NAME

ddd, xddd - the data display debugger

SYNOPSIS

ddd [**−−gdb**] [**−−dbx**] [**−−xdb**] [**−−jdb**] [**−−pydb**] [**−−perl**] [**−−debugger** *name*] [**−−**[**r**]**host**
[*username*@]*hostname*]] [**−−help**] [**−−trace**] [**−−version**] [**−−configuration**] [*options...*]
[*program* [*core* | *process-id*]]

but usually just

ddd *program*

DESCRIPTION

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.

“Classical” UNIX debuggers such as the GNU debugger (GDB) provide a command-line interface and a multitude of commands for these and other debugging purposes. DDD is a comfortable *graphical user interface* around an inferior GDB, DBX, XDB, JDB, Python debugger, or Perl debugger.

INVOKING DDD

You can run DDD with no arguments or options. However, the most usual way to start DDD is with one argument or two, specifying an executable program as the argument:

ddd program

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).

By default, DDD determines the inferior debugger automatically. Use

ddd −−gdb *program*

or

ddd −−dbx *program*

or

ddd −−xdb *program*

or

ddd −−jdb *class*

or

ddd −−pydb *module*

or

ddd −−perl *programfile*

to run GDB, DBX, XDB, JDB, PYDB or Perl as inferior debugger.

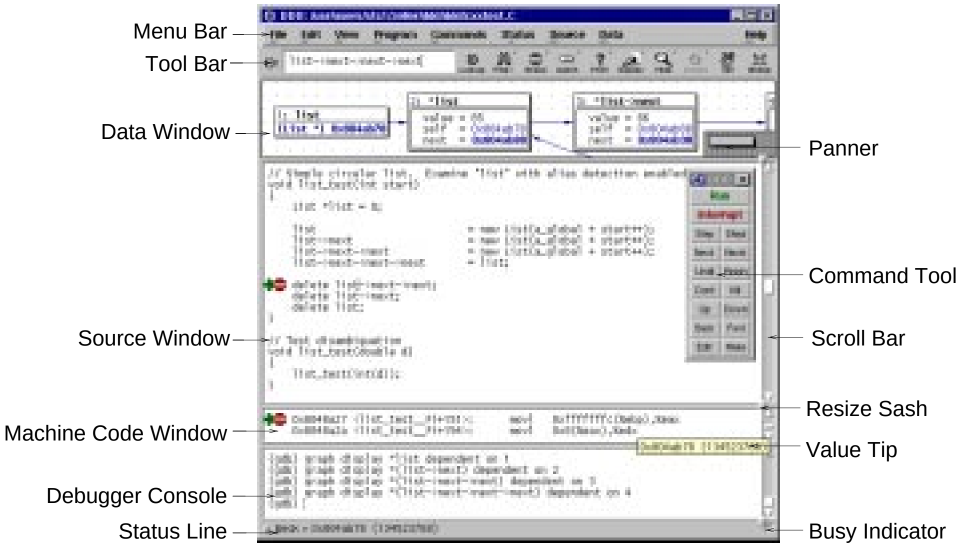
To learn more about DDD options, run

ddd −−help

to get a list of frequently used options, or see the '**OPTIONS**' section, below.

THE DDD WINDOWS

The DDD Main Windows

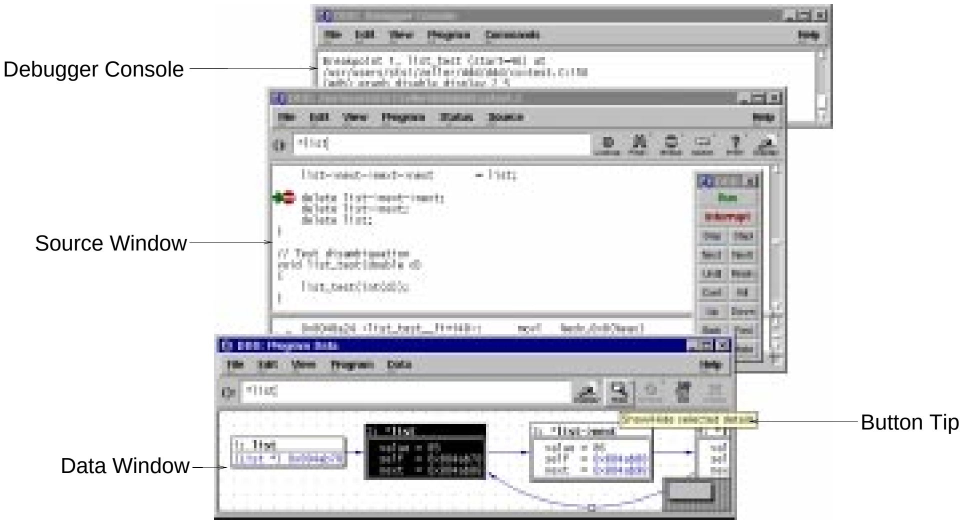


The DDD Layout using Stacked Windows

DDD is composed of three main windows:

- The *Data Window* shows the current data of the debugged program.
- The *Source Window* shows the current source code of the debugged program.
- The *Debugger Console* accepts debugger commands and shows debugger messages.

By default, DDD places these main windows stacked into one single top-level window, but DDD can also be configured to treat each one separately.



The DDD Layout using Separate Windows

Besides these 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.
- The *Machine Code Window* shows the current machine code. It is usually placed beneath the current source.
- The *Execution Window* shows the input and output of the debugged program.

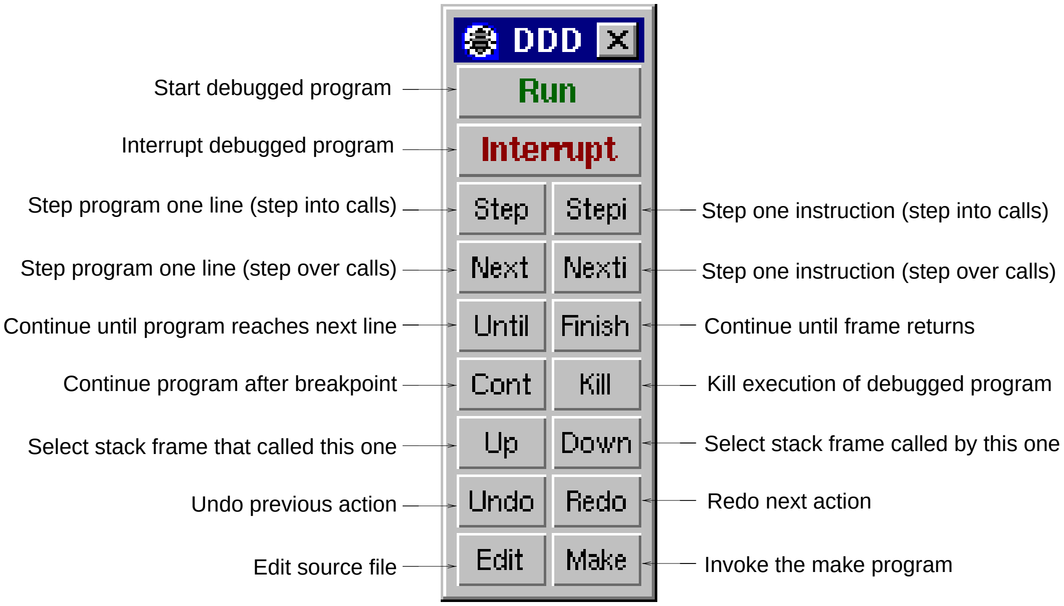
DDD also has several temporary *dialogs* for showing and entering additional information.

