

- have DDD determine the appropriate inferior debugger automatically from its command-line arguments. Set '`Determine Automatically from Arguments`' to enable.
- have DDD start the debugger of your choice, as specified in '`Debugger Type`'.

The following DDD resources control the invocation of the inferior debugger (see Section 3.6 [Customizing], page 60).

autoDebugger (class AutoDebugger) Resource

If this is 'on' (default), DDD will attempt to determine the debugger type from its arguments, possibly overriding the 'debugger' resource (see below). If this is 'off', DDD will invoke the debugger specified by the 'debugger' resource regardless of DDD arguments.

debugger (class Debugger) Resource

The type of the inferior debugger to invoke ('gdb', 'dbx', 'ladebug', 'xdb', 'jdb', 'pydb', 'perl', or 'bash').

This resource is usually set through the '--gdb', '--dbx', '--ladebug', '--xdb', '--jdb', '--pydb', '--perl', and '--bash' options; See Section 2.1.2 [Options], page 16, for details.

debuggerCommand (class DebuggerCommand) Resource

The name under which the inferior debugger is to be invoked. If this string is empty (default), the debugger type ('debugger' resource) is used.

This resource is usually set through the '--debugger' option; See Section 2.1.2 [Options], page 16, for details.

2.5.2 Initializing the Inferior Debugger

DDD uses a number of resources to initialize the inferior debugger (see Section 3.6 [Customizing], page 60).

2.5.2.1 GDB Initialization

gdbInitCommands (class InitCommands) Resource

This string contains a list of newline-separated commands that are initially sent to GDB. As a side-effect, all settings specified in this resource are considered fixed and cannot be changed through the GDB settings panel, unless preceded by white space. By default, the 'gdbInitCommands' resource contains some settings vital to DDD:

```
Ddd*gdbInitCommands: \
set height 0\n\
set width 0\n\
 set verbose off\n\
set prompt (gdb) \n
```

While the 'set height', 'set width', and 'set prompt' settings are fixed, the 'set verbose' settings can be changed through the GDB settings panel (although being reset upon each new DDD invocation).

Do not use this resource to customize GDB; instead, use a personal '~/.gdbinit' file. See your GDB documentation for details.

gdbSettings (class Settings) Resource

This string contains a list of newline-separated commands that are also initially sent to GDB. Its default value is

```
Ddd*gdbSettings: \
set print asm-demangle on\n
```

This resource is used to save and restore the debugger settings.

sourceInitCommands (class SourceInitCommands) Resource

If '`on`' (default), DDD writes all GDB initialization commands into a temporary file and makes GDB read this file, rather than sending each initialization command separately. This results in faster startup (especially if you have several user-defined commands). If '`off`', DDD makes GDB process each command separately.

2.5.2.2 DBX Initialization

dbxInitCommands (class InitCommands) Resource

This string contains a list of newline-separated commands that are initially sent to DBX. By default, it is empty.

Do not use this resource to customize DBX; instead, use a personal '`~/.dbxinit`' or '`~/.dbxrc`' file. See your DBX documentation for details.

dbxSettings (class Settings) Resource

This string contains a list of newline-separated commands that are also initially sent to DBX. By default, it is empty.

2.5.2.3 XDB Initialization

xdbInitCommands (class InitCommands) Resource

This string contains a list of newline-separated commands that are initially sent to XDB. By default, it is empty.

Do not use this resource to customize DBX; instead, use a personal '`~/.xdbrc`' file. See your XDB documentation for details.

xdbSettings (class Settings) Resource

This string contains a list of newline-separated commands that are also initially sent to XDB. By default, it is empty.

2.5.2.4 JDB Initialization

jdbInitCommands (class InitCommands) Resource

This string contains a list of newline-separated commands that are initially sent to JDB. This resource may be used to customize JDB. By default, it is empty.

jdbSettings (class Settings) Resource

This string contains a list of newline-separated commands that are also initially sent to JDB. By default, it is empty.

This resource is used by DDD to save and restore JDB settings.

2.5.2.5 PYDB Initialization

pydbInitCommands (class InitCommands) Resource

This string contains a list of newline-separated commands that are initially sent to PYDB. By default, it is empty.

This resource may be used to customize PYDB.

pydbSettings (class Settings) Resource

This string contains a list of newline-separated commands that are also initially sent to PYDB. By default, it is empty.

This resource is used by DDD to save and restore PYDB settings.

2.5.2.6 Perl Initialization

perlInitCommands (class InitCommands) Resource

This string contains a list of newline-separated commands that are initially sent to the Perl debugger. By default, it is empty.

This resource may be used to customize the Perl debugger.

perlSettings (class Settings) Resource

This string contains a list of newline-separated commands that are also initially sent to the Perl debugger. By default, it is empty.

This resource is used by DDD to save and restore Perl debugger settings.

2.5.2.7 Bash Initialization

bashInitCommands (class InitCommands) Resource

This string contains a list of newline-separated commands that are initially sent to the Bash debugger. By default, it is empty.

This resource may be used to customize the Bash debugger.

bash (class Settings) Resource

This string contains a list of newline-separated commands that are also initially sent to the Perl debugger. By default, it is empty.

This resource is used by DDD to save and restore Bash debugger settings.

2.5.2.8 Finding a Place to Start

initSymbols (class InitSymbols) Resource

When loading an executable, DDD queries the inferior debugger for the initial source location—typically the `main` function. If this location is not found, DDD tries other symbols from this newline-separated list. The default value makes DDD look for a variety of main functions (especially FORTRAN main functions):

```
main\n\
MAIN\n\
main_\n\
MAIN_\n\
main__\n\
MAIN__\n\
_main\n\
_MAIN\n\
__main\n\
__MAIN
```

2.5.2.9 Opening the Selection

openSelection (class OpenSelection) Resource

If this is 'on', DDD invoked without argument checks whether the current selection or clipboard contains the file name or URL of an executable program. If this is so, DDD will automatically open this program for debugging. If this resource is 'off' (default), DDD invoked without arguments will always start without a debugged program.

2.5.3 Communication with the Inferior Debugger

The following resources control the communication with the inferior debugger.

blockTTYInput (class BlockTTYInput) Resource

Whether DDD should block when reading data from the inferior debugger via the pseudo-tty interface. Most UNIX systems except GNU/Linux *require* this; set it to 'on'. On GNU/Linux, set it to 'off'. The value 'auto' (default) will always select the "best" choice (that is, the best choice known to the DDD developers).

bufferGDBOutput (class BufferGDBOutput) Resource

If this is 'on', all output from the inferior debugger is buffered until a debugger prompt appears. This makes it easier for DDD to parse the output, but has the drawback that interaction with a running debuggee in the debugger console is not possible. If 'off', output is shown as soon as it arrives, enabling interaction, but making it harder for DDD to parse the output. If 'auto' (default), output is buffered if and only if the execution window is open, which redirects debuggee output and thus enables interaction. See Section 6.2 [Using the Execution Window], page 93, for details.

contInterruptDelay (class InterruptDelay) Resource

The time (in ms) to wait before automatically interrupting a 'cont' command. DDD cannot interrupt a 'cont' command immediately, because this may disturb the status change of the process. Default is 200.

displayTimeout (class DisplayTimeout) Resource

The time (in ms) to wait for the inferior debugger to finish a partial display information. Default is 2000.

positionTimeout (class PositionTimeout) Resource

The time (in ms) to wait for the inferior debugger to finish a partial position information. Default is `500`.

questionTimeout (class QuestionTimeout) Resource

The time (in seconds) to wait for the inferior debugger to reply. Default is `10`.

runInterruptDelay (class InterruptDelay) Resource

The time (in ms) to wait before automatically interrupting a '`run`' command. DDD cannot interrupt a '`cont`' command immediately, because this may disturb process creation. Default is `2000`.

stopAndContinue (class StopAndContinue) Resource

If '`on`' (default), debugger commands interrupt program execution, resuming execution after the command has completed. This only happens if the last debugger command was either a '`run`' or a '`continue`' command. If '`off`', debugger commands do not interrupt program execution.

synchronousDebugger (class SynchronousDebugger) Resource

If '`on`', X events are not processed while the debugger is busy. This may result in slightly better performance on single-processor systems. See Section 2.1.2 [Options], page 16, for the '`--sync-debugger`' option.

terminateOnEOF (class TerminateOnEOF) Resource

If '`on`', DDD terminates the inferior debugger when DDD detects an EOF condition (that is, as soon as the inferior debugger closes its output channel). This was the default behavior in DDD 2.x and earlier. If '`off`' (default), DDD takes no special action.

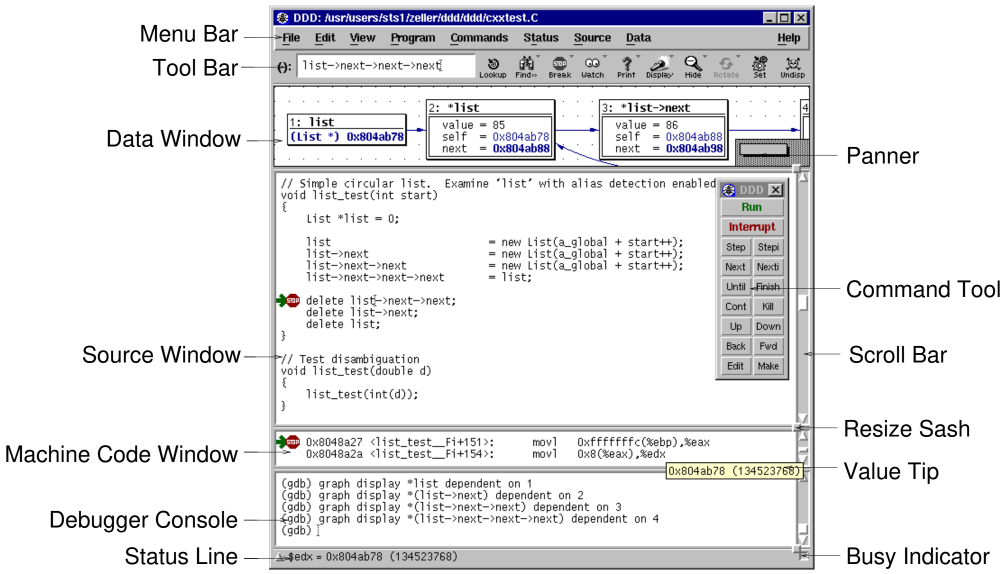
useTTYCommand (class UseTTYCommand) Resource

If '`on`', use the GDB `tty` command for redirecting input/output to the separate execution window. If '`off`', use explicit redirection through shell redirection operators '<' and '>'. The default is '`off`' (explicit redirection), since on some systems, the `tty` command does not work properly on some GDB versions.

3 The DDD Windows

DDD is composed of three main windows. From top to bottom, we have:

- The *Data Window* shows the current data of the debugged program. See Section 7.3 [Displaying Values], page 107, for details.
- The *Source Window* shows the current source code of the debugged program. See Chapter 4 [Navigating], page 73, for details.
- The *Debugger Console* accepts debugger commands and shows debugger messages. See Chapter 10 [Commands], page 147, for details.



The DDD Layout using Stacked Windows

Besides these three main windows, there are some other optional windows:

- The *Command Tool* offers buttons for frequently used commands. It is usually placed on the source window. See Section 3.3 [Command Tool], page 55, for details.
- The *Machine Code Window* shows the current machine code. It is usually placed beneath the current source. See Section 8.1 [Machine Code], page 141, for details.
- The *Execution Window* shows the input and output of the debugged program. See Section 6.2 [Using the Execution Window], page 93, for details.

3.1 The Menu Bar

The DDD Menu Bar gives you access to all DDD functions.

`File` Perform file-related operations such as selecting programs, processes, and sessions, printing graphs, recompiling, as well as exiting DDD.

`Edit` Perform standard editing operations, such as cutting, copying, pasting, and killing selected text. Also allows editing DDD options and preferences.

`View` Allows accessing the individual DDD windows.

`Program` Perform operations related to the program being debugged, such as starting and stopping the program.

`Commands`
Perform operations related to DDD commands, such as accessing the command history or defining new commands.

`Status` Examine the program status, such as the stack traces, registers, or threads.

`Source` Perform source-related operations such as looking up items or editing breakpoints.

`Data` Perform data-related operations such as editing displays or layouting the display graph.

`Maintenance`
Perform operations that are useful for debugging DDD. By default, this menu is disabled.

`Help` Give help on DDD usage.

There are two ways of selecting an item from a pull-down menu:

- Select an item in the menu bar by moving the cursor over it and click *mouse button 1*. Then move the cursor over the menu item you want to choose and click left again.
- Select an item in the menu bar by moving the cursor over it and click and hold *mouse button 1*. With the mouse button depressed, move the cursor over the menu item you want, then release it to make your selection.

The menus can also be *torn off* (i.e. turned into a persistent window) by selecting the dashed line at the top.

If a command in the pull-down menu is not applicable in a given situation, the command is *disabled* and its name appears faded. You cannot invoke items that are faded. For example, many commands on the '`Edit`' menu appear faded until you select text on which they are to operate; after you select a block of text, edit commands are enabled.

3.1.1 The File Menu

The '`File`' menu contains file-related operations such as selecting programs, processes, and sessions, printing graphs, recompiling, as well as exiting DDD.

`Open Program`
`Open Class`
Open a program or class to be debugged (⟨Ctrl+O⟩). See Section 4.2.1 [Opening Programs], page 73, for details.

`Open Recent`
Re-open a recently opened program to be debugged. See Section 4.2.1 [Opening Programs], page 73, for details.

`Open Core Dump`
Open a core dump for the currently debugged program. See Section 4.2.2 [Opening Core Dumps], page 74, for details.

`Open Source`
: Open a source file of the currently debugged program. See Section 4.2.3 [Opening Source Files], page 74, for details.

`Open Session`
: Resume a previously saved DDD session (⟨Ctrl+N⟩). See Section 2.3.2 [Resuming Sessions], page 30, for details.

`Save Session As`
: Save the current DDD session such that you can resume it later (⟨Ctrl+S⟩). See Section 2.3.1 [Saving Sessions], page 29, for details.

`Attach to Process`
: Attach to a running process of the debugged program. See Section 6.3 [Attaching to a Process], page 94, for details.

`Detach Process`
: Detach from the running process. See Section 6.3 [Attaching to a Process], page 94, for details.

`Print Graph`
: Print the current graph on a printer. See Section 7.3.7 [Printing the Graph], page 131, for details.

`Change Directory`
: Change the working directory of your program. See Section 6.1.3 [Working Directory], page 92, for details.

`Make`
: Run the `make` program (⟨Ctrl+M⟩). See Section 9.2 [Recompiling], page 146, for details.

`Close`
: Close this DDD window (⟨Ctrl+W⟩). See Section 2.2 [Quitting], page 28, for details.

`Restart`
: Restart DDD.

`Exit`
: Exit DDD (⟨Ctrl+Q⟩). See Section 2.2 [Quitting], page 28, for details.

3.1.2 The Edit Menu

The '`Edit`' menu contains standard editing operations, such as cutting, copying, pasting, and killing selected text. Also allows editing DDD options and preferences.

`Undo`
: Undo the most recent action (⟨Ctrl+Z⟩). Almost all commands can be undone this way. See Section 3.5 [Undo and Redo], page 60, for details.

`Redo`
: Redo the action most recently undone (⟨Ctrl+Y⟩). Every command undone can be redone this way. See Section 3.5 [Undo and Redo], page 60, for details.

`Cut`
: Removes the selected text block from the current text area and makes it the X clipboard selection (⟨Ctrl+X⟩ or ⟨Shift+Del⟩; See Section 3.1.11.2 [Customizing the Edit Menu], page 51, for details). Before executing this command, you have to select a region in a text area—either with the mouse or with the usual text selection keys.

  This item can also be applied to displays (see Section 7.3.1.12 [Deleting Displays], page 116).

`Copy` Makes a selected text block the X clipboard selection (⟨Ctrl+C⟩ or ⟨Ctrl+Ins⟩; See Section 3.1.11.2 [Customizing the Edit Menu], page 51, for details). You can select text by selecting a text region with the usual text selection keys or with the mouse. See Section 3.1.11.2 [Customizing the Edit Menu], page 51, for changing the default accelerator.

This item can also be applied to displays (see Section 7.3.1.12 [Deleting Displays], page 116).

`Paste` Inserts the current value of the X clipboard selection in the most recently selected text area (⟨Ctrl+V⟩ or ⟨Shift+Ins⟩; See Section 3.1.11.2 [Customizing the Edit Menu], page 51, for details). You can paste in text you have placed in the clipboard using '`Copy`' or '`Cut`'. You can also use '`Paste`' to insert text that was pasted into the clipboard from other applications.

`Clear` Clears the most recently selected text area (⟨Ctrl+U⟩).

`Delete` Removes the selected text block from the most recently selected text area, but does not make it the X clipboard selection.

This item can also be applied to displays (see Section 7.3.1.12 [Deleting Displays], page 116).

`Select All`
Selects all characters from the most recently selected text area (⟨Ctrl+A⟩ or or ⟨Ctrl+Shift+A⟩; see Section 3.1.11.2 [Customizing the Edit Menu], page 51, for details).

`Preferences`
Allows you to customize DDD interactively. See Section 3.6 [Customizing], page 60, for details.

`Debugger Settings`
Allows you to customize the inferior debugger. See Section 3.6.5 [Debugger Settings], page 71, for details.

`Save Options`
If set, all preferences and settings will be saved for the next DDD invocation. See Section 3.6.1.3 [Saving Options], page 61, for details.

3.1.3 The View Menu

The '`View`' menu allows accessing the individual DDD windows.

`Command Tool`
Open and recenter the command tool (⟨Alt+8⟩). See Section 3.3 [Command Tool], page 55, for details.

`Execution Window`
Open the separate execution window (⟨Alt+9⟩). See Section 6.2 [Using the Execution Window], page 93, for details.

`Debugger Console`
Open the debugger console (⟨Alt+1⟩). See Chapter 10 [Commands], page 147, for details.

`Source Window`
: Open the source window (⟨Alt+2⟩). See Chapter 4 [Navigating], page 73, for details.

`Data Window`
: Open the data window (⟨Alt+3⟩). See Section 7.3 [Displaying Values], page 107, for details.

`Machine Code Window`
: Show machine code (⟨Alt+4⟩). See Section 8.1 [Machine Code], page 141, for details.

3.1.4 The Program Menu

The '`Program`' menu performs operations related to the program being debugged, such as starting and stopping the program.

Most of these operations are also found on the command tool (see Section 3.3 [Command Tool], page 55).

`Run`
: Start program execution, prompting for program arguments (⟨F2⟩). See Section 6.1 [Starting Program Execution], page 91, for details.

`Run Again`
: Start program execution with the most recently used arguments (⟨F3⟩). See Section 6.1 [Starting Program Execution], page 91, for details.

`Run in Execution Window`
: If enabled, start next program execution in separate execution window. See Section 6.2 [Using the Execution Window], page 93, for details.

`Step`
: Continue running your program until control reaches a different source line, then stop it and return control to DDD (⟨F5⟩). See Section 6.5 [Resuming Execution], page 96, for details.

`Step Instruction`
: Execute one machine instruction, then stop and return to DDD (⟨Shift+F5⟩). See Section 8.2 [Machine Code Execution], page 142, for details.

`Next`
: Continue to the next source line in the current (innermost) stack frame (⟨F6⟩). This is similar to '`Step`', but function calls that appear within the line of code are executed without stopping. See Section 6.5 [Resuming Execution], page 96, for details.

`Next Instruction`
: Execute one machine instruction, but if it is a function call, proceed until the function returns (⟨Shift+F6⟩). See Section 8.2 [Machine Code Execution], page 142, for details.

`Until`
: Continue running until a source line past the current line, in the current stack frame, is reached (⟨F7⟩). See Section 6.5 [Resuming Execution], page 96, for details.

`Finish`
: Continue running until just after function in the selected stack frame returns (⟨F8⟩). Print the returned value (if any). See Section 6.5 [Resuming Execution], page 96, for details.

`Continue`
: Resume program execution, at the address where your program last stopped (⟨F9⟩); any breakpoints set at that address are bypassed. See Section 6.5 [Resuming Execution], page 96, for details.

`Continue Without Signal`
: Continue execution without giving a signal (⟨Shift+F9⟩). This is useful when your program stopped on account of a signal and would ordinary see the signal when resumed with '`Continue`'. See Section 6.10 [Signals], page 102, for details.

`Kill`
: Kill the process of the debugged program (⟨F4⟩). See Section 6.11 [Killing the Program], page 104, for details.

`Interrupt`
: Interrupt program execution (⟨Esc⟩ or ⟨Ctrl+C⟩; see Section 3.1.11.2 [Customizing the Edit Menu], page 51, for details). This is equivalent to sending an interrupt signal to the process. See Section 5.3 [Interrupting], page 88, for details.

`Abort`
: Abort program execution (and maybe debugger execution, too; ⟨Ctrl+\⟩). This is equivalent to sending a `SIGABRT` signal to the process. See Section 2.2 [Quitting], page 28, for details.

3.1.5 The Commands Menu

The '`Commands`' menu performs operations related to DDD commands, such as accessing the command history or defining new commands.

Most of these items are not meant to be actually executed via the menu; instead, they serve as *reminder* for the equivalent keyboard commands.

`Command History`
: View the command history. See Section 10.1.2 [Command History], page 148, for details.

`Previous`
: Show the previous command from the command history (⟨Up⟩). See Section 10.1.2 [Command History], page 148, for details.

`Next`
: Show the next command from the command history (⟨Down⟩). See Section 10.1.2 [Command History], page 148, for details.

`Find Backward`
: Do an incremental search backward through the command history (⟨Ctrl+B⟩). See Section 10.1.2 [Command History], page 148, for details.

`Find Forward`
: Do an incremental search forward through the command history (⟨Ctrl+F⟩). See Section 10.1.2 [Command History], page 148, for details.

`Quit Search`
: Quit incremental search through the command history (⟨Esc⟩). See Section 10.1.2 [Command History], page 148, for details.

`Complete`
: Complete the current command in the debugger console (⟨Tab⟩). See Section 10.1 [Entering Commands], page 147, for details.

`Apply`
: Apply the current command in the debugger console (⟨Apply⟩). See Section 10.1 [Entering Commands], page 147, for details.

`Clear Line`
: Clear the current command line in the debugger console (⟨Ctrl+U⟩). See Section 10.1 [Entering Commands], page 147, for details.

`Clear Window`
: Clear the debugger console (⟨Shift+Ctrl+U⟩). See Section 10.1 [Entering Commands], page 147, for details.

`Define Command`
: Define a new debugger command. See Section 10.5 [Defining Commands], page 154, for details.

`Edit Buttons`
: Customize DDD buttons. See Section 10.4 [Defining Buttons], page 151, for details.

3.1.6 The Status Menu

The '`Status`' menu lets you examine the program status, such as the stack traces, registers, or threads.

`Backtrace`
: View the current backtrace. See Section 6.7.2 [Backtraces], page 99, for a discussion.

`Registers`
: View the current register contents. See Section 8.3 [Registers], page 142, for details.

`Threads`
: View the current threads. See Section 6.9 [Threads], page 101, for details.

`Signals`
: View and edit the current signal handling. See Section 6.10 [Signals], page 102, for details.

`Up`
: Select the stack frame (i.e. the function) that called this one (⟨Ctrl+Up⟩). This advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. See Section 6.7 [Stack], page 98, for details.

`Down`
: Select the stack frame (i.e. the function) that was called by this one (⟨Ctrl+Down⟩). This advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. See Section 6.7 [Stack], page 98, for details.

3.1.7 The Source Menu

The '`Source`' menu performs source-related operations such as looking up items or editing breakpoints.

`Breakpoints`
: Edit all Breakpoints. See Section 5.1.11 [Editing all Breakpoints], page 86, for details.

`Lookup ()`
: Look up the argument '`()`' in the source code (⟨Ctrl+/⟩). See Section 4.3.1 [Looking up Definitions], page 75, for details.

`Find >> ()`
: Look up the next occurrence of the argument '`()`' in the current source code (⟨Ctrl+.⟩). See Section 4.3.2 [Textual Search], page 76, for details.

`Find << ()`
: Look up the previous occurrence of the argument '`()`' in the current source code (⟨Ctrl+,⟩). See Section 4.3.2 [Textual Search], page 76, for details.

`Find Words Only`
: If enabled, find only complete words (⟨Alt+W⟩). See Section 4.3.2 [Textual Search], page 76, for details.

`Find Case Sensitive`
: If enabled, find is case-sensitive (⟨Alt+I⟩). See Section 4.3.2 [Textual Search], page 76, for details.

`Display Line Numbers`
: If enabled, prefix source lines with their line number (⟨Alt+N⟩). See Section 4.4 [Customizing Source], page 77, for details.

`Display Machine Code`
: If enabled, show machine code (⟨Alt+4⟩). See Section 8.1 [Machine Code], page 141, for details.

`Edit Source`
: Invoke an editor for the current source file (⟨Shift+Ctrl+V⟩). See Section 9.1 [Editing Source Code], page 145, for details.

`Reload Source`
: Reload the current source file (⟨Shift+Ctrl+L⟩). See Section 9.1 [Editing Source Code], page 145, for details.

3.1.8 The Data Menu

The '`Data`' menu performs data-related operations such as editing displays or layouting the display graph.

`Displays`
: Invoke the Display Editor. See Section 7.3.1.11 [Editing all Displays], page 115, for details.

`Watchpoints`
: Edit all Watchpoints. See Section 5.2.3 [Editing all Watchpoints], page 88, for details.

`Memory`
: View a memory dump. See Section 7.5 [Examining Memory], page 138, for details.

`Print ()`
: Print the value of '`()`' in the debugger console (⟨Ctrl+=⟩). See Section 7.2 [Printing Values], page 106, for details.

`Display ()`
: Display the value of '`()`' in the data window (⟨Ctrl+-⟩). See Section 7.3 [Displaying Values], page 107, for details.

`Detect Aliases`
: If enabled, detect shared data structures (⟨Alt+A⟩). See Section 7.3.4.3 [Shared Structures], page 121, for a discussion.

`Display Local Variables`
: Show all local variables in a display (⟨Alt+L⟩). See Section 7.3.1.5 [Displaying Local Variables], page 111, for details.

`Display Arguments`
: Show all arguments of the current function in a display (⟨Alt+U⟩). See Section 7.3.1.5 [Displaying Local Variables], page 111, for details.

`Status Displays`
: Show current debugging information in a display. See Section 7.3.1.6 [Displaying Program Status], page 112, for details.

`Align on Grid`
: Align all displays on the grid (⟨Alt+G⟩). See Section 7.3.6.3 [Aligning Displays], page 130, for a discussion.

`Rotate Graph`
: Rotate the graph by 90 degrees (⟨Alt+R⟩). See Section 7.3.6.5 [Rotating the Graph], page 131, for details.

`Layout Graph`
: Layout the graph (⟨Alt+Y⟩). See Section 7.3.6 [Layouting the Graph], page 129, for details.

`Refresh`
: Update all values in the data window (⟨Ctrl+L⟩). See Section 7.3.1.7 [Refreshing the Data Window], page 113, for details.

3.1.9 The Maintenance Menu

The '`Maintenance`' menu performs operations that are useful for debugging DDD.

By default, this menu is disabled; it is enabled by specifically requesting it at DDD invocation (via the '`--maintenance`' option; see Section 2.1.2 [Options], page 16). It is also enabled when DDD gets a fatal signal.

`Debug DDD`
: Invoke a debugger (typically, GDB) and attach it to this DDD process (⟨F12⟩). This is useful only if you are a DDD maintainer.

`Dump Core Now`
: Make this DDD process dump core. This can also be achieved by sending DDD a `SIGUSR1` signal.

`Tic Tac Toe`
: Invoke a Tic Tac Toe game. You must try to get three stop signs in a row, while preventing DDD from doing so with its skulls. Click on '`New Game`' to restart.

`When DDD Crashes`
: Select what to do when DDD gets a fatal signal.

    `Debug DDD`
    : Invoke a debugger on the DDD core dump when DDD crashes. This is useful only if you are a DDD maintainer.

`Dump Core`
: Just dump core when DDD crashes; don't invoke a debugger. This is the default setting, as the core dump may contain important information required for debugging DDD.

`Do Nothing`
: Do not dump core or invoke a debugger when DDD crashes.

`Remove Menu`
: Make this menu inaccessible again.

3.1.10 The Help Menu

The '`Help`' menu gives help on DDD usage. See Section 3.4 [Getting Help], page 59, for a discussion on how to get help within DDD.

`Overview`
: Explains the most important concepts of DDD help.

`On Item`
: Lets you click on an item to get help on it.

`On Window`
: Gives you help on this DDD window.

`What Now?`
: Gives a hint on what to do next.

`Tip of the Day`
: Shows the current tip of the day.

`DDD Reference`
: Shows the DDD Manual.

`DDD News`
: Shows what's new in this DDD release.

`Debugger Reference`
: Shows the on-line documentation for the inferior debugger.

`DDD License`
: Shows the DDD License (see Appendix G [License], page 187).

`DDD WWW Page`
: Invokes a WWW browser for the DDD WWW page.

`About DDD`
: Shows version and copyright information.

3.1.11 Customizing the Menu Bar

The Menu Bar can be customized in various ways (see Section 3.6 [Customizing], page 60).

3.1.11.1 Auto-Raise Menus

You can cause pull-down menus to be raised automatically.

autoRaiseMenu (class AutoRaiseMenu) Resource

If '`on`' (default), DDD will always keep the pull down menu on top of the DDD main window. If this setting interferes with your window manager, or if your window manager does not auto-raise windows, set this resource to '`off`'.

autoRaiseMenuDelay (class AutoRaiseMenuDelay) Resource

The time (in ms) during which an initial auto-raised window blocks further auto-raises. This is done to prevent two overlapping auto-raised windows from entering an *auto-raise loop*. Default is `100`.

3.1.11.2 Customizing the Edit Menu

In the Menu Bar, the '`Edit`' Menu can be customized in various ways. Use '`Edit` ⇒ `Preferences` ⇒ `Startup`' to customize these keys.

The ⟨Ctrl+C⟩ key can be bound to different actions, each in accordance with a specific style guide.

`Copy` This setting binds ⟨Ctrl+C⟩ to the Copy operation, as specified by the KDE style guide. In this setting, use ⟨ESC⟩ to interrupt the debuggee.

`Interrupt`

This (default) setting binds ⟨Ctrl+C⟩ to the Interrupt operation, as used in several UNIX command-line programs. In this setting, use ⟨Ctrl+Ins⟩ to copy text to the clipboard.

The ⟨Ctrl+A⟩ key can be bound to different actions, too.

`Select All`

This (default) setting binds ⟨Ctrl+A⟩ to the '`Select All`' operation, as specified by the KDE style guide. In this setting, use ⟨Home⟩ to move the cursor to the beginning of a line.

`Beginning of Line`

This setting binds ⟨Ctrl+A⟩ to the '`Beginning of Line`' operation, as used in several UNIX text-editing programs. In this setting, use ⟨Ctrl+Shift+A⟩ to select all text.

Here are the related DDD resources:

cutCopyPasteBindings (class BindingStyle) Resource

Controls the key bindings for clipboard operations.

- If this is '`Motif`' (default), Cut/Copy/Paste is on ⟨Shift+Del⟩/⟨Ctrl+Ins⟩/⟨Shift+Ins⟩. This is conformant to the M*tif style guide.
- If this is '`KDE`', Cut/Copy/Paste is on ⟨Ctrl+X⟩/⟨Ctrl+C⟩/⟨Ctrl+V⟩. This is conformant to the KDE style guide. Note that this means that ⟨Ctrl+C⟩ no longer interrupts the debuggee; use ⟨ESC⟩ instead.

selectAllBindings (class BindingStyle) Resource

Controls the key bindings for the '`Select All`' operation.

- If this is '`Motif`', Select All is on ⟨Shift+Ctrl+A⟩.
- If this is '`KDE`' (default), Select All is on ⟨Ctrl+A⟩. This is conformant to the KDE style guide. Note that this means that ⟨Ctrl+A⟩ no longer moves the cursor to the beginning of a line; use ⟨Home⟩ instead.

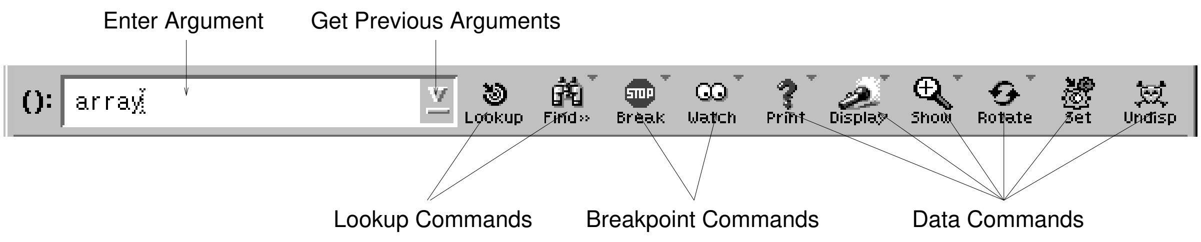
3.2 The Tool Bar

Some DDD commands require an *argument*. This argument is specified in the *argument field*, labeled '`():`'. Basically, there are four ways to set arguments:

- You can *key in* the argument manually.
- You can *paste* the current selection into the argument field (typically using ***mouse button 2***). To clear old contents beforehand, click on the '`():`' label.
- You can *select an item* from the source and data windows. This will automatically copy the item to the argument field.
- You can select a *previously used argument* from the drop-down menu at the right of the argument field.

Using GDB and Perl, the argument field provides a completion mechanism. You can enter the first few characters of an item an press the ⟨TAB⟩ key to complete it. Pressing ⟨TAB⟩ again shows alternative completions.

After having entered an argument, you can select one of the buttons on the right. Most of these buttons also have menus associated with them; this is indicated by a small arrow in the upper right corner. Pressing and holding *mouse button 1* on such a button will pop up a menu with further operations.



The Tool Bar

These are the buttons of the tool bar. Note that not all buttons may be inactive, depending on the current state and the capabilities of the inferior debugger.

`Lookup`

Look up the argument '`()`' in the source code. See Section 4.3.1 [Looking up Definitions], page 75, for details.

`Find >>`

Look up the next occurrence of the argument '`()`' in the current source code. See Section 4.3.2 [Textual Search], page 76, for details.

`Break/Clear`
: Toggle a breakpoint (see Section 5.1 [Breakpoints], page 81) at the location '`()`'.

  `Break` If there is no breakpoint at '`()`', then this button is labeled '`Break`'. Clicking on '`Break`' sets a breakpoint at the location '`()`'. See Section 5.1.1 [Setting Breakpoints], page 81, for details.

  `Clear` If there already is a breakpoint at '`()`', then this button is labeled '`Clear`'. Clicking on '`Clear`' clears (deletes) the breakpoint at the location '`()`'. See Section 5.1.2 [Deleting Breakpoints], page 82, for details.

`Watch/Unwatch`
: Toggle a watchpoint (see Section 5.2 [Watchpoints], page 87) on the expression '`()`'.

  `Watch` If '`()`' is not being watched, then this button is labeled '`Watch`'. Clicking on '`Watch`' creates a watchpoint on the expression '`()`'. See Section 5.2.1 [Setting Watchpoints], page 88, for details.

  `Unwatch` If '`()`' is being watched, then this button is labeled '`Unwatch`'. Clicking on '`Unwatch`' clears (deletes) the watchpoint on '`()`'. See Section 5.2.4 [Deleting Watchpoints], page 88, for details.

`Print`
: Print the value of '`()`' in the debugger console. See Section 7.2 [Printing Values], page 106, for details.

`Display`
: Display the value of '`()`' in the data window. See Section 7.3 [Displaying Values], page 107, for details.

`Plot`
: Plot '`()`' in a plot window. See Section 7.4 [Plotting Values], page 133, for details.

`Show/Hide`
: Toggle details of the selected display(s). See Section 7.3.1.3 [Showing and Hiding Details], page 109, for a discussion.

`Rotate`
: Rotate the selected display(s). See Section 7.3.1.4 [Rotating Displays], page 110, for details.

`Set`
: Set (change) the value of '`()`'. See Section 7.3.3 [Assignment], page 119, for details.

`Undisp`
: Undisplay (delete) the selected display(s). See Section 7.3.1.12 [Deleting Displays], page 116, for details.