Using the Command Tool

The command tool is a small window containing frequently used DDD commands. It can be moved around on top of the DDD windows, but it can also be placed besides them. 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**'.

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.


DDD
Start debugged program
Run
Interrupt debugged program
Interrupt
Step program one line (step into calls)
Step
Stepi
Step one instruction (step into calls)
Step program one line (step over calls)
Next
Nexti
Step one instruction (step over calls)
Continue until program reaches next line
Until
Finish
Continue until frame returns
Continue program after breakpoint
Cont
Kill
Kill execution of debugged program
Select stack frame that called this one
Up
Down
Select stack frame called by this one
Undo previous action
Undo
Redo
Redo next action
Edit source file
Edit
Make
Invoke the make program

The Command Tool

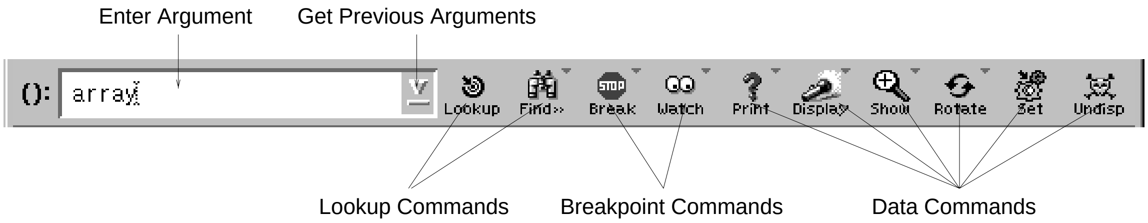
Using the Tool Bar

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

GETTING HELP

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

Button Tips

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*) 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

The status line also displays information about the currently selected item. By clicking on the status line, you can redisplay the most recent messages.

Context-Sensitive Help

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.

Help on Debugger Commands

You can get help on debugger commands by entering ‘**help**’ at the debugger prompt.

See ‘**Entering Commands**’, below, for details on entering commands.

Are You Stuck?

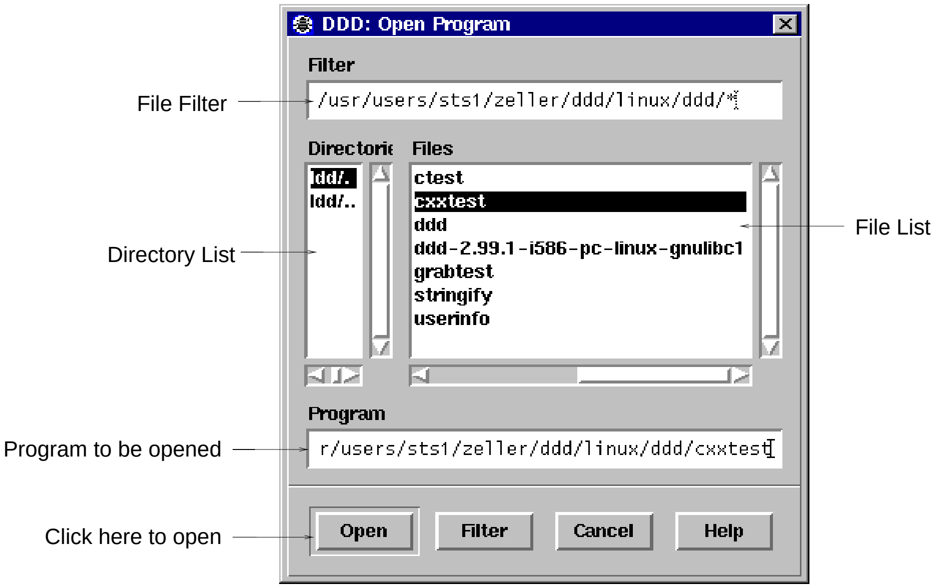
If you are stuck, try ‘**Help→What Now?**’ (the ‘**What Now**’ item in the ‘**Help**’ menu) or press **Ctrl+F1**. Depending on the current state, DDD will give you some hints on what you can do next.

Undoing Commands

And if, after all, you made a mistake, don’t worry. 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.

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.



Filter

/usr/users/sts1/zeller/ddd/linux/ddd/*

Directories

ddd/.
ddd/..

Files

ctest
cxxtest
ddd
ddd-2.99.1-i586-pc-linux-gnulibc1
grabtest
stringify
userinfo

Program

r/users/sts1/zeller/ddd/linux/ddd/cxxtest

Opening a program to be debugged

To open a program to be debugged, select ‘**File→Open Program**’.

In JDB, select ‘**File→Open Class**’ instead. This gives you a list of available classes to choose from.

To re-open a recently debugged program or class, select ‘**File→Open Recent**’ and choose a program or class from the list.

Note: With XDB and some DBX versions, the debugged program must be specified upon invocation and cannot be changed at run time.

To open a core dump for the program, select ‘**File→Open 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.

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.

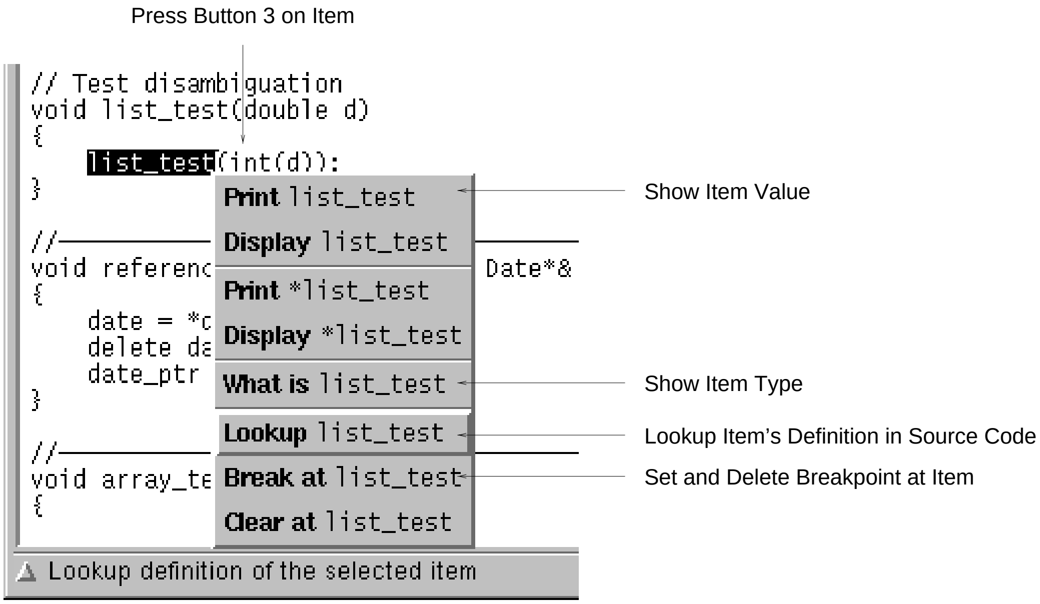
LOOKING UP ITEMS

As soon as the source of the debugged program is available, the *source window* displays its current source text. (If a source text cannot be found, use ‘**Edit→GDB Settings**’ to specify source text directories.)

In the source window, you can lookup and examine function and variable definitions as well as search for arbitrary occurrences in the source text.

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. Alter the name if desired and click on the ‘**Lookup ()**’ button to find its definition.



```
// Test disambiguation
void list_test(double d)
{
    list_test(int(d));
}

//————————
void referenc                  Date*&
{
    date = *c
    delete da
    date_ptr
}

//————————
void array_te
{
```

Lookup definition of the selected item

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.

Textual Search

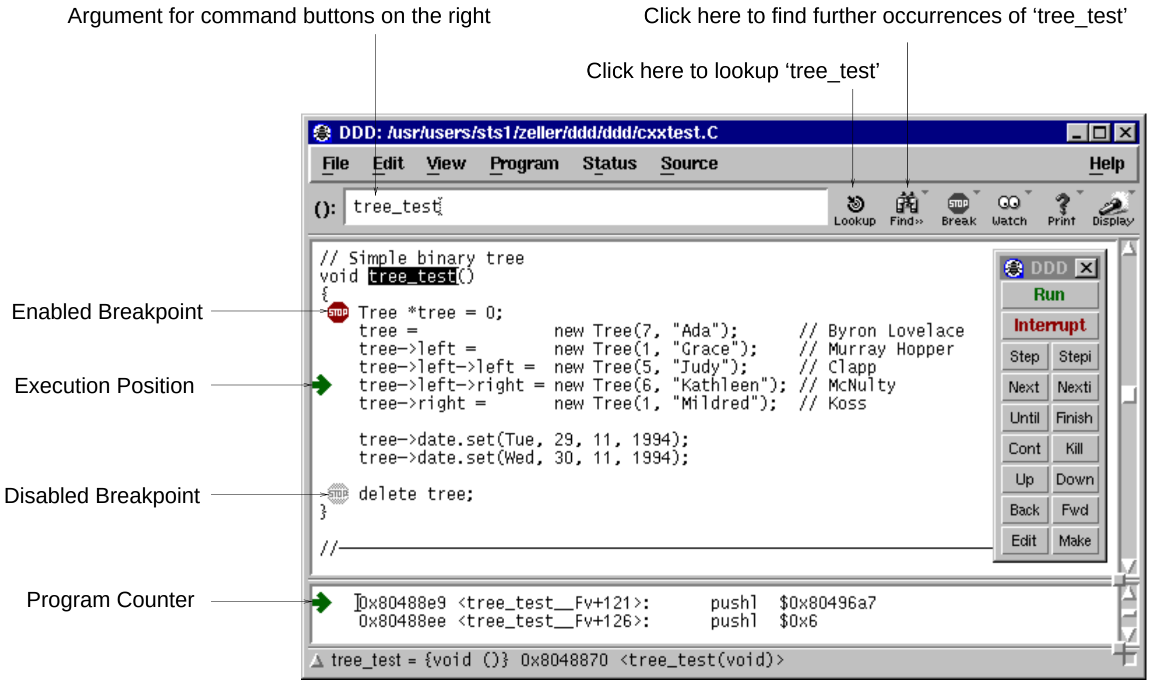
If the item you wish to search is visible in the source text, click with *mouse button 1* on it. The identifier is copied to the argument field. Click on the '**Find**>> **()**' button to find following occurrences and on the '**Find**<< **()**' button to find previous occurrences.

As an alternative, you can enter the item in the argument field and click on one of the '**Find**' buttons.

By default, DDD finds only complete words. To search for arbitrary substrings, change the value of the '**Source**→**Find Words Only**' option.

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.



Argument for command buttons on the right

Click here to find further occurrences of 'tree_test'

Click here to lookup 'tree_test'

Enabled Breakpoint

Execution Position

Disabled Breakpoint

Program Counter

(): tree_test

```
// Simple binary tree
void tree_test()
{
    Tree *tree = 0;
    tree =               new Tree(7, "Ada");      // Byron Lovelace
    tree->left =         new Tree(1, "Grace");    // Murray Hopper
    tree->left->left =   new Tree(5, "Judy");     // Clapp
    tree->left->right =  new Tree(6, "Kathleen"); // McNulty
    tree->right =        new Tree(1, "Mildred");  // Koss

    tree->date.set(Tue, 29, 11, 1994);
    tree->date.set(Wed, 30, 11, 1994);

    delete tree;
}

//
```

```
0x80488e9 <tree_test__Fv+121>:      pushl  $0x80496a7
0x80488ee <tree_test__Fv+126>:      pushl  $0x6
```

```
tree_test = {void ()} 0x8048870 <tree_test(void)>
```