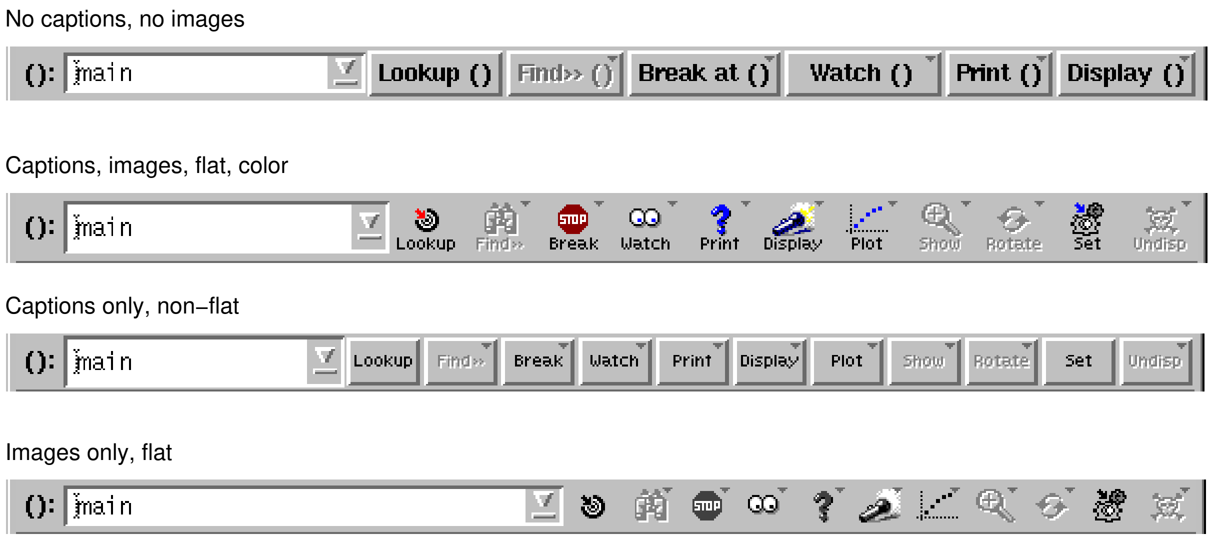
3.2.1 Customizing the Tool Bar

The DDD tool bar buttons can appear in a variety of styles, customized via '`Edit` ⇒ `Preferences` ⇒ `Startup`'.

`Images` This lets each tool bar button show an image illustrating the action.

`Captions`
This shows the action name below the image.

The default is to have images as well as captions, but you can choose to have only images (saving space) or only captions.



Tool Bar Appearance

If you choose to have neither images nor captions, tool bar buttons are labeled like other buttons, as in DDD 2.x. Note that this implies that in the stacked window configuration, the common tool bar cannot be displayed; it is replaced by two separate tool bars, as in DDD 2.x.

If you enable '`Flat`' buttons (default), the border of tool bar buttons will appear only if the mouse pointer is over them. This latest-and-greatest GUI invention can be disabled, such that the button border is always shown.

If you enable '`Color`' buttons, tool bar images will be colored when entered. If DDD was built using M*tif 2.0 and later, you can also choose a third setting, where buttons appear in color all the time.

Here are the related resources (see Section 3.6 [Customizing], page 60):

activeButtonColorKey (class ColorKey) Resource

The XPM color key to use for the images of active buttons (entered or armed). '`c`' means color, '`g`' (default) means grey, and '`m`' means monochrome.

buttonCaptions (class ButtonCaptions) Resource

Whether the tool bar buttons should be shown using captions (‘on’, default) or not (‘off’). If neither captions nor images are enabled, tool bar buttons are shown using ordinary labels. See also ‘buttonImages’, below.

buttonCaptionGeometry (class ButtonCaptionGeometry) Resource

The geometry of the caption subimage within the button icons. Default is ‘29x7+0-0’.

buttonImages (class ButtonImages) Resource

Whether the tool bar buttons should be shown using images (‘on’, default) or not (‘off’). If neither captions nor images are enabled, tool bar buttons are shown using ordinary labels. See also ‘buttonCaptions’, above.

buttonImageGeometry (class ButtonImageGeometry) Resource

The geometry of the image within the button icon. Default is ‘25x21+2+0’.

buttonColorKey (class ColorKey) Resource

The XPM color key to use for the images of inactive buttons (non-entered or insensitive). ‘c’ means color, ‘g’ (default) means grey, and ‘m’ means monochrome.

flatToolbarButtons (class FlatButtons) Resource

If ‘on’ (default), all tool bar buttons with images or captions are given a ‘flat’ appearance—the 3-D border only shows up when the pointer is over the icon. If ‘off’, the 3-D border is shown all the time.

flatDialogButtons (class FlatButtons) Resource

If ‘on’ (default), all dialog buttons with images or captions are given a ‘flat’ appearance—the 3-D border only shows up when the pointer is over the icon. If ‘off’, the 3-D border is shown all the time.

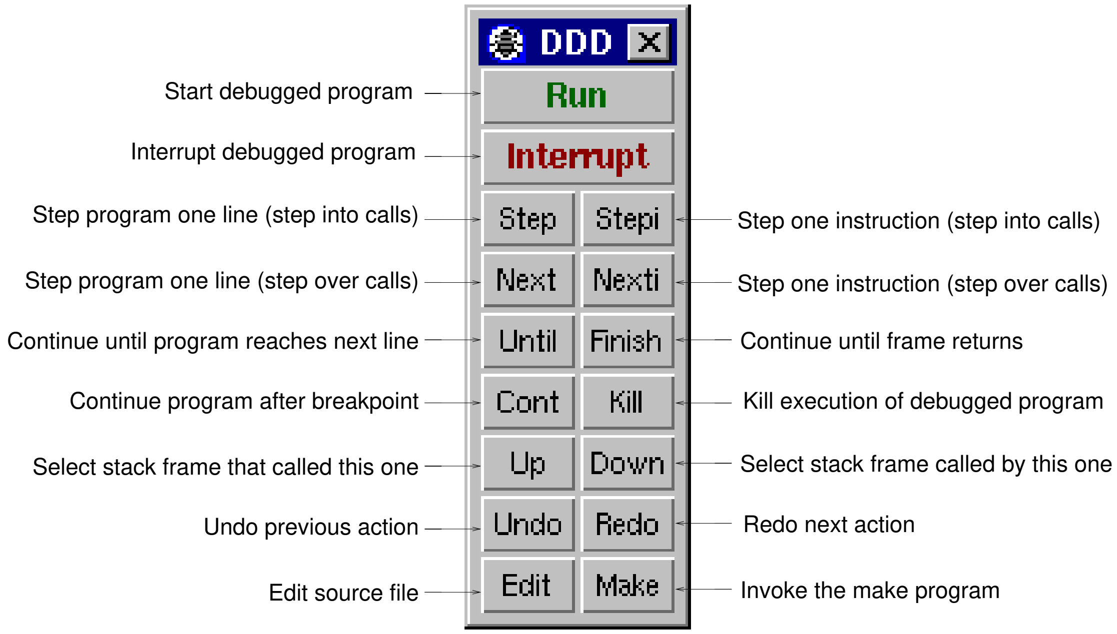
3.3 The Command Tool

The command tool is a small window that gives you access to the most frequently used DDD commands. It can be moved around on top of the DDD windows, but it can also be placed besides them.

By default, the command tool *sticks* to the DDD source window: Whenever you move the DDD source window, the command tool follows such that the distance between source window and command tool remains the same. By default, the command tool is also *auto-raised*, such that it stays on top of other DDD windows.

The command tool can be configured to appear as a command tool bar above the source window; see ‘Edit ⇒ Preferences ⇒ Source ⇒ Tool Buttons Location’ for details.

Whenever you save DDD state, DDD also saves the distance between command tool and source window, such that you can select your own individual command tool placement. To move the command tool to its saved position, use ‘View ⇒ Command Tool’.



The Command Tool

These are the buttons of the command tool. Note that not all buttons may be inactive, depending on the current state and the capabilities of the inferior debugger.

`Run` Start program execution. When you click this button, your program will begin to execute immediately. See Section 6.1 [Starting Program Execution], page 91, for details.

`Interrupt`
Interrupt program execution. This is equivalent to sending an interrupt signal to the process. See Section 5.3 [Interrupting], page 88, for details.

`Step` Continue running your program until control reaches a different source line, then stop it and return control to DDD. See Section 6.5 [Resuming Execution], page 96, for details.

`Stepi` Execute one machine instruction, then stop and return to DDD. See Section 8.2 [Machine Code Execution], page 142, for details.

`Next` Continue to the next source line in the current (innermost) stack frame. This is similar to '`Step`', but function calls that appear within the line of code are executed without stopping. See Section 6.5 [Resuming Execution], page 96, for details.

`Nexti` Execute one machine instruction, but if it is a function call, proceed until the function returns. See Section 8.2 [Machine Code Execution], page 142, for details.

`Until` Continue running until a source line past the current line, in the current stack frame, is reached. See Section 6.5 [Resuming Execution], page 96, for details.

`Finish` Continue running until just after function in the selected stack frame returns. Print the returned value (if any). See Section 6.5 [Resuming Execution], page 96, for details.

`Cont` Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. See Section 6.5 [Resuming Execution], page 96, for details.

`Kill` Kill the process of the debugged program. See Section 6.11 [Killing the Program], page 104, for details.

`Up` Select the stack frame (i.e. the function) that called this one. This advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. See Section 6.7 [Stack], page 98, for details.

`Down` Select the stack frame (i.e. the function) that was called by this one. This advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. See Section 6.7 [Stack], page 98, for details.

`Undo` Undo the most recent action. Almost all commands can be undone this way. See Section 3.5 [Undo and Redo], page 60, for details.

`Redo` Redo the action most recently undone. Every command undone can be redone this way. See Section 3.5 [Undo and Redo], page 60, for details.

`Edit` Invoke an editor for the current source file. See Section 9.1 [Editing Source Code], page 145, for details.

`Make` Run the `make` program with the most recently given arguments. See Section 9.2 [Recompiling], page 146, for details.

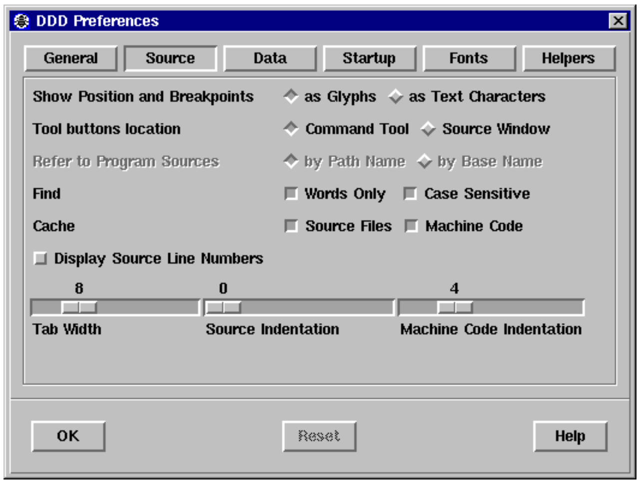
3.3.1 Customizing the Command Tool

The Command Tool can be customized in various ways.

See Section 10.4.1 [Customizing Buttons], page 152, for details on customizing the tool buttons.

3.3.1.1 Disabling the Command Tool

You can disable the command tool and show its buttons in a separate row beneath the tool bar. To disable the command tool, set '`Edit ⇒ Preferences ⇒ Source ⇒ Tool Buttons Location ⇒ Source Window`'.



Source Preferences

Here's the related resource:

commandToolBar (class ToolBar) Resource

Whether the tool buttons should be shown in a tool bar above the source window ('`on`') or within the command tool ('`off`', default). Enabling the command tool bar disables the command tool and vice versa.

3.3.2 Command Tool Position

The following resources control the position of the command tool (see Section 3.6 [Customizing], page 60):

autoRaiseTool (class AutoRaiseTool) Resource

If '`on`' (default), DDD will always keep the command tool on top of other DDD windows. If this setting interferes with your window manager, or if your window manager keeps the command tool on top anyway, set this resource to '`off`'.

stickyTool (class StickyTool) Resource

If '`on`' (default), the command tool automatically follows every movement of the source window. Whenever the source window is moved, the command tool is moved by the same offset such that its position relative to the source window remains unchanged. If '`off`', the command tool does not follow source window movements.

toolRightOffset (class Offset) Resource

The distance between the right border of the command tool and the right border of the source text (in pixels). Default is 8.

toolTopOffset (class Offset) Resource

The distance between the upper border of the command tool and the upper border of the source text (in pixels). Default is 8.

3.3.2.1 Customizing Tool Decoration

The following resources control the decoration of the command tool (see Section 3.6 [Customizing], page 60):

decorateTool (class Decorate) Resource

This resource controls the decoration of the command tool.

- If this is '`off`', the command tool is created as a *transient window*. Several window managers keep transient windows automatically on top of their parents, which is appropriate for the command tool. However, your window manager may be configured not to decorate transient windows, which means that you cannot easily move the command tool around.
- If this is '`on`', DDD realizes the command tool as a *top-level window*. Such windows are always decorated by the window manager. However, top-level windows are not automatically kept on top of other windows, such that you may wish to set the '`autoRaiseTool`' resource, too.
- If this is '`auto`' (default), DDD checks whether the window manager decorates transients. If yes, the command tool is realized as a transient window (as in the '`off`' setting); if no, the command tool is realized as a top-level window (as in the '`on`' setting). Hence, the command tool is always decorated using the "best" method, but the extra check takes some time.

3.4 Getting Help

DDD has an extensive on-line help system. Here's how to get help while working with DDD.

- You can get a short help text on most DDD buttons by simply moving the mouse pointer on it and leave it there. After a second, a small window (called *button tip*; also known as *tool tip* or *balloon help*) pops up, giving a hint on the button's meaning. The button tip disappears as soon as you move the mouse pointer to another item.
- The *status line* also displays information about the currently selected item. By clicking on the status line, you can redisplay the most recent messages.
- You can get detailed help on any visible DDD item. Just point on the item you want help and press the '`F1`' key. This pops up a detailed help text.
- The DDD dialogs all contain '`Help`' buttons that give detailed information about the dialog.
- You can get help on debugger commands by entering `help` at the debugger prompt. See Section 10.1 [Entering Commands], page 147, for details on entering commands.
- If you are totally stuck, try '`Help` ⇒ `What Now?`' (the '`What Now?`' item in the '`Help`' menu) or press ⟨Ctrl+F1⟩. Depending on the current state, DDD will give you some hints on what you can do next.
- Of course, you can always refer to the *on-line documentation*:
  - '`Help` ⇒ `DDD Reference`' gives you access to the DDD manual, the ultimate DDD reference.
  - '`Help` ⇒ `Debugger Reference`' shows you the on-line documentation of the inferior debugger.

   – '`Help` ⇒ DDD WWW `Page`' gives you access to the latest and greatest information on DDD.

- Finally, the DDD *Tip Of The Day* gives you important hints with each new DDD invocation.

All these functions can be customized in various ways (see Section 3.6.2 [Customizing Help], page 61).

If, after all, you made a mistake, don't worry: almost every DDD command can be undone. See Section 3.5 [Undo and Redo], page 60, for details.

3.5 Undoing and Redoing Commands

Almost every DDD command can be undone, using '`Edit` ⇒ `Undo`' or the '`Undo`' button on the command tool.

Likewise, '`Edit` ⇒ `Redo`' repeats the command most recently undone.

The '`Edit`' menu shows which commands are to be undone and redone next; this is also indicated by the popup help on the '`Undo`' and '`Redo`' buttons.

3.6 Customizing DDD

DDD is controlled by several *resources*—user-defined variables that take specific values in order to control and customize DDD behavior.

Most DDD resources can be set interactively while DDD is running or when invoking DDD. See [Resource Index], page 211, for the full list of DDD resources.

We first discuss how customizing works in general; then we turn to customizing parts of DDD introduced so far.

3.6.1 How Customizing DDD Works

3.6.1.1 Resources

Just like any X program, DDD has a number of places to get resource values from. For DDD, the most important places to specify resources are:

- The '`~/.ddd/init`' file ('`~`' stands for your home directory). This file is read in by DDD upon start-up; the resources specified herein override all other sources (except for resources given implicitly by command-line options).

  If the environment variable `DDD_STATE` is set, its value is used instead of '`~/.ddd/`'.
- The '`Ddd`' application-defaults file. This file is typically compiled into the DDD executable. If it exists, its resource values override the values compiled into DDD. If the versions of the '`Ddd`' application-defaults file and the DDD executable do not match, DDD may not function properly; DDD will give you a warning in this case.[1]
- The command-line options. These options override all other resource settings.

---

[1] If you use a '`Ddd`' application-defaults file, you will not be able to maintain multiple DDD versions at the same time. This is why the suiting '`Ddd`' is normally compiled into the DDD executable.

- If the environment variable `DDD_SESSION` is set, it indicates the name of a session to start, overriding all options and resources. This is used by DDD when restarting itself.

Not every resource has a matching command-line option. Each resource (whether in '`~/.ddd/init`' or '`Ddd`') is specified using a line

`Ddd*`*resource*`:` *value*

For instance, to set the '`pollChildStatus`' resource to '`off`', you would specify in '`~/.ddd/init`':

```
Ddd*pollChildStatus: off
```

For more details on the syntax of resource specifications, see the section *RESOURCES* in the *X(1)* manual page.

3.6.1.2 Changing Resources

You can change DDD resources by three methods:

- Use DDD to change the options, notably '`Edit` ⇒ `Preferences`'. This works for the most important DDD resources. Be sure to save the options (see Section 3.6.1.3 [Saving Options], page 61) such that they apply to future DDD sessions, too.
- You can also invoke DDD with an appropriate command-line option. This changes the related DDD resource for this particular DDD invocation. However, if you save the options (see Section 3.6.1.3 [Saving Options], page 61), the changed resource will also apply to future invocations.
- Finally, you can set the appropriate resource in a file named '`.ddd/init`' in your home directory. See [Resource Index], page 211, for a list of DDD resources to be set.

3.6.1.3 Saving Options

You can save the current option settings by setting '`Edit` ⇒ `Save Options`'. Options are saved in a file named '`.ddd/init`' in your home directory when DDD exits. If a session *session* is active, options will be saved in '`~/.ddd/sessions/`*session*`/init`' instead.

The options are automatically saved when exiting DDD. You can turn off this feature by unsetting '`Edit` ⇒ `Save Options`'. This is tied to the following resource:

saveOptionsOnExit (class SaveOnExit) Resource

If '`on`' (default), the current option settings are automatically saved when DDD exits.

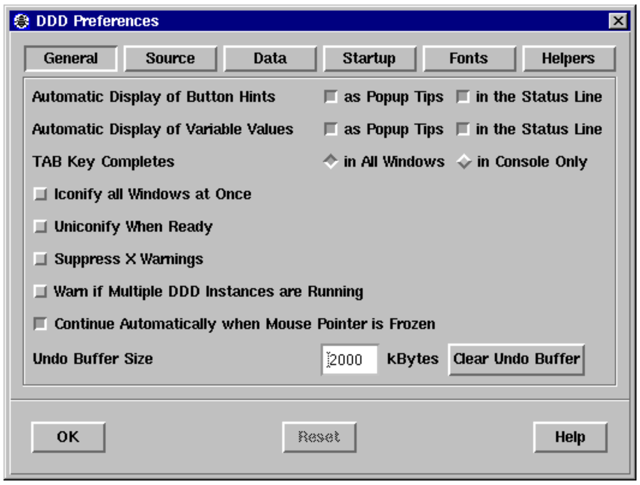
3.6.2 Customizing DDD Help

DDD Help can be customized in various ways.

3.6.2.1 Button Tips

Button tips are helpful for novices, but may be distracting for experienced users. You can turn off button tips via '`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Automatic display of Button Hints` ⇒ `as Popup Tips`'.

You can also turn off the hint that is displayed in the status line. Just toggle 'Edit ⇒ Preferences ⇒ General ⇒ Automatic Display of Button Hints ⇒ in the Status Line'.



General Preferences

These are the related DDD resources (see Section 3.6 [Customizing], page 60):

buttonTips (class Tips) Resource

If 'on' (default), enable button tips.

buttonDocs (class Docs) Resource

If 'on' (default), show button hints in the status line.

3.6.2.2 Tip of the day

You can turn off the tip of the day by toggling 'Edit ⇒ Preferences ⇒ Startup ⇒ Startup Windows ⇒ Tip of the Day'.

Here is the related DDD resource (see Section 3.6 [Customizing], page 60):

startupTips (class StartupTips) Resource

If 'on' (default), show a tip of the day upon DDD startup.

See Section 2.1.2 [Options], page 16, for options to set this resource upon DDD invocation.

The actual tips are controlled by these resources (see Section 3.6 [Customizing], page 60):

startupTipCount (class StartupTipCount) Resource

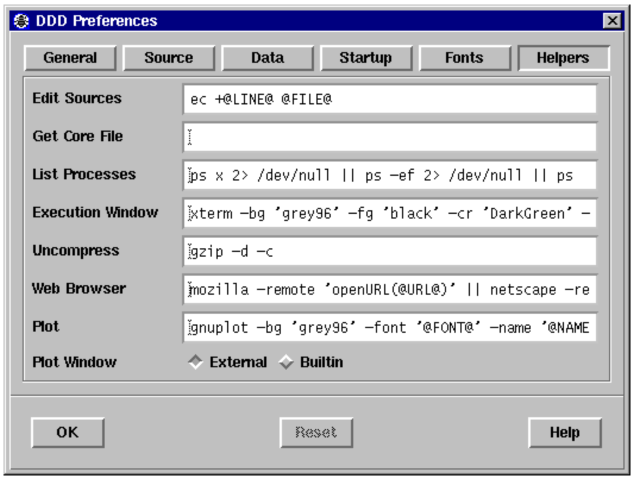
The number *n* of the tip of the day to be shown at startup. See also the 'tip*n*' resources.

tip*n* (class Tip) Resource

The tip of the day numbered *n* (a string).

3.6.2.3 Help Helpers

DDD relies on a number of external commands, specified via 'Edit ⇒ Preferences ⇒ Helpers'.



Setting Helpers Preferences

To uncompress help texts, you can define a 'Uncompress' command:

uncompressCommand (class UncompressCommand) Resource
The command to uncompress the built-in DDD manual, the DDD license, and the DDD news. Takes a compressed text from standard input and writes the uncompressed text to standard output. The default value is `gzip -d -c`; typical values include `zcat` and `gunzip -c`.

To view WWW pages, you can define a 'Web Browser' command:

wwwCommand (class WWWCommand) Resource
The command to invoke a WWW browser. The string '@URL@' is replaced by the URL to open. Default is to try a running Netscape first (trying `mozilla`, then `netscape`), then `$WWWBROWSER`, then to invoke a new Netscape process, then to let a running Emacs or XEmacs do the job (via `gnudoit`), then to invoke Mosaic, then to invoke Lynx in an xterm.

To specify 'netscape-6.0' as browser, use the setting:

```
Ddd*wwwCommand: \
     netscape-6.0 -remote 'openURL(@URL@)' \
  || netscape-6.0 '@URL@'
```

This command first tries to connect to a running `netscape-6.0` browser; if this fails, it starts a new `netscape-6.0` process.

This is the default WWW Page shown by 'Help ⇒ DDD WWW Page':

wwwPage (class WWWPage) Resource

The DDD WWW page. Value: `http://www.gnu.org/software/ddd/`

3.6.3 Customizing Undo

DDD Undo can be customized in various ways.

To set a maximum size for the undo buffer, set '`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Undo Buffer Size`'.

This is related to the '`maxUndoSize`' resource:

maxUndoSize (class MaxUndoSize) Resource

The maximum memory usage (in bytes) of the undo buffer. Useful for limiting DDD memory usage. A negative value means to place no limit. Default is `2000000`, or 2000 kBytes.

You can also limit the number of entries in the undo buffer, regardless of size (see Section 3.6 [Customizing], page 60):

maxUndoDepth (class MaxUndoDepth) Resource

The maximum number of entries in the undo buffer. This limits the number of actions that can be undone, and the number of states that can be shown in historic mode. Useful for limiting DDD memory usage. A negative value (default) means to place no limit.

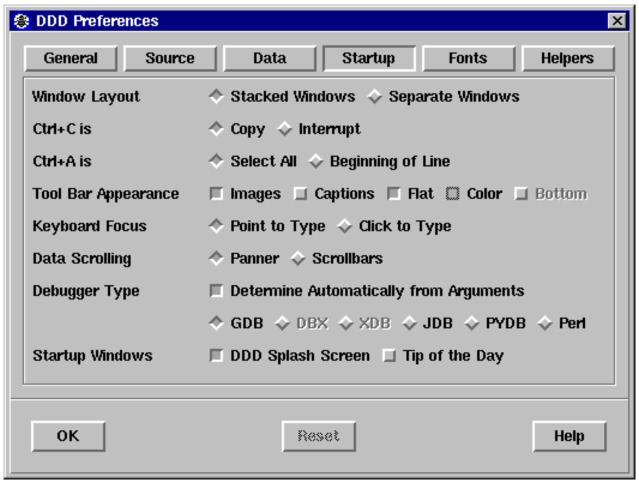
To clear the undo buffer at any time, thus reducing memory usage, use '`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Clear Undo Buffer`'

3.6.4 Customizing the DDD Windows

You can customize the DDD Windows in various ways.

3.6.4.1 Splash Screen

You can turn off the DDD splash screen shown upon startup. Just select '`Edit` ⇒ `Preferences` ⇒ `Startup DDD Splash Screen`'.



Startup Preferences

The value applies only to the next DDD invocation.

This setting is related to the following resource:

splashScreen (class SplashScreen) Resource

If 'on' (default), show a DDD splash screen upon start-up.

You can also customize the appearance of the splash screen (see Section 3.6 [Customizing], page 60):

splashScreenColorKey (class ColorKey) Resource

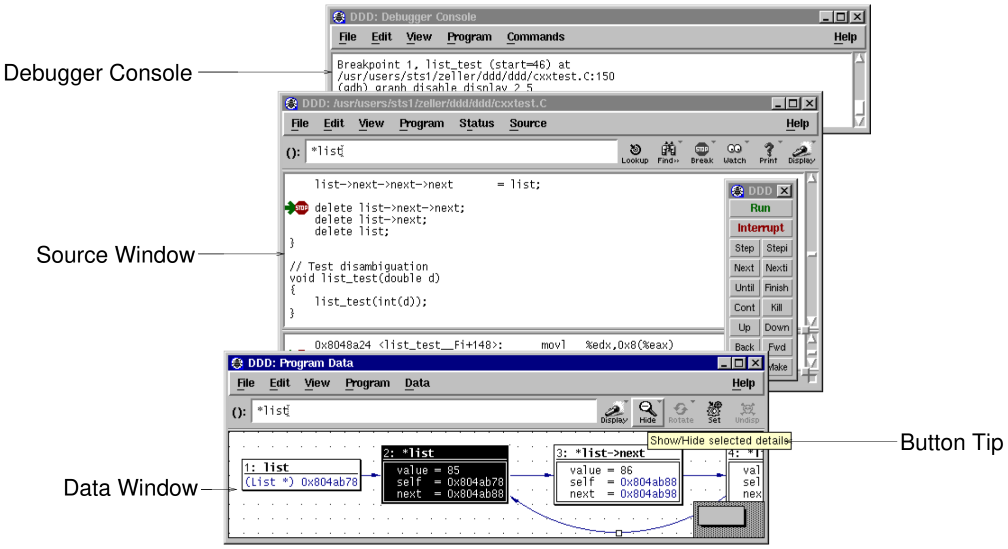
The color key to use for the DDD splash screen. Possible values include:

- 'c' (default) for a color visual,
- 'g' for a multi-level greyscale visual,
- 'g4' for a 4-level greyscale visual, and
- 'm' for a dithered monochrome visual.
- 'best' chooses the best visual available for your display.

Please note: if DDD runs on a monochrome display, or if DDD was compiled without the XPM library, only the monochrome version ('m') can be shown.

3.6.4.2 Window Layout

By default, DDD stacks commands, source, and data in one single top-level window. To have separate top-level windows for source, data, and debugger console, set 'Edit ⇒ Preferences ⇒ Startup ⇒ Window Layout ⇒ Separate Windows'.



The DDD Layout using Separate Windows

Here are the related DDD resources:

separateDataWindow (class Separate) Resource

If '`on`', the data window and the debugger console are realized in different top-level windows. If '`off`' (default), the data window is attached to the debugger console.

separateSourceWindow (class Separate) Resource

If '`on`', the source window and the debugger console are realized in different top-level windows. If '`off`' (default), the source window is attached to the debugger console.

By default, the DDD tool bars are located on top of the window. If you prefer the tool bar being located at the bottom, as in DDD 2.x and earlier, set '`Edit` ⇒ `Preferences` ⇒ `Startup` ⇒ `Tool Bar Appearance` ⇒ `Bottom`'.

This is related to the '`toolbarsAtBottom`' resource:

toolbarsAtBottom (class ToolbarsAtBottom) Resource

Whether source and data tool bars should be placed above source and data, respectively ('`off`', default), or below, as in DDD 2.x ('`on`').

The bottom setting is only supported for separate tool bars—that is, you must either choose separate windows or configure the tool bar to have neither images nor captions (see Section 3.2.1 [Customizing the Tool Bar], page 54).

If you use stacked windows, you can choose whether there should be one tool bar or two tool bars. By default, DDD uses two tool bars if you use separate windows and disable captions and images, but you can also explicitly change the setting via this resource:

commonToolBar (class ToolBar) Resource

Whether the tool bar buttons should be shown in one common tool bar at the top of the common DDD window (‘`on`’, default), or whether they should be placed in two separate tool bars, one for data, and one for source operations, as in DDD 2.x (‘`off`’).

You can also change the location of the *status line* (see Section 3.6 [Customizing], page 60):

statusAtBottom (class StatusAtBottom) Resource

If ‘`on`’ (default), the status line is placed at the bottom of the DDD source window. If ‘`off`’, the status line is placed at the top of the DDD source window (as in DDD 1.x).

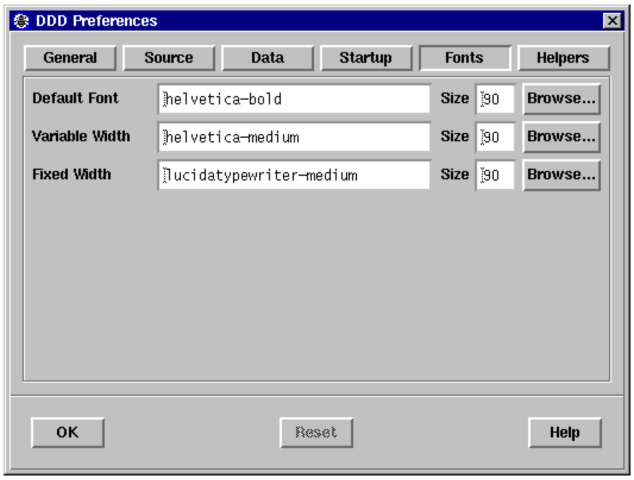
See Section 2.1.2 [Options], page 16, for options to set these resources upon DDD invocation.

3.6.4.3 Customizing Fonts

You can configure the basic DDD fonts at run-time. Each font is specified using two members:

- The *font family* is an X font specifications, where the initial ‘*foundry*−’ specification may be omitted, as well as any specification after *family*. Thus, a pair ‘*family*−*weight*’ usually suffices.
- The *font size* is given as (resolution-independent) 1/10 points.

To specify fonts, select ‘`Edit` ⇒ `Preferences` ⇒ `Fonts`’.



Setting Font Preferences

The ‘`Browse`’ button opens a font selection program, where you can select fonts and attributes interactively. Clicking ‘`quit`’ or ‘`select`’ in the font selector causes all non-default values to be transferred to the DDD font preferences panel.

The following fonts can be set using the preferences panel:

`Default Font`
: The default DDD font to use for labels, menus, and buttons. Default is '`helvetica-bold`'.

`Variable Width`
: The variable width DDD font to use for help texts and messages. Default is '`helvetica-medium`'.

`Fixed Width`
: The fixed width DDD font to use for source code, the debugger console, text fields, and the execution window. Default is '`lucidatypewriter-medium`'.

`Data`
: The DDD font to use for data displays. Default is '`lucidatypewriter-medium`'.

Changes in this panel will take effect only in the next DDD session. To make it effective right now, restart DDD (using '`File` ⇒ `Restart DDD`').

After having made changes in the panel, DDD will automatically offer you to restart itself, such that you can see the changes taking effect.

The '`Reset`' button restores the most recently saved preferences.

Here are the resources related to font specifications:

defaultFont (class Font) Resource

The default DDD font to use for labels, menus, buttons, etc. The font is specified as an X font spec, where the initial *Foundry* specification may be omitted, as well as any specification after *Family*.

Default value is '`helvetica-bold`'.

To set the default DDD font to, say, '`helvetica medium`', insert a line

```
Ddd*defaultFont: helvetica-medium
```

in your '`~/.ddd/init`' file.

defaultFontSize (class FontSize) Resource

The size of the default DDD font, in 1/10 points. This resource overrides any font size specification in the '`defaultFont`' resource (see above). The default value is `120` for a 12.0 point font.

variableWidthFont (class Font) Resource

The variable width DDD font to use for help texts and messages. The font is specified as an X font spec, where the initial *Foundry* specification may be omitted, as well as any specification after *Family*.

Default value is '`helvetica-medium-r`'.

To set the variable width DDD font family to, say, '`times`', insert a line

```
Ddd*fixedWidthFont: times-medium
```

in your '`~/.ddd/init`' file.

variableWidthFontSize (class FontSize) Resource

The size of the variable width DDD font, in 1/10 points. This resource overrides any font size specification in the '`variableWidthFont`' resource (see above). The default value is `120` for a 12.0 point font.

fixedWidthFont (class Font) Resource

The fixed width DDD font to use for source code, the debugger console, text fields, and the execution window. The font is specified as an X font spec, where the initial *Foundry* specification may be omitted, as well as any specification after *Family*.

Default value is '`lucidatypewriter-medium`'.

To set the fixed width DDD font family to, say, '`courier`', insert a line

```
Ddd*fixedWidthFont: courier-medium
```

in your '`~/.ddd/init`' file.

fixedWidthFontSize (class FontSize) Resource

The size of the fixed width DDD font, in 1/10 points. This resource overrides any font size specification in the '`fixedWidthFont`' resource (see above). The default value is `120` for a 12.0 point font.

dataFont (class Font) Resource

The fixed width DDD font to use data displays. The font is specified as an X font spec, where the initial *Foundry* specification may be omitted, as well as any specification after *Family*.

Default value is '`lucidatypewriter-medium`'.

To set the DDD data font family to, say, '`courier`', insert a line

```
Ddd*dataFont: courier-medium
```

in your '`~/.ddd/init`' file.

dataFontSize (class FontSize) Resource

The size of the DDD data font, in 1/10 points. This resource overrides any font size specification in the '`dataFont`' resource (see above). The default value is `120` for a 12.0 point font.

As all font size resources have the same class (and by default the same value), you can easily change the default DDD font size to, say, 9.0 points by inserting a line

```
Ddd*FontSize: 90
```

in your '`~/.ddd/init`' file.

Here's how to specify the command to select fonts:

fontSelectCommand (class FontSelectCommand) Resource

A command to select from a list of fonts. The string '`@FONT@`' is replaced by the current DDD default font; the string '`@TYPE@`' is replaced by a symbolic name of the DDD font to edit. The program must either place the name of the selected font in the `PRIMARY` selection or print the selected font on standard output. A typical value is:

```
Ddd*fontSelectCommand: xfontsel -print
```

See Section 2.1.2 [Options], page 16, for options to set these resources upon DDD invocation.

3.6.4.4 Toggling Windows

In the default stacked window setting, you can turn the individual DDD windows on and off by toggling the respective items in the 'View' menu (see Section 3.1.3 [View Menu], page 44). When using separate windows (see Section 3.6.4.2 [Window Layout], page 65), you can close the individual windows via 'File ⇒ Close' or by closing them via your window manager.

Whether windows are opened or closed when starting DDD is controlled by the following resources, immediately tied to the 'View' menu items:

openDataWindow (class Window) Resource
    If 'off' (default), the data window is closed upon start-up.

openDebuggerConsole (class Window) Resource
    If 'off', the debugger console is closed upon start-up.

openSourceWindow (class Window) Resource
    If 'off', the source window is closed upon start-up.

See Section 2.1.2 [Options], page 16, for options to set these resources upon DDD invocation.

3.6.4.5 Text Fields

The DDD text fields can be customized using the following resources:

popdownHistorySize (class HistorySize) Resource
    The maximum number of items to display in pop-down value histories. A value of 0 (default) means an unlimited number of values.

sortPopdownHistory (class SortPopdownHistory) Resource
    If 'on' (default), items in the pop-down value histories are sorted alphabetically. If 'off', most recently used values will appear at the top.

3.6.4.6 Icons

If you frequently switch between DDD and other multi-window applications, you may like to set 'Edit ⇒ Preferences ⇒ General ⇒ Iconify all windows at once'. This way, all DDD windows are iconified and deiconified as a group.

This is tied to the following resource:

groupIconify (class GroupIconify) Resource
    If this is 'on', (un)iconifying any DDD window causes all other DDD windows to (un)iconify as well. Default is 'off', meaning that each DDD window can be iconified on its own.

If you want to keep DDD off your desktop during a longer computation, you may like to set 'Edit ⇒ Preferences ⇒ General ⇒ Uniconify when ready'. This way, you can iconify DDD while it is busy on a command (e.g. running a program); DDD will automatically pop up again after becoming ready (e.g. after the debugged program has stopped at a breakpoint). See Section 6.4 [Program Stop], page 96, for a discussion.

Here is the related resource:

uniconifyWhenReady (class UniconifyWhenReady) Resource

If this is '`on`' (default), the DDD windows are uniconified automatically whenever GDB becomes ready. This way, you can iconify DDD during some longer operation and have it uniconify itself as soon as the program stops. Setting this to '`off`' leaves the DDD windows iconified.