The Source Window

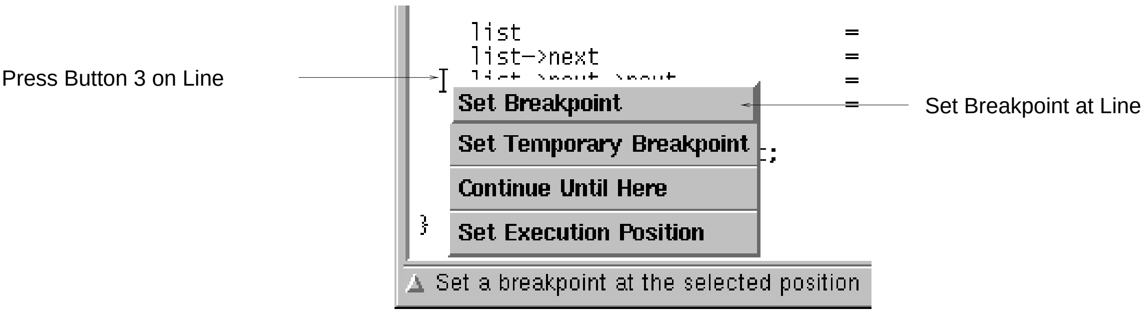
BREAKPOINTS

You can make the program stop at certain *breakpoints* and trace its execution.

Setting Breakpoints by Location

If the source line is visible, click with *mouse button 1* on the left of the source line and then on the '**Break at ()**' 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.



```
list                  =
list->next            =
```

Set a breakpoint at the selected position

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→Edit 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.

Note: We have received reports that some Motif versions fail to display stop signs correctly. If this happens, try writing in your ‘**$HOME/.ddd/init**’ file:

```
Ddd*cacheGlyphImages: off
```

and restart DDD. See also the ‘**cacheGlyphImages**’ resource in the ‘**RESOURCES**’ section, below.

Setting Breakpoints by Name

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 ‘**Break at ()**’ 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**’ item from the popup menu.

As yet another alternative, you can click on ‘**New**’ from the Breakpoint editor (invoked through ‘**Source→Edit Breakpoints**’) and enter the function name.

Setting Regexp Breakpoints

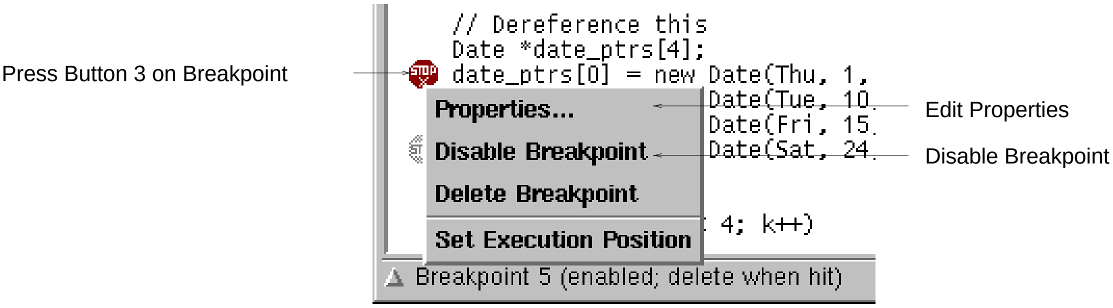
Using GDB, you can also set a breakpoint on all functions that match a given string. ‘**Break at ()→Set Breakpoints at Regexp ()**’ sets a breakpoint on all functions whose name matches the *regular expression* given in ‘**()**’. Here are some examples:

- 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$**’.

Once these multiple breakpoints are set, they are treated just like the breakpoints set with the ‘**Break at ()**’ button. You can delete them, disable them, or make them conditional the same way as any other breakpoint. Use ‘**Source→Edit Breakpoints**’ to view and edit the list of breakpoints.

Disabling Breakpoints

To temporarily 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**’.



```
// Dereference this
Date *date_ptrs[4];
date_ptrs[0] = new Date(Thu, 1,
                    Date(Tue, 10.
                    Date(Fri, 15.
                    Date(Sat, 24.

                  4; k++)
```

Breakpoint 5 (enabled; delete when hit)

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→Edit 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 at ()**’ button. Just press and hold *mouse button 1* on the button to get a popup menu.

Note: JDB does not support breakpoint disabling.

Temporary Breakpoints

A *temporary breakpoint* is immediately deleted as soon as it is reached. 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.

The ‘**Set Temporary Breakpoint**’ and ‘**Continue Until Here**’ items are also accessible via the ‘**Break at ()**’ button. Just press and hold *mouse button 1* on the button to get a popup menu.

Note: JDB does not support temporary breakpoints.

Deleting Breakpoints

If the breakpoint is visible, click with *mouse button 1* on the breakpoint. The breakpoint location is copied to the argument field. Click on the ‘**Clear at ()**’ 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 at ()**’ button to set a breakpoint 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.

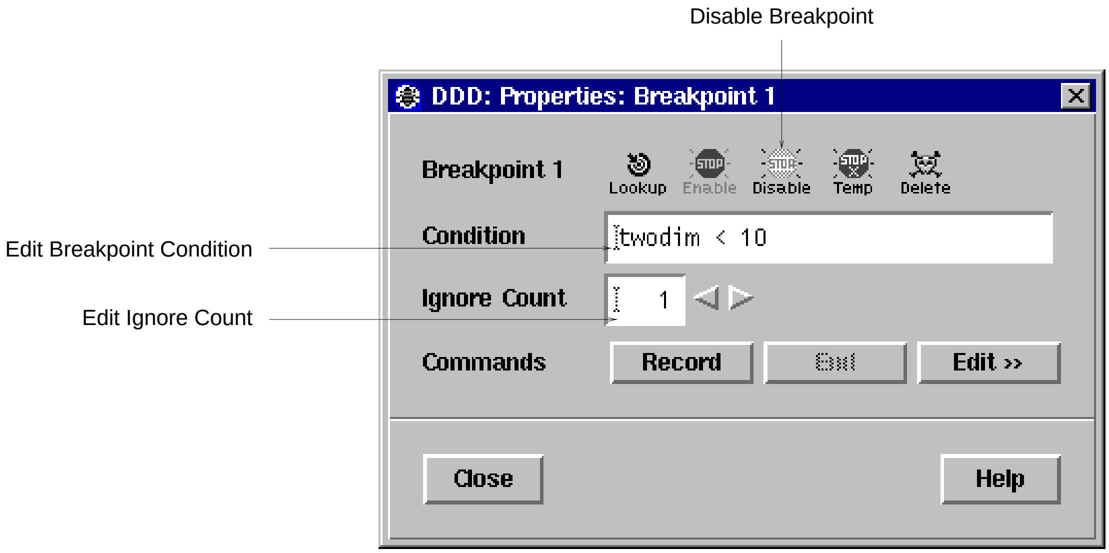
As yet another alternative, you can select the breakpoint and click on ‘**Delete**’ in the Breakpoint editor (invoked through ‘**Source→Edit Breakpoints**’).

As an even faster alternative, you can simply double-click on the breakpoint while holding **Ctrl**.

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.

As an even faster alternative, you can simply double-click on the breakpoint.



Condition: twodim < 10

Ignore Count: 1

Breakpoint Properties

- 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. Note: GDB has no way to make a temporary breakpoint non-temporary again.
- Click on '**Delete**' to delete the breakpoint.

Breakpoint Conditions

In the field '**Condition**' of the '**Breakpoint Properties**' panel, you can specify a *breakpoint condition*. If a breakpoint condition is set, the breakpoint stops the program only if the associated condition is met—that is, if the condition expression evaluates to a non-zero value.

Note: JDB does not support breakpoint conditions.

Breakpoint Ignore Counts

In the field '**Ignore Count**' of the '**Breakpoint Properties**' panel, you can specify a *breakpoint ignore count*. If the ignore count is set to some value *N*, the next *N* crossings of the breakpoint will be ignored: Each crossing of the breakpoint decrements the ignore count; the program stops only if the ignore count is zero.

Note: JDB, Perl and some DBX variants do not support breakpoint ignore counts.

Breakpoint Commands

Note: Breakpoint commands are currently available on GDB only.

Using the '**Commands**' buttons of the '**Breakpoint Properties**' panel, you can record and edit commands to be executed when the breakpoint is hit.

To record a command sequence, follow these steps:

- Click on ‘**Record**’ to begin the recording of the breakpoint commands.
- 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.
- To stop the recording, click on ‘**End**’ or enter ‘**end**’ at the GDB prompt. To *cancel* the recording, click on ‘**Interrupt**’ or press **ESC**.
- Click on ‘**Edit** >>’ to edit the recorded commands. When done with editing, click on ‘**Edit** <<’ to close the commands editor.

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. 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 the **Shift** key while dragging.

Note: Dragging breakpoints is not possible when glyphs are disabled. Delete and set breakpoints instead.

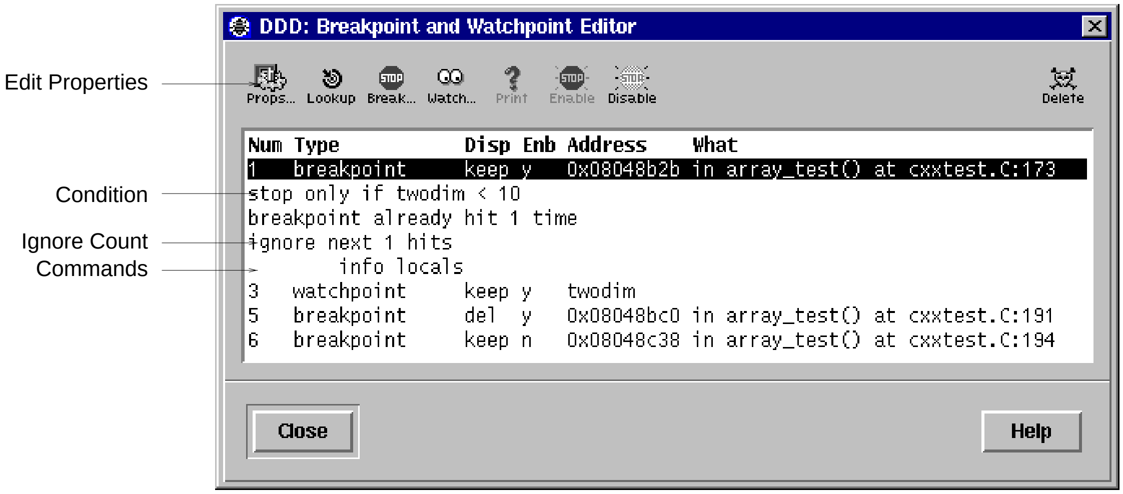
Looking up Breakpoints

If you wish to lookup a specific breakpoint, select ‘**Source**→**Edit 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.

Editing all Breakpoints

To view and edit all breakpoints at once, select ‘**Source**→**Edit Breakpoints**’. This will popup the *Breakpoint Editor* which displays the state of all breakpoints.



```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048b2b in array_test() at cxxtest.C:173
stop only if twodim < 10
breakpoint already hit 1 time
ignore next 1 hits
        info locals
3   watchpoint     keep y   twodim
5   breakpoint     del  y   0x08048bc0 in array_test() at cxxtest.C:191
6   breakpoint     keep n   0x08048c38 in array_test() at cxxtest.C:194
```

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**’.

More Breakpoint Features

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 the GDB documentation for details on these commands.

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'.

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.

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.

Note: 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.

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, you can set different types of watchpoints. Click and hold *mouse button 1* on the '**Watch ()**' button to get a menu.

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 '**Editing Breakpoint Properties**', above, for details. As an additional feature, you can click on '**Print ()**' to see the current value of a watched variable.

Editing all Watchpoints

To view and edit all watchpoints at once, select '**Data→Edit 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 '**Editing All Breakpoints**', above, for details. As an additional feature, you can click on '**Print ()**' to see the current value of a watched variable.

Deleting Watchpoints

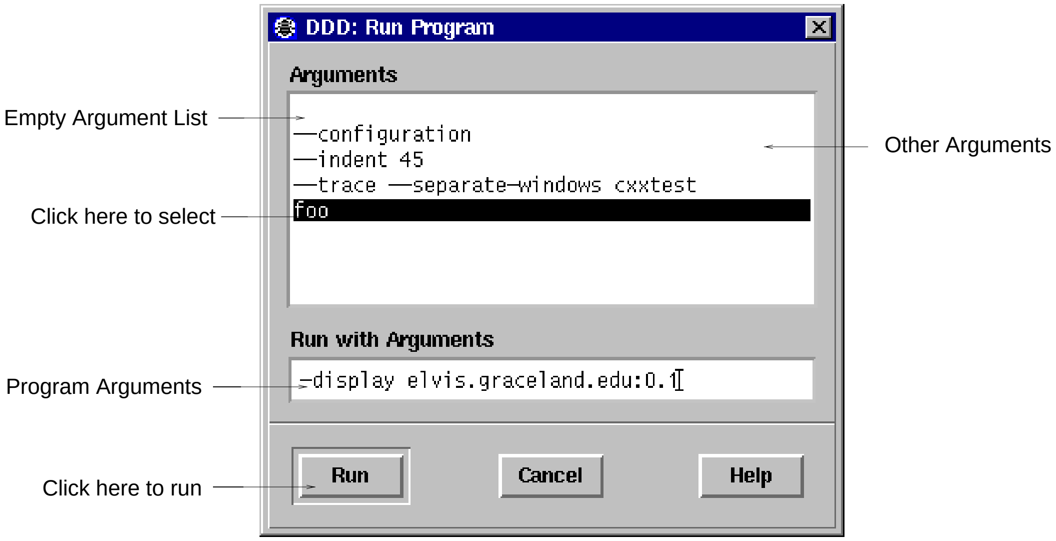
To delete a watchpoint, enter the name of the watched variable in the argument field and click the '**Unwatch ()**' button.