3.6.4.7 Adding Buttons

You can extend DDD with new buttons. See Section 10.4 [Defining Buttons], page 151, for details.

3.6.4.8 More Customizations

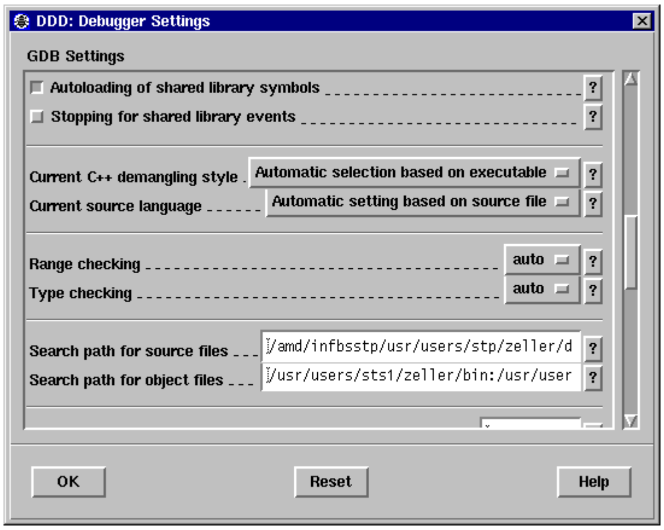
You can change just about any label, color, keyboard mapping, etc. by changing resources from the '`Ddd`' application defaults file which comes with the DDD source distribution. Here's how it works:

- Identify the appropriate resource in the '`Ddd`' file.
- Copy the resource line to your '`~/.ddd/init`' file and change it at will.

See Appendix A [Application Defaults], page 159, for details on the application-defaults file.

3.6.5 Debugger Settings

For most inferior debuggers, you can change their internal settings using '`Edit ⇒ Settings`'. Using the settings editor, you can determine whether C++ names are to be demangled, how many array elements are to print, and so on.



GDB Settings Panel (Excerpt)

The capabilities of the settings editor depend on the capabilities of your inferior debugger. Clicking on '?' gives an an explanation on the specific item; the GDB documentation gives more details.

Use 'Edit ⇒ Undo' to undo changes. Clicking on 'Reset' restores the most recently saved settings.

Some debugger settings are insensitive and cannot be changed, because doing so would endanger DDD operation. See the 'gdbInitCommands' and 'dbxInitCommands' resources for details.

All debugger settings (except source and object paths) are saved with DDD options.

4 Navigating through the Code

This chapter discusses how to access code from within DDD.

4.1 Compiling for Debugging

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.[1]

To request debugging information, specify the '-g' option when you run the compiler.

Many C compilers are unable to handle the '-g' and '-O' options together. Using those compilers, you cannot generate optimized executables containing debugging information.

GCC, the GNU C compiler, supports '-g' with or without '-O', making it possible to debug optimized code. We recommend that you *always* use '-g' whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck.

When you debug a program compiled with '-g -O', remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example: if you define a variable, but never use it, DDD never sees that variable—because the compiler optimizes it out of existence.

4.2 Opening Files

If you did not invoke DDD specifying a program to be debugged, you can use the 'File' menu to open programs, core dumps and sources.

4.2.1 Opening Programs

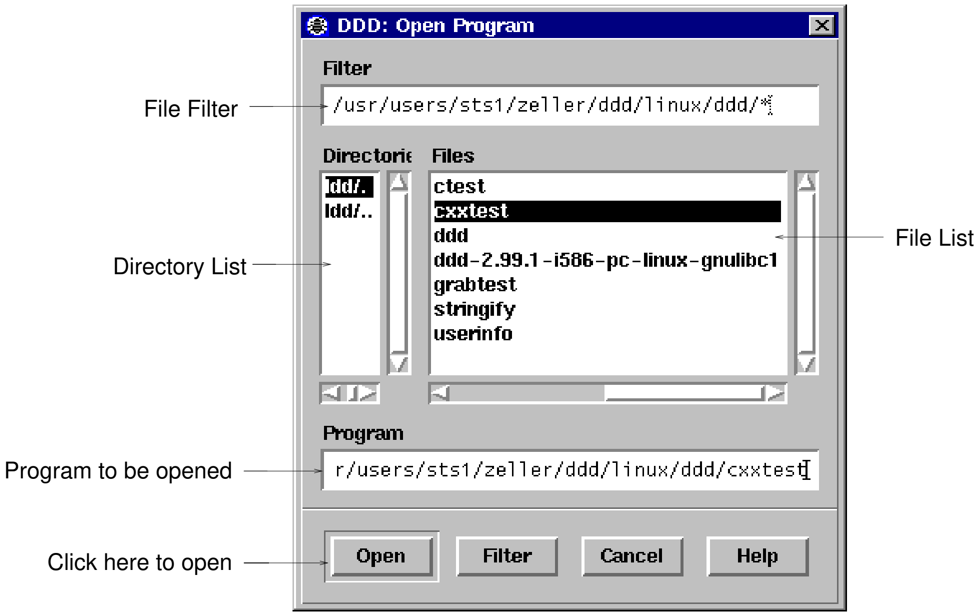
To open a program to be debugged, select 'File ⇒ Open Program'.[2] Click on 'Open' to open the program

In JDB, select 'File ⇒ Open Class' instead. This gives you a list of available classes to choose from.

---

[1] If you use DDD to debug Perl, Python or Bash scripts, then this section does not apply.

[2] With XDB and some DBX variants, the debugged program must be specified upon invocation and cannot be changed at run time.



Opening a program to be debugged

To re-open a recently debugged program or class, select '`File` ⇒ `Open Recent`' and choose a program or class from the list.

If no sources are found, See Section 4.3.4 [Source Path], page 76, for specifying source directories.

4.2.2 Opening Core Dumps

If a previous run of the program has crashed and you want to find out why, you can have DDD examine its *core dump*.[3]

To open a core dump for the program, select '`File` ⇒ `Open Core Dump`'. Click on '`Open`' to open the core dump.

Before '`Open Core Dump`', you should first use '`File` ⇒ `Open Program`' to specify the program that generated the core dump and to load its symbol table.

4.2.3 Opening Source Files

To open a source file of the debugged program, select '`File` ⇒ `Open Source`'.

- Using GDB, this gives you a list of the sources used for compiling your program.
- Using other inferior debuggers, this gives you a list of accessible source files, which may or may not be related to your program.

Click on '`Open`' to open the source file. See Section 4.3.4 [Source Path], page 76, if no sources are found.

[3] JDB, PYDB, Perl, and Bash do not support core dumps.

4.2.4 Filtering Files

When presenting files to be opened, DDD by default filters files when opening execution files, core dumps, or source files, such that the selection shows only suitable files. This requires that DDD opens each file, which may take time. See Section 4.4.6 [Customizing File Filtering], page 80, if you want to turn off this feature.

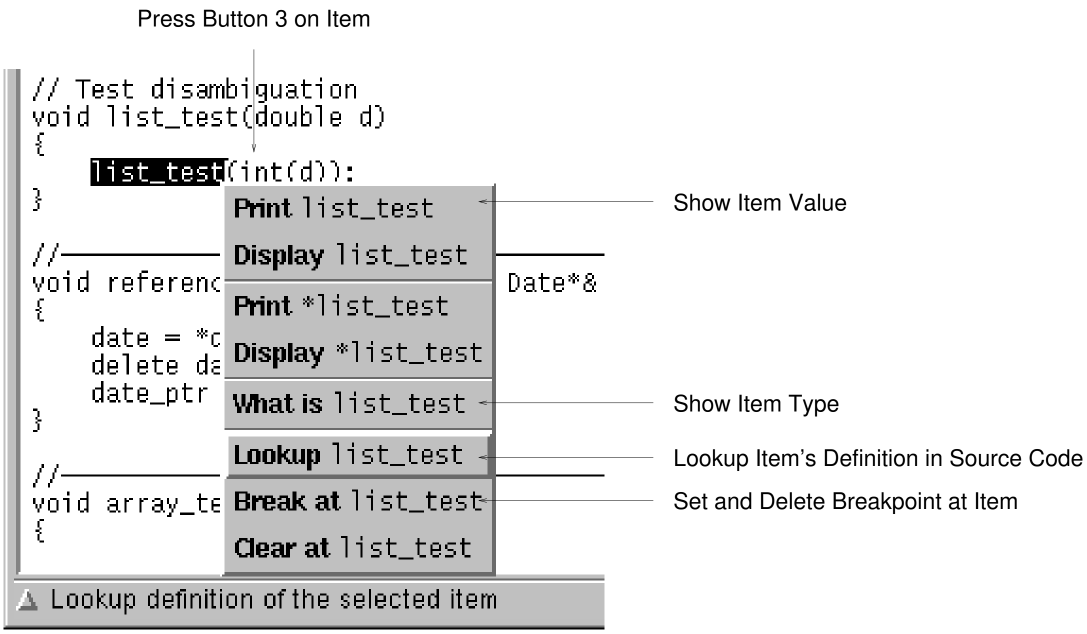
4.3 Looking up Items

As soon as the source of the debugged program is available, the *source window* displays its current source text. (see Section 4.3.4 [Source Path], page 76, if a source text cannot be found.)

In the source window, you can lookup and examine function and variable definitions as well as search for arbitrary occurrences in the source text.

4.3.1 Looking up Definitions

If you wish to lookup a specific function or variable definition whose name is visible in the source text, click with *mouse button 1* on the function or variable name. The name is copied to the argument field. Change the name if desired and click on the '`Lookup`' button to find its definition.



The Source Popup Menu

As a faster alternative, you can simply press *mouse button 3* on the function name and select the '`Lookup`' item from the source popup menu.

As an even faster alternative, you can also double-click on a function call (an identifier followed by a '`(`' character) to lookup the function definition.

If a source file is not found, See Section 4.3.4 [Source Path], page 76, for specifying source directories.

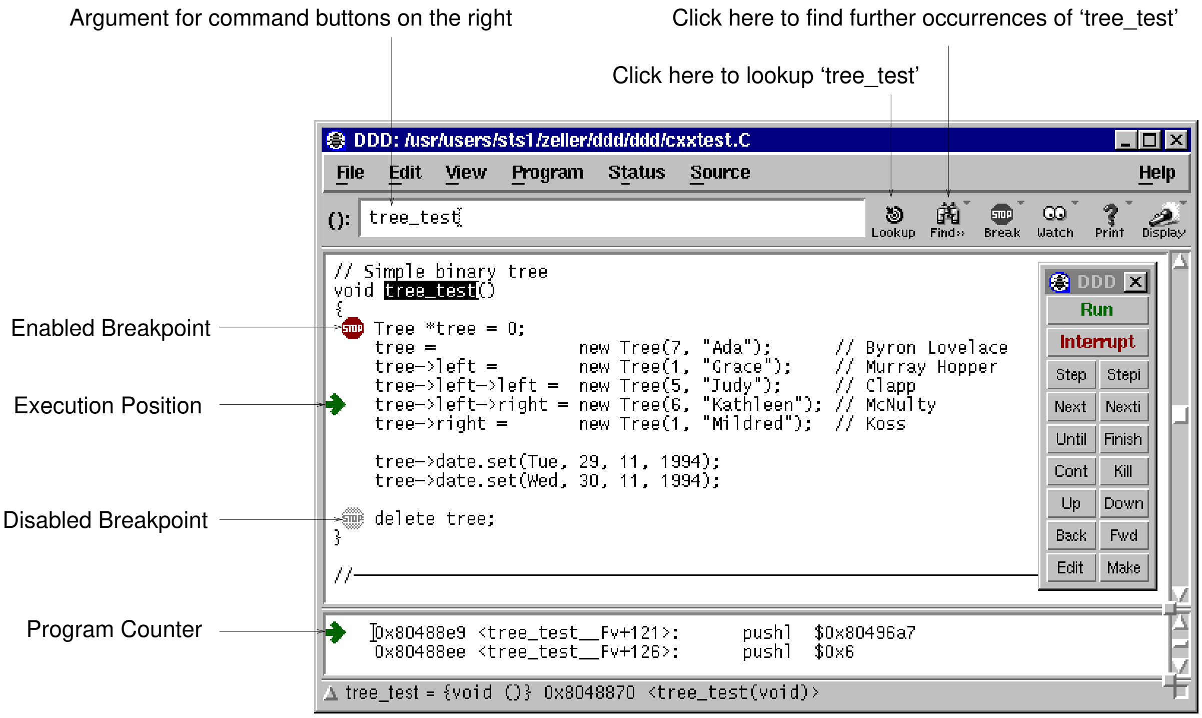
4.3.2 Textual Search

If the item you wish to search is visible in the source text, click with *mouse button 1* on it. The identifier is copied to the argument field. Click on the '`Find >>`' button to find following occurrences and on '`Find >>` ⇒ `Find << ()`' to find previous occurrences.

By default, DDD finds only complete words. To search for arbitrary substrings, change the value of the '`Source` ⇒ `Find Words Only`' option.

4.3.3 Looking up Previous Locations

After looking up a location, use '`Edit` ⇒ `Undo`' (or the '`Undo`' button on the command tool) to go back to the original locations. '`Edit` ⇒ `Redo`' brings you back again to the location you looked for.



The Source Window

4.3.4 Specifying Source Directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, just the names. Even when they do, the directories could be moved between the compilation and your debugging session.

Here's how GDB accesses source files; other inferior debuggers have similar methods.

GDB has a list of directories to search for source files; this is called the *source path*. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name. Note that the executable search path is *not* used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

If GDB cannot find a source file in the source path, and the object program records a directory, GDB tries that directory too. If the source path is empty, and there is no record of the compilation directory, GDB looks in the current directory as a last resort.

To specify a source path for your inferior debugger, use '`Edit ⇒ Debugger Settings`' (see Section 3.6.5 [Debugger Settings], page 71 and search for appropriate entries (in GDB, this is '`Search path for source files`').

If '`Debugger Settings`' has no suitable entry, you can also specify a source path for the inferior debugger when invoking DDD. See Section 2.1.4 [Inferior Debugger Options], page 24, for details.

When using JDB, you can set the `CLASSPATH` environment variable to specify directories where JDB (and DDD) should search for classes.

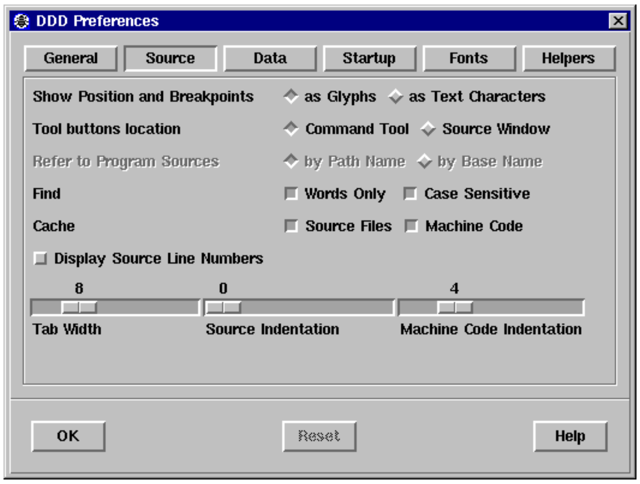
If DDD does not find a source file for any reason, check the following issues:

- In order to debug a program effectively, you need to generate debugging information when you compile it. Without debugging information, the inferior debugger will be unable to locate the source code. To request debugging information, specify the '`-g`' option when you run the compiler. See Section 4.1 [Compiling for Debugging], page 73, for details.
- You may need to tell your inferior debugger where the source code files are. See Section 4.3.4 [Source Path], page 76, for details.

  Using GDB, you can also create a local '`.gdbinit`' file that contains a line `directory` *path*. Here, *path* is a colon-separated list of source paths.

4.4 Customizing the Source Window

The source window can be customized in a number of ways, most of them accessed via '`Edit ⇒ Preferences ⇒ Source`'.



Source Preferences

4.4.1 Customizing Glyphs

In the source text, the current execution position and breakpoints are indicated by symbols (*glyphs*). As an alternative, DDD can also indicate these positions using text characters. If you wish to disable glyphs, set '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Show Position and Breakpoints` ⇒ `as Text Characters`' option. This also makes DDD run slightly faster, especially when scrolling.

This setting is tied to this resource:

displayGlyphs (class DisplayGlyphs) Resource

If this is '`on`', the current execution position and breakpoints are displayed as glyphs; otherwise, they are shown through characters in the text. The default is '`on`'. See Section 2.1.2 [Options], page 16, for the '`--glyphs`' and '`--no-glyphs`' options.

You can further control glyphs using the following resources:

cacheGlyphImages (class CacheMachineCode) Resource

Whether to cache (share) glyph images ('`on`') or not ('`off`'). Caching glyph images requires less X resources, but has been reported to fail with OSF/Motif 2.1 on XFree86 servers. Default is '`off`' for OSF/Motif 2.1 or later on GNU/Linux machines, and '`on`' otherwise.

glyphUpdateDelay (class GlyphUpdateDelay) Resource

A delay (in ms) that says how much time to wait before updating glyphs while scrolling the source text. A small value results in glyphs being scrolled with the text, a large value disables glyphs while scrolling and makes scrolling faster. Default: `10`.

maxGlyphs (class MaxGlyphs) Resource

The maximum number of glyphs to be displayed (default: `10`). Raising this value causes more glyphs to be allocated, possibly wasting resources that are never needed.

4.4.2 Customizing Searching

Searching in the source text (see Section 4.3.2 [Textual Search], page 76) is controlled by these resources, changed via the '`Source`' menu:

findCaseSensitive (class FindCaseSensitive) Resource
: If this is '`on`' (default), the '`Find`' commands are case-sensitive. Otherwise, occurrences are found regardless of case.

findWordsOnly (class FindWordsOnly) Resource
: If this is '`on`' (default), the '`Find`' commands find complete words only. Otherwise, arbitrary occurrences are found.

4.4.3 Customizing Source Appearance

You can have DDD show line numbers within the source window. Use '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Display Source Line Numbers`'.

displayLineNumbers (class DisplayLineNumbers) Resource
: If this is '`on`', lines in the source text are prefixed with their respective line number. The default is '`off`'.

You can instruct DDD to indent the source code, leaving more room for breakpoints and execution glyphs. This is done using the '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Source indentation`' slider. The default value is `0` for no indentation at all.

indentSource (class Indent) Resource
: The number of columns to indent the source code, such that there is enough place to display breakpoint locations. Default: `0`.

By default, DDD uses a minimum indentation for script languages.

indentScript (class Indent) Resource
: The minimum indentation for script languages, such as Perl, Python, and Bash. Default: `4`.

The maximum width of line numbers is controlled by this resource.

lineNumberWidth (class LineNumberWidth) Resource
: The number of columns to use for line numbers (if displaying line numbers is enabled). Line numbers wider than this value extend into the breakpoint space. Default: `4`.

If your source code uses a tab width different from `8` (the default), you can set an alternate width using the '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Tab width`' slider.

tabWidth (class TabWidth) Resource
: The tab width used in the source window (default: `8`)

4.4.4 Customizing Source Scrolling

These resources control when the source window is scrolled:

linesAboveCursor (class LinesAboveCursor) Resource
: The minimum number of lines to show before the current location. Default is `2`.

linesBelowCursor (class LinesBelowCursor) Resource
: The minimum number of lines to show after the current location. Default is `3`.

4.4.5 Customizing Source Lookup

Some DBX and XDB variants do not properly handle paths in source file specifications. If you want the inferior debugger to refer to source locations by source base names only, unset the '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Refer to Program Sources by full path name`' option.

This is related to the following resource:

useSourcePath (class UseSourcePath) Resource
: If this is '`off`' (default), the inferior debugger refers to source code locations only by their base names. If this is '`on`' (default), DDD uses the full source code paths.

By default, DDD caches source files in memory. This is convenient for remote debugging, since remote file access may be slow. If you want to reduce memory usage, unset the '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Cache source files`' option.

This is related to the following resource:

cacheSourceFiles (class CacheSourceFiles) Resource
: Whether to cache source files ('`on`', default) or not ('`off`'). Caching source files requires more memory, but makes DDD run faster.

4.4.6 Customizing File Filtering

You can control whether DDD should filter files to be opened.

filterFiles (class FilterFiles) Resource
: If this is '`on`' (default), DDD filters files when opening execution files, core dumps, or source files, such that the selection shows only suitable files. This requires that DDD opens each file, which may take time. If this is '`off`', DDD always presents all available files.

5 Stopping the Program

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and find out why.

Inside DDD, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a DDD command such as '`Step`'. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution.

The inferior debuggers supported by DDD support two mechanisms for stopping a program upon specific events:

- A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. Typically, breakpoints are set before running the program.
- A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes.

5.1 Breakpoints

5.1.1 Setting Breakpoints

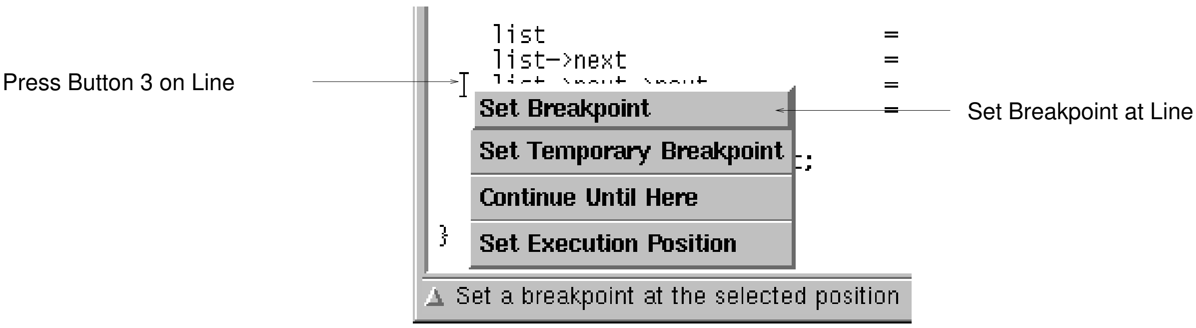
You can set breakpoints by location or by name.

5.1.1.1 Setting Breakpoints by Location

Breakpoints are set at a specific location in the program.

If the source line is visible, click with *mouse button 1* on the left of the source line and then on the '`Break`' button.

As a faster alternative, you can simply press *mouse button 3* on the left of the source line and select the '`Set Breakpoint`' item from the line popup menu.



The Line Popup Menu

As an even faster alternative, you can simply double-click on the left of the source line to set a breakpoint.

As yet another alternative, you can select '`Source` ⇒ `Breakpoints`'. Click on the '`Break`' button and enter the location.

(If you find this number of alternatives confusing, be aware that DDD users fall into three categories, which must all be supported. *Novice users* explore DDD and may prefer to use one single mouse button. *Advanced users* know how to use shortcuts and prefer popup menus. *Experienced users* prefer the command line interface.)

Breakpoints are indicated by a plain stop sign, or as '`#`*n*', where *n* is the breakpoint number. A greyed out stop sign (or '_*n*_') indicates a disabled breakpoint. A stop sign with a question mark (or '`?`*n*`?`') indicates a conditional breakpoint or a breakpoint with an ignore count set.

If you set a breakpoint by mistake, use '`Edit` ⇒ `Undo`' to delete it again.

5.1.1.2 Setting Breakpoints by Name

If the function name is visible, click with *mouse button 1* on the function name. The function name is then copied to the argument field. Click on the '`Break`' button to set a breakpoint there.

As a shorter alternative, you can simply press *mouse button 3* on the function name and select the '`Break at`' item from the popup menu.

As yet another alternative, you can click on '`Break...`' from the Breakpoint editor (invoked through '`Source` ⇒ `Breakpoints`') and enter the function name.

5.1.1.3 Setting Regexp Breakpoints

Using GDB, you can also set a breakpoint on all functions that match a given string. '`Break` ⇒ `Set Breakpoints at Regexp ()`' sets a breakpoint on all functions whose name matches the *regular expression* given in '`()`'. Here are some examples:

- To set a breakpoint on every function that starts with '`Xm`', set '`()`' to '`^Xm`'.
- To set a breakpoint on every member of class '`Date`', set '`()`' to '`^Date::`'.
- To set a breakpoint on every function whose name contains '`_fun`', set '`()`' to '`_fun`'.
- To set a breakpoint on every function that ends in '`_test`', set '`()`' to '`_test$`'.

5.1.2 Deleting Breakpoints

To delete a visible breakpoint, click with *mouse button 1* on the breakpoint. The breakpoint location is copied to the argument field. Click on the '`Clear`' button to delete all breakpoints there.

If the function name is visible, click with *mouse button 1* on the function name. The function name is copied to the argument field. Click on the '`Clear`' button to clear all breakpoints there.

As a faster alternative, you can simply press *mouse button 3* on the breakpoint and select the '`Delete Breakpoint`' item from the popup menu.

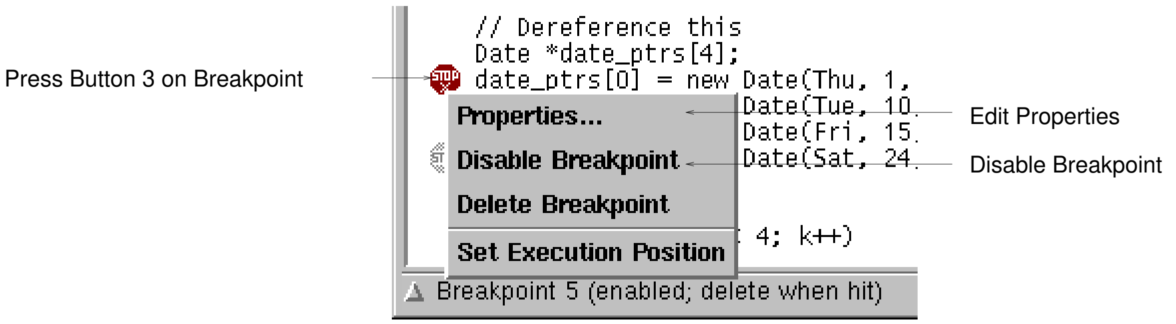
As yet another alternative, you can select the breakpoint and click on '`Delete`' in the Breakpoint editor (invoked through '`Source` ⇒ `Breakpoints`').

As an even faster alternative, you can simply double-click on the breakpoint while holding ⟨Ctrl⟩.

5.1.3 Disabling Breakpoints

Rather than deleting a breakpoint or watchpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can *enable* it again later.[1]

To disable a breakpoint, press *mouse button 3* on the breakpoint symbol and select the '`Disable Breakpoint`' item from the breakpoint popup menu. To enable it again, select '`Enable Breakpoint`'.



The Breakpoint Popup Menu

As an alternative, you can select the breakpoint and click on '`Disable`' or '`Enable`' in the Breakpoint editor (invoked through '`Source ⇒ Breakpoints`'.

Disabled breakpoints are indicated by a grey stop sign, or '_*n*_', where *n* is the breakpoint number.

The '`Disable Breakpoint`' item is also accessible via the '`Clear`' button. Just press and hold *mouse button 1* on the button to get a popup menu.

5.1.4 Temporary Breakpoints

A *temporary breakpoint* is immediately deleted as soon as it is reached.[2]

To set a temporary breakpoint, press *mouse button 3* on the left of the source line and select the '`Set Temporary Breakpoint`' item from the popup menu.

As a faster alternative, you can simply double-click on the left of the source line while holding ⟨Ctrl⟩.

Temporary breakpoints are convenient to make the program continue up to a specific location: just set the temporary breakpoint at this location and continue execution.

The '`Continue Until Here`' item from the popup menu sets a temporary breakpoint on the left of the source line and immediately continues execution. Execution stops when the temporary breakpoint is reached.

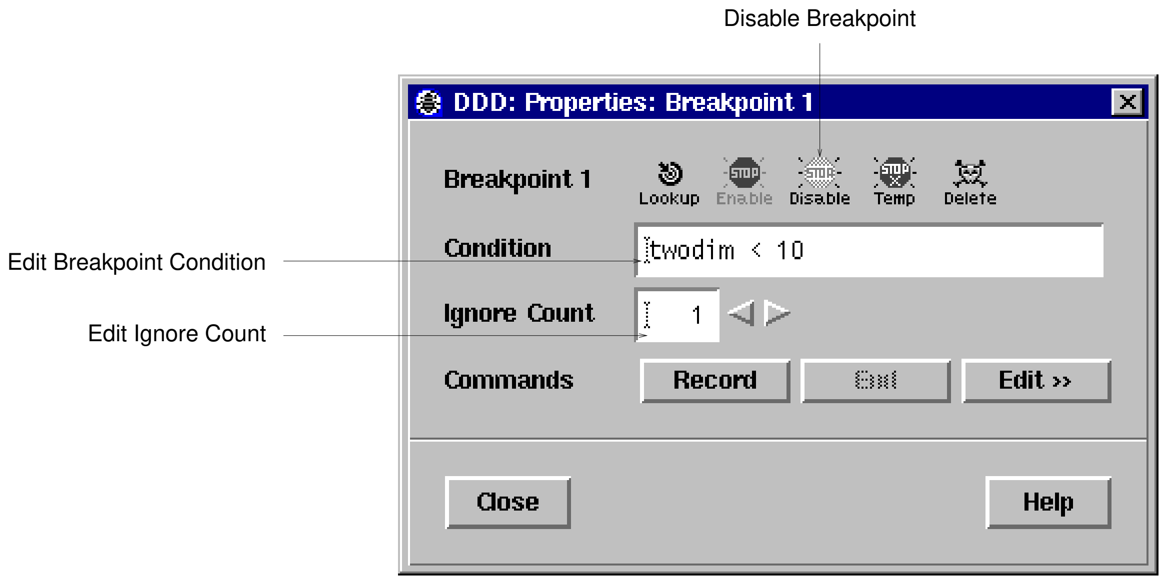
[1] JDB does not support breakpoint disabling.

[2] JDB does not support temporary breakpoints.

The 'Set Temporary Breakpoint' and 'Continue Until Here' items are also accessible via the 'Break' button. Just press and hold *mouse button 1* on the button to get a popup menu.

5.1.5 Editing Breakpoint Properties

You can change all properties of a breakpoint by pressing *mouse button 3* on the breakpoint symbol and select 'Properties' from the breakpoint popup menu. This will pop up a dialog showing the current properties of the selected breakpoint.



Breakpoint Properties

As an even faster alternative, you can simply double-click on the breakpoint.

- Click on 'Lookup' to move the cursor to the breakpoint's location.
- Click on 'Enable' to enable the breakpoint.
- Click on 'Disable' to disable the breakpoint.
- Click on 'Temp' to make the breakpoint temporary.[3]
- Click on 'Delete' to delete the breakpoint.

5.1.6 Breakpoint Conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your

[3] GDB has no way to make a temporary breakpoint non-temporary again.

programming language. A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated–that is, when the condition is false. In C, if you want to test an assertion expressed by the condition *assertion*, you should set the condition '`!`*assertion*' on the appropriate breakpoint.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, DDD might see the other breakpoint first and stop your program without checking the condition of this one.)

Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached. See Section 5.1.8 [Breakpoint Commands], page 85, for details.

5.1.7 Breakpoint Ignore Counts

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. This is so useful that there is a special way to do it, using the *ignore count* of the breakpoint. Every breakpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if your program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is *n*, the breakpoint does not stop the next *n* times your program reaches it.

In the field '`Ignore Count`' of the '`Breakpoint Properties`' panel, you can specify the breakpoint ignore count.[4]

If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, DDD resumes checking the condition.

5.1.8 Breakpoint Commands

You can give any breakpoint (or watchpoint) a series of DDD commands to execute when your program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.[5]

Using the '`Commands`' buttons of the '`Breakpoint Properties`' panel, you can edit commands to be executed when the breakpoint is hit.

To edit breakpoint commands, click on '`Edit >>`' and enter the commands in the commands editor. When done with editing, click on '`Edit <<`' to close the commands editor.

Using GDB, you can also *record* a command sequence to be executed. To record a command sequence, follow these steps:

1. Click on '`Record`' to begin the recording of the breakpoint commands.

---

[4] JDB, Perl and some DBX variants do not support breakpoint ignore counts.

[5] JDB, PYDB, and some DBX variants do not support breakpoint commands.

2. Now interact with DDD. While recording, DDD does not execute commands, but simply records them to be executed when the breakpoint is hit. The recorded debugger commands are shown in the debugger console.

3. To stop the recording, click on '`End`' or enter '`end`' at the GDB prompt. To *cancel* the recording, click on '`Interrupt`' or press ⟨ESC⟩.

4. You can edit the breakpoint commands just recorded using '`Edit >>`'.

5.1.9 Moving and Copying Breakpoints

To move a breakpoint to a different location, press *mouse button 1* on the stop sign and drag it to the desired location.[6] This is equivalent to deleting the breakpoint at the old location and setting a breakpoint at the new location. The new breakpoint inherits all properties of the old breakpoint, except the breakpoint number.

To copy a breakpoint to a new location, press ⟨Shift⟩ while dragging.

5.1.10 Looking up Breakpoints

If you wish to lookup a specific breakpoint, select '`Source ⇒ Breakpoints ⇒ Lookup`'. After selecting a breakpoint from the list and clicking the '`Lookup`' button, the breakpoint location is displayed.

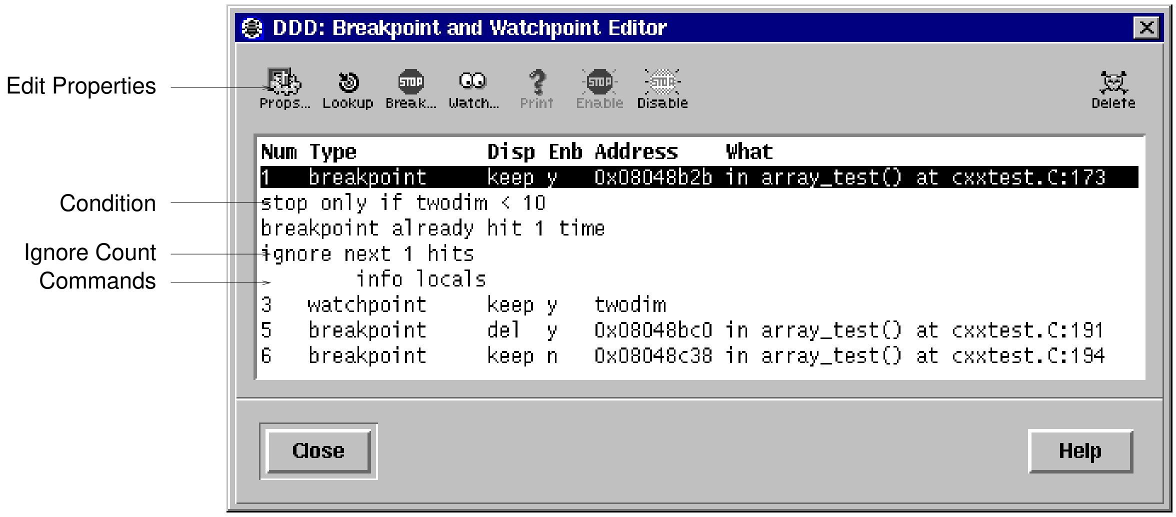
As an alternative, you can enter '`#`*n*' in the argument field, where *n* is the breakpoint number, and click on the '`Lookup`' button to find its definition.

5.1.11 Editing all Breakpoints

To view and edit all breakpoints at once, select '`Source ⇒ Breakpoints`'. This will popup the *Breakpoint Editor* which displays the state of all breakpoints.

---

[6] When glyphs are disabled (see Section 4.4 [Customizing Source], page 77), breakpoints cannot be dragged. Delete and set breakpoints instead.



The Breakpoint Editor

In the breakpoint editor, you can select individual breakpoints by clicking on them. Pressing ⟨Ctrl⟩ while clicking toggles the selection. To edit the properties of all selected breakpoints, click on '`Props`'.

5.1.12 Hardware-Assisted Breakpoints

Using GDB, a few more commands related to breakpoints can be invoked through the debugger console:

`hbreak` *position*
: Sets a hardware-assisted breakpoint at *position*. This command requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction.

`thbreak` *pos*
: Set a temporary hardware-assisted breakpoint at *pos*.

See section "Setting Breakpoints" in *Debugging with* GDB, for details.

5.2 Watchpoints

You can make the program stop as soon as some variable value changes, or when some variable is read or written. This is called *setting a watchpoint on a variable*.[7]

Watchpoints have much in common with breakpoints: in particular, you can enable and disable them. You can also set conditions, ignore counts, and commands to be executed when a watched variable changes its value.

[7] Watchpoints are available in GDB and some DBX variants only. In XDB, a similar feature is available via XDB *assertions*; see the XDB documentation for details.

Please note: on architectures without special watchpoint support, watchpoints currently make the program execute two orders of magnitude more slowly. This is so because the inferior debugger must interrupt the program after each machine instruction in order to examine whether the watched value has changed. However, this delay can be well worth it to catch errors when you have no clue what part of your program is the culprit.

5.2.1 Setting Watchpoints

If the variable name is visible, click with *mouse button 1* on the variable name. The variable name is copied to the argument field. Otherwise, enter the variable name in the argument field. Click on the '`Watch`' button to set a watchpoint there.