RUNNING THE PROGRAM

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.

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.



```
--configuration
--indent 45
--trace --separate-windows cxxtest
foo
```

```
-display elvis.graceland.edu:0.1
```

Starting a Program with Arguments

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. To activate the execution window, select ‘**Program→Run in Execution Window**’.

While the execution window is active, DDD invokes your program such that its standard input, output, and error streams are redirected 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 a 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).

The execution window is not available in JDB and Perl.

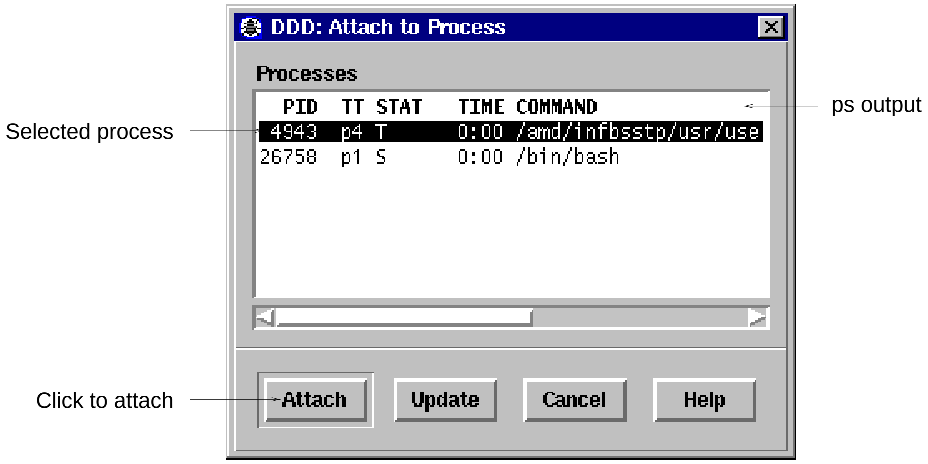
Attaching to a Running Process

If the debugged program is already running in some process, you can *attach* to this process (instead of starting a new one with ‘**Run**’). Select ‘**File→Attach to Process**’ to choose from a list of processes. Afterwards, press the ‘**Attach**’ button to attach to the specified process.

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**’.



```
  PID  TT STAT   TIME COMMAND
 4943  p4 T      0:00 /amd/infbsstp/usr/use
26758  p1 S      0:00 /bin/bash
```

Selecting a Process to Attach

You can customize the list of processes shown by defining an alternate command to list processes. See ‘**Edit→Preferences→Helpers→List Processes**’.

Note: JDB, PYDB, and Perl do not support attaching the debugger to running processes.

Stopping the Program

The program stops as soon as a breakpoint is reached. The current execution position is highlighted by an arrow.

You can interrupt a running program any time by clicking the ‘**Interrupt**’ button or typing **ESC** in a DDD window.

Resuming Execution

To resume execution, at the address where your program last stopped, click on the ‘**Continue**‘ button. Any breakpoints set at that address are bypassed.

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.

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.

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.

To continue running until the current function returns, use the ‘**Finish**’ button. The returned value (if any) is printed.

To continue running until a line after the current source line is reached, use the ‘**Continue Until Here**’ facility from the line popup menu. See the ‘**Temporary Breakpoints**’ section, above, for a discussion.

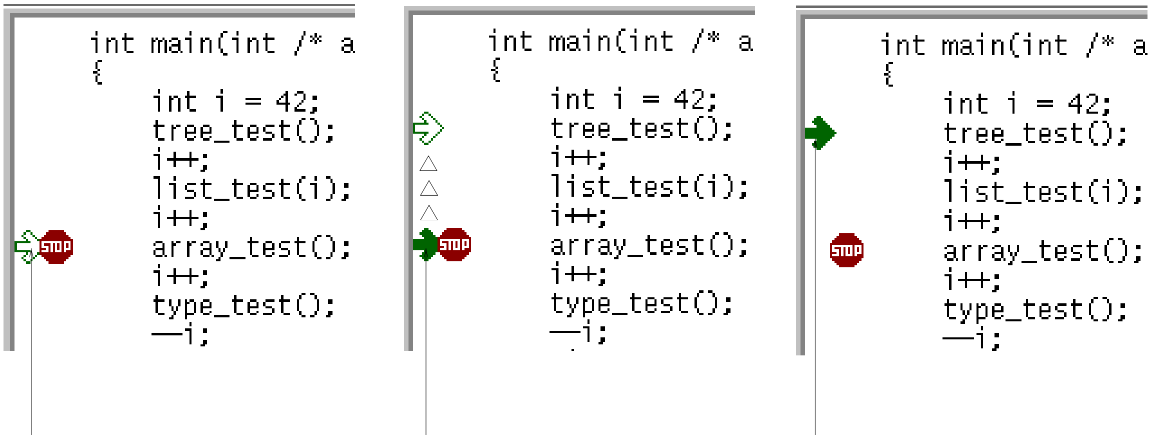
Altering the Execution Position

To resume execution at a different location, press *mouse button 1* on the arrow and drag it to a different location. 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.

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 the ‘**Continue**’ button to resume execution.



```
int main(int /* a
{
    int i = 42;
    tree_test();
    i++;
    list_test(i);
    i++;
    array_test();
    i++;
    type_test();
    --i;
```

Click on arrow, ...

```
int main(int /* a
{
    int i = 42;
    tree_test();
    i++;
    list_test(i);
    i++;
    array_test();
    i++;
    type_test();
    --i;
```

... hold mouse button and move ...

```
int main(int /* a
{
    int i = 42;
    tree_test();
    i++;
    list_test(i);
    i++;
    array_test();
    i++;
    type_test();
    --i;
```

... to the final position.

Changing the Execution Position by Dragging the Execution Arrow

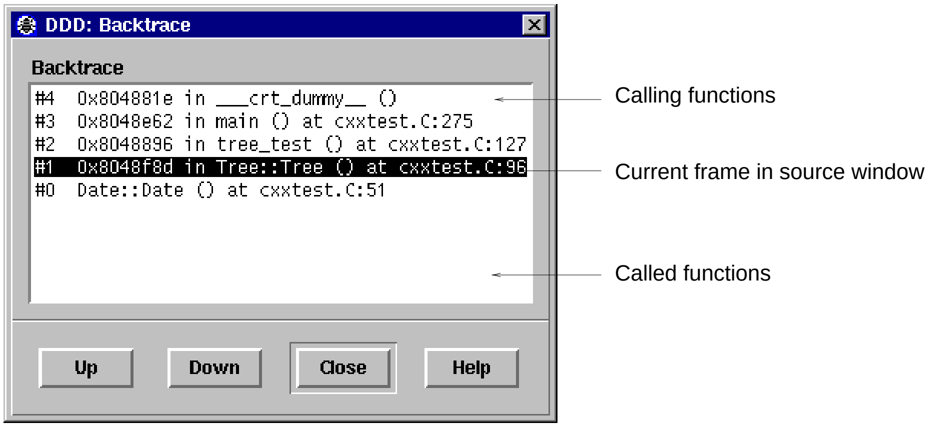
Note: Dragging the execution position is not possible when glyphs are disabled. Use ‘**Set Execution Position**’ from the breakpoint popup menu instead to set the execution position to the current location. This item is also accessible by pressing and holding the ‘**Break at ()**/**Clear at ()**’ button.

Note: JDB does not support altering the execution position.

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

DDD provides a *backtrace window* showing a summary of how your program got where it is. To enable the backtrace window, select ‘**Status→Backtrace**’.



```
#4  0x804881e in ___crt_dummy__ ()
#3  0x8048e62 in main () at cxxtest.C:275
#2  0x8048896 in tree_test () at cxxtest.C:127
#1  0x8048f8d in Tree::Tree () at cxxtest.C:96
#0  Date::Date () at cxxtest.C:51
```

Calling functions

Current frame in source window

Called functions

Selecting a Frame from the Backtrace Viewer

The ‘**Up**’ button selects the function that called the current one.

The ‘**Down**’ button selects the function that was called by the current one.

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**’.

“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 ‘**Breakpoint Commands**’, above, for details. 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.

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*

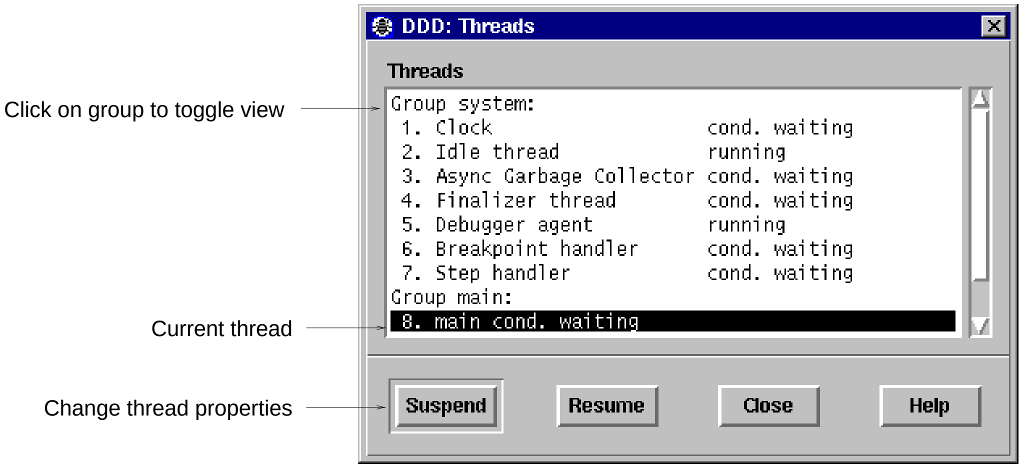
This returns a list of all previously recorded values of the variable *name*, using array syntax. Note that *name* must have been displayed at earlier program stops in order to record values.

Examining Threads

Note: Thread support is available with GDB and JDB only.

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.



```
Group system:
 1. Clock                    cond. waiting
 2. Idle thread              running
 3. Async Garbage Collector cond. waiting
 4. Finalizer thread         cond. waiting
 5. Debugger agent           running
 6. Breakpoint handler       cond. waiting
 7. Step handler             cond. waiting
Group main:
 8. main cond. waiting
```

Click on group to toggle view

Current thread

Change thread properties

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:

- 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.

Handling Signals

Note: Signal support is available with GDB only.

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.

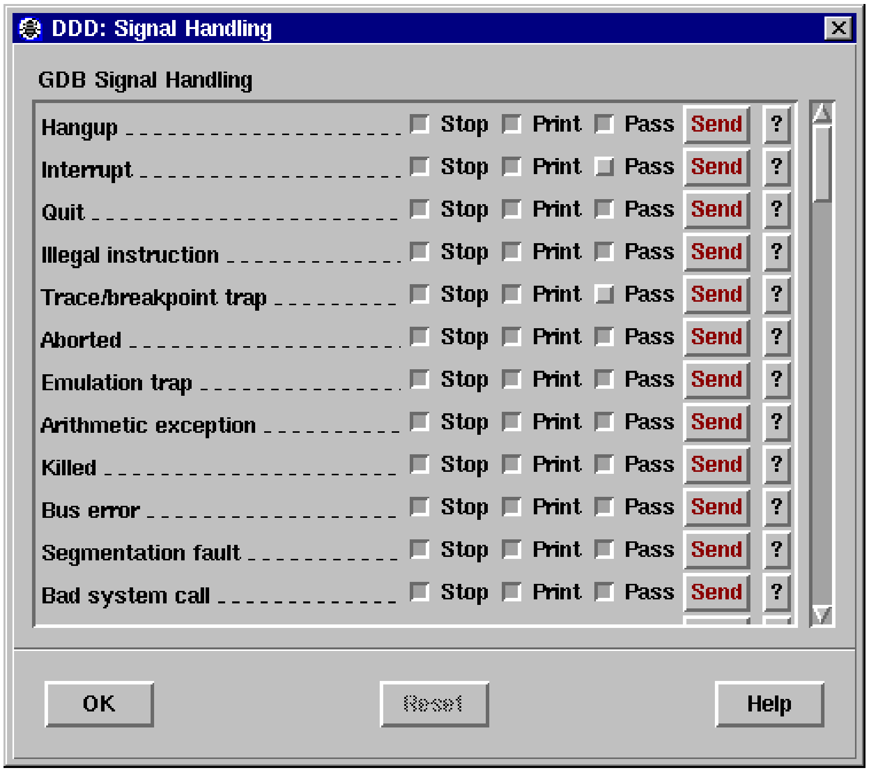
Normally, DDD is set up to ignore non-erroneous signals like **SIGALRM** (so as not to interfere with their role in the functioning of your program) but to stop your program immediately whenever an error signal happens. In DDD, you can change these settings via ‘**Status**→**Signals**’.