Using GDB and JDB 1.2, you can set different types of watchpoints. Click and hold *mouse button 1* on the '`Watch`' button to get a menu.

5.2.2 Editing Watchpoint Properties

To change the properties of a watchpoint, enter the name of the watched variable in the argument field. Click and hold *mouse button 1* on the '`Watch`' button and select '`Watchpoint Properties`'.

The Watchpoint Properties panel has the same functionality as the Breakpoint Properties panel (see Section 5.1.5 [Editing Breakpoint Properties], page 84). As an additional feature, you can click on '`Print`' to see the current value of a watched variable.

5.2.3 Editing all Watchpoints

To view and edit all watchpoints at once, select '`Data` ⇒ `Watchpoints`'. This will popup the *Watchpoint Editor* which displays the state of all watchpoints.

The Watchpoint Editor has the same functionality as the Breakpoint Editor (see Section 5.1.11 [Editing all Breakpoints], page 86). As an additional feature, you can click on '`Print`' to see the current value of a watched variable.

5.2.4 Deleting Watchpoints

To delete a watchpoint, enter the name of the watched variable in the argument field and click the '`Unwatch`' button.

5.3 Interrupting

If the program is already running (see Chapter 6 [Running], page 91), you can interrupt it any time by clicking the '`Interrupt`' button or typing ⟨ESC⟩ in a DDD window.[8] Using GDB, this is equivalent to sending a `SIGINT` (Interrupt) signal.

'`Interrupt`' and ⟨ESC⟩ also interrupt a running debugger command, such as printing data.

[8] If ⟨Ctrl+C⟩ is not bound to '`Copy`' (see Section 3.1.11.2 [Customizing the Edit Menu], page 51), you can also use ⟨Ctrl+C⟩ to interrupt the running program.

5.4 Stopping X Programs

If your program is a modal X application, DDD may interrupt it while it has grabbed the mouse pointer, making further interaction impossible—your X display will be unresponsive to any user actions.

By default, DDD will check after each interaction whether the pointer is grabbed. If the pointer is grabbed, DDD will continue the debugged program such that you can continue to use your X display.

This is how this feature works: When the program stops, DDD checks for input events such as keyboard or mouse interaction. If DDD does not receive any event within the next 5 seconds, DDD checks whether the mouse pointer is grabbed by attempting to grab and ungrab it. If this attempt fails, then DDD considers the pointer grabbed.

Unfortunately, DDD cannot determine the program that grabbed the pointer—it may be the debugged program, or another program. Consequently, you have another 10 seconds to cancel continuation before DDD continues the program automatically.

There is one situation where this fails: if you lock your X display while DDD is running, then DDD will consider a resulting pointer grab as a result of running the program—and automatically continue execution of the debugged program. Consequently, you can turn off this feature via '`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Continue Automatically when Mouse Pointer is Frozen`'.

5.4.1 Customizing Grab Checking

The grab checks are controlled by the following resources:

checkGrabs (class CheckGrabs) Resource

If this is '`on`' (default), DDD will check after each interaction whether the pointer is grabbed. If this is so, DDD will automatically continue execution of debugged program.

checkGrabDelay (class CheckGrabDelay) Resource

The time to wait (in ms) after a debugger command before checking for a grabbed pointer. If DDD sees some pointer event within this delay, the pointer cannot be grabbed and an explicit check for a grabbed pointer is unnecessary. Default is `5000`, or 5 seconds.

grabAction (class grabAction) Resource

The action to take after having detected a grabbed mouse pointer. This is a list of newline-separated commands. Default is `cont`, meaning to continue the debuggee. Other possible choices include `kill` (killing the debuggee) or `quit` (exiting DDD).

grabActionDelay (class grabActionDelay) Resource

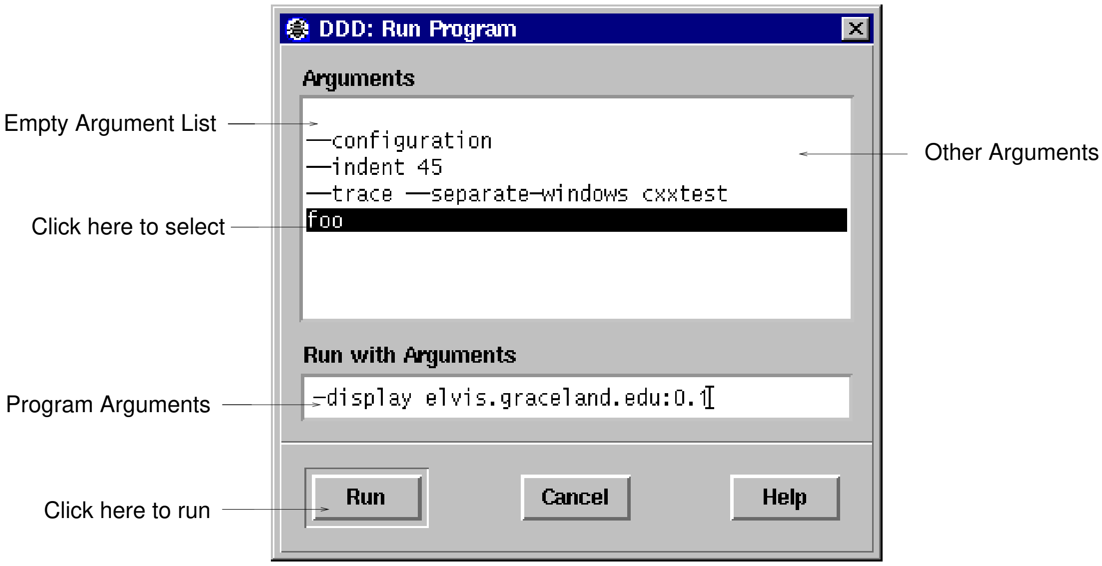
The time to wait (in ms) before taking an action due to having detected a grabbed pointer. During this delay, a working dialog pops up telling the user about imminent execution of the grab action (see the '`grabAction`' resource, above). If the pointer grab is released within this delay, the working dialog pops down and no action is taken. This is done to exclude pointer grabs from sources other than the debugged program (including DDD). Default is `10000`, or 10 seconds.

6 Running the Program

You may start the debugged program with its arguments, if any, in an environment of your choice. You may redirect your program's input and output, debug an already running process, or kill a child process.

6.1 Starting Program Execution

To start execution of the debugged program, select '`Program` ⇒ `Run`'. You will then be prompted for the arguments to pass to your program. You can either select from a list of previously used arguments or enter own arguments in the text field. Afterwards, press the '`Run`' button to start execution with the selected arguments.



Starting a Program with Arguments

To run your program again, with the same arguments, select '`Program` ⇒ `Run Again`' or press the '`Run`' button on the command tool. You may also enter `run`, followed by arguments at the debugger prompt instead.

When you click on '`Run`', your program begins to execute immediately. See Chapter 5 [Stopping], page 81, for a discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program to examine data. See Chapter 7 [Examining Data], page 105, for details.

If the modification time of your symbol file has changed since the last time GDB read its symbols, GDB discards its symbol table, and reads it again. When it does this, GDB and DDD try to retain your current debugger state, such as breakpoints.

6.1.1 Your Program's Arguments

The arguments to your program are specified by the arguments of the 'run' command, as composed in 'Program ⇒ Run'.

In GDB, the arguments are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to your program. Your SHELL environment variable (if it exists) specifies what shell GDB uses. If you do not define SHELL, GDB uses '/bin/sh'.

If you use another inferior debugger, the exact semantics on how the arguments are interpreted depend on the inferior debugger you are using. Normally, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments.

6.1.2 Your Program's Environment

Your program normally inherits its environment from the inferior debugger, which again inherits it from DDD, which again inherits it from its parent process (typically the shell or desktop).

In GDB, you can use the commands `set environment` and `unset environment` to change parts of the environment that affect your program. See section "Your Program's Environment" in *Debugging with* GDB, for details.

The following environment variables are set by DDD:

`DDD` Set to a string indicating the DDD version. By testing whether DDD is set, a debuggee (or inferior debugger) can determine whether it was invoked by DDD.

`TERM` Set to 'dumb', the DDD terminal type. This is set for the inferior debugger only.[1]

`TERMCAP` Set to '' (none), the DDD terminal capabilities.

`PAGER` Set to 'cat', the preferred DDD pager.

The inferior debugger, in turn, might also set or unset some environment variables.

6.1.3 Your Program's Working Directory

Your program normally inherits its working directory from the inferior debugger, which again inherits it from DDD, which again inherits it from its parent process (typically the shell or desktop).

You can change the working directory of the inferior debugger via 'File ⇒ Change Directory' or via the 'cd' command of the inferior debugger.

6.1.4 Your Program's Input and Output

By default, the program you run under DDD does input and output to the debugger console. Normally, you can redirect your program's input and/or output using *shell redirections* with the arguments—that is, additional arguments like '< *input*' or '> *output*'. You can enter these shell redirections just like other arguments (see Section 6.1.1 [Arguments], page 92).

[1] If the debuggee runs in a separate execution window, the debuggee's TERM value is set according to the 'termType' resource; See Section 6.2.1 [Customizing the Execution Window], page 94, for details.

Warning: While input and output redirection work, you cannot use pipes to pass the output of the program you are debugging to another program; if you attempt this, DDD may wind up debugging the wrong program. See Section 6.3 [Attaching to a Process], page 94, for an alternative.

If command output is sent to the debugger console, it is impossible for DDD to distinguish between the output of the debugged program and the output of the inferior debugger.

Program output that confuses DDD includes:

- Primary debugger prompts (e.g. '`(gdb) `', '`(dbx) `' or '`(ladebug) `')
- Secondary debugger prompts (e.g. '`>`')
- Confirmation prompts (e.g. '`(y or n) `')
- Prompts for more output (e.g. '`Press RETURN to continue`')
- Display output (e.g. '`$pc = 0x1234`')

If your program outputs any of these strings, you may encounter problems with DDD mistaking them for debugger output. These problems can easily be avoided by redirecting program I/O, for instance to the separate execution window (see Section 6.2 [Using the Execution Window], page 93).

If the inferior debugger changes the default TTY settings, for instance through a `stty` command in its initialization file, DDD may also become confused. The same applies to debugged programs which change the default TTY settings.

The behavior of the debugger console can be controlled using the following resource:

lineBufferedConsole (class LineBuffered) Resource

If this is '`on`' (default), each line from the inferior debugger is output on each own, such that the final line is placed at the bottom of the debugger console. If this is '`off`', all lines are output as a whole. This is faster, but results in a random position of the last line.

6.2 Using the Execution Window

By default, input and output of your program go to the debugger console. As an alternative, DDD can also invoke an *execution window*, where the program terminal input and output is shown.[2]

To activate the execution window, select '`Program` ⇒ `Run in Execution Window`'.

Using the execution window has an important side effect: The output of your program no longer gets intermixed with the output of the inferior debugger. This makes it far easier for DDD to parse the debugger output correctly. See Section 2.5.3 [Debugger Communication], page 38, for details on the '`bufferGDBOutput`' resource.

The execution window is opened automatically as soon as you start the debugged program. While the execution window is active, DDD redirects the standard input, output, and error streams of your program to the execution window. Note that the device '`/dev/tty`' still refers to the debugger console, *not* the execution window.

You can override the DDD stream redirection by giving alternate redirection operations as arguments. For instance, to have your program read from *file*, but to write to the execution window, invoke your program with '`<` *file*' as argument. Likewise, to redirect the standard error output to the debugger console, use '`2> /dev/tty`' (assuming the inferior debugger and/or your UNIX shell support standard error redirection).

[2] The execution window is not available in JDB.

6.2.1 Customizing the Execution Window

You can customize the DDD execution window and use a different TTY command. The command is set by '`Edit` ⇒ `Preferences` ⇒ `Helpers` ⇒ `Execution Window`':

termCommand (class TermCommand) Resource

The command to invoke for the execution window—a TTY emulator that shows the input/output of the debugged program. A Bourne shell command to run in the separate TTY is appended to this string. The string '`@FONT@`' is replaced by the name of the fixed width font used by DDD. A simple value is

```
Ddd*termCommand: xterm -fn @FONT@ -e /bin/sh -c
```

You can also set the terminal type:

termType (class TermType) Resource

The terminal type provided by the '`termCommand`' resource—that is, the value of the `TERM` environment variable to be passed to the debugged program. Default: '`xterm`'.

Whether the execution window is active or not, as set by '`Program` ⇒ `Run in Execution Window`', is saved using this resource:

separateExecWindow (class Separate) Resource

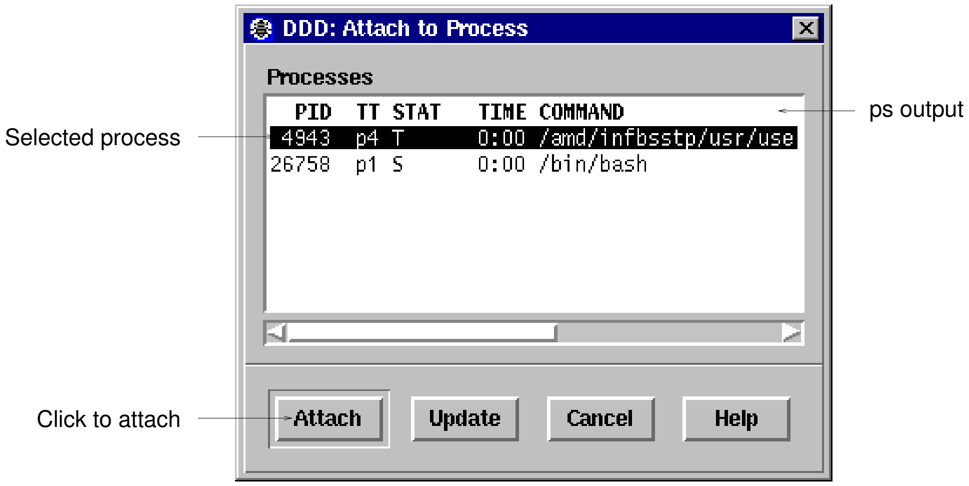
If '`on`', the debugged program is executed in a separate execution window. If '`off`' (default), the debugged program is executed in the console window.

6.3 Attaching to a Running Process

If the debugged program is already running in some process, you can *attach* to this process (instead of starting a new one with '`Run`').[3]

To attach DDD to a process, select '`File` ⇒ `Attach to Process`'. You can now choose from a list of processes. Then, press the '`Attach`' button to attach to the specified process.

[3] JDB, PYDB, Perl, and Bash do not support attaching the debugger to running processes.



Selecting a Process to Attach

The first thing DDD does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the DDD commands that are ordinarily available when you start processes with 'Run'. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, you may use 'Continue' after attaching DDD to the process.

When using 'Attach to Process', you should first use 'Open Program' to specify the program running in the process and load its symbol table.

When you have finished debugging the attached process, you can use the 'File ⇒ Detach Process' to release it from DDD control. Detaching the process continues its execution. After 'Detach Process', that process and DDD become completely independent once more, and you are ready to attach another process or start one with 'Run'.

You can customize the list of processes shown by defining an alternate command to list processes. See 'Edit ⇒ Preferences ⇒ Helpers ⇒ List Processes'; See Section 6.3.1 [Customizing Attaching to Processes], page 95, for details.

6.3.1 Customizing Attaching to Processes

When attaching to a process (see Section 6.3 [Attaching to a Process], page 94), DDD uses a `ps` command to get the list of processes. This command is defined by the 'psCommand' resource.

psCommand (class PsCommand) Resource

The command to get a list of processes. Usually `ps`. Depending on your system, useful alternate values include `ps -ef` and `ps ux`. The first line of the output must either contain a 'PID' title, or each line must begin with a process ID.

Note that the output of this command is filtered by DDD; a process is only shown if it can be attached to. The DDD process itself as well as the process of the inferior debugger are suppressed, too.

6.4 Program Stops

After the program has been started, it runs until one of the following happens:

- A breakpoint is reached (see Section 5.1 [Breakpoints], page 81).
- A watched value changes (see Section 5.2 [Watchpoints], page 87).
- The program is interrupted (see Section 5.3 [Interrupting], page 88).
- A signal is received (see Section 6.10 [Signals], page 102).
- Execution completes.

DDD shows the current program status in the debugger console. The current execution position is highlighted by an arrow.

If '`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Uniconify When Ready`' is set, DDD automatically deiconifies itself when the program stops. This way, you can iconify DDD during a lengthy computation and have it uniconify as soon as the program stops.

6.5 Resuming Execution

6.5.1 Continuing

To resume execution, at the current execution position, click on the '`Continue`' button. Any breakpoints set at the current execution position are bypassed.

6.5.2 Stepping one Line

To execute just one source line, click on the '`Step`' button. The program is executed until control reaches a different source line, which may be in a different function. Then, the program is stopped and control returns to DDD.

Warning: If you use the '`Step`' button while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, it will not step into a function which is compiled without debugging information. To step through functions without debugging information, use the '`Stepi`' button (see Section 8.2 [Machine Code Execution], page 142).

In GDB, the '`Step`' button only stops at the first instruction of a source line. This prevents the multiple stops that used to occur in switch statements, for loops, etc. '`Step`' continues to stop if a function that has debugging information is called within the line.

Also, the '`Step`' in GDB only enters a subroutine if there is line number information for the subroutine. Otherwise it acts like the '`Next`' button.

6.5.3 Continuing to the Next Line

To continue to the next line in the current function, click on the '`Next`' button. This is similar to '`Step`', but any function calls appearing within the line of code are executed without stopping.

Execution stops when control reaches a different line of code at the original stack level that was executing when you clicked on '`Next`'.

6.5.4 Continuing Until Here

To continue running until a specific location is reached, use the '`Continue Until Here`' facility from the line popup menu. See Section 5.1.4 [Temporary Breakpoints], page 83, for a discussion.

6.5.5 Continuing Until a Greater Line is Reached

To continue until a greater line in the current function is reached, click on the '`Until`' button. This is useful to avoid single stepping through a loop more than once.

'`Until`' is like '`Next`', except that when '`Until`' encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.

This means that when you reach the end of a loop after single stepping though it, '`until`' makes your program continue execution until it exits the loop. In contrast, clicking on '`Next`' at the end of a loop simply steps back to the beginning of the loop, which forces you to step through the next iteration.

'`Until`' always stops your program if it attempts to exit the current stack frame.

'`Until`' works by means of single instruction stepping, and hence is slower than continuing until a breakpoint is reached.

6.5.6 Continuing Until Function Returns

To continue running until the current function returns, use the '`Finish`' button. The returned value (if any) is printed.

6.6 Continuing at a Different Address

Ordinarily, when you continue your program, you do so at the place where it stopped. You can instead continue at an address of your own choosing.

The most common occasion to use this feature is to back up—perhaps with more breakpoints set-over a portion of a program that has already executed, in order to examine its execution in more detail.

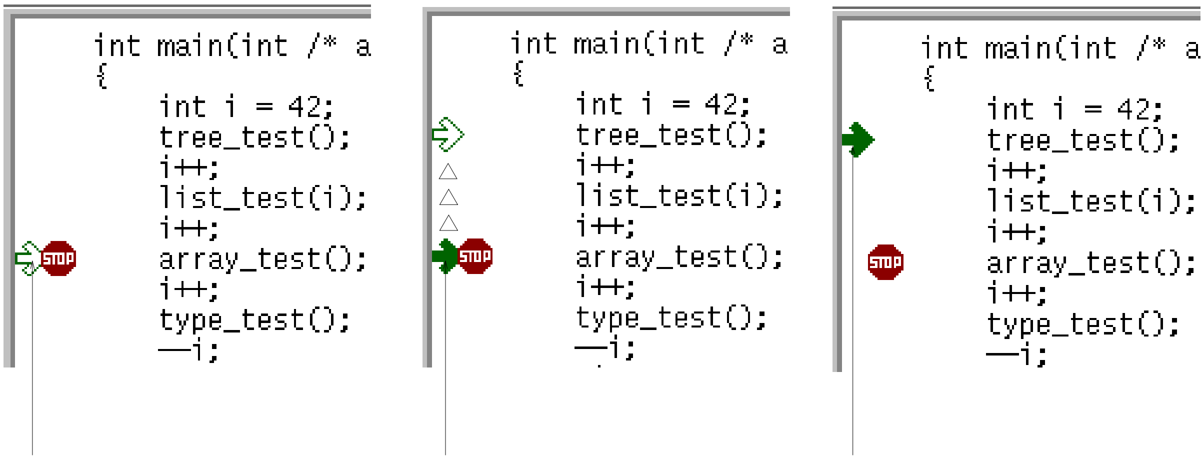
To set the execution position to the current location, use '`Set Execution Position`' from the breakpoint popup menu. This item is also accessible by pressing and holding the '`Break/Clear`' button.[4]

As a quicker alternative, you can also press *mouse button 1* on the arrow and drag it to a different location.[5]

---

[4] JDB, PYDB, Perl, and Bash do not support altering the execution position.

[5] When glyphs are disabled (see Section 4.4 [Customizing Source], page 77), dragging the execution position is not possible. Set the execution position explicitly instead.



Click on arrow, ... hold mouse button and move ... ... to the final position.

Changing the Execution Position by Dragging the Execution Arrow

Moving the execution position does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter.

Some inferior debuggers (notably GDB) allow you to set the new execution position into a different function from the one currently executing. This may lead to bizarre results if the two functions expect different patterns of arguments or of local variables. For this reason, moving the execution position requests confirmation if the specified line is not in the function currently executing.

After moving the execution position, click on '`Continue`' to resume execution.

6.7 Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*.

When your program stops, the DDD commands for examining the stack allow you to see all of this information.

One of the stack frames is *selected* by DDD and many DDD commands refer implicitly to the selected frame. In particular, whenever you ask DDD for the value of a variable in your program, the value is found in the selected frame. There are special DDD commands to select whichever frame you are interested in.

6.7.1 Stack Frames

The call stack is divided up into contiguous pieces called *stack frames*, or *frames* for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function `main`. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

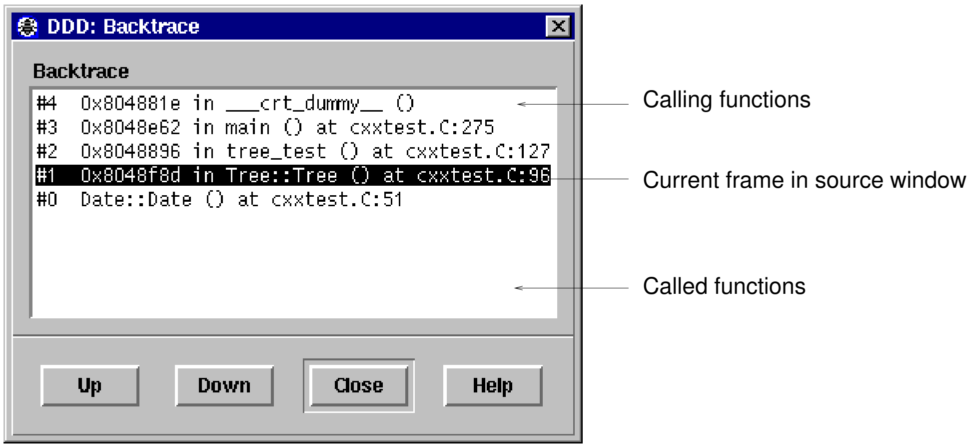
Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one byte whose address serves as the address of the frame. Usually this address is kept in a register called the *frame pointer register* while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by GDB to give you a way of designating stack frames in GDB commands.

6.7.2 Backtraces

DDD provides a *backtrace window* showing a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

To enable the backtrace window, select '`Status` ⇒ `Backtrace`'.



Selecting a Frame from the Backtrace Viewer

Using GDB, each line in the backtrace shows the frame number and the function name. The program counter value is also shown—unless you use the GDB command '`set print address off`'. The backtrace also shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number.

6.7.3 Selecting a Frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame.[6]

In the backtrace window, you can *select* an arbitrary frame to move from one stack frame to another. Just click on the desired frame.

The '`Up`' button selects the function that called the current one—that is, it moves one frame up.

The '`Down`' button selects the function that was called by the current one—that is, it moves one frame down.

You can also directly type the `up` and `down` commands at the debugger prompt. Typing ⟨Ctrl+Up⟩ and ⟨Ctrl+Down⟩, respectively, will also move you through the stack.

'`Up`' and '`Down`' actions can be undone via '`Edit` ⇒ `Undo`'.

6.8 "Undoing" Program Execution

If you take a look at the '`Edit` ⇒ `Undo`' menu item after an execution command, you'll find that DDD offers you to undo execution commands just as other commands. Does this mean that DDD allows you to go backwards in time, undoing program execution as well as undoing any side-effects of your program?

Sorry—we must disappoint you. DDD cannot undo what your program did. (After a little bit of thought, you'll find that this would be impossible in general.) However, DDD can do something different: it can show *previously recorded states* of your program.

After "undoing" an execution command (via '`Edit` ⇒ `Undo`', or the '`Undo`' button), the execution position moves back to the earlier position and displayed variables take their earlier values. Your program state is in fact unchanged, but DDD gives you a *view* on the earlier state as recorded by DDD.

In this so-called *historic mode*, most normal DDD commands that would query further information from the program are disabled, since the debugger cannot be queried for the earlier state. However, you can examine the current execution position, or the displayed variables. Using '`Undo`' and '`Redo`', you can move back and forward in time to examine how your program got into the present state.

To let you know that you are operating in historic mode, the execution arrow gets a dashed-line appearance (indicating a past position); variable displays also come with dashed lines. Furthermore, the status line informs you that you are seeing an earlier program state.

Here's how historic mode works: each time your program stops, DDD collects the current execution position and the values of displayed variables. Backtrace, thread, and register information is also collected if the corresponding dialogs are open. When "undoing" an execution command, DDD updates its view from this collected state instead of querying the program.

If you want to collect this information without interrupting your program—within a loop, for instance—you can place a breakpoint with an associated `cont` command (see Section 5.1.8 [Breakpoint Commands], page 85). When the breakpoint is hit, DDD will stop, collect the data, and execute the '`cont`' command, resuming execution. Using a later '`Undo`', you can step back and look at every single loop iteration.

---

[6] Perl does not allow changing the current stack frame.

To leave historic mode, you can use '`Redo`' until you are back in the current program state. However, any DDD command that refers to program state will also leave historic mode immediately by applying to the current program state instead. For instance, '`Up`' leaves historic mode immediately and selects an alternate frame in the restored current program state.

If you want to see the history of a specific variable, as recorded during program stops, you can enter the DDD command

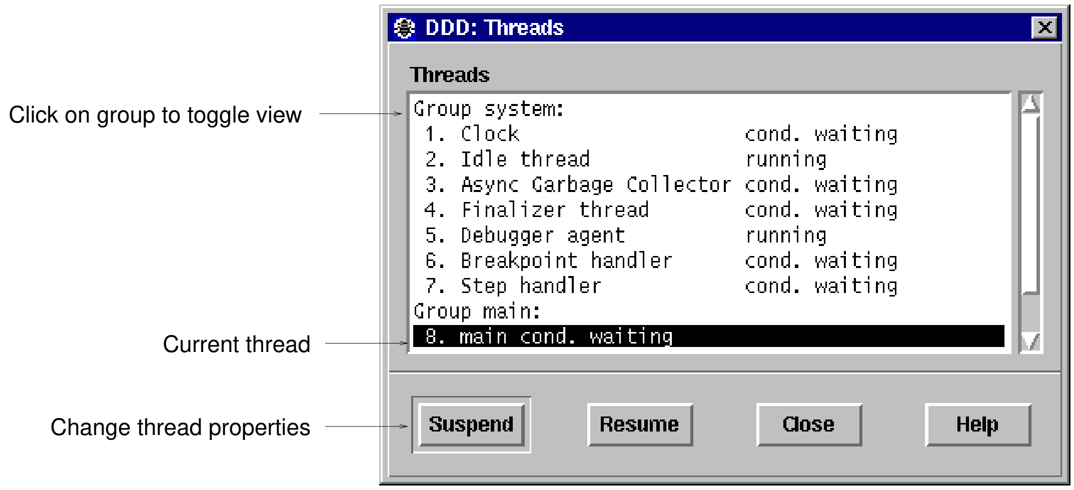
`graph history` *name*

This returns a list of all previously recorded values of the variable *name*, using array syntax. Note that *name* must have been displayed at earlier program stops in order to record values.

6.9 Examining Threads

In some operating systems, a single program may have more than one *thread* of execution. The precise semantics of threads differ from one operating system to another, but in general the threads of a single program are akin to multiple processes—except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and perhaps private memory.

For debugging purposes, DDD lets you display the list of threads currently active in your program and lets you select the *current thread*—the thread which is the focus of debugging. DDD shows all program information from the perspective of the current thread.[7]



Selecting Threads

To view all currently active threads in your program, select '`Status` ⇒ `Threads`'. The current thread is highlighted. Select any thread to make it the current thread.

Using JDB, additional functionality is available:

[7] Currently, threads are supported in GDB and JDB only.

- Select a *thread group* to switch between viewing all threads and the threads of the selected thread group;
- Click on '`Suspend`' to suspend execution of the selected threads;
- Click on '`Resume`' to resume execution of the selected threads.

For more information on threads, see the JDB and GDB documentation (see section "Debugging Programs with Multiple Threads" in *Debugging with* GDB).

6.10 Handling Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, in UNIX, `SIGINT` is the signal a program gets when you type an interrupt; `SIGSEGV` is the signal a program gets from referencing a place in memory far away from all the areas in use; `SIGALRM` occurs when the alarm clock timer goes off (which happens only if your program has requested an alarm).

Some signals, including `SIGALRM`, are a normal part of the functioning of your program. Others, such as `SIGSEGV`, indicate errors; these signals are *fatal* (kill your program immediately) if the program has not specified in advance some other way to handle the signal. `SIGINT` does not indicate an error in your program, but it is normally fatal so it can carry out the purpose of the interrupt: to kill the program.

GDB has the ability to detect any occurrence of a signal in your program. You can tell GDB in advance what to do for each kind of signal.

Normally, DDD is set up to ignore non-erroneous signals like `SIGALRM` (so as not to interfere with their role in the functioning of your program) but to stop your program immediately whenever an error signal happens. In DDD, you can view and edit these settings via '`Status` ⇒ `Signals`'.

'`Status` ⇒ `Signals`' pops up a panel showing all the kinds of signals and how GDB has been told to handle each one. The settings available for each signal are:

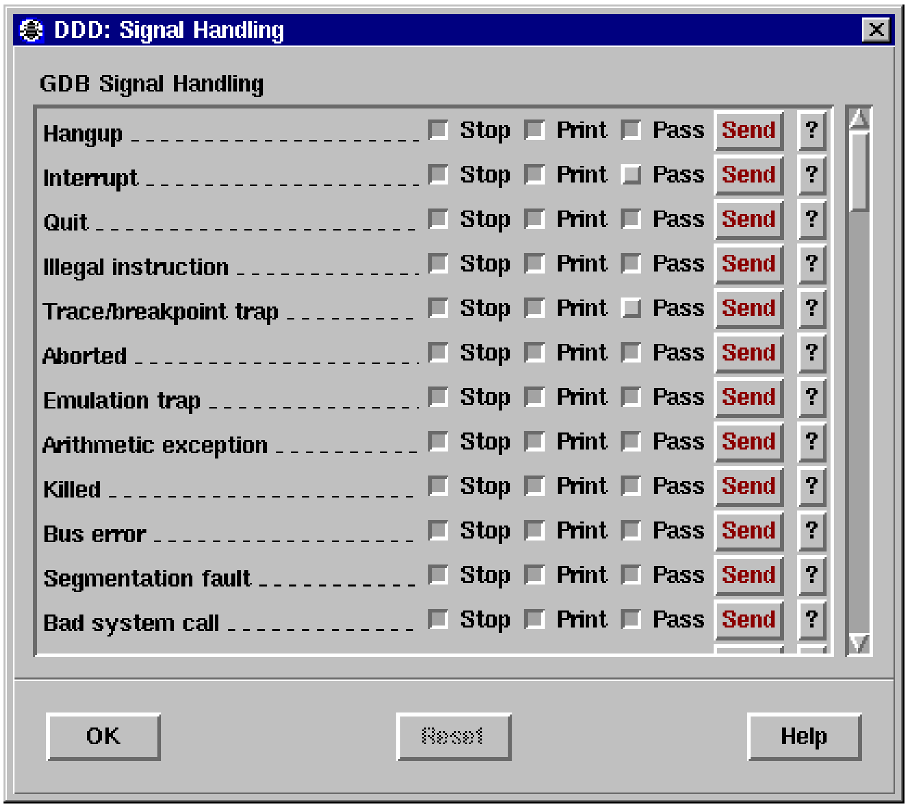
`Stop`
: If set, GDB should stop your program when this signal happens. This also implies '`Print`' being set.

`Print`
: If set, GDB should print a message when this signal happens.

: If unset, GDB should not mention the occurrence of the signal at all. This also implies '`Stop`' being unset.

`Pass`
: If set, GDB should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled.

: If unset, GDB should not allow your program to see this signal.



GDB Signal Handling Panel (Excerpt)

The entry '`All Signals`' is special. Changing a setting here affects *all signals at once*—except those used by the debugger, typically `SIGTRAP` and `SIGINT`.

To undo any changes, use '`Edit ⇒ Undo`'. The '`Reset`' button restores the saved settings.

When a signal stops your program, the signal is not visible until you continue. Your program sees the signal then, if '`Pass`' is in effect for the signal in question *at that time*. In other words, after GDB reports a signal, you can change the '`Pass`' setting in '`Status ⇒ Signals`' to control whether your program sees that signal when you continue.

You can also cause your program to see a signal it normally would not see, or to give it any signal at any time. The '`Send`' button will resume execution where your program stopped, but immediately give it the signal shown.

On the other hand, you can also prevent your program from seeing a signal. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can resume execution using '`Commands ⇒ Continue Without Signal`'.

Signal settings are not saved across DDD invocations, since changed signal settings are normally useful within specific projects only. Instead, signal settings are saved with the current session, using '`File ⇒ Save Session As`'.

6.11 Killing the Program

You can kill the process of the debugged program at any time using the 'Kill' button.

Killing the process is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump file while your program is running.

The 'Kill' button is also useful if you wish to recompile and relink your program, since on many systems it is impossible to modify an executable file while it is running in a process. In this case, when you next click on 'Run', GDB notices that the file has changed, and reads the symbol table again (while trying to preserve your current debugger state).

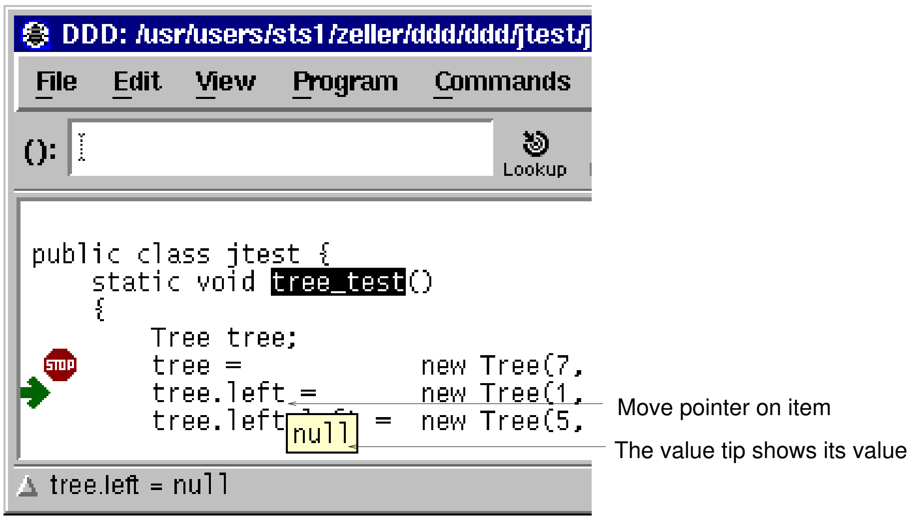
7 Examining Data

DDD provides several means to examine data.

- The quickest way to examine variables is to move the pointer on an occurrence in the source text. The value is displayed in the source line; after a second, a popup window (called *value tip*) shows the variable value. This is useful for quick examination of several simple values.
- If you want to refer to variable values at a later time, you can *print* the value in the debugger console. This allows for displaying and examining larger data structures.
- If you want to examine complex data structures, you can *display* them graphically in the data window. Displays remain effective until you delete them; they are updated each time the program stops. This is useful for large dynamic structures.
- If you want to examine arrays of numeric values, you can *plot* them graphically in a separate plot window. The plot is updated each time the program stops. This is useful for large numeric arrays.
- Using GDB or DBX, you can also *examine memory contents* in any of several formats, independently of your program's data types.

7.1 Showing Simple Values using Value Tips

To display the value of a simple variable, move the mouse pointer on its name. After a second, a small window (called *value tip*) pops up showing the value of the variable pointed at. The window disappears as soon as you move the mouse pointer away from the variable. The value is also shown in the status line.



Displaying Simple Values using Value Tips

You can disable value tips via '`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Automatic display of variable values as popup tips`'.

You can disable displaying variable values in the status line via ‘`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Automatic display of variable values in the status line`’.

These customizations are tied to the following resources:

valueTips (class Tips) Resource
: Whether value tips are enabled (‘`on`’, default) or not (‘`off`’). Value tips affect DDD performance and may be distracting for some experienced users.

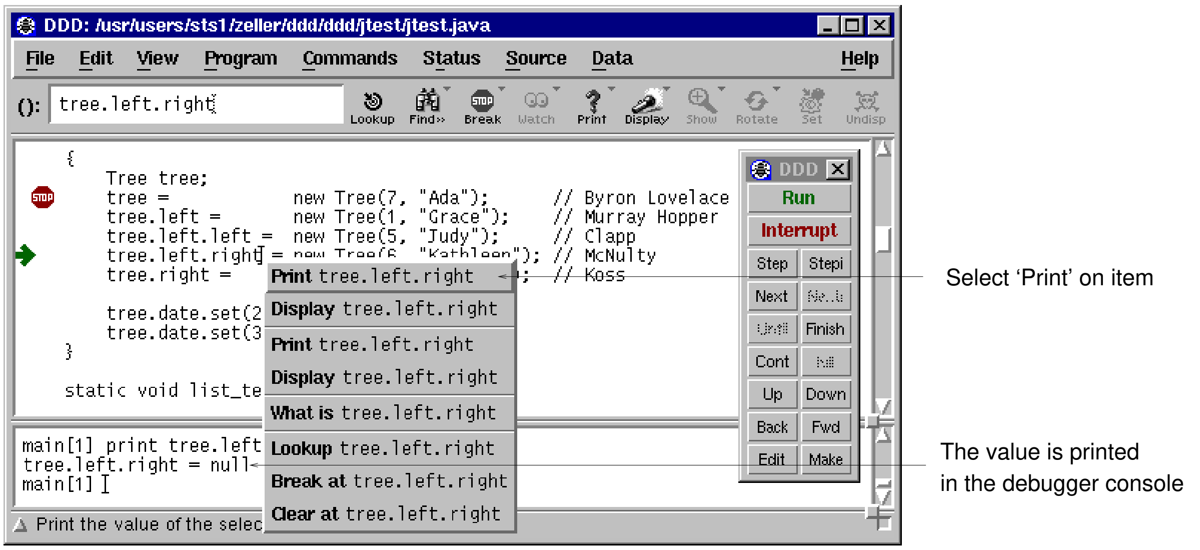
valueDocs (class Docs) Resource
: Whether the display of variable values in the status line is enabled (‘`on`’, default) or not (‘`off`’).

You can turn off value tips via ‘`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `Automatic Display of Variable Values`’.

7.2 Printing Simple Values in the Debugger Console

The variable value can also be printed in the debugger console, making it available for future operations. To print a variable value, select the desired variable by clicking *mouse button 1* on its name. The variable name is copied to the argument field. By clicking the ‘`Print`’ button, the value is printed in the debugger console. The printed value is also shown in the status line.

As a shorter alternative, you can simply press *mouse button 3* on the variable name and select the ‘`Print`’ item from the popup menu.



Displaying Simple Values in the Debugger Console

In GDB, the 'Print' button generates a `print` command, which has several more options. See section "Examining Data" in *Debugging with* GDB, for GDB-specific expressions, variables, and output formats.

7.3 Displaying Complex Values in the Data Window

To explore complex data structures, you can *display* them permanently in the *data window*. The data window displays selected data of your program, showing complex data structures graphically. It is updated each time the program stops.

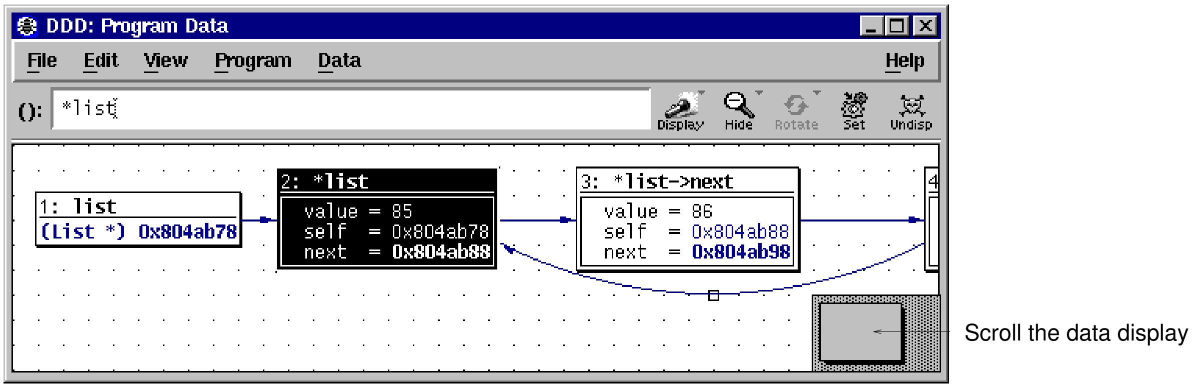
7.3.1 Display Basics

This section discusses how to create, manipulate, and delete displays. The essentials are:

- Click on '`Display`' to display the variable in '`()`'.
- Click on a display to select it.
- Click on '`Undisplay`' to delete the selected display.

7.3.1.1 Creating Single Displays

To create a new display showing a specific variable, select the variable by clicking *mouse button 1* on its name. The variable name is copied to the argument field. By clicking the '`Display`' button, a new display is created in the data window. The data window opens automatically as soon as you create a display.



Displaying Data

As a shorter alternative, you can simply press *mouse button 3* on the variable name and select '`Display`' from the popup menu.

As an even faster alternative, you can also double-click on the variable name.

As another alternative, you may also enter the expression to be displayed in the argument field and press the '`Display`' button.

Finally, you may also type in a command at the debugger prompt:

```
graph display expr [clustered] [at (x, y)]
    [dependent on display] [[now or] when in scope]
```

This command creates a new display showing the value of the expression *expr*. The optional parts have the following meaning:

`clustered`
: If given, the new display is created in a cluster. See Section 7.3.1.9 [Clustering], page 114, for a discussion.

`at (x, y)`
: If given, the new display is created at the position (*x*, *y*). Otherwise, a default position is assigned.

`dependent on` *display*
: If given, an edge from the display numbered or named *display* to the new display is created. Otherwise, no edge is created. See Section 7.3.4.1 [Dependent Values], page 120, for details.

`when in` *scope*
`now or when in` *scope*
: If '`when in`' is given, the display creation is *deferred* until execution reaches the given *scope* (a function name, as in the backtrace output).

If '`now or when in`' is given, DDD first attempts to create the display immediately. The display is deferred only if display creation fails.

If neither '`when in`' suffix nor '`now or when in`' suffix is given, the display is created immediately.

7.3.1.2 Selecting Displays

Each display in the data window has a *title bar* containing the *display number* and the displayed expression (the *display name*). Below the title, the *display value* is shown.

You can select single displays by clicking on them with *mouse button 1*.

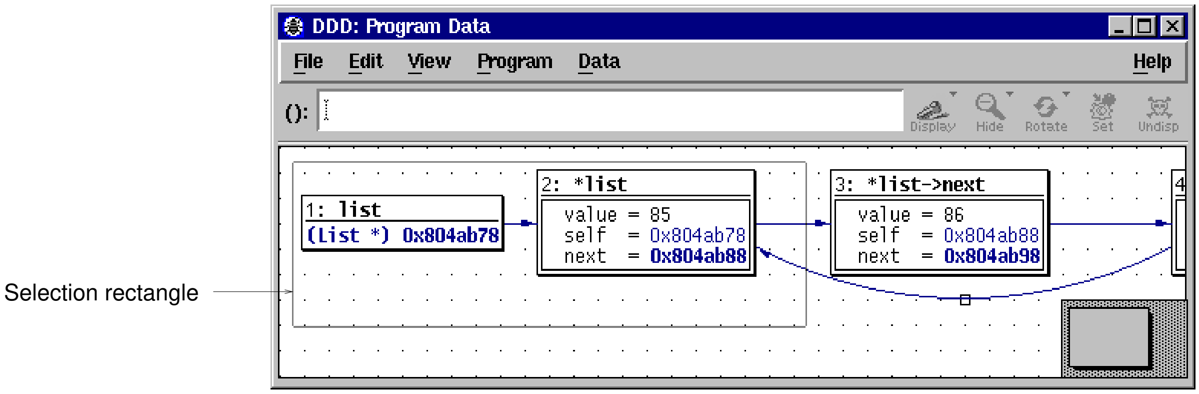
You can *extend* an existing selection by pressing the ⟨Shift⟩ key while selecting. You can also *toggle* an existing selection by pressing the ⟨Shift⟩ key while selecting already selected displays.

Single displays may also be selected by using the arrow keys ⟨Up⟩, ⟨Down⟩, ⟨Left⟩, and ⟨Right⟩.

Multiple displays are selected by pressing and holding *mouse button 1* somewhere on the window background. By moving the pointer while holding the button, a selection rectangle is shown; all displays fitting in the rectangle are selected when mouse button 1 is released.

If the ⟨Shift⟩ key is pressed while selecting, the existing selection is *extended*.

By double-clicking on a display title, the display itself and all connected displays are automatically selected.



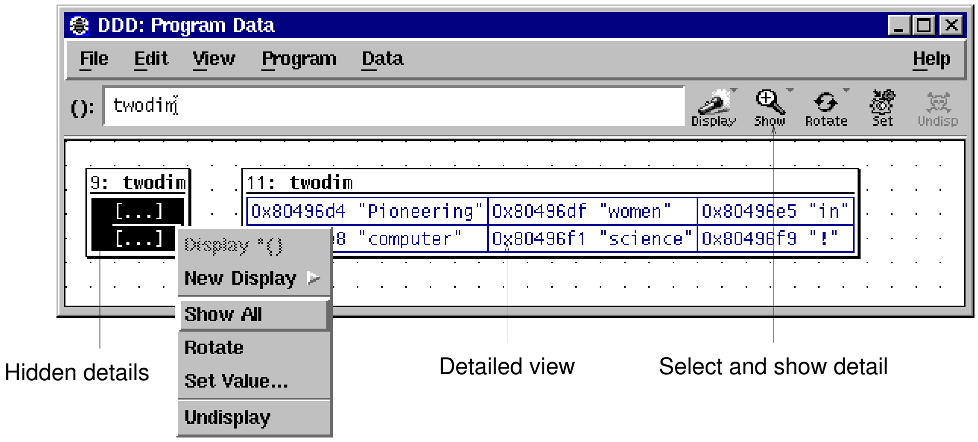
Selecting Multiple Displays

7.3.1.3 Showing and Hiding Details

Aggregate values (i.e. records, structs, classes, and arrays) can be shown *expanded*, that is, displaying all details, or *hidden*, that is, displayed as '{...}'.

To show details about an aggregate, select the aggregate by clicking *mouse button 1* on its name or value and click on the 'Show' button. Details are shown for the aggregate itself as well as for all contained sub-aggregates.

To hide details about an aggregate, select the aggregate by clicking *mouse button 1* on its name or value and click on the 'Hide' button.



Showing Display Detail

When pressing and holding *mouse button 1* on the 'Show/Hide' button, a menu pops up with even more alternatives:

`Show More ()`
: Shows details of all aggregates currently hidden, but not of their sub-aggregates. You can invoke this item several times in a row to reveal more and more details of the selected aggregate.

`Show Just ()`
: Shows details of the selected aggregate, but hides all sub-aggregates.

`Show All ()`
: Shows all details of the selected aggregate and of its sub-aggregates. This item is equivalent to the '`Show`' button.

`Hide ()`
: Hide all details of the selected aggregate. This item is equivalent to the '`Hide`' button.

As a faster alternative, you can also press *mouse button 3* on the aggregate and select the appropriate menu item.

As an even faster alternative, you can also double-click *mouse button 1* on a value. If some part of the value is hidden, more details will be shown; if the entire value is shown, double-clicking will *hide* the value instead. This way, you can double-click on a value until you get the right amount of details.

If *all* details of a display are hidden, the display is called *disabled*; this is indicated by the string '`(Disabled)`'.

Displays can also be disabled or enabled via a DDD command, which you enter at the debugger prompt:

`graph disable display` *displays...*

disables the given displays.

`graph enable display` *displays...*

re-enables the given displays.

In both commands, *displays...* is either

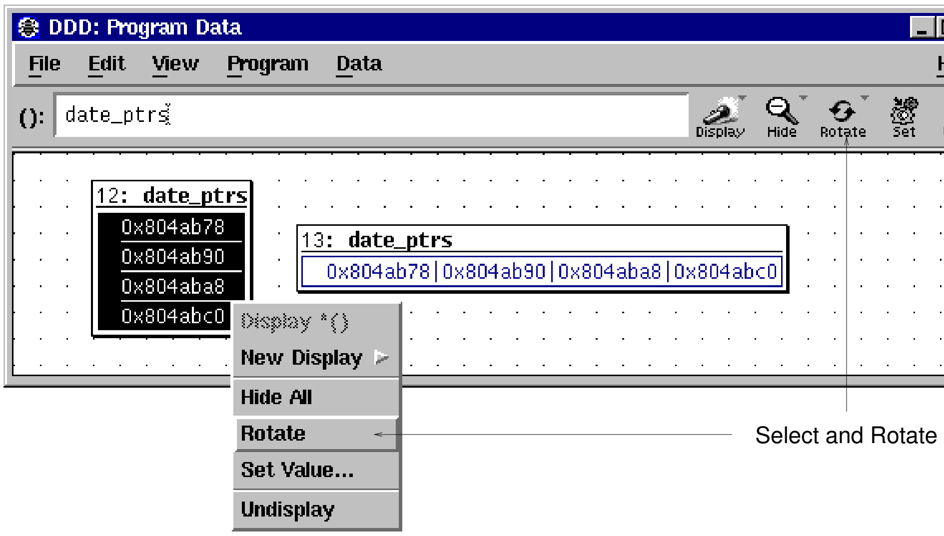
- a space-separated list of display numbers to disable or enable, or
- a single display name. If you specify a display by name, all displays with this name will be affected.

Use '`Edit` ⇒ `Undo`' to undo disabling or enabling displays.

7.3.1.4 Rotating Displays

Arrays, structures and lists can be oriented horizontally or vertically. To change the orientation of a display, select it and then click on the '`Rotate`' button.

As a faster alternative, you can also press *mouse button 3* on the array and select '`Rotate`' from the popup menu.



Rotating an Array

If a structure or list is oriented horizontally, DDD automatically suppresses the member names. This can be handy for saving space.

The last chosen display orientation is used for the creation of new displays. If you recently rotated an array to horizontal orientation, the next array you create will also be oriented horizontally. These settings are tied to the following resources:

arrayOrientation (class Orientation) Resource
: How arrays are to be oriented. Possible values are '`XmVERTICAL`' (default) and '`XmHORIZONTAL`'.

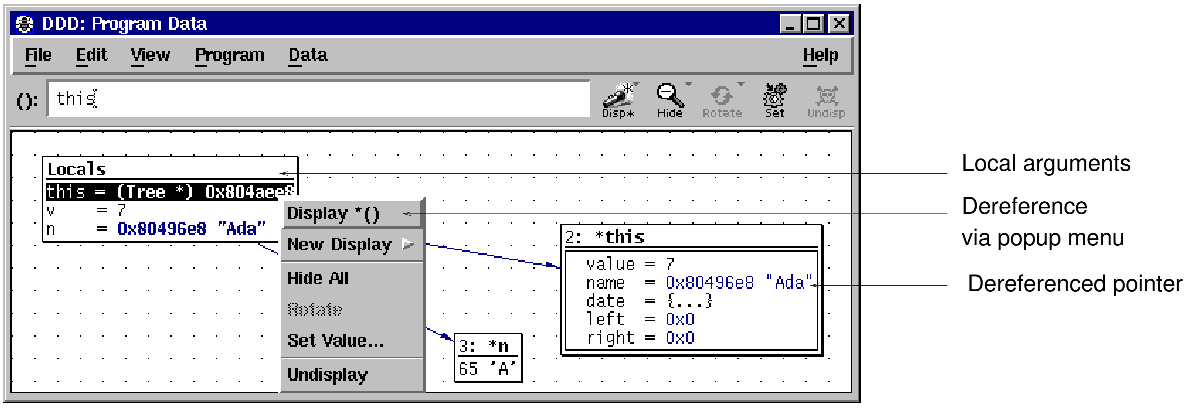
showMemberNames (class ShowMemberNames) Resource
: Whether to show struct member names or not. Default is '`on`'.

structOrientation (class Orientation) Resource
: How structs are to be oriented. Possible values are '`XmVERTICAL`' (default) and '`XmHORIZONTAL`'.

7.3.1.5 Displaying Local Variables

You can display all local variables at once by choosing '`Data` ⇒ `Display Local Variables`'. When using DBX, XDB, JDB, or Perl, this displays all local variables, including the arguments of the current function. When using GDB or PYDB, function arguments are contained in a separate display, activated by '`Data` ⇒ `Display Arguments`'.

The display showing the local variables can be manipulated just like any other data display. Individual variables can be selected and dereferenced.



Dereferencing a Local Variable

7.3.1.6 Displaying Program Status

You can create a display from the output of an arbitrary debugger command. By entering

`graph display `command``

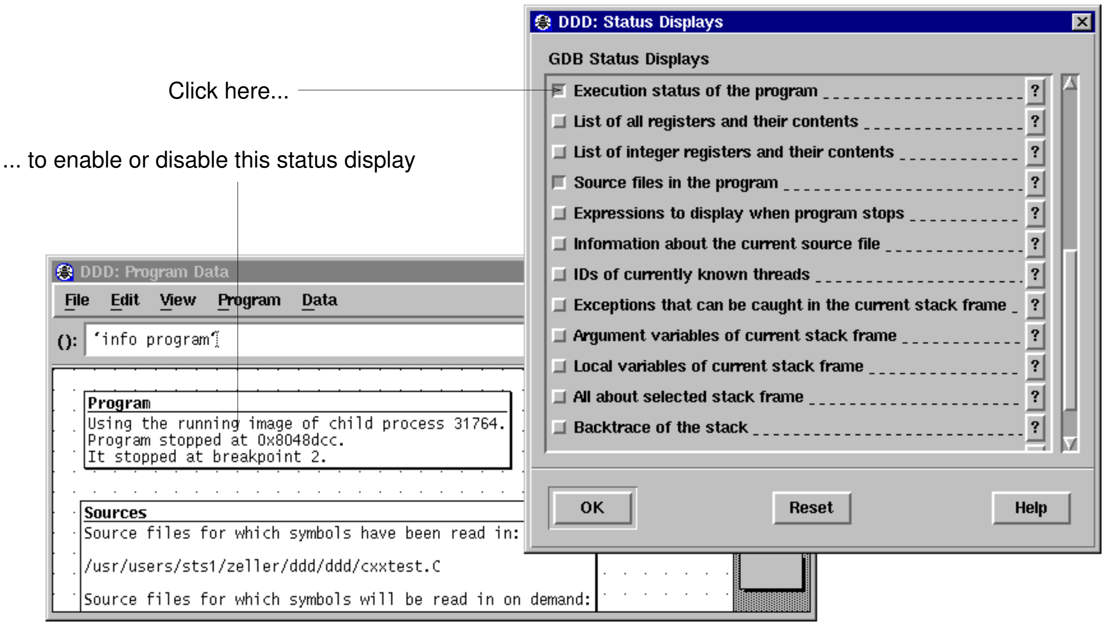
the output of *command* is turned into a *status display* updated each time the program stops.

For instance, the command

`graph display `where``

creates a status display named '`Where`' that shows the current backtrace.

If you are using GDB, DDD provides a panel from which you can choose useful status displays. Select '`Data` ⇒ `Status Displays`' and pick your choice from the list.



Activating Status Displays

Refreshing status displays at each stop takes time; you should delete status displays as soon as you don't need them any more.

7.3.1.7 Refreshing the Data Window

The data window is automatically updated or *refreshed* each time the program stops. Values that have changed since the last refresh are highlighted.

However, there may be situations where you should refresh the data window explicitly. This is especially the case whenever you changed debugger settings that could affect the data format, and want the data window to reflect these settings.

You can refresh the data window by selecting '`Data` ⇒ `Refresh Displays`'.

As an alternative, you can press *mouse button 3* on the background of the data window and select the '`Refresh Displays`' item.

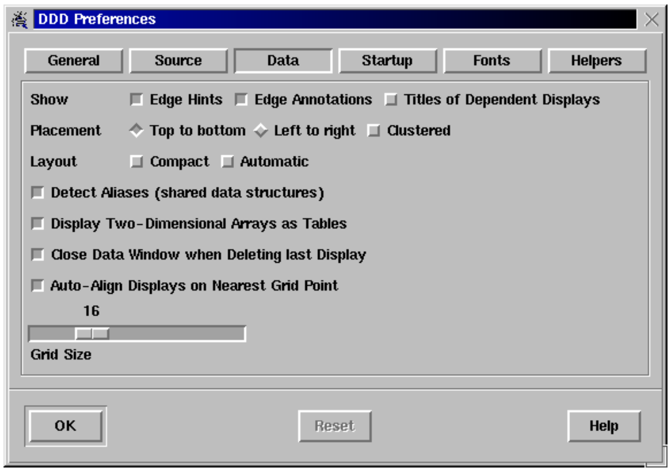
Typing

```
graph refresh
```

at the debugger prompt has the same effect.

7.3.1.8 Display Placement

By default, displays are created from *top to bottom*—that is, each new display is placed below the downmost one. You can change this setting to *left to right* via '`Edit` ⇒ `Preferences` ⇒ `Data` ⇒ `Placement` ⇒ `Left to right`'.



Data Preferences

This setting is tied to the following resource:

displayPlacement (class Orientation) Resource

If this is '`XmVERTICAL`' (default), DDD places each new independent display below the downmost one. If this is '`XmHORIZONTAL`', each new independent display is placed on the right of the rightmost one.

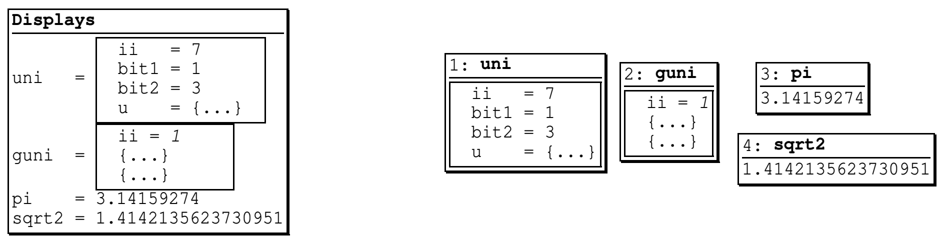
Note that changing the placement of new displays also affects the placement of *dependent displays* (see Section 7.3.4.1 [Dependent Values], page 120). In *top to bottom* mode, dependent displays are created on the right of the originating display; in *left to right* mode, dependent displays are created on the below the originating display.

7.3.1.9 Clustering Displays

If you examine several variables at once, having a separate display for each of them uses a lot of screen space. This is why DDD supports *clusters*. A cluster merges several logical data displays into one physical display, saving screen space.

There are two ways to create clusters:

- You can create clusters *manually*. This is done by selecting the displays to be clustered and choosing '`Undisp` ⇒ `Cluster ()`'. This creates a new cluster from all selected displays. If an already existing cluster is selected, too, the selected displays will be clustered into the selected cluster.
- You can create a cluster *automatically* for all independent data displays, such that all new data displays will automatically be clustered, too. This is achieved by enabling '`Edit` ⇒ `Preferences` ⇒ `Data` ⇒ `Placement` ⇒ `clustered`'.



Clustered and Unclustered Displays

Displays in a cluster can be selected and manipulated like parts of an ordinary display; in particular, you can show and hide details, or dereference pointers. However, edges leading to clustered displays can not be shown, and you must either select one or all clustered displays.

Disabling a cluster is called *unclustering*, and again, there are two ways of doing it:

- You can uncluster displays *manually*, by selecting the cluster and choosing '`Undisp` ⇒ `Uncluster ()`'.
- You can uncluster all current and future displays by disabling '`Edit` ⇒ `Preferences` ⇒ `Data` ⇒ `Placement` ⇒ `clustered`'.

7.3.1.10 Creating Multiple Displays

To display several successive objects of the same type (a section of an array, or an array of dynamically determined size), you can use the notation '*from*`..`*to*' in display expressions.

from and *to* are numbers that denote the first and last expression to display. Thus,

```
graph display argv[0..9]
```

creates 10 new displays for '`argv[0]`', '`argv[1]`', ..., '`argv[9]`'. The displays are clustered automatically (see Section 7.3.1.9 [Clustering], page 114), such that you can easily handle the set just like an array.

The '*from*`..`*to*' notation can also be used multiple times. For instance,

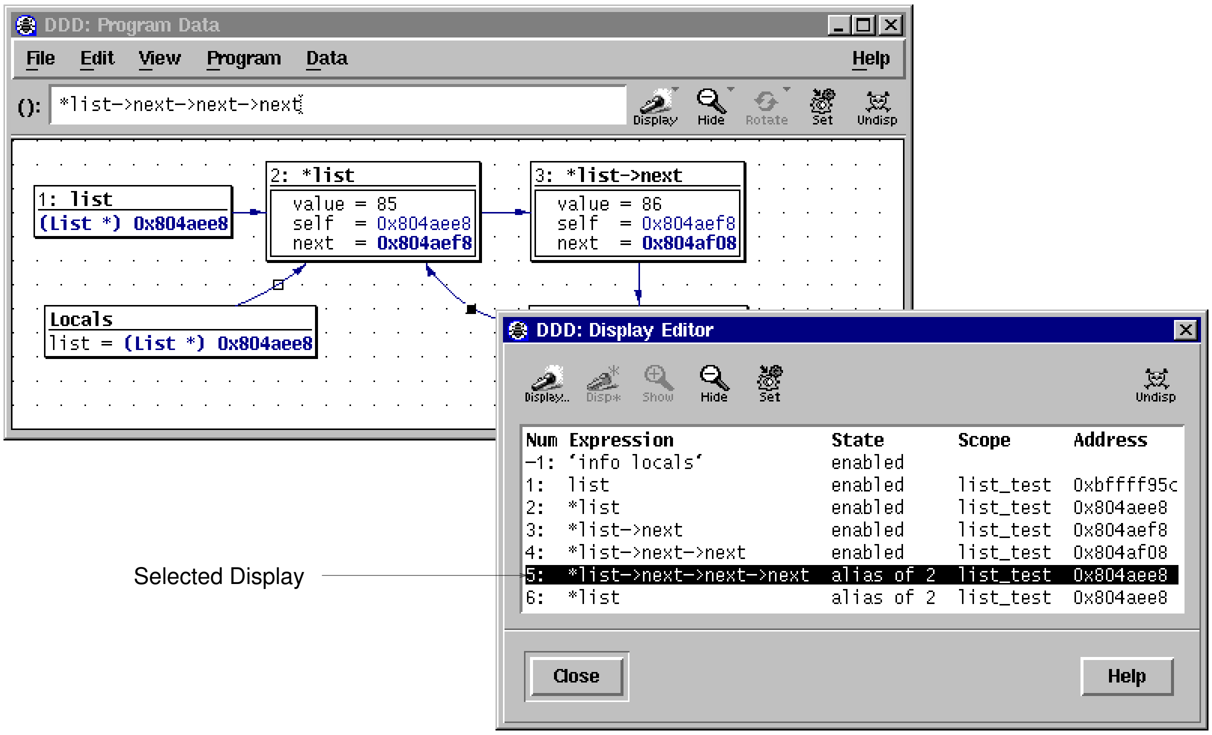
```
graph display 1..5 * 1..5
```

creates a handy small multiplication table.

The '*from*`..`*to*' notation creates several displays, which takes time to create and update. If you want to display only a part of an array, *array slices* are a more efficient way. See Section 7.3.2.1 [Array Slices], page 117, for a discussion.

7.3.1.11 Editing all Displays

You can view the state of all displays by selecting '`Data` ⇒ `Displays`'. This invokes the *Display Editor*.



The Display Editor

The Display Editor shows the properties of each display, using the following fields:

'Num' The display number.

'Expression'
: The displayed expression.

'State' One of

- 'enabled'
  Normal state.

- 'disabled'
  Disabled; all details are hidden. Use 'Show' to enable.

- 'not active'
  Out of scope.

- 'deferred'
  Will be created as soon as its 'Scope' is reached (see Section 7.3.1.1 [Creating Single Displays], page 107).

- 'clustered'
  Part of a cluster (see Section 7.3.1.9 [Clustering], page 114). Use 'Undisp ⇒ Uncluster' to uncluster.

- 'alias of *display*'
  A suppressed alias of display *display* (see Section 7.3.4.3 [Shared Structures], page 121).

'Scope' The scope in which the display was created. For deferred displays, this is the scope in which the display will be created.

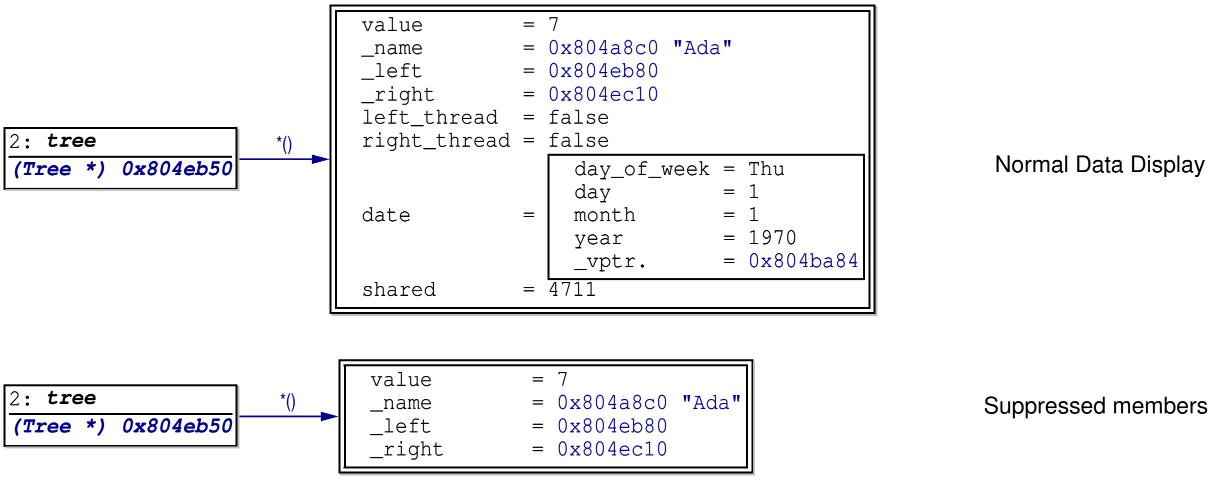
'Address'
: The address of the displayed expression. Used for resolving aliases (see Section 7.3.4.3 [Shared Structures], page 121).

7.3.1.12 Deleting Displays

To delete a single display, select its title or value and click on the 'Undisp' button. As an alternative, you can also press *mouse button 3* on the display and select the 'Undisplay' item.

When a display is deleted, its immediate ancestors and descendants are automatically selected, so that you can easily delete entire graphs.

If you have selected only part of a display, clicking on the 'Undisp' button allows you to *suppress* this part—by applying the *Suppress Values* theme on the part. You'll be asked for confirmation first. See Section 7.3.5.1 [Using Data Themes], page 125, for details.



Suppressing Values

To delete several displays at once, use the '`Undisp`' button in the Display Editor (invoked via '`Data ⇒ Displays`'). Select any number of display items in the usual way and delete them by pressing '`Undisp`'.

As an alternative, you can also use a DDD command:

`graph undisplay` *displays...*

Here, *displays...* is either

- a space-separated list of display numbers to disable or enable, or
- a single display name. If you specify a display by name, all displays with this name will be affected.

If you are using stacked windows, deleting the last display from the data window also automatically closes the data window. (You can change this via '`Edit ⇒ Preferences ⇒ Data ⇒ Close data window when deleting last display`'.)

If you deleted a display by mistake, use '`Edit ⇒ Undo`' to re-create it.

Finally, you can also cut, copy, and paste displays using the '`Cut`', '`Copy`', and '`Paste`' items from the '`Edit`' menu. The clipboard holds the *commands* used to create the displays; '`Paste`' inserts the display commands in the debugger console. This allows you to save displays for later usage or to copy displays across multiple DDD instances.

7.3.2 Arrays

DDD has some special features that facilitate handling of arrays.

7.3.2.1 Array Slices

It is often useful to print out several successive objects of the same type in memory; a *slice* (section) of an array, or an array of dynamically determined size for which only a pointer exists in the program.

Using DDD, you can display slices using the '*from*`..`*to*' notation (see Section 7.3.1.10 [Creating Multiple Displays], page 115). But this requires that you already know *from* and *to*; it is also inefficient to create several single displays. If you use GDB, you have yet another alternative.

Using GDB, you can display successive objects by referring to a contiguous span of memory as an *artificial array*, using the binary operator '`@`'. The left operand of '`@`' should be the first element of the desired array and be an individual object. The right operand should be the desired length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on.

Here is an example. If a program says

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of `array` with

```
print array[0]@len
```

and display the contents with

```
graph display array[0]@len
```

The general form of displaying an array slice is thus

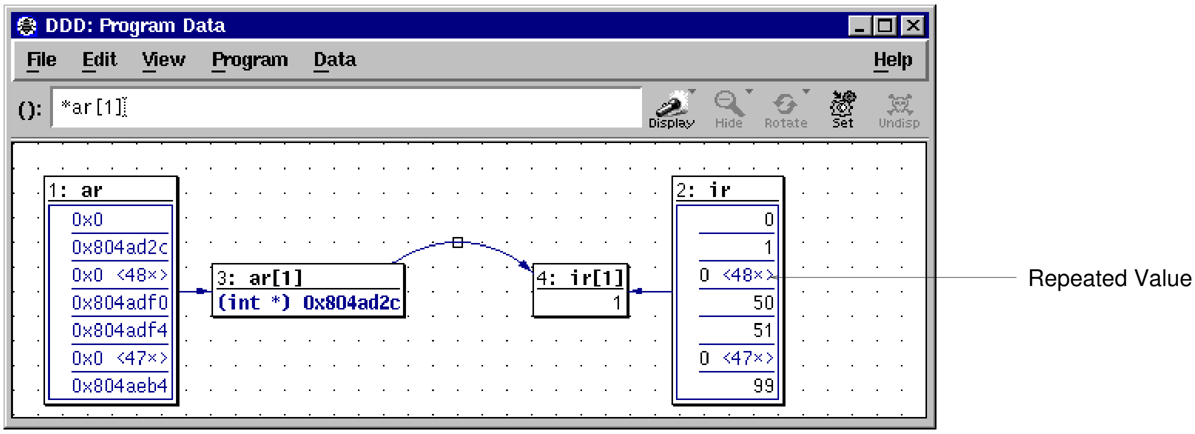
`graph display` *array*`[`*first*`]@`*nelems*

where *array* is the name of the array to display, *first* is the index of the first element, and *nelems* is the number of elements to display.

The left operand of '`@`' must reside in memory. Array values made with '`@`' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions.

7.3.2.2 Repeated Values

Using GDB, an array value that is repeated 10 or more times is displayed only once. The value is shown with a '`<`*n*`x>`' postfix added, where *n* is the number of times the value is repeated. Thus, the display '`0x0 <30x>`' stands for 30 array elements, each with the value '`0x0`'. This saves a lot of display space, especially with homogeneous arrays.



Displaying Repeated Array Values

The default GDB threshold for repeated array values is 10. You can change it via 'Edit ⇒ GDB Settings ⇒ Threshold for repeated print elements'. Setting the threshold to 0 will cause GDB (and DDD) to display each array element individually. Be sure to refresh the data window via 'Data ⇒ Refresh Displays' after a change in GDB settings.

You can also configure DDD to display each array element individually:

expandRepeatedValues (class ExpandRepeatedValues) Resource

GDB can print repeated array elements as '*value* <repeated *n* times>'. If 'expandRepeatedValues' is 'on', DDD will display *n* instances of *value* instead. If 'expandRepeatedValues' is 'off' (default), DDD will display *value* with '<*n*x>' appended to indicate the repetition.

7.3.2.3 Arrays as Tables

By default, DDD lays out two-dimensional arrays as tables, such that all array elements are aligned with each other.[1] To disable this feature, unset 'Edit ⇒ Preferences ⇒ Data ⇒ Display Two-Dimensional Arrays as Tables'. This is tied to the following resource:

align2dArrays (class Align2dArrays) Resource

If 'on' (default), DDD lays out two-dimensional arrays as tables, such that all array elements are aligned with each other. If 'off', DDD treats a two-dimensional array as an array of one-dimensional arrays, each aligned on its own.

7.3.3 Assignment to Variables

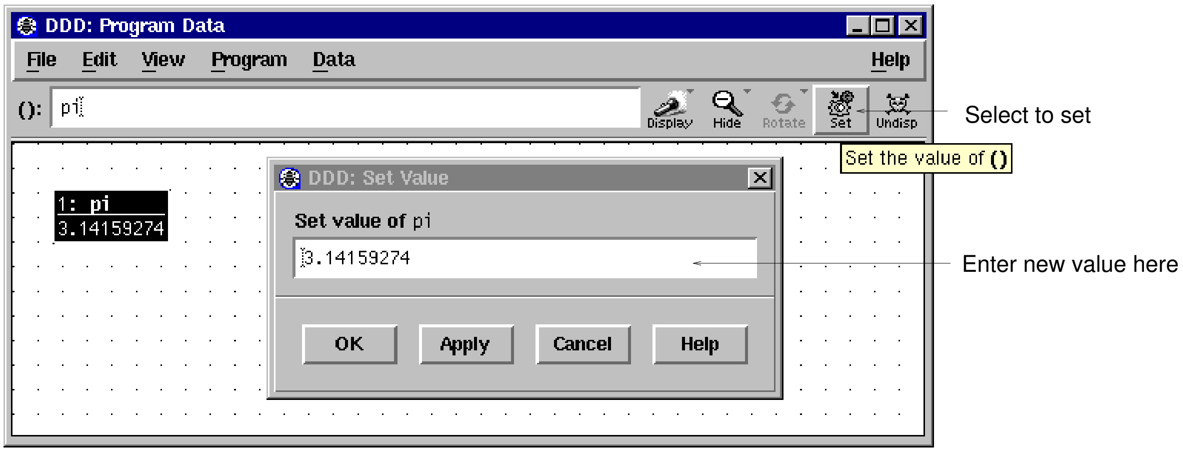
During program execution, you can change the values of arbitrary variables.[2]

To change the value of a variable, enter its name in '()'—for instance, by selecting an occurrence or a display. Then, click on the 'Set' button. In a dialog, you can edit the variable value at will; clicking the 'OK' or 'Apply' button commits your change and assigns the new value to the variable.

---

[1] This requires that the full array size is known to the debugger.

[2] JDB 1.1 does not support changing variable values.



Changing Variable Values

To change a displayed value, you can also select '`Set Value`' menu from the data popup menu,

If you made a mistake, you can use '`Edit` ⇒ `Undo`' to re-set the variable to its previous value.

7.3.4 Examining Structures

Besides displaying simple values, DDD can also visualize the *Dependencies* between values—especially pointers and other references that make up complex data structures.

7.3.4.1 Displaying Dependent Values

Dependent displays are created from an existing display. The dependency is indicated by an *edge* leading from the originating display to the dependent display.

To create a dependent display, select the originating display or display part and enter the dependent expression in the '`():`' argument field. Then click on the '`Display`' button.

Using dependent displays, you can investigate the data structure of a tree for example and lay it out according to your intuitive image of the tree data structure.

By default, DDD does not recognize shared data structures (i.e. a data object referenced by multiple other data objects). See Section 7.3.4.3 [Shared Structures], page 121, for details on how to examine such structures.

7.3.4.2 Dereferencing Pointers

There are special shortcuts for creating dependent displays showing the value of a dereferenced pointer. This allows for rapid examination of pointer-based data structures.

To dereference a pointer, select the originating pointer value or name and click on the '`Disp *`' button. A new display showing the dereferenced pointer value is created.

As a faster alternative, you can also press *mouse button 3* on the originating pointer value or name and select the '`Display *`' menu item.

As an even faster alternative, you can also double-click *mouse button 1* on the originating pointer value or name. If you press ⟨Ctrl⟩ while double-clicking, the display will be dereferenced *in place*–that is, it will be replaced by the dereferenced display.

The '`Display *()`' function is also accessible by pressing and holding the '`Display`' button.

7.3.4.3 Shared Structures

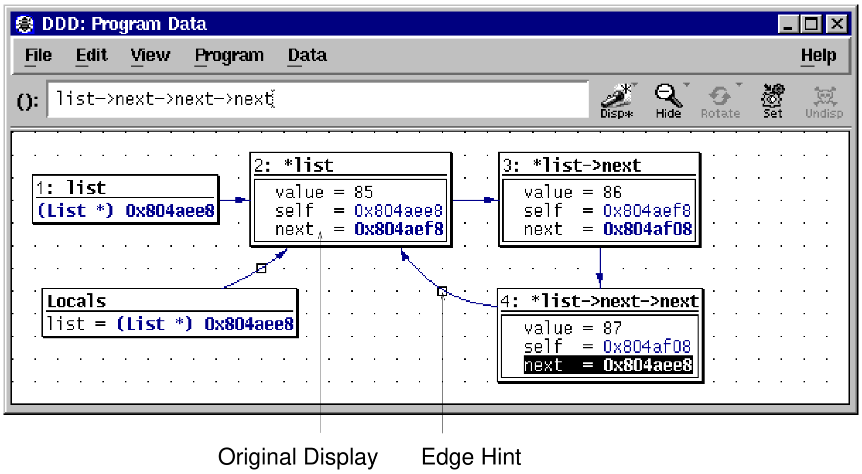
By default, DDD does not recognize shared data structures—that is, a data object referenced by multiple other data objects. For instance, if two pointers '`p1`' and '`p2`' point at the same data object '`d`', the data displays '`d`', '`*p1`', and '`*p2`' will be separate, although they denote the same object.

DDD provides a special mode which makes it detect these situations. DDD recognizes if two or more data displays are stored at the same physical address, and if this is so, merges all these *aliases* into one single data display, the *original data display*. This mode is called *Alias Detection*; it is enabled via '`Data ⇒ Detect Aliases`'.

When alias detection is enabled, DDD inquires the memory location (the *address*) of each data display after each program step. If two displays have the same address, they are merged into one. More specifically, only the one which has least recently changed remains (the *original data display*); all other aliases are *suppressed*, i.e. completely hidden. The edges leading to the aliases are replaced by edges leading to the original data display.

An edge created by alias detection is somewhat special: rather than connecting two displays directly, it goes through an *edge hint*, describing an arc connecting the two displays and the edge hint.

Each edge hint is a placeholder for a suppressed alias; selecting an edge hint is equivalent to selecting the alias. This way, you can easily delete display aliases by simply selecting the edge hint and clicking on '`Undisp`'.



Examining Shared Data Structures

To access suppressed display aliases, you can also use the Display Editor. Suppressed displays are listed in the Display Editor as *aliases* of the original data display. Via the Display Editor, you can select, change, and delete suppressed displays.

Suppressed displays become visible again as soon as

- alias detection is disabled,
- their address changes such that they are no more aliases, or
- the original data display is deleted, such that the least recently changed alias becomes the new original data display.

Please note the following *caveats* with alias detection:

- Alias detection requires that the current programming language provides a means to determine the address of an arbitrary data object. Currently, only C, C++, and Java are supported.
- Some inferior debuggers (for instance, SunOS DBX) produce incorrect output for address expressions. Given a pointer *p*, you may verify the correct function of your inferior debugger by comparing the values of *p* and '`&`*p*' (unless *p* actually points to itself). You can also examine the data display addresses, as shown in the Display Editor.
- Alias detection slows down DDD slightly, which is why you can turn it off. You may consider to enable it only at need—for instance, while examining some complex data structure—and disable it while examining control flow (i.e., stepping through your program). DDD will automatically restore edges and data displays when switching modes.

Alias detection is controlled by the following resources:

deleteAliasDisplays (class DeleteAliasDisplays) Resource
: If this is '`on`' (default), the '`Undisplay ()`' button also deletes all aliases of the selected displays. If this is '`off`', only the selected displays are deleted; the aliases remain, and one of the aliases will be unsuppressed.

detectAliases (class DetectAliases) Resource
: If '`on`' (default), DDD attempts to recognize shared data structures. If '`off`', shared data structures are not recognized.

typedAliases (class TypedAliases) Resource
: If '`on`' (default), DDD requires structural equivalence in order to recognize shared data structures. If this is '`off`', two displays at the same address are considered aliases, regardless of their structure.

7.3.4.4 Display Shortcuts

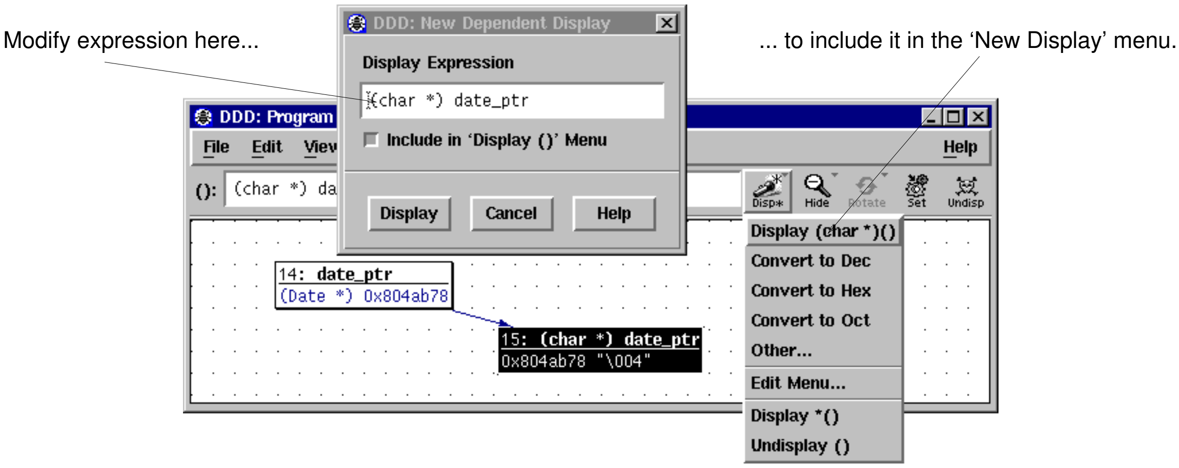
DDD maintains a *shortcut menu* of frequently used display expressions. This menu is activated

- by pressing and holding the '`Display`' button, or
- by pressing *mouse button 3* on some display and selecting '`New Display`', or
- by pressing ⟨Shift⟩ and *mouse button 3* on some display.

By default, the shortcut menu contains frequently used base conversions.

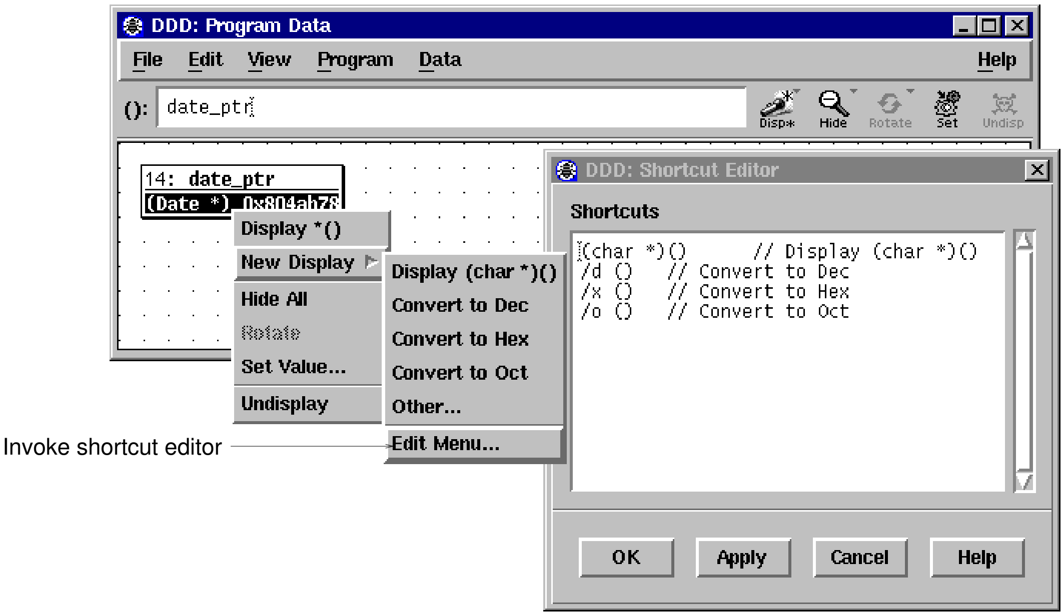
The ‘Other’ entry in the shortcut menu lets you create a new display that *extends* the shortcut menu.

As an example, assume you have selected a display named ‘date_ptr’. Selecting ‘Display ⇒ Other’ pops up a dialog that allows you to enter a new expression to be displayed—for instance, you can cast the display ‘date_ptr’ to a new display ‘(char *)date_ptr’. If the ‘Include in 'New Display' Menu’ toggle was activated, the shortcut menu will then contain a new entry ‘Display (char *)()’ that will cast *any* selected display *display* to ‘(char *)*display*’. Such shortcuts can save you a lot of time when examining complex data structures.



Using Display Shortcuts

You can edit the contents of the ‘New Display’ menu by selecting its ‘Edit Menu’ item. This pops up the *Shortcut Editor* containing all shortcut expressions, which you can edit at leisure. Each line contains the expression for exactly one menu item. Clicking on ‘Apply’ re-creates the ‘New Display’ menu from the text. If the text is empty, the ‘New Display’ menu will be empty, too.



Editing Display Shortcuts

DDD also allows you to specify individual labels for user-defined buttons. You can write such a label after the expression, separated by '`//`'. This feature is used in the default contents of the GDB '`New Display`' menu, where each of the base conversions has a label:

```
/t ()   // Convert to Bin
/d ()   // Convert to Dec
/x ()   // Convert to Hex
/o ()   // Convert to Oct
```

Feel free to add other conversions here. DDD supports up to 20 '`New Display`' menu items.

The shortcut menu is controlled by the following resources:

dbxDisplayShortcuts (class DisplayShortcuts) Resource

A newline-separated list of display expressions to be included in the '`New Display`' menu for DBX.

If a line contains a label delimiter[3], the string before the delimiter is used as *expression*, and the string after the delimiter is used as label. Otherwise, the label is '`Display` *expression*'. Upon activation, the string '`()`' in *expression* is replaced by the name of the currently selected display.

gdbDisplayShortcuts (class DisplayShortcuts) Resource

A newline-separated list of display expressions to be included in the '`New Display`' menu for GDB. See the description of '`dbxDisplayShortcuts`', above.

[3] The string '`//`'; can be changed via the '`labelDelimiter`' resource. See Section 10.4.1 [Customizing Buttons], page 152, for details.

jdbDisplayShortcuts (class DisplayShortcuts) Resource
A newline-separated list of display expressions to be included in the 'New Display' menu for JDB. See the description of 'dbxDisplayShortcuts', above.

perlDisplayShortcuts (class DisplayShortcuts) Resource
A newline-separated list of display expressions to be included in the 'New Display' menu for Perl. See the description of 'dbxDisplayShortcuts', above.

bashDisplayShortcuts (class DisplayShortcuts) Resource
A newline-separated list of display expressions to be included in the 'New Display' menu for Bash. See the description of 'dbxDisplayShortcuts', above.

pydbDisplayShortcuts (class DisplayShortcuts) Resource
A newline-separated list of display expressions to be included in the 'New Display' menu for PYDB. See the description of 'dbxDisplayShortcuts', above.

xdbDisplayShortcuts (class DisplayShortcuts) Resource
A newline-separated list of display expressions to be included in the 'New Display' menu for XDB. See the description of 'dbxDisplayShortcuts', above.

7.3.5 Customizing Displays

7.3.5.1 Using Data Themes

DDD provides a simple method to customize displays. DDD comes with a number of *visual modifiers*, called *data themes*.

Each theme modifies a particular aspect of a data display. It can be applied to individual displays or to a number of displays. The themes installed with DDD include:

'Small Titles'
: Apply this theme to show display titles in a smaller font.

'Small Values'
: Apply this theme to display values in a smaller font.

'Tiny Values'
: Apply this theme to display values in a tiny font.

'Suppress Values'
: Apply this theme to display values not at all.

Each of these themes can be applied for specific displays.

Some DDD Themes

To apply a theme on a display,

1. Press *mouse button 3* on the display.
2. Select '`Theme`'
3. Select the theme to apply.

For instance, to display the variable `s` in a tiny font, click *mouse button 3* on the display of `s`, and select '`Theme` ⇒ `Tiny Values` ⇒ `Apply`'.

To unapply a theme, just click on '`Undo`' (if you just applied it) or repeat the sequence as above.

7.3.5.2 Applying Data Themes to Several Values

Whenever you want to apply a theme on a *struct member* or an *array element,* you will be asked whether to

- apply the theme on the single value only, or
- apply the theme on all similar values.

Suppose, for instance, that you don't want to see '`vptr`' members anymore. Then you'd apply the theme *Suppress Values* on all similar values.

On the other hand, if you want to highlight one single value only, you'd apply the theme *Red Background* on only one single value.

If you find this confirmation annoying, you can define a command button which directly applies the theme. See Section 10.5 [Defining Commands], page 154, for details on defining commands.

Applying and unapplying themes is associated with the following commands:

`graph apply theme` *name pattern*

applies the theme *name* on *pattern*.

`graph unapply theme` *name pattern*

unapplies the theme *name* on *pattern*.

`graph toggle theme` *name pattern*

applies the theme *name* on *pattern* if it was not already applied, and unapplies it otherwise.

7.3.5.3 Editing Themes

Each theme can be globally activated or not. If a theme is activated, it is applied to all expressions that match its *pattern*.

Normally, these patterns are automatically maintained by simply selecting the themes for the individual displays. However, you can also edit patterns directly.

Patterns are separated by '`;`' and contain shell-like metacharacters:

- '`*`' matches any sequence of characters.
- '`?`' matches any single character.
- '`[`*set*`]`' matches any character in *set*. Character ranges can be expressed using *from-to*: '`[0-9a-zA-Z_]`' is the set of characters allowed in C characters.
- '`[!`*set*`]`' matches any character not in *set*.
- To suppress the special syntactic significance of any metacharacter\n\ and match the character exactly, precede it with '`\`' (backslash).
- To suppress the syntactic significance of *all* metacharacters,\n\ enclose the pattern in double or single quotes.\n\

To edit the set of themes, invoke '`Data` ⇒ `Themes`'.

To apply changes you made to the themes, click on '`Apply`'. To revert the themes to the last saved, click on '`Reset`'.

7.3.5.4 Writing Data Themes

You can write your own data themes, customizing the display to match your need. See section "Top" in *Writing DDD Themes*, for details.

7.3.5.5 Display Resources

You can use these resources to control display appearance:

autoCloseDataWindow (class AutoClose) Resource
: If this is '`on`' (default) and DDD is in stacked window mode, deleting the last display automatically closes the data window. If this is '`off`', the data window stays open even after deleting the last display.

bumpDisplays (class BumpDisplays) Resource
: If some display *d* changes size and this resource is '`on`' (default), DDD assigns new positions to displays below and on the right of *d* such that the distance between displays remains constant. If this is '`off`', other displays are not rearranged.

clusterDisplays (class ClusterDisplays) Resource
: If '`on`', new independent data displays will automatically be clustered. Default is '`off`', meaning to leave new displays unclustered.

hideInactiveDisplays (class HideInactiveDisplays) Resource
: If some display gets out of scope and this resource is '`on`' (default), DDD removes it from the data display. If this is '`off`', it is simply disabled.

showBaseDisplayTitles (class ShowDisplayTitles) Resource
: Whether to assign titles to base (independent) displays or not. Default is '`on`'.

showDependentDisplayTitles (class ShowDisplayTitles) Resource
: Whether to assign titles to dependent displays or not. Default is '`off`'.

suppressTheme (class Theme) Resource
: The theme to apply when selecting '`Undisp`' on a data value. Default is '`suppress.vsl`'.

themes (class Themes) Resource
: A newline-separated list of themes. Each theme has the format *name*, tabulator character, *pattern*.

7.3.5.6 VSL Resources

The following resources control the VSL interpreter:

vslBaseDefs (class VSLDefs) Resource
: A string with additional VSL definitions that are appended to the builtin VSL library. This resource is prepended to the '`vslDefs`' resource below and set in the DDD application defaults file; don't change it.

vslDefs (class VSLDefs) Resource

A string with additional VSL definitions that are appended to the builtin VSL library. The default value is an empty string. This resource can be used to override specific VSL definitions that affect the data display. The preferred method, though, is to write a specific data theme (see Section 7.3.5.4 [Writing Data Themes], page 127).

vslLibrary (class VSLLibrary) Resource

The VSL library to use. '`builtin`' (default) means to use the built-in library, any other value is used as file name.

vslPath (class VSLPath) Resource

A colon-separated list of directories to search for VSL include files. The following directory names are special:

- The special directory name '`user_themes`' stands for your individual theme directory, typically '`~/.ddd/themes/`'.
- The special directory name '`ddd_themes`' stands for the installed theme directory, typically '`/usr/local/share/ddd-3.3.11/themes/`'.

Default is '`user_themes:ddd_themes:.`', which means that DDD first searches your theme directory, followed by the system directory and the current directory.

If your DDD source distribution is installed in '`/opt/src`', you can use the following settings to read the VSL library from '`/home/joe/ddd.vsl`':

```
Ddd*vslLibrary: /home/joe/ddd.vsl
Ddd*vslPath:    user_themes:.:/opt/src/ddd/ddd:/opt/src/ddd/vsllib
```

VSL include files referenced by '`/home/joe/ddd.vsl`' are searched first in the current directory '`.`', then in your theme directory, then in '`/opt/src/ddd/ddd/`', and then in '`/opt/src/ddd/vsllib/`'.

Instead of supplying another VSL library, it is often easier to specify some minor changes to the built-in library (see Section 7.3.5.4 [Writing Data Themes], page 127).

7.3.6 Layouting the Graph

If you have several displays at once, you may wish to arrange them according to your personal preferences. This section tells you how you can do this.

7.3.6.1 Moving Displays

From time to time, you may wish to move displays at another place in the data window. You can move a single display by pressing and holding *mouse button 1* on the display title. Moving the pointer while holding the button causes all selected displays to move along with the pointer.

Edge hints can be selected and moved around like other displays. If an arc goes through the edge hint, you can change the shape of the arc by moving the edge hint around.

For fine-grain movements, selected displays may also be moved using the arrow keys. Pressing ⟨Shift⟩ and an arrow key moves displays by single pixels. Pressing ⟨Ctrl⟩ and arrow keys moves displays by grid positions.

7.3.6.2 Scrolling Data

If the data window becomes too small to hold all displays, scroll bars are created. If your DDD is set up to use *panners* instead, a panner is created in the lower right edge. When the panner is moved around, the window view follows the position of the panner.

To change from scroll bars to panners, use '`Edit` ⇒ `Startup` ⇒ `Data Scrolling`' and choose either '`Panner`' or '`Scrollbars`'.

This setting is tied to the following resource:

pannedGraphEditor (class PannedGraphEditor) Resource

The control to scroll the graph.

- If this is '`on`', an Athena panner is used (a kind of two-directional scrollbar).
- If this is '`off`' (default), two M*tif scrollbars are used.

See Section 2.1.2 [Options], page 16, for the '`--scrolled-graph-editor`' and '`--panned-graph-editor`' options.

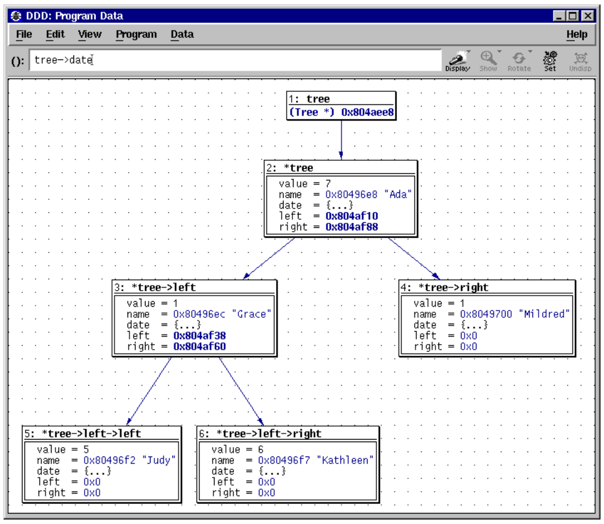
7.3.6.3 Aligning Displays

You can align all displays on the nearest grid position by selecting '`Data` ⇒ `Align on Grid`'. This is useful for keeping edges strictly horizontal or vertical.

You can enforce alignment by selecting '`Edit` ⇒ `Preferences` ⇒ `Data` ⇒ `Auto-align Displays on Nearest Grid Point`'. If this feature is enabled, displays can be moved on grid positions only.

7.3.6.4 Automatic Layout

You can layout the entire graph as a tree by selecting '`Data` ⇒ `Layout Graph`'. The layout direction is determined from the display placement (see Section 7.3.1.8 [Placement], page 113) and from the last rotation (see Section 7.3.6.5 [Rotating the Graph], page 131).



A Layouted Graph (with Compact Layout)

Layouting the graph may introduce *edge hints*; that is, edges are no more straight lines, but lead to an edge hint and from there to their destination. Edge hints can be moved around like arbitrary displays.

To enable a more compact layout, you can set the 'Edit ⇒ Preferences ⇒ Data ⇒ Compact Layout' option. This realizes an alternate layout algorithm, where successors are placed next to their parents. This algorithm is suitable for homogeneous data structures only.

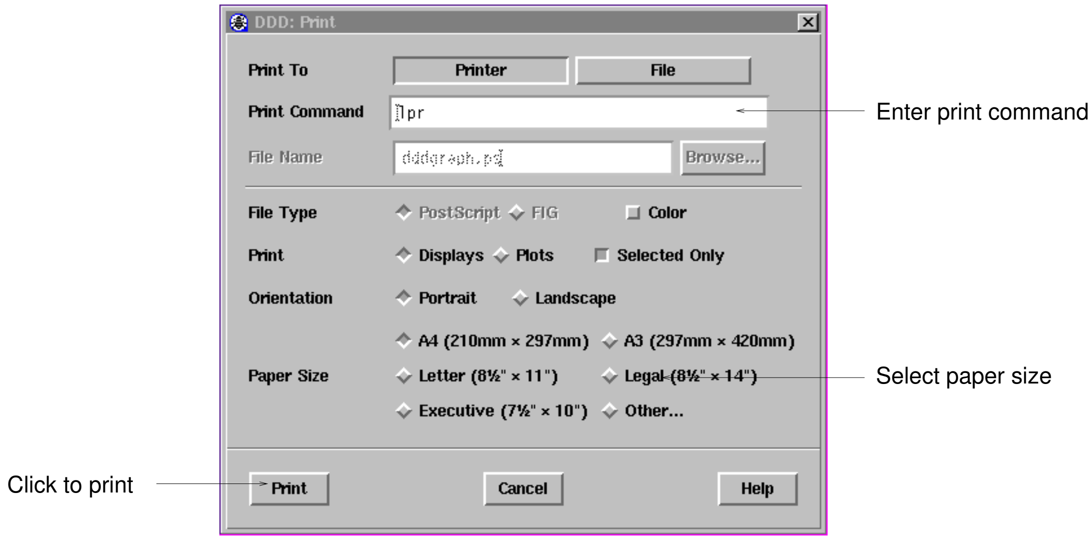
You can enforce layout by setting 'Edit ⇒ Preferences ⇒ Data ⇒ Automatic Layout'. If automatic layout is enabled, the graph is layouted after each change.

7.3.6.5 Rotating the Graph

You can rotate the entire graph clockwise by 90 degrees by selecting 'Data ⇒ Rotate Graph'. You may need to layout the graph after rotating it; See Section 7.3.6.4 [Automatic Layout], page 130, for details.

7.3.7 Printing the Graph

DDD allows for printing the graph picture on PostScript printers or into files. This is useful for documenting program states.



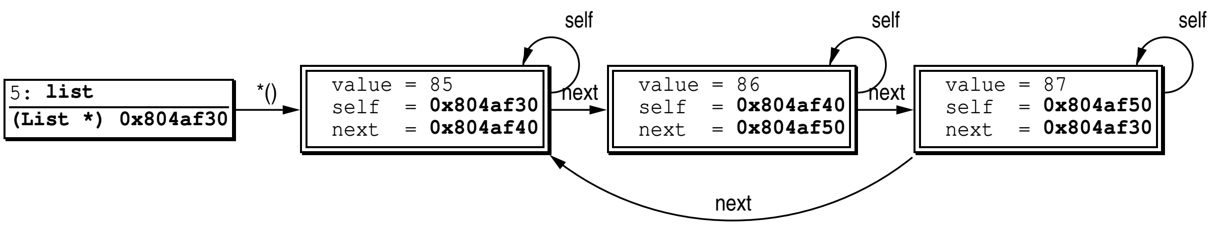
Printing displays

To print the graph on a PostScript printer, select '`File` ⇒ `Print Graph`'. Enter the printing command in the '`Print Command`' field. Click on the '`OK`' or the '`Apply`' button to start printing.

As an alternative, you may also print the graph in a file. Click on the '`File`' button and enter the file name in the '`File Name`' field. Click on the '`Print`' button to create the file.

When the graph is printed in a file, two formats are available:

- '`PostScript`'—suitable for enclosing the graph in another document;
- '`FIG`'—suitable for post-processing, using the `xfig` graphic editor, or for conversion into other formats (among others, IBMGL, TEX, PIC), using the `transfig` or `fig2dev` programs.



Output of the 'Print Graph' Command

Please note the following *caveats* related to printing graphs:

- If any displays were selected when invoking the '`Print`' dialog, the option '`Selected Only`' is set. This makes DDD print only the selected displays.

- The '`Color`', '`Orientation`', and '`Paper Size`' options are meaningful for PostScript only.

These settings are tied to the following resources:

printCommand (class PrintCommand) Resource
The command to print a PostScript file. Usually '`lp`' or '`lpr`'.

paperSize (class PaperSize) Resource
The paper size used for printing, in format '*width* `x` *height*'. The default is ISO A4 format, or '`210mm x 297mm`'.

7.4 Plotting Values

If you have huge amounts of numerical data to examine, a picture often says more than a thousand numbers. Therefore, DDD allows you to draw numerical values in nice 2-D and 3-D plots.

7.4.1 Plotting Arrays

Basically, DDD can plot two types of numerical values:

- One-dimensional arrays. These are drawn in a 2-D x/y space, where x denotes the array index, and y the element value.
- Two-dimensional arrays. These are drawn in a 3-D $x/y/z$ space, where x and y denote the array indexes, and z the element value.

To plot a fixed-size array, select its name by clicking *mouse button 1* on an occurrence. The array name is copied to the argument field. By clicking the '`Plot`' button, a new display is created in the data window, followed by a new top-level window containing the value plot.

To plot a dynamically sized array, you must use an array slice (see Section 7.3.2.1 [Array Slices], page 117). In the argument field, enter

```
array[first]@nelems
```

where *array* is the name of the array to display, *first* is the index of the first element, and *nelems* is the number of elements to display. Then, click on '`Plot`' to start the plot.

To plot a value, you can also enter a command at the debugger prompt:

```
graph plot expr
```

works like '`graph display` *expr*' (and takes the same arguments; see Section 7.3.1.1 [Creating Single Displays], page 107), but the value is additionally shown in the plot window.

Each time the value changes during program execution, the plot is updated to reflect the current values. The plot window remains active until you close it (via '`File` ⇒ `Close`') or until the associated display is deleted.

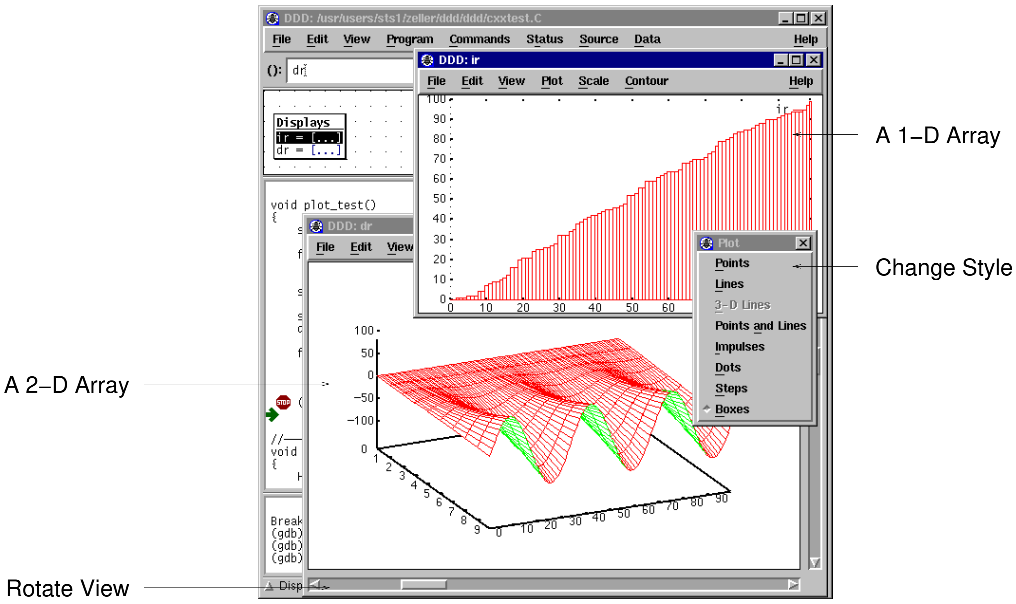
7.4.2 Changing the Plot Appearance

The actual drawing is not done by DDD itself. Instead, DDD relies on an external `gnuplot` program to create the drawing.

DDD adds a menu bar to the Gnuplot plot window that lets you influence the appearance of the plot:

- The '`View`' menu toggles optional parts of the plot, such as border lines or a background grid.
- The '`Plot`' menu changes the *plotting style*. The '`3-D Lines`' option is useful for plotting two-dimensional arrays.
- The '`Scale`' menu allows you to enable logarithmic scaling and to enable or disable the scale tics.
- The '`Contour`' menu adds contour lines to 3-D plots.

In a 3-D plot, you can use the scroll bars to change your view position. The horizontal scroll bar rotates the plot around the *z* axis, that is, to the left and right. The vertical scroll bar rotates the plot around the *y* axis, that is, up and down.



Plotting 1–D and 2–D Arrays

You can also resize the plot window as desired.

7.4.3 Plotting Scalars and Composites

Besides plotting arrays, DDD also allows you to plot scalars (simple numerical values). This works just like plotting arrays—you select the numerical variable, click on '`Plot`', and here comes the plot. However, plotting a scalar is not very exciting. A plot that contains nothing but a scalar simply draws the scalar's value as a *y* constant—that is, a horizontal line.

So why care about scalars at all? DDD allows you to combine *multiple values into one plot*. The basic idea is: if you want to plot something that is neither an array nor a scalar, DDD takes all numerical sub-values it can find and plots them all together in one window. For instance, you can plot all local variables by selecting '`Data ⇒ Display Local Variables`', followed by '`Plot`'. This will create a plot containing all numerical values as found in the current local variables. Likewise, you can plot all numeric members contained in a structure by selecting it, followed by '`Plot`'.

If you want more control about what to include in a plot and what not, you can use *display clusters* (see Section 7.3.1.9 [Clustering], page 114). A common scenario is to plot a one-dimensional array together with the current index position. This is done in three steps:

1. Display the array and the index, using '`Display`'.
2. Cluster both displays: select them and choose '`Undisp ⇒ Cluster ()`'.
3. Plot the cluster by pressing '`Plot`'.

Scalars that are displayed together with arrays can be displayed either as vertical lines or horizontal lines. By default, scalars are plotted as horizontal lines. However, if a scalar is a valid index for an array that was previously plotted, it is shown as a vertical line. You can change this initial orientation by selecting the scalar display, followed by '`Rotate`'.

7.4.4 Plotting Display Histories

At each program stop, DDD records the values of all displayed variables, such that you can "undo" program execution (see Section 6.8 [Undoing Program Execution], page 100). These *display histories* can be plotted, too. The menu item '`Plot ⇒ Plot history of ()`' creates a plot that shows all previously recorded values of the selected display.

7.4.5 Printing Plots

If you want to print the plot, select '`File ⇒ Print Plot`'. This pops up the DDD printing dialog, set up for printing plots. Just as when printing graphs, you have the choice between printing to a printer or a file and setting up appropriate options.

The actual printing is also performed by Gnuplot, using the appropriate driver. Please note the following *caveats* related to printing:

- Creating '`FIG`' files requires an appropriate driver built into Gnuplot. Your Gnuplot program may not contain such a driver. In this case, you will have to recompile Gnuplot, including the line '`#define FIG`' in the Gnuplot '`term.h`' file.
- The '`Portrait`' option generates an EPS file useful for inclusion in other documents. The '`Landscape`' option makes DDD print the plot in the size specified in the '`Paper Size`' option; this is useful for printing on a printer. In '`Portrait`' mode, the '`Paper Size`' option is ignored.
- The Gnuplot device drivers for PostScript and X11 each have their own set of colors, such that the printed colors may differ from the displayed colors.
- The '`Selected Only`' option is set by default, such that only the currently selected plot is printed. (If you select multiple plots to be printed, the respective outputs will all be concatenated, which may not be what you desire.)

7.4.6 Entering Plotting Commands

Via '`File` ⇒ `Command`', you can enter Gnuplot commands directly. Each command entered at the '`gnuplot>`' prompt is passed to Gnuplot, followed by a Gnuplot '`replot`' command to update the view. This is useful for advanced Gnuplot tasks.

Here's a simple example. The Gnuplot command

```
set xrange [xmin:xmax]
```

sets the horizontal range that will be displayed to *xmin*...*xmax*. To plot only the elements 10 to 20, enter:

```
gnuplot> set xrange [10:20]
gnuplot> _
```

After each command entered, DDD adds a `replot` command, such that the plot is updated automatically.

Here's a more complex example. The following sequence of Gnuplot commands saves the plot in TEX format:

```
gnuplot> set output "plot.tex" # Set the output filename
gnuplot> set term latex        # Set the output format
gnuplot> set term x11          # Show original picture again
gnuplot> _
```

Due to the implicit `replot` command, the output is automatically written to '`plot.tex`' after the `set term latex` command.

The dialog keeps track of the commands entered; use the arrow keys to restore previous commands. Gnuplot error messages (if any) are also shown in the history area.

The interaction between DDD and Gnuplot is logged in the file '`~/.ddd/log`' (see Section B.5.1 [Logging], page 170). The DDD '`--trace`' option logs this interaction on standard output.

7.4.7 Exporting Plot Data

If you want some external program to process the plot data (a stand-alone Gnuplot program or the `xmgr` program, for instance), you can save the plot data in a file, using '`File` ⇒ `Save Data As`'. This pops up a dialog that lets you choose a data file to save the plotted data in.

The generated file starts with a few comment lines. The actual data follows in X/Y or X/Y/Z format. It is the same file as processed by Gnuplot.

7.4.8 Animating Plots

If you want to see how your data evolves in time, you can set a breakpoint whose command sequence ends in a `cont` command (see Section 5.1.8 [Breakpoint Commands], page 85. Each time this "continue" breakpoint is reached, the program stops and DDD updates the displayed values, including the plots. Then, DDD executes the breakpoint command sequence, resuming execution.

This way, you can set a "continue" breakpoint at some decisive point within an array-processing algorithm and have DDD display the progress graphically. When your program has stopped for good, you can use '`Undo`' and '`Redo`' to redisplay and examine previous program states. See Section 6.8 [Undoing Program Execution], page 100, for details.

7.4.9 Customizing Plots

You can customize the Gnuplot program to invoke, as well as a number of basic settings.

7.4.9.1 Gnuplot Invocation

Using '`Edit` ⇒ `Preferences` ⇒ `Helpers` ⇒ `Plot`', you can choose the Gnuplot program to invoke. This is tied to the following resource:

plotCommand (class PlotCommand) Resource

The name of a Gnuplot executable. Default is '`gnuplot`', followed by some options to set up colors and the initial geometry.

Using '`Edit` ⇒ `Preferences` ⇒ `Helpers` ⇒ `Plot Window`', you can choose whether to use the Gnuplot plot window ('`External`') or to use the plot window supplied by DDD ('`builtin`'). This is tied to the following resource:

plotTermType (class PlotTermType) Resource

The Gnuplot terminal type. Can have one of two values:

- If this is '`x11`', DDD "swallows" the *external* Gnuplot output window into its own user interface. Some window managers, notably `mwm`, have trouble with swallowing techniques.
- Setting this resource to '`xlib`' (default) makes DDD provide a *builtin plot window* instead. In this mode, plots work well with any window manager, but are less customizable (Gnuplot resources are not understood).

You can further control interaction with the external plot window:

plotWindowClass (class PlotWindowClass) Resource

The class of the Gnuplot output window. When invoking Gnuplot, DDD waits for a window with this class and incorporates it into its own user interface (unless '`plotTermType`' is '`xlib`'; see above). Default is '`Gnuplot`'.

plotWindowDelay (class WindowDelay) Resource

The time (in ms) to wait for the creation of the Gnuplot window. Before this delay, DDD looks at each newly created window to see whether this is the plot window to swallow. This is cheap, but unfortunately, some window managers do not pass the creation event to DDD. If this delay has passed, and DDD has not found the plot window, DDD searches *all* existing windows, which is pretty expensive. Default time is `2000`.

7.4.9.2 Gnuplot Settings

To change Gnuplot settings, use these resources:

plotInitCommands (class PlotInitCommands) Resource

The initial Gnuplot commands issued by DDD. Default is:

```
set parametric
set urange [0:1]
set vrange [0:1]
set trange [0:1]
```

The '`parametric`' setting is required to make Gnuplot understand the data files as generated DDD. The range commands are used to plot scalars.

See the Gnuplot documentation for additional commands.

plot2dSettings (class PlotSettings) Resource

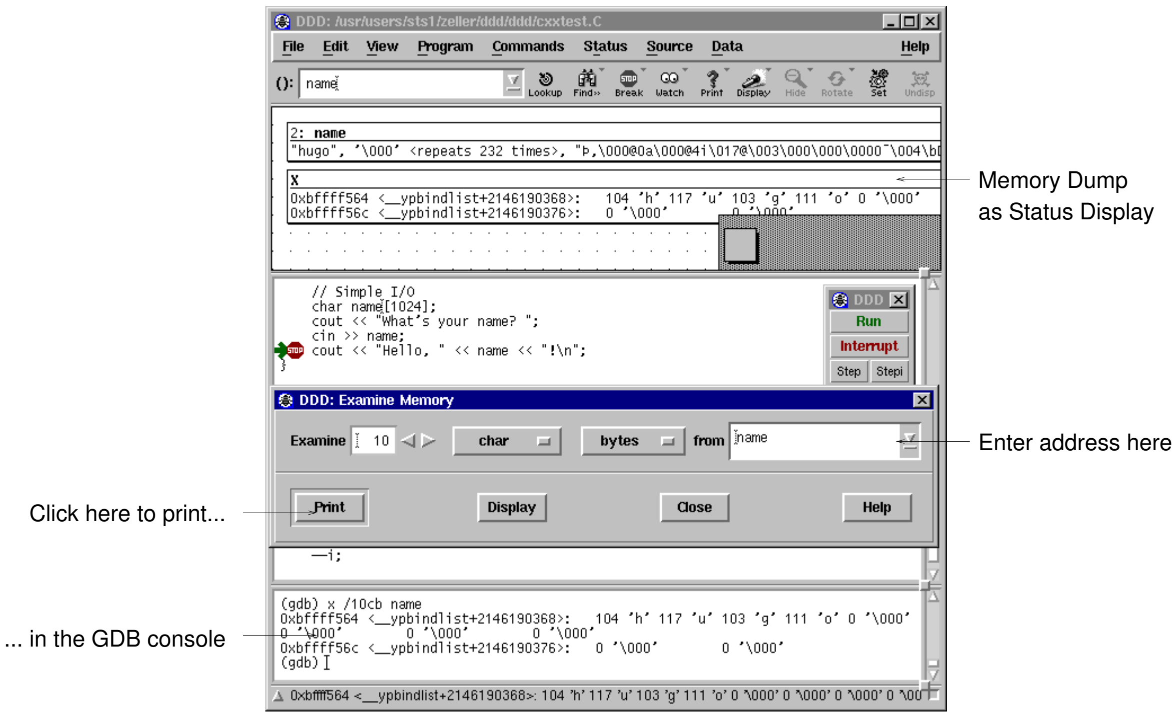
Additional initial settings for 2-D plots. Default is '`set noborder`'. Feel free to customize these settings as desired.

plot3dSettings (class PlotSettings) Resource

Additional initial settings for 3-D plots. Default is '`set border`'. Feel free to customize these settings as desired.

7.5 Examining Memory

Using GDB or DBX, you can examine memory in any of several formats, independently of your program's data types. The item '`Data` ⇒ `Memory`' pops up a panel where you can choose the format to be shown.



Examining Memory

In the panel, you can enter

- a *repeat count*, a decimal integer that specifies how much memory (counting by units) to display
- a *display format*—one of

  ‘octal’ Print as integer in octal

  ‘hex’ Regard the bits of the value as an integer, and print the integer in hexadecimal.

  ‘decimal’
  Print as integer in signed decimal.

  ‘unsigned’
  Print as integer in unsigned decimal.

  ‘binary’ Print as integer in binary.

  ‘float’ Regard the bits of the value as a floating point number and print using typical floating point syntax.

  ‘address’
  Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol.

  ‘instruction’
  Print as machine instructions. The unit size is ignored for this display format.

  ‘char’ Regard as an integer and print it as a character constant.

  ‘string’ Print as null-terminated string. The unit size is ignored for this display format.

- a *unit size*—one of

  ‘bytes’ Bytes.

  ‘halfwords’
  Halfwords (two bytes).

  ‘words’ Words (four bytes).

  ‘giants’ Giant words (eight bytes).

- an *address*—the starting display address. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory.

There are two ways to examine the values:

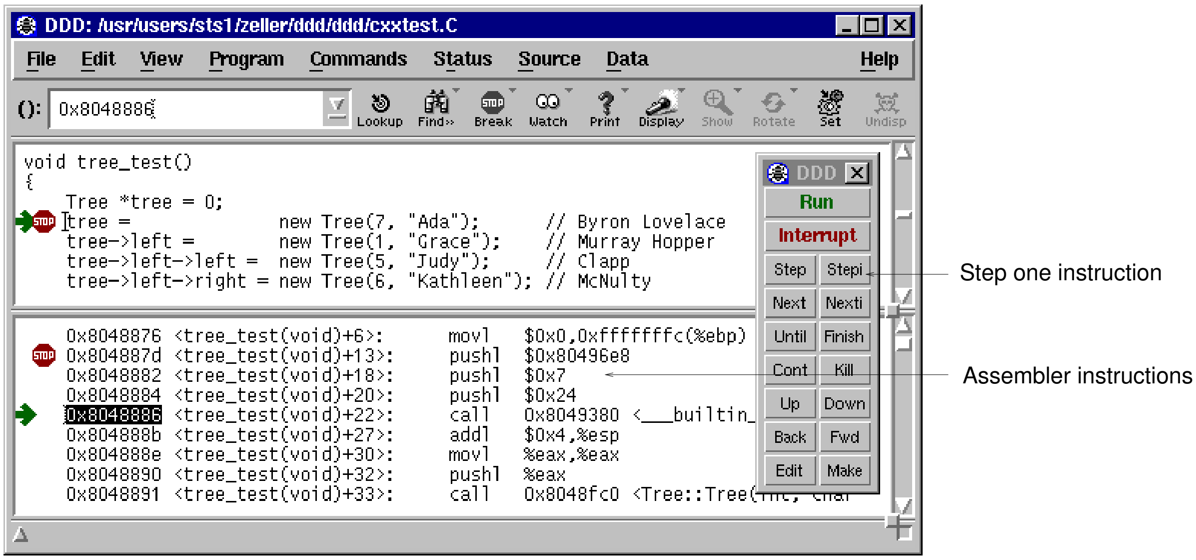
- You can dump the memory in the debugger console (using ‘Print’). If you repeat the resulting ‘x’ command by pressing ⟨Return⟩ in the debugger console (see Section 10.1.2 [Command History], page 148), the following area of memory is shown.
- You can also display the memory dump in the data window (using ‘Display’). If you choose to display the values, the values will be updated automatically each time the program stop.

8 Machine-Level Debugging

Sometimes, it is desirable to examine a program not only at the source level, but also at the machine level. DDD provides special machine code and register windows for this task.

8.1 Examining Machine Code

To enable machine-level support, select '`Source` ⇒ `Display Machine Code`'. With machine code enabled, an additional *machine code window* shows up, displaying the machine code of the current function.[1] By moving the sash at the right of the separating line between source and machine code, you can resize the source and machine code windows.



Showing Machine Code

The machine code window works very much like the source window. You can set, clear, and change breakpoints by selecting the address and pressing a '`Break`' or '`Clear`' button; the usual popup menus are also available. Breakpoints and the current execution position are displayed simultaneously in both source and machine code.

The '`Lookup`' button can be used to look up the machine code for a specific function—or the function for a specific address. Just click on the location in one window and press '`Lookup`' to see the corresponding code in the other window.

If source code is not available, only the machine code window is updated.

You can customize various aspects of the disassembling window. See Section 8.4 [Customizing Machine Code], page 143, for details.

[1] The machine code window is available with GDB and some DBX variants only.

8.2 Machine Code Execution

All execution facilities available in the source code window are available in the machine code window as well. Two special facilities are convenient for machine-level debugging:

To execute just one machine instruction, click on the '`Stepi`' button or select '`Program` ⇒ `Step Instruction`'.

To continue to the next instruction in the current function, click on the '`Nexti`' button select '`Program` ⇒ `Next Instruction`'.. This is similar to '`Stepi`', but any subroutine calls are executed without stopping.

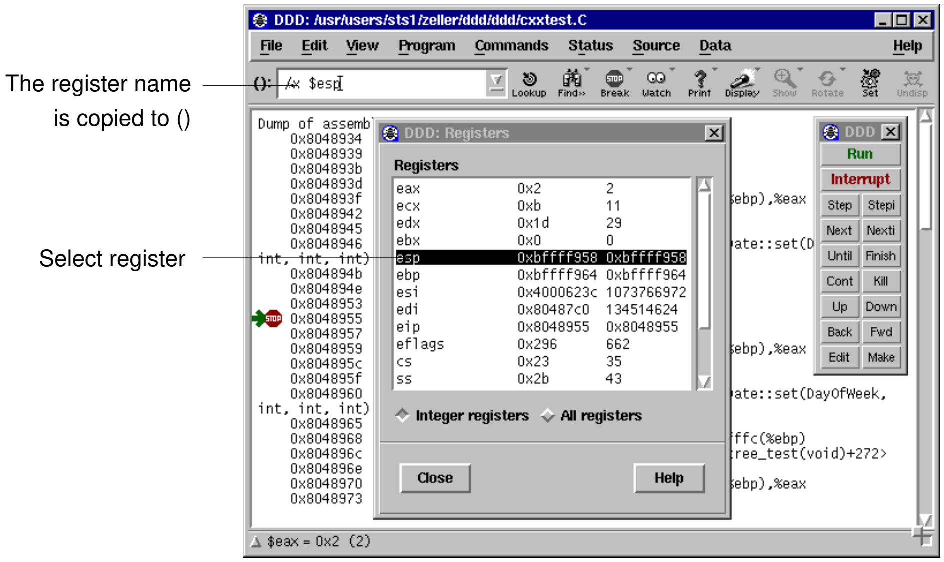
Using GDB, it is often useful to do

```
graph display /i $pc
```

when stepping by machine instructions. This makes DDD automatically display the next instruction to be executed, each time your program stops.

8.3 Examining Registers

DDD provides a *register window* showing the machine register values after each program stop. To enable the register window, select '`Status` ⇒ `Registers`'.[2]



Displaying Register Values

By selecting one of the registers, its name is copied to the argument field. You can use it as value for '`Display`', for instance, to have its value displayed in the data window.

[2] The machine code window is available with GDB and some DBX variants only.

8.4 Customizing Machine Code

Enabling machine code via '`Source` ⇒ `Display Machine Code`' (see Section 8.1 [Machine Code], page 141) toggles the following resource:

disassemble (class Disassemble) Resource

If this is '`on`', the source code is automatically disassembled. The default is '`off`'. See Section 2.1.2 [Options], page 16, for the '`--disassemble`' and '`--no-disassemble`' options.

You can keep disassembled code in memory, using '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Cache Machine Code`':

cacheMachineCode (class CacheMachineCode) Resource

Whether to cache disassembled machine code ('`on`', default) or not ('`off`'). Caching machine code requires more memory, but makes DDD run faster.

You can control the indentation of machine code, using '`Edit` ⇒ `Preferences` ⇒ `Source` ⇒ `Machine Code Indentation`':

indentCode (class Indent) Resource

The number of columns to indent the machine code, such that there is enough place to display breakpoint locations. Default: `4`.

The '`maxDisassemble`' resource controls how much is to be disassembled. If '`maxDisassemble`' is set to 256 (default) and the current function is larger than 256 bytes, DDD only disassembles the first 256 bytes below the current location. You can set the '`maxDisassemble`' resource to a larger value if you prefer to have a larger machine code view.

maxDisassemble (class MaxDisassemble) Resource

Maximum number of bytes to disassemble (default: `256`). If this is zero, the entire current function is disassembled.

9 Changing the Program

DDD offers some basic facilities to edit and recompile the source code, as well as patching executables and core files.

9.1 Editing Source Code

In DDD itself, you cannot change the source file currently displayed. Instead, DDD allows you to invoke a *text editor*. To invoke a text editor for the current source file, select the '`Edit`' button or '`Source` ⇒ `Edit Source`'.

By default, DDD tries a number of common editors. You can customize DDD to use your favorite editor; See Section 9.1.1 [Customizing Editing], page 145, for details.

After the editor has exited, the source code shown is automatically updated.

If you have DDD and an editor running in parallel, you can also update the source code manually via '`Source` ⇒ `Reload Source`'. This reloads the source code shown from the source file. Since DDD automatically reloads the source code if the debugged program has been recompiled, this should seldom be necessary.

9.1.1 Customizing Editing

You can customize the editor to be used via '`Edit` ⇒ `Preferences` ⇒ `Helpers` ⇒ `Edit Sources`'. This is tied to the following resource:

editCommand (class EditCommand) Resource

A command string to invoke an editor on the specific file. '`@LINE@`' is replaced by the current line number, '`@FILE@`' by the file name. The default is to invoke `$XEDITOR` first, then `$EDITOR`, then `vi`:

```
Ddd*editCommand: \
${XEDITOR-false} +@LINE@ @FILE@ || \
xterm -e ${EDITOR-vi} +@LINE@ @FILE@
```

This '`~/.ddd/init`' setting invokes an editing session for an XEmacs editor running `gnuserv`:

```
Ddd*editCommand: gnuclient +@LINE@ @FILE@
```

This '`~/.ddd/init`' setting invokes an editing session for an Emacs editor running `emacsserver`:

```
Ddd*editCommand: emacsclient +@LINE@ @FILE@
```

9.1.2 In-Place Editing

This resource is experimental:

sourceEditing (class SourceEditing) Resource

If this is '`on`', the displayed source code becomes editable. This is an experimental feature; Default is '`off`'.

9.2 Recompiling

To recompile the source code using `make`, you can select '`File` ⇒ `Make`'. This pops up a dialog where you can enter a *Make Target*—typically the name of the executable. Clicking on the '`Make`' button invokes the `make` program with the given target.

The '`Make`' button on the command tool re-invokes `make` with the most recently given arguments.

9.3 Patching

Using GDB, you can open your program's executable code (and the core file) for both reading and writing. This allows alterations to machine code, such that you can intentionally patch your program's binary. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

Note that depending on your operating system, special preparation steps, such as setting permissions, may be needed before you can change executable files.

To patch the binary, enable '`Edit` ⇒ `GDB Settings` ⇒ `Writing into executable and core files`'. This makes GDB open executable and core files for both reading and writing. If you have already loaded a file, you must load it again (using '`Edit` ⇒ `Open File`' or '`Edit` ⇒ `Open Core`'), for your new setting to take effect.

Be sure to turn off '`Writing into executable and core files`' as soon as possible, to prevent accidental alterations to machine code.

10 The Command-Line Interface

All the buttons you click within DDD get eventually translated into some debugger command, shown in the debugger console. You can also type in and edit these commands directly.

10.1 Entering Commands

In the *debugger console*, you can interact with the command interface of the inferior debugger. Enter commands at the *debugger prompt*—that is, '`(gdb)`' for GDB, '`(dbx)`' for DBX, '`(ladebug)`' for Ladebug, '`>`' for XDB, '`>`' and '*thread* `[`*depth*`]`' for JDB, or '`(Pydb)`' for PYDB, or '`DB<>`' for Perl, or '`bashdb<>`' for Bash. You can use arbitrary debugger commands; use the ⟨Return⟩ key to enter them.

10.1.1 Command Completion

When using GDB or Perl, you can use the ⟨TAB⟩ key for *completing* commands and arguments. This works in the debugger console as well as in all other text windows.

GDB can fill in the rest of a word in a command for you, if there is only one possibility; it can also show you what the valid possibilities are for the next word in a command, at any time. This works for GDB commands, GDB subcommands, and the names of symbols in your program.

Press the ⟨TAB⟩ key whenever you want GDB to fill out the rest of a word. If there is only one possibility, GDB fills in the word, and waits for you to finish the command (or press ⟨RET⟩ to enter it). For example, if you type

```
(gdb) info bre_⟨TAB⟩
```

GDB fills in the rest of the word '`breakpoints`', since that is the only `info` subcommand beginning with '`bre`':

```
(gdb) info breakpoints
```

You can either press ⟨RET⟩ at this point, to run the `info breakpoints` command, or backspace and enter something else, if '`breakpoints`' does not look like the command you expected. (If you were sure you wanted `info breakpoints` in the first place, you might as well just type ⟨RET⟩ immediately after '`info bre`', to exploit command abbreviations rather than command completion).

If there is more than one possibility for the next word when you press ⟨TAB⟩, DDD sounds a bell. You can either supply more characters and try again, or just press ⟨TAB⟩ a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with '`make_`', but when you type ***b make_***⟨TAB⟩, DDD just sounds the bell. Typing ⟨TAB⟩ again displays all the function names in your program that begin with those characters. If you type ⟨TAB⟩ again, you cycle through the list of completions, for example:

```
(gdb) b make_ ⟨TAB⟩
```

DDD sounds bell; press ⟨TAB⟩ again, to see:

```
make_a_section_from_file      make_environ
make_abs_section              make_function_type
make_blockvector              make_pointer_type
make_cleanup                  make_reference_type
make_command                  make_symbol_completion_list
(gdb) b make_ ⟨TAB⟩
```

DDD presents one expansion after the other:

```
(gdb) b make_a_section_from_file ⟨TAB⟩
(gdb) b make_abs_section ⟨TAB⟩
(gdb) b make_blockvector ⟨TAB⟩
```

After displaying the available possibilities, GDB copies your partial input (‘`b make_`’ in the example) so you can finish the command—by pressing ⟨TAB⟩ again, or by entering the remainder manually.

Sometimes the string you need, while logically a “word”, may contain parentheses or other characters that GDB normally excludes from its notion of a word. To permit word completion to work in this situation, you may enclose words in `'` (single quote marks) in GDB commands.

The most likely situation where you might need this is in typing the name of a C++ function. This is because C++ allows function overloading (multiple definitions of the same function, distinguished by argument type). For example, when you want to set a breakpoint you may need to distinguish whether you mean the version of `name` that takes an `int` parameter, `name(int)`, or the version that takes a `float` parameter, `name(float)`. To use the word-completion facilities in this situation, type a single quote `'` at the beginning of the function name. This alerts GDB that it may need to consider more information than usual when you press ⟨TAB⟩ to request word completion:

```
(gdb) b 'bubble(_⟨TAB⟩
bubble(double,double)    bubble(int,int)
(gdb) b 'bubble(_
```

In some cases, DDD can tell that completing a name requires using quotes. When this happens, DDD inserts the quote for you (while completing as much as it can) if you do not type the quote in the first place:

```
(gdb) b bub_⟨TAB⟩
```

DDD alters your input line to the following, and rings a bell:

```
(gdb) b 'bubble(_
```

In general, DDD can tell that a quote is needed (and inserts it) if you have not yet started typing the argument list when you ask for completion on an overloaded symbol.

If you prefer to use the ⟨TAB⟩ key for switching between items, unset ‘`Edit` ⇒ `Preferences` ⇒ `General` ⇒ `TAB Key completes in All Windows`’. This is useful if you have pointer-driven keyboard focus (see below) and no special usage for the ⟨TAB⟩ key. If the option is set, the ⟨TAB⟩ key completes in the debugger console only.

This option is tied to the following resource:

globalTabCompletion (class GlobalTabCompletion) Resource

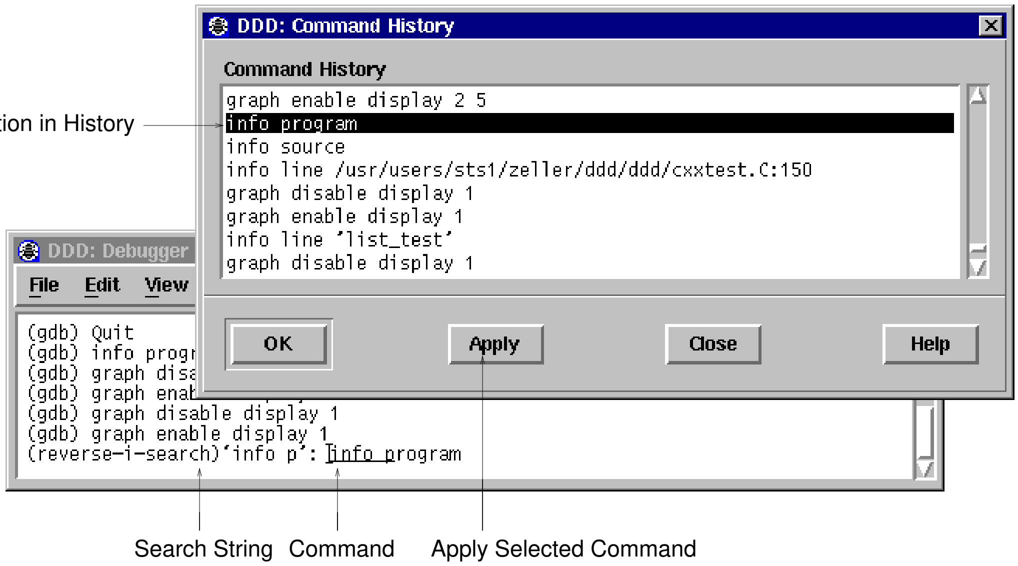
If this is ‘`on`’ (default), the ⟨TAB⟩ key completes arguments in all windows. If this is ‘`off`’, the ⟨TAB⟩ key completes arguments in the debugger console only.

10.1.2 Command History

You can *repeat* previous and next commands by pressing the ⟨Up⟩ and ⟨Down⟩ arrow keys, respectively. This presents you previous and later commands on the command line; use ⟨Return⟩ to apply the current command.

If you enter an empty line (just use ⟨Return⟩ at the debugger prompt), the last command is repeated as well.

'Commands ⇒ Command History' shows the command history.



Searching with Ctrl+B in the Command History

You can *search* for previous commands by pressing ⟨Ctrl+B⟩. This invokes *incremental search mode,* where you can enter a string to be searched in previous commands. Press ⟨Ctrl+B⟩ again to repeat the search, or ⟨Ctrl+F⟩ to search in the reverse direction. To return to normal mode, press ⟨ESC⟩, or use any cursor command.

The command history is automatically saved when exiting DDD. You can turn off this feature by setting the following resource to 'off':

saveHistoryOnExit (class SaveOnExit) Resource

If 'on' (default), the command history is automatically saved when DDD exits.

10.1.3 Typing in the Source Window

As a special convenience, anything you type into the source window is automatically forwarded to the debugger console. Thus, you don't have to change the keyboard focus explicitly in order to enter commands.

You can change this behaviour using the following resource:

consoleHasFocus (class ConsoleHasFocus) Resource

If 'on' (default), all keyboard events in the source window are automatically forwarded to the debugger console. If 'off', keyboard events are not forwarded. If 'auto', keyboard events forwarded only if the debugger console is open.

10.2 Entering Commands at the TTY

Rather than entering commands at the debugger console, you may prefer to enter commands at the terminal window DDD was invoked from.

When DDD is invoked using the '`--tty`' option, it enables its TTY *interface*, taking additional debugger commands from standard input and forwarding debugger output to standard output, just as if the inferior debugger had been invoked directly. All remaining DDD functionality stays unchanged.

By default, the debugger console remains closed if DDD is invoked using the '`--tty`' option. Use '`View` ⇒ `Debugger Console`' to open it.

DDD can be configured to use the '`readline`' library for reading in commands from standard input. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are GNU Emacs-style or *vi*-style inline editing of commands, `csh`-like history substitution, and a storage and recall of command history across debugging sessions. See section "Command Line Editing" in *Debugging with* GDB, for details on command-line editing via the TTY interface.

10.3 Integrating DDD

You can run DDD as an inferior debugger in other debugger front-ends, combining their special abilities with those of DDD.

To have DDD run as an inferior debugger in other front-ends, the general idea is to set up your debugger front-end such that '`ddd --tty`' is invoked instead of the inferior debugger. When DDD is invoked using the '`--tty`' option, it enables its TTY *interface*, taking additional debugger commands from standard input and forwarding debugger output to standard output, just as if the inferior debugger had been invoked directly. All remaining DDD functionality stays unchanged.

In case your debugger front-end uses the GDB '`-fullname`' option to have GDB report source code positions, the '`--tty`' option is not required. DDD recognizes the '`-fullname`' option, finds that it has been invoked from a debugger front-end and automatically enables the TTY interface.

If DDD is invoked with the '`-fullname`' option, the debugger console and the source window are initially disabled, as their facilities are supposed to be provided by the integrating front-end. In case of need, you can use the '`View`' menu to re-enable these windows.

10.3.1 Using DDD with Emacs

To integrate DDD with Emacs, use ***M-x gdb*** or ***M-x dbx*** in Emacs to start a debugging session. At the prompt, enter `ddd --tty` (followed by '`--dbx`' or '`--gdb`', if required), and the name of the program to be debugged. Proceed as usual.

10.3.2 Using DDD with XEmacs

To integrate DDD with XEmacs, set the variable `gdb-command-name` to '`"ddd"`', by inserting the following line in your '`~/.emacs`' file:

```
(setq gdb-command-name "ddd")
```

You can also evaluate this expression by pressing ⟨ESC⟩ ⟨:⟩ and entering it directly (⟨ESC⟩ ⟨ESC⟩ for XEmacs 19.13 and earlier).

To start a DDD debugging session in XEmacs, use '`M-x gdb`' or '`M-x gdbsrc`'. Proceed as usual.

10.3.3 Using DDD with XXGDB

To integrate DDD with XXGDB, invoke `xxgdb` as

```
xxgdb -db_name ddd -db_prompt '(gdb) '
```

10.4 Defining Buttons

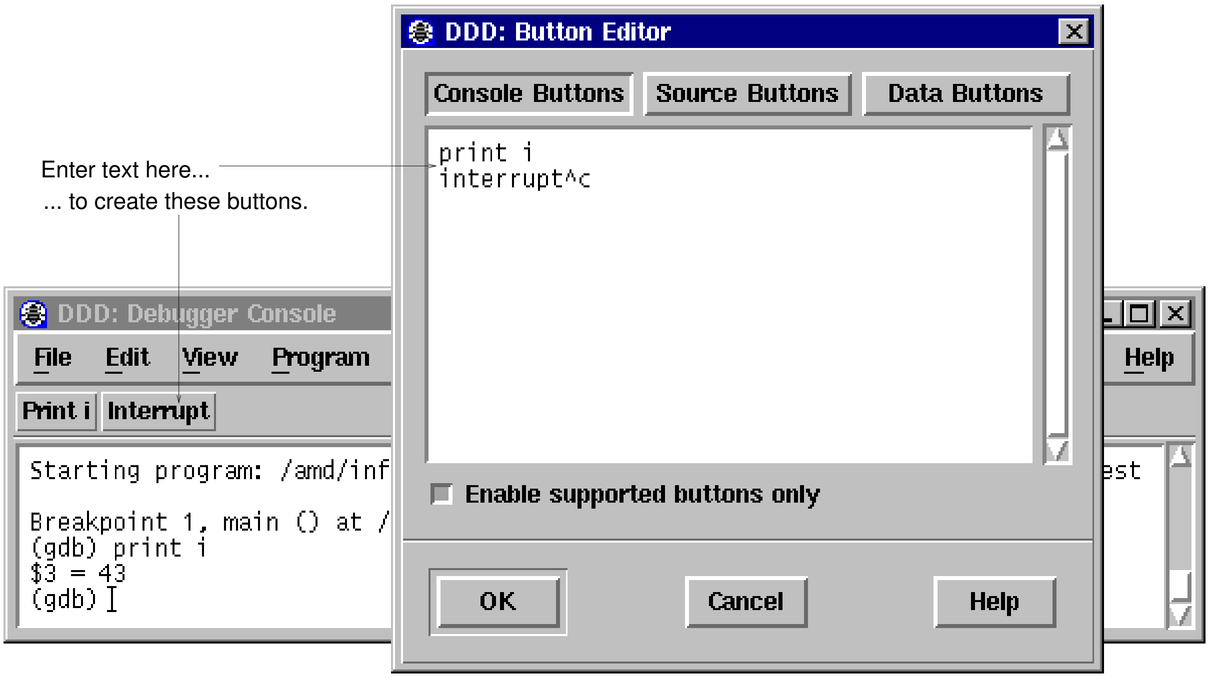
To facilitate interaction, you can add own command buttons to DDD. These buttons can be added below the debugger console ('`Console Buttons`'), the source window ('`Source Buttons`'), or the data window ('`Data Buttons`').

To define individual buttons, use the *Button Editor*, invoked via '`Commands` ⇒ `Edit Buttons`'. The button editor displays a text, where each line contains the command for exactly one button. Clicking on '`OK`' creates the appropriate buttons from the text. If the text is empty (the default), no button is created.

As a simple example, assume you want to create a '`print i`' button. Invoke '`Commands` ⇒ `Edit Buttons`' and enter a line saying '`print i`' in the button editor. Then click on '`OK`'. A button named '`Print i`' will now appear below the debugger console—try it! To remove the button, reopen the button editor, clear the '`print i`' line and press '`OK`' again.

If a button command contains '`()`', the string '`()`' will automatically be replaced by the contents of the argument field. For instance, a button named '`return ()`' will execute the GDB '`return`' command with the current content of the argument field as argument.

By default, DDD disables buttons whose commands are not supported by the inferior debugger. To enable such buttons, unset the '`Enable supported buttons only`' toggle in the button editor.



Defining individual buttons

DDD also allows you to specify control sequences and special labels for user-defined buttons. See Section 10.4.1 [Customizing Buttons], page 152, for details.

10.4.1 Customizing Buttons

DDD allows defining additional command buttons; See Section 10.4 [Defining Buttons], page 151, for doing this interactively. This section describes the resources that control user-defined buttons.

consoleButtons (class Buttons) Resource

A newline-separated list of buttons to be added under the debugger console. Each button issues the command given by its name.

The following characters have special meanings:

- Commands ending with `...` insert their name, followed by a space, in the debugger console.
- Commands ending with a control character (that is, '`^`' followed by a letter or '`?`') insert the given control character.
- The string '`()`' is replaced by the current contents of the argument field '`()`'.
- The string specified in the '`labelDelimiter`' resource (usually '`//`') separates the command name from the button label. If no button label is specified, the capitalized command will be used as button label.

The following button names are reserved:

'`Apply`' Send the given command to the debugger.

'Back' Lookup previously selected source position.

'Clear' Clear current command

'Complete'
: Complete current command.

'Edit' Edit current source file.

'Forward'
: Lookup next selected source position.

'Make' Invoke the 'make' program, using the most recently given arguments.

'Next' Show next command

'No' Answer current debugger prompt with 'no'. This button is visible only if the debugger asks a yes/no question.

'Prev' Show previous command

'Reload' Reload source file.

'Yes' Answer current debugger prompt with 'yes'. This button is visible only if the debugger asks a yes/no question.

The default resource value is empty—no console buttons are created.

Here are some examples to insert into your '~/.ddd/init' file. These are the settings of DDD 1.x:

```
Ddd*consoleButtons: Yes\nNo\nbreak^C
```

This setting creates some more buttons:

```
Ddd*consoleButtons: \
Yes\nNo\nrun\nClear\nPrev\nNext\nApply\nbreak^C
```

See also the 'dataButtons', 'sourceButtons' and 'toolButtons' resources.

dataButtons (class Buttons) Resource

A newline-separated list of buttons to be added under the data display. Each button issues the command given by its name. See the 'consoleButtons' resource, above, for details on button syntax.

The default resource value is empty—no source buttons are created.

sourceButtons (class Buttons) Resource

A newline-separated list of buttons to be added under the debugger console. Each button issues the command given by its name. See the 'consoleButtons' resource, above, for details on button syntax.

The default resource value is empty—no source buttons are created.

Here are some example to insert into your '~/.ddd/init' file. These are the settings of DDD 1.x:

```
Ddd*sourceButtons: \
  run\nstep\nnext\nstepi\nnexti\ncont\n\
  finish\nkill\nup\ndown\n\
  Back\nForward\nEdit\ninterrupt^C
```

This setting creates some buttons which are not found on the command tool:

```
Ddd*sourceButtons: \
print *()\ngraph display *()\nprint /x ()\n\
whatis ()\nptype ()\nwatch ()\nuntil\nshell
```

An even more professional setting uses customized button labels.

```
Ddd*sourceButtons: \
print *(()) // Print *()\n\
graph display *(()) // Display *()\n\
print /x ()\n\
whatis () // What is ()\n\
ptype ()\n\
watch ()\n\
until\n\
shell
```

See also the 'consoleButtons' and 'dataButtons' resources, above, and the 'toolButtons' resource, below.

toolButtons (class Buttons) Resource

A newline-separated list of buttons to be included in the command tool or the command tool bar (see Section 3.3.1.1 [Disabling the Command Tool], page 57). Each button issues the command given by its name. See Section 10.4 [Defining Buttons], page 151, for details on button syntax.

The default resource value is

```
Ddd*toolButtons: \
run\nbreak^C\nstep\nstepi\nnext\nnexti\n\
until\nfinish\ncont\n\kill\n\
up\ndown\nBack\nForward\nEdit\nMake
```

For each button, its location in the command tool must be specified using 'XmForm' constraint resources. See the 'Ddd' application defaults file for instructions.

If the 'toolButtons' resource value is empty, the command tool is not created.

The following resources set up button details:

labelDelimiter (class LabelDelimiter) Resource

The string used to separate labels from commands and shortcuts. Default is '//'.

verifyButtons (class VerifyButtons) Resource

If 'on' (default), verify for each button whether its command is actually supported by the inferior debugger. If the command is unknown, the button is disabled. If this resource is 'off', no checking is done: all commands are accepted "as is".

10.5 Defining Commands

Aside from breakpoint commands (see Section 5.1.8 [Breakpoint Commands], page 85), DDD also allows you to define user-defined commands. A *user-defined command* is a sequence of commands to which you assign a new name as a command. This new command can be entered at the debugger prompt or invoked via a button.

10.5.1 Defining Simple Commands using GDB

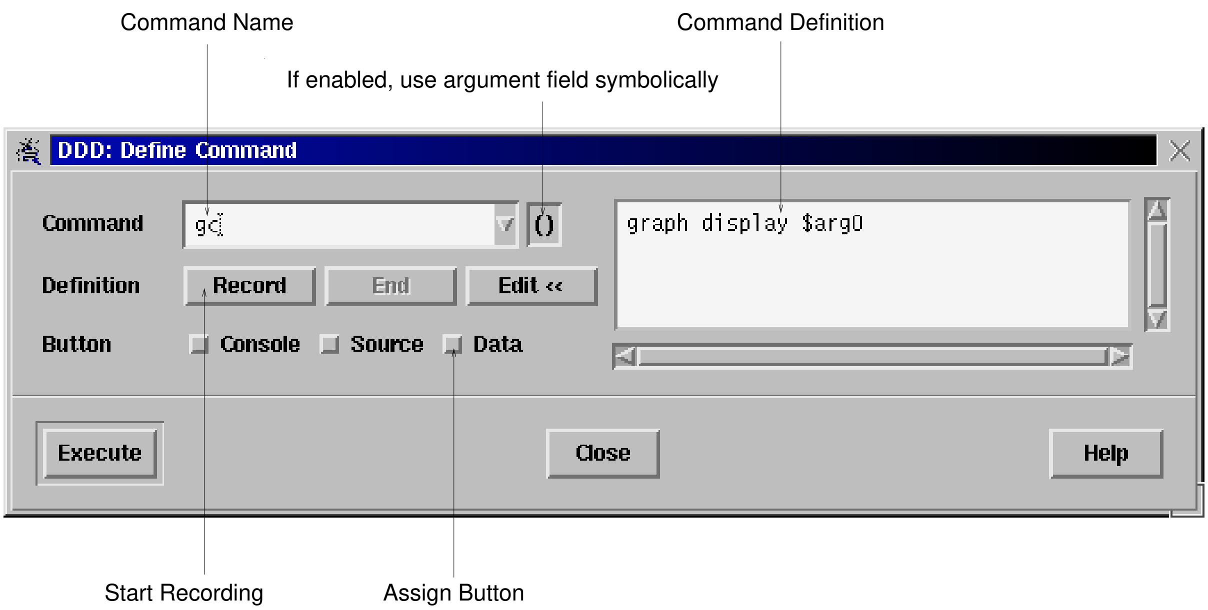
Aside from breakpoint commands (see '`Breakpoint commands`', above), DDD also allows you to store sequences of commands as a user-defined GDB command. A *user-defined command* is a sequence of GDB commands to which you assign a new name as a command. Using DDD, this is done via the *Command Editor*, invoked via '`Commands ⇒ Define Command`'.

A GDB command is created in five steps:

1. Enter the name of the command in the '`Command`' field. Use the drop-down list on the right to select from already defined commands.
2. Click on '`Record`' to begin the recording of the command sequence.
3. Now interact with DDD. While recording, DDD does not execute commands, but simply records them to be executed when the breakpoint is hit. The recorded debugger commands are shown in the debugger console.
4. To stop the recording, click on '`End`' or enter '`end`' at the GDB prompt. To *cancel* the recording, click on '`Interrupt`' or press ⟨ESC⟩.
5. Click on '`Edit >>`' to edit the recorded commands. When done with editing, click on '`Edit <<`' to close the commands editor.

After the command is defined, you can enter it at the GDB prompt. You may also click on '`Execute`' to test the given user-defined command.

For convenience, you can assign a button to the defined command. Enabling one of the '`Button`' locations will add a button with the given command to the specified location. If you want to edit the button, select '`Commands ⇒ Edit Buttons`'. See Section 10.4 [Defining Buttons], page 151, for a discussion.



Defining GDB Commands

When user-defined GDB commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.[1]

If used interactively, commands that would ask for confirmation proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

Command definitions are saved across DDD sessions.

10.5.2 Defining Argument Commands using GDB

If you want to pass arguments to user-defined commands, you can enable the '`()`' toggle button in the Command Editor. Enabling '`()`' has two effects:

- While recording commands, all references to the argument field are taken *symbolically* instead of literally. The argument field value is frozen to '`$arg0`', which is how GDB denotes the argument of a user-defined command. When GDB executes the command, it will replace '`$arg0`' by the current command argument.
- When assigning a button to the command, the command will be suffixed by the current contents of the argument field.

While defining a command, you can toggle the '`()`' button as you wish to switch between using the argument field symbolically and literally.

As an example, let us define a command `contuntil` that will set a breakpoint in the given argument and continue execution.

1. Enter '`contuntil`' in the '`Command`' field.
2. Enable the '`()`' toggle button.
3. Now click on '`Record`' to start recording. Note that the contents of the argument field change to '`$arg0`'.
4. Click on '`Break at ()`' to create a breakpoint. Note that the recorded breakpoint command refers to '`$arg0`'.
5. Click on '`Cont`' to continue execution.
6. Click on '`End`' to end recording. Note that the argument field is restored to its original value.
7. Finally, click on one of the '`Button`' locations. This creates a '`Contuntil ()`' button where '`()`' will be replaced by the current contents of the argument field—and thus passed to the '`contuntil`' command.
8. You can now either use the '`Contuntil ()`' button or enter a '`contuntil`' command at the GDB prompt. (If you plan to use the command frequently, you may wish to define a '`cu`' command, which again calls '`contuntil`' with its argument. This is a nice exercise.)

There is a little drawback with argument commands: a user-defined command in GDB has no means to access the argument list as a whole; only the first argument (up to whitespace) is processed. This may change in future GDB releases.

---

[1] If you use DDD commands within command definitions, or if you include debugger commands that resume execution, these commands will be realized transparently as *auto-commands*—that is, they won't be executed directly by the inferior debugger, but result in a command string being sent to DDD. This command string is then interpreted by DDD and sent back to the inferior debugger, possibly prefixed by some other commands such that DDD can update its state. See Section 10.5.3 [Commands with Other Debuggers], page 157, for a discussion.

10.5.3 Defining Commands using Other Debuggers

If your inferior debugger allows you to define own command sequences, you can also use these user-defined commands within DDD; just enter them at the debugger prompt.

However, you may encounter some problems:

- In contrast to the well-documented commands of the inferior debugger, DDD does not know what a user-defined command does. This may lead to inconsistencies between DDD and the inferior debugger. For instance, if your the user-defined command '`bp`' sets a breakpoint, DDD may not display it immediately, because DDD does not know that '`bp`' changes the breakpoint state.
- You cannot use DDD '`graph`' commands within user-defined commands. This is only natural, because user-defined commands are interpreted by the inferior debugger, which does not know about DDD commands.

As a solution, DDD provides a simple facility called *auto-commands*. If DDD receives any output from the inferior debugger in the form '*prefix command*', it will interpret *command* as if it had been entered at the debugger prompt. *prefix* is a user-defined string, for example '`ddd: `'.

Suppose you want to define a command `gd` that serves as abbreviation for `graph display`. All the command `gd` has to do is to issue a string

```
ddd: graph display argument
```

where *argument* is the argument given to `gd`. Using GDB, this can be achieved using the `echo` command. In your '`~/.gdbinit`' file, insert the lines

```
define gd
  echo ddd: graph display $arg0\n
end
```

To complete the setting, you must also set the '`autoCommandPrefix`' resource to the '`ddd: `' prefix you gave in your command. In '`~/.ddd/init`', write:

```
Ddd*autoCommandPrefix: ddd:\
```

(Be sure to leave a space after the trailing backslash.)

Entering `gd foo` will now have the same effect as entering `graph display foo` at the debugger prompt.

Please note: In your commands, you should choose some other prefix than '`ddd: `'. This is because auto-commands raise a security problem, since arbitrary commands can be executed. Just imagine some malicious program issuing a string like '*prefix* `shell rm -fr ~`' when being debugged! As a consequence, be sure to choose your own *prefix*; it must be at least three characters long.

Appendix A Application Defaults

Like any good X citizen, DDD comes with a large application-defaults file named 'Ddd'. This appendix documents the actions and images referenced in 'Ddd', such that you can easily modify them.

A.1 Actions

The following DDD actions may be used in translation tables.

A.1.1 General Actions

ddd-get-focus () Action
: Assign focus to the element that just received input.

ddd-next-tab-group () Action
: Assign focus to the next tab group.

ddd-prev-tab-group () Action
: Assign focus to the previous tab group.

ddd-previous-tab-group () Action
: Assign focus to the previous tab group.

A.1.2 Data Display Actions

These actions are used in the DDD graph editor.

end () Action
: End the action initiated by `select`. Bound to a button up event.

extend () Action
: Extend the current selection. Bound to a button down event.

extend-or-move () Action
: Extend the current selection. Bound to a button down event. If the pointer is dragged, move the selection.

follow () Action
: Continue the action initiated by `select`. Bound to a pointer motion event.

graph-select () Action
: Equivalent to `select`, but also updates the current argument.

graph-select-or-move () Action
: Equivalent to `select-or-move`, but also updates the current argument.

graph-extend () Action

Equivalent to `extend`, but also updates the current argument.

graph-extend-or-move () Action

Equivalent to `extend-or-move`, but also updates the current argument.

graph-toggle () Action

Equivalent to `toggle`, but also updates the current argument.

graph-toggle-or-move () Action

Equivalent to `toggle-or-move`, but also updates the current argument.

graph-popup-menu ([`graph`|`node`|`shortcut`]) Action

Pops up a menu. `graph` pops up a menu with global graph operations, `node` pops up a menu with node operations, and `shortcut` pops up a menu with display shortcuts.

If no argument is given, pops up a menu depending on the context: when pointing on a node with the ⟨Shift⟩ key pressed, behaves like `shortcut`; when pointing on a without the ⟨Shift⟩ key pressed, behaves like `node`; otherwise, behaves as if `graph` was given.

graph-dereference () Action

Dereference the selected display.

graph-detail () Action

Show or hide detail of the selected display.

graph-rotate () Action

Rotate the selected display.

graph-dependent () Action

Pop up a dialog to create a dependent display.

hide-edges ([`any`|`both`|`from`|`to`]) Action

Hide some edges. `any` means to process all edges where either source or target node are selected. `both` means to process all edges where both nodes are selected. `from` means to process all edges where at least the source node is selected. `to` means to process all edges where at least the target node is selected. Default is `any`.

layout ([`regular`|`compact`], [[+|−] *degrees*]) Action

Layout the graph. `regular` means to use the regular layout algorithm; `compact` uses an alternate layout algorithm, where successors are placed next to their parents. Default is `regular`. *degrees* indicates in which direction the graph should be layouted. Default is the current graph direction.

move-selected (*x-offset*, *y-offset*) Action

Move all selected nodes in the direction given by *x-offset* and *y-offset*. *x-offset* and *y-offset* is either given as a numeric pixel value, or as '`+grid`', or '`-grid`', meaning the current grid size.

normalize () Action

Place all nodes on their positions and redraw the graph.

rotate ([[+|−]*degrees*]) Action

Rotate the graph around *degrees* degrees. *degrees* must be a multiple of 90. Default is `+90`.

select () Action

Select the node pointed at. Clear all other selections. Bound to a button down event.

select-all () Action

Select all nodes in the graph.

select-first () Action

Select the first node in the graph.

select-next () Action

Select the next node in the graph.

select-or-move () Action

Select the node pointed at. Clear all other selections. Bound to a button down event. If the pointer is dragged, move the selected node.

select-prev () Action

Select the previous node in the graph.

show-edges ([`any`|`both`|`from`|`to`]) Action

Show some edges. `any` means to process all edges where either source or target node are selected. `both` means to process all edges where both nodes are selected. `from` means to process all edges where at least the source node is selected. `to` means to process all edges where at least the target node is selected. Default is `any`.

snap-to-grid () Action

Place all nodes on the nearest grid position.

toggle () Action

Toggle the current selection—if the node pointed at is selected, it will be unselected, and vice versa. Bound to a button down event.

toggle-or-move () Action

Toggle the current selection—if the node pointed at is selected, it will be unselected, and vice versa. Bound to a button down event. If the pointer is dragged, move the selection.

unselect-all () Action

Clear the selection.

A.1.3 Debugger Console Actions

These actions are used in the debugger console and other text fields.

gdb-backward-character () Action
: Move one character to the left. Bound to `Left`.

gdb-beginning-of-line () Action
: Move cursor to the beginning of the current line, after the prompt. Bound to `HOME`.

gdb-control (*control-character*) Action
: Send the given *control-character* to the inferior debugger. *control-character* must be specified in the form '`^`X', where X is an upper-case letter, or '`?`'.

gdb-command (*command*) Action
: Execute *command* in the debugger console. The following replacements are performed on *command*:

- If *command* has the form '*name*...', insert *name*, followed by a space, in the debugger console.
- All occurrences of '`()`' are replaced by the current contents of the argument field '`()`'.

gdb-complete-arg (*command*) Action
: Complete current argument as if *command* was prepended. Bound to ⟨Ctrl+T⟩.

gdb-complete-command () Action
: Complete current command line in the debugger console. Bound to ⟨TAB⟩.

gdb-complete-tab (*command*) Action
: If global ⟨TAB⟩ completion is enabled, complete current argument as if *command* was prepended. Otherwise, proceed as if the ⟨TAB⟩ key was hit. Bound to ⟨TAB⟩.

gdb-delete-or-control (*control-character*) Action
: Like `gdb-control`, but effective only if the cursor is at the end of a line. Otherwise, *control-character* is ignored and the character following the cursor is deleted. Bound to ⟨Ctrl+D⟩.

gdb-end-of-line () Action
: Move cursor to the end of the current line. Bound to `End`.

gdb-forward-character () Action
: Move one character to the right. Bound to `Right`.

gdb-insert-graph-arg () Action
: Insert the contents of the data display argument field '`()`'.

gdb-insert-source-arg () Action
: Insert the contents of the source argument field '`()`'.

gdb-interrupt () Action
: If DDD is in incremental search mode, exit it; otherwise call `gdb-control(^C)`.

gdb-isearch-prev () Action
: Enter reverse incremental search mode. Bound to ⟨Ctrl+B⟩.

gdb-isearch-next () Action
: Enter incremental search mode. Bound to ⟨Ctrl+F⟩.

gdb-isearch-exit () Action
: Exit incremental search mode. Bound to ⟨ESC⟩.

gdb-next-history () Action
: Recall next command from history. Bound to `Down`.

gdb-prev-history () Action
: Recall previous command from history. Bound to `Up`.

gdb-previous-history () Action
: Recall previous command from history. Bound to `Up`.

gdb-process ([*action* [, *args*. . .]]) Action
: Process the given event in the debugger console. Bound to key events in the source and data window. If this action is bound to the source window, and the source window is editable, perform *action*`(`*args*`...)` on the source window instead; if *action* is not given, perform '`self-insert()`'.

gdb-select-all () Action
: If the '`selectAllBindings`' resource is set to `Motif`, perform '`beginning-of-line`'. Otherwise, perform '`select-all`'. Bound to ⟨Ctrl+A⟩.

gdb-set-line (*value*) Action
: Set the current line to *value*. Bound to ⟨Ctrl+U⟩.

A.1.4 Source Window Actions

These actions are used in the source and code windows.

source-delete-glyph () Action
: Delete the breakpoint related to the glyph at cursor position.

source-double-click ([*text-action* [, *line-action* [, *function-action*]]]) Action

The double-click action in the source window.

- If this action is taken on a breakpoint glyph, edit the breakpoint properties.
- If this action is taken in the breakpoint area, invoke '`gdb-command(`*line-action*`)`'. If *line-action* is not given, it defaults to '`break ()`'.
- If this action is taken in the source text, and the next character following the current selection is '`(`', invoke '`gdb-command(`*function-action*`)`'. If *function-action* is not given, it defaults to '`list ()`'.
- Otherwise, invoke '`gdb-command(`*text-action*`)`'. If *text-action* is not given, it defaults to '`graph display ()`'.

source-drag-glyph () Action

Initiate a drag on the glyph at cursor position.

source-drop-glyph ([*action*]) Action

Drop the dragged glyph at cursor position. *action* is either '`move`', meaning to move the dragged glyph, or '`copy`', meaning to copy the dragged glyph. If no *action* is given, '`move`' is assumed.

source-end-select-word () Action

End selecting a word.

source-follow-glyph () Action

Continue a drag on the glyph at cursor position. Usually bound to some motion event.

source-popup-menu () Action

Pop up a menu, depending on the location.

source-set-arg () Action

Set the argument field to the current selection. Typically bound to some selection operation.

source-start-select-word () Action

Start selecting a word.

source-update-glyphs () Action

Update all visible glyphs. Usually invoked after a scrolling operation.

A.2 Images

DDD installs a number of images that may be used as pixmap resources, simply by giving a symbolic name. For button images, three variants are installed as well:

- The suffix '`-hi`' indicates a highlighted variant (Button is entered).
- The suffix '`-arm`' indicates an armed variant (Button is pushed).
- The suffix '`-xx`' indicates a disabled (insensitive) variant.

break_at Image
: ‘`Break at ()`’ button.

clear_at Image
: ‘`Clear at ()`’ button.

ddd Image
: `DDD` icon.

delete Image
: ‘`Delete ()`’ button.

disable Image
: ‘`Disable`’ button.

dispref Image
: ‘`Display * ()`’ button.

display Image
: ‘`Display ()`’ button.

drag_arrow Image
: The execution pointer (being dragged).

drag_cond Image
: A conditional breakpoint (being dragged).

drag_stop Image
: A breakpoint (being dragged).

drag_temp Image
: A temporary breakpoint (being dragged).

enable Image
: ‘`Enable`’ button.

find_forward Image
: ‘`Find>> ()`’ button.

find_backward Image
: ‘`Find<< ()`’ button.

grey_arrow Image
: The execution pointer (not in lowest frame).

grey_cond Image
: A conditional breakpoint (disabled).

grey_stop Image
: A breakpoint (disabled).

grey_temp Image
: A temporary breakpoint (disabled).

hide Image
: '`Hide ()`' button.

lookup Image
: '`Lookup ()`' button.

maketemp Image
: '`Make Temporary`' button.

new_break Image
: '`New Breakpoint`' button.

new_display Image
: '`New Display`' button.

new_watch Image
: '`New Watchpoint`' button.

plain_arrow Image
: The execution pointer.

plain_cond Image
: A conditional breakpoint (enabled).

plain_stop Image
: A breakpoint (enabled).

plain_temp Image
: A temporary breakpoint (enabled).

print Image
: '`Print ()`' button.

properties Image
: '`Properties`' button.

rotate Image

'`Rotate ()`' button.

set Image

'`Set ()`' button.

show Image

'`Show ()`' button.

signal_arrow Image

The execution pointer (stopped by signal).

undisplay Image

'`Undisplay ()`' button.

unwatch Image

'`Unwatch ()`' button.

watch Image

'`Watch ()`' button.

Appendix B Bugs and How To Report Them

Sometimes you will encounter a bug in DDD. Although we cannot promise we can or will fix the bug, and we might not even agree that it is a bug, we want to hear about bugs you encounter in case we do want to fix them.

To make it possible for us to fix a bug, you must report it. In order to do so effectively, you must know when and how to do it.

B.1 Where to Send Bug Reports

Send bug reports for DDD via electronic mail to

`bug-ddd@gnu.org`

B.2 Is it a DDD Bug?

Before sending in a bug report, try to find out whether the problem cause really lies within DDD. A common cause of problems are incomplete or missing X or M*tif installations, for instance, or bugs in the X server or M*tif itself. Running DDD as

```
$ ddd --check-configuration
```

checks for common problems and gives hints on how to repair them.

Another potential cause of problems is the inferior debugger; occasionally, they show bugs, too. To find out whether a bug was caused by the inferior debugger, run DDD as

```
$ ddd --trace
```

This shows the interaction between DDD and the inferior debugger on standard error while DDD is running. (If '`--trace`' is not given, this interaction is logged in the file '`~/.ddd/log`'; see Section B.5.1 [Logging], page 170) Compare the debugger output to the output of DDD and determine which one is wrong.

B.3 How to Report Bugs

Here are some guidelines for bug reports:

- The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!
- Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. It is not very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.
- Your bug report should be self-contained. Do not refer to information sent in previous mails; your previous mail may have been forwarded to somebody else.
- Please report each bug in a separate message. This makes it easier for us to track which bugs have been fixed and to forward your bugs reports to the appropriate maintainer.
- Please report bugs in English; this increases the chances of finding someone who can fix the bug. Do not assume one particular person will receive your bug report.

B.4 What to Include in a Bug Report

To enable us to fix a DDD bug, you *must* include the following information:

- Your DDD configuration. Invoke DDD as

  ```
  $ ddd --configuration
  ```

  to get the configuration information. If this does not work, please include at least the DDD version, the type of machine you are using, and its operating system name and version number.
- The debugger you are using and its version (e.g., '`gdb-4.17`' or '`dbx as shipped with Solaris 2.6`').
- The compiler you used to compile DDD and its version (e.g., '`gcc-2.8.1`').
- A description of what behavior you observe that you believe is incorrect. For example, "DDD gets a fatal signal" or "DDD exits immediately after attempting to create the data window".
- A *log file* showing the interaction between DDD and the inferior debugger. By default, this interaction is logged in the file '`~/.ddd/log`'. Include all trace output from the DDD invocation up to the first bug occurrence; insert own comments where necessary.
- If you wish to suggest changes to the DDD source, send us context diffs. If you even discuss something in the DDD source, refer to it by context, *never* by line number.

Be sure to include this information in *every* single bug report.

B.5 Getting Diagnostics

B.5.1 Logging

If things go wrong, the first and most important information source is the DDD *log file*. This file, created in '`~/.ddd/log`' ('`~`' stands for your home directory), records the following information:

- Your DDD configuration (at the top)
- All programs invoked by DDD, shown as '`$` *program args*`...`'
- All DDD messages, shown as '`#` *message*'.
- All information sent from DDD to the inferior debugger, shown as '`->` *text*'.
- All information sent from the inferior debugger standard output to DDD, shown as '`<-` *text*'.
- All information sent from the inferior debugger standard error to DDD, shown as '<= *text*'.[1]
- All information sent from DDD to Gnuplot, shown as '>> *text*'.
- All information sent from Gnuplot standard output to DDD, shown as '<< *text*'.
- All information sent from Gnuplot standard error to DDD, shown as '<= *text*'.
- If DDD crashes, a GDB backtrace of the DDD core dump is included at the end.

This information, all in one place, should give you (and anyone maintaining DDD) a first insight of what's going wrong.

---

[1] Since the inferior debugger is invoked through a virtual TTY, standard error is normally redirected to standard output, so DDD never receives standard error from the inferior debugger.

B.5.1.1 Disabling Logging

The log files created by DDD can become quite large, so you might want to turn off logging. There is no explicit DDD feature that allows you to do that. However, you can easily create a *symbolic link* from '~/.ddd/log' to '/dev/null', such that logging information is lost. Enter the following commands at the shell prompt:

```
$ cd
$ rm .ddd/log
$ ln -s /dev/null .ddd/log
$ _
```

Be aware, though, that having logging turned off makes diagnostics much more difficult; in case of trouble, it may be hard to reproduce the error.

B.5.2 Debugging DDD

As long as DDD is compiled with '-g' (see Section 4.1 [Compiling for Debugging], page 73), you can invoke a debugger on DDD—even DDD itself, if you wish. From within DDD, a special 'Maintenance' menu is provided that invokes GDB on the running DDD process. See Section 3.1.9 [Maintenance Menu], page 49, for details.

The DDD distribution comes with a '.gdbinit' file that is suitable for debugging DDD. Among others, this defines a 'ddd' command that sets up an environment for debugging DDD and a 'string' command that lets you print the contents of DDD 'string' variables; just use 'print *var*' followed by 'string'.

You can cause DDD to dump core at any time by sending it a SIGUSR1 signal. DDD resumes execution while you can examine the core file with GDB.

When debugging DDD, it can be useful to make DDD not catch fatal errors. This can be achieved by setting the environment variable DDD_NO_SIGNAL_HANDLERS before invoking DDD.

B.5.3 Customizing Diagnostics

You can use these additional resources to obtain diagnostics about DDD. Most of them are tied to a particular invocation option.

appDefaultsVersion (class Version) Resource

The version of the DDD app-defaults file. If this string does not match the version of the current DDD executable, DDD issues a warning.

checkConfiguration (class CheckConfiguration) Resource

If 'on', check the DDD environment (in particular, the X configuration), report any possible problem causes and exit. See Section 2.1.2 [Options], page 16, for the '--check-configuration' option.

dddinitVersion (class Version) Resource

The version of the DDD executable that last wrote the '~/.ddd/init' file. If this string does not match the version of the current DDD executable, DDD issues a warning.

debugCoreDumps (class DebugCoreDumps) Resource

If 'on', DDD invokes a debugger on itself when receiving a fatal signal. See Section 3.1.9 [Maintenance Menu], page 49, for setting this resource.

dumpCore (class DumpCore) Resource

If 'on' (default), DDD dumps core when receiving a fatal signal. See Section 3.1.9 [Maintenance Menu], page 49, for setting this resource.

maintenance (class Maintenance) Resource

If 'on', enables the top-level 'Maintenance' menu (see Section 3.1.9 [Maintenance Menu], page 49) with additional options. See Section 2.1.2 [Options], page 16, for the '--maintenance' option.

showConfiguration (class ShowConfiguration) Resource

If 'on', show the DDD configuration on standard output and exit. See Section 2.1.2 [Options], page 16, for the '--configuration' option.

showFonts (class ShowFonts) Resource

If 'on', show the DDD font definitions on standard output and exit. See Section 2.1.2 [Options], page 16, for the '--fonts' option.

showInvocation (class ShowInvocation) Resource

If 'on', show the DDD invocation options on standard output and exit. See Section 2.1.2 [Options], page 16, for the '--help' option.

showLicense (class ShowLicense) Resource

If 'on', show the DDD license on standard output and exit. See Section 2.1.2 [Options], page 16, for the '--license' option.

showManual (class ShowManual) Resource

If 'on', show this DDD manual page on standard output and exit. If the standard output is a terminal, the manual page is shown in a pager ($PAGER, less or more). See Section 2.1.2 [Options], page 16, for the '--manual' option.

showNews (class ShowNews) Resource

If 'on', show the DDD news on standard output and exit. See Section 2.1.2 [Options], page 16, for the '--news' option.

showVersion (class ShowVersion) Resource

If 'on', show the DDD version on standard output and exit. See Section 2.1.2 [Options], page 16, for the '--version' option.

suppressWarnings (class SuppressWarnings) Resource

If 'on', X warnings are suppressed. This is sometimes useful for executables that were built on a machine with a different X or M*tif configuration. By default, this is 'off'. See Section 2.1.6 [X Warnings], page 28, for details.

trace (class Trace) Resource

If 'on', show the dialog between DDD and the inferior debugger on standard output. Default is 'off'. See Section 2.1.2 [Options], page 16, for the '--trace' option.

Appendix C Configuration Notes

C.1 Using DDD with GDB

Some GDB settings are essential for DDD to work correctly. These settings with their correct values are:

```
set height 0
set width 0
set verbose off
set annotate 1
set prompt (gdb)
```

DDD sets these values automatically when invoking GDB; if these values are changed, there may be some malfunctions, especially in the data display.

When debugging at the machine level with GDB 4.12 and earlier as inferior debugger, use a '`display /x $pc`' command to ensure the program counter value is updated correctly at each stop. You may also enter the command in '`~/.gdbinit`' or (better yet) upgrade to the most recent GDB version.

C.1.1 Using DDD with WDB

HP's WildeBeest (WDB) is essentially a variant of GDB. To start DDD with WDB as inferior debugger, use

`ddd --wdb` *program*

See Section C.1 [GDB], page 175, for further configuration notes.

C.1.2 Using DDD with WindRiver GDB (Tornado)

DDD now supports WindRiver's version of GDB.[1] DDD can be integrated into the '`Launch`' window by placing the '`launch.tcl`' script (see below) into the the directory '`~/.wind`'.

Currently, DDD only supports the PowerPC and has been only tested on a Solaris 2.6 host.

DDD launches the version of GDB that is either in the current path, or the one specified on the command line using the '`--debugger`' command.

Normally, the Tornado environment is set up by sourcing a script file which, among other things, sets up the PATH variable.

It is suggested that a soft link for the version of GDB used for the target (i.e. '`gdbppc`') be made in the same directory:

```
bin>ls -l gdb*
39 Mar  6 16:14 gdb -> /usr/wind/host/sun4-solaris2/bin/gdbppc*
1619212 Mar 11  1997 gdbppc*
bin>_
```

This way DDD will start the correct version of GDB automatically.

[1] This section was contributed by Gary Cliff from Computing Devices Canada Ltd., `gary.cliff@cdott.com`.

It is also suggested that you use DDD's execution window to facilitate parsing of GDB output. See Section 2.5.3 [Debugger Communication], page 38, for details.

Tornado reads the default TCL scripts first, then the ones in the users '`.wind`' directory. The following procedures can be cut and pasted into the user's '`launch.tcl`' file:

```
# Launch.tcl - Launch application Tcl user customization file.
#

######
#
# setupDDD - sets up DDD for use by the launcher
#
# This routine adds the DDD to the application bar
#
# SYNOPSIS:
# setupDDD
#
# PARAMETERS: N/A
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc setupDDD {} {
    # Add to the default application bar
    objectCreate app ddd DDD {launchDDD}
}
```

```
######
#
# launchDDD - launch the DDD debugger
#
# SYNOPSIS:
# launchDDD
#
# PARAMETERS: N/A
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc launchDDD {} {

    global tgtsvr_selected
    global tgtsvr_cpuid

    if {$tgtsvr_selected == "" || $tgtsvr_cpuid == 0} {
        noticePost error "Select an attached target first."
        return
    }

    set startFileName /tmp/dddstartup.[pid]

    if [catch {open $startFileName w} file] {
         # couldn't create a startup file.  Oh, well.
         exec ddd --gdb &
    }
    else
    {
         # write out a little /tmp file that attaches to the
         # selected target server and then deletes itself.
         puts $file "set wtx-tool-name ddd"
         puts $file "target wtx $tgtsvr_selected"
         puts $file "tcl exec rm $startFileName"
         close $file
         exec ddd --gdb --command=$startFileName &
    }
}
```

```
######
#
# Launch.tcl - Initialization
#
# The user's resource file sourced from the initial Launch.tcl
#

# Add DDD to the laucher
  setupDDD
```

In order for DDD to automatically display the source of a previously loaded file, the entry point must be named either 'vxworks_main' or 'main_vxworks'.

See Section C.1 [GDB], page 175, for further configuration notes.

C.2 Using DDD with DBX

When used for debugging Pascal-like programs, DDD does not infer correct array subscripts and always starts to count with 1.

With some DBX versions (notably Solaris DBX), DDD strips C-style and C++-style comments from the DBX output in order to interpret it properly. This also affects the output of the debugged program when sent to the debugger console. Using the separate execution window avoids these problems.

In some DBX versions (notably DEC DBX and AIX DBX), there is no automatic data display. As an alternative, DDD uses the DBX 'print' command to access data values. This means that variable names are interpreted according to the current frame; variables outside the current frame cannot be displayed.

C.3 Using DDD with Ladebug

All DBX limitations (see Section C.2 [DBX], page 178) apply to Ladebug as well.

C.4 Using DDD with XDB

There is no automatic data display in XDB. As a workaround, DDD uses the 'p' command to access data values. This means that variable names are interpreted according to the current frame; variables outside the current frame cannot be displayed.

C.5 Using DDD with JDB

There is no automatic data display in JDB. As a workaround, DDD uses the 'dump' command to access data values. This means that variable names are interpreted according to the current frame; variables outside the current frame cannot be displayed.

In JDB 1.1, the 'dump' and 'print' commands do not support expression evaluation. Hence, you cannot display arbitrary expressions.

Parsing of JDB output is quite CPU-intensive, due to the recognition of asynchronous prompts (any thread may output anything at any time, including prompts). Hence, a program producing much console output is likely to slow down DDD considerably. In such a case, have the program run with '-debug' in a separate window and attach JDB to it using the '-passwd' option.

C.6 Using DDD with Perl

There is no automatic data display in Perl. As a workaround, DDD uses the '`x`' command to access data values. This means that variable names are interpreted according to the current frame; variables outside the current frame cannot be displayed.

C.7 Using DDD with Bash

BASH support is rather new. As a programming language, BASH is not feature rich: there are no record structures or hash tables (yet), no pointers, package variable scoping or methods. So much of the data display and visualization features of DDD are disabled.

As with any scripting or interpreted language like Perl, stepping a machine-language instructions (commands Stepi/Nexti) doesn't exist.

Some BASH settings are essential for DDD to work correctly. These settings with their correct values are:

```
set annotate 1
set prompt set prompt bashdb$_Dbg_less$_Dbg_greater$_Dbg_space
```

DDD sets these values automatically when invoking BASH; if these values are changed, there may be some malfunctions.

Pay special attention when the prompt has extra angle brackets (a nested shell) or has any parenthesis (is in a subshell). Quitting may merely exit out of one of these nested (sub)shells rather than leave the program.

C.8 Using DDD with LessTif

DDD includes a number of hacks that make DDD run with *LessTif,* a free M*tif library without loss of functionality. Since a DDD binary may be dynamically bound and used with either an OSF/Motif or LessTif library, these *lesstif hacks* can be enabled and disabled at run time.

Whether the *lesstif hacks* are included at run-time depends on the setting of the '`lessTifVersion`' resource:

lessTifVersion (class LessTifVersion) Resource

Indicates the LessTif version DDD is running against. For LessTif version *x.y*, the value is *x* multiplied by 1000 plus *y*—for instance, the value `79` stands for LessTif 0.79 and the value `1005` stands for LessTif 1.5.

If the value of this resource is less than 1000, indicating LessTif 0.99 or earlier, DDD enables version-specific hacks to make DDD work around LessTif bugs and deficiencies.

If DDD was compiled against LessTif, the default value is the value of the '`LessTifVersion`' macro in '`<Xm/Xm.h>`'. If DDD was compiled against OSF/Motif, the default value is `1000`, disabling all LessTif-specific hacks.

To set the '`lessTifVersion`' resource at DDD invocation and to specify the version number of the LessTif library, you can also use the option '`--lesstif-version`' *version*.

The default value of the '`lessTifVersion`' resource is derived from the LessTif library DDD was compiled against (or `1000` when compiled against OSF/Motif). Hence, you normally don't need to worry about the value of this resource. However, if you use a dynamically linked DDD binary with a library other than the one DDD was compiled against, you must specify the version number of the library using this resource. (Unfortunately, DDD cannot detect this at run-time.)

Here are a few scenarios to illustrate this scheme:

- Your DDD binary was compiled against OSF/Motif, but you use a LessTif 0.88 dynamic library instead. Invoke DDD with '`--lesstif-version 88`'.
- Your DDD binary was compiled against LessTif, but you use a OSF/Motif dynamic library instead. Invoke DDD with '`--lesstif-version 1000`'.
- Your DDD binary was compiled against LessTif 0.85, and you have upgraded to LessTif 0.90. Invoke DDD with '`--lesstif-version 90`'.

To find out the LessTif or OSF/Motif version DDD was compiled against, invoke DDD with the '`--configuration`' option.

In the DDD source, LessTif-specific hacks are controlled by the string '`lesstif_version`'.

Appendix D Dirty Tricks

Do you miss anything in this manual? Do you have any material that should be added? Please send any contributions to `ddd@gnu.org`.

Appendix E Extending DDD

If you have any contributions to be incorporated into DDD, please send them to `ddd@gnu.org`. For suggestions on what might be done, see the file '`TODO`' in the DDD distribution.

Appendix F Frequently Answered Questions

See the DDD WWW page (`http://www.gnu.org/software/ddd/`) for frequently answered questions not covered in this manual.

Appendix G GNU General Public License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The

"Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

    NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT

HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix H Help and Assistance

We have set up a *mailing list* for general DDD discussions. If you need help and assistance for solving a DDD problem, you find the right people here.

Send message to all receivers of the mailing list to:

`ddd@gnu.org`

This mailing list is also the place where new DDD releases are announced. If you want to subscribe the list, or get more information, send a mail to

`ddd-request@gnu.org`

See also the DDD WWW page (`http://www.gnu.org/software/ddd/`) for recent announcements and other news related to DDD.

Appendix I GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

   A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

   The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

   Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Label Index

Key Index

A

C

D

E

F

H

L

R

S

T

U

Command Index

Resource Index

A

B

C

D

E

F

Q

R

S

T

U

V

W

X

File Index

W

X

Concept Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z