‘**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.
If unset, GDB should not stop your program when this signal happens. It may still print a message telling you that the signal has come in.

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.



Hangup	☑ Stop	☑ Print	☑ Pass	Send	?
Interrupt	☑ Stop	☑ Print	□ Pass	Send	?
Quit	☑ Stop	☑ Print	☑ Pass	Send	?
Illegal instruction	☑ Stop	☑ Print	☑ Pass	Send	?
Trace/breakpoint trap	☑ Stop	☑ Print	□ Pass	Send	?
Aborted	☑ Stop	☑ Print	☑ Pass	Send	?
Emulation trap	☑ Stop	☑ Print	☑ Pass	Send	?
Arithmetic exception	☑ Stop	☑ Print	☑ Pass	Send	?
Killed	☑ Stop	☑ Print	☑ Pass	Send	?
Bus error	☑ Stop	☑ Print	☑ Pass	Send	?
Segmentation fault	☑ Stop	☑ Print	☑ Pass	Send	?
Bad system call	☑ Stop	☑ Print	☑ Pass	Send	?

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**'.

'**Edit→Save Options**' does not save changed signal settings, 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**'.

EXAMINING DATA

DDD provides several means to examine data.

Value Hints

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 shows the variable value. This is useful for quick examination of several simple variables.

Printing Values

If you want to reuse variable values at a later time, you can print the value in the debugger console. This allows for displaying and examining larger data structures.

Displaying Values

If you want to examine complex data structures, you can display them graphically in the data display. Displays remain effective until you delete them; they are updated each time the program stops. This is useful for large dynamic structures.

Plotting Values

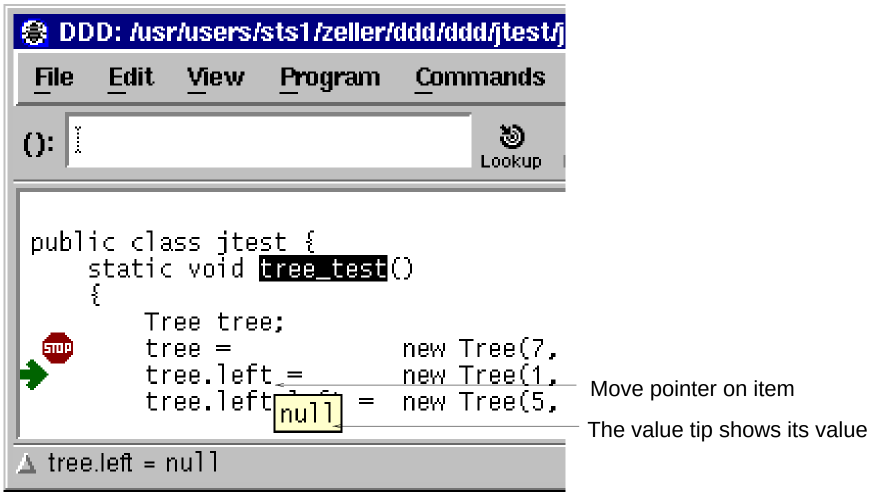
If you want to examine arrays of numeric values, you can plot them graphically in a separate plot window. The plot is updated each time the program stops. This is useful for large numeric arrays.

Memory Dumps

This feature, available using GDB only, allows you to dump memory contents in any of several formats, independently of your program's data types. This is described under '**MACHINE-LEVEL DEBUGGING**', below.

Showing Simple Values using Value Hints

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.



```
public class jtest {
    static void tree_test()
    {
        Tree tree;
        tree =                  new Tree(7,
        tree.left =             new Tree(1,
        tree.left          =    new Tree(5,
```

tree.left = null

Move pointer on item

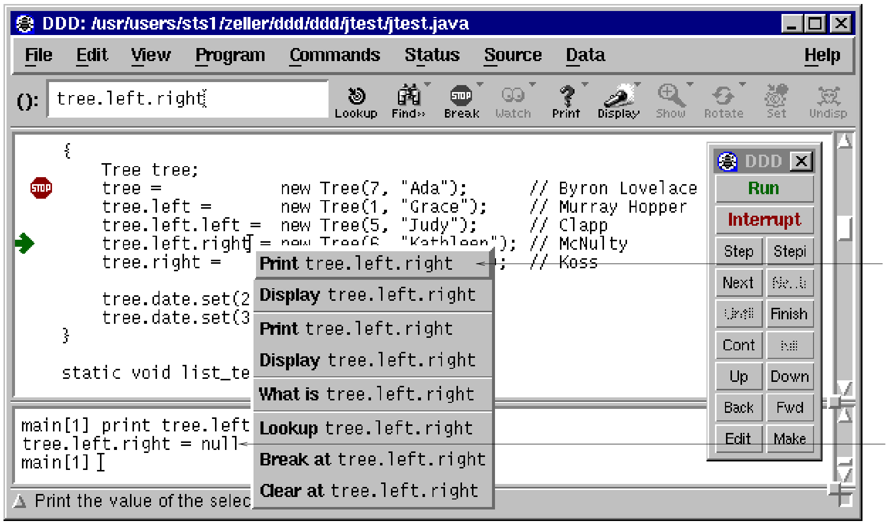
The value tip shows its value

Displaying Simple Values using Value Tips

Printing Simple Values in the Debugger Console

The variable value can also be printed in the debugger console, making it available for further 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. Note that the 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.



```
    {
        Tree tree;
        tree =             new Tree(7, "Ada");       // Byron Lovelace
        tree.left =        new Tree(1, "Grace");     // Murray Hopper
        tree.left.left =   new Tree(5, "Judy");      // Clapp
        tree.left.right = new Tree(6, "Kathleen"); // McNulty
        tree.right =                               ; // Koss

        tree.date.set(2
        tree.date.set(3
    }

    static void list_te
```

```
main[1] print tree.left
tree.left.right = null
main[1]
```

Print the value of the selec

Select ‘Print’ on item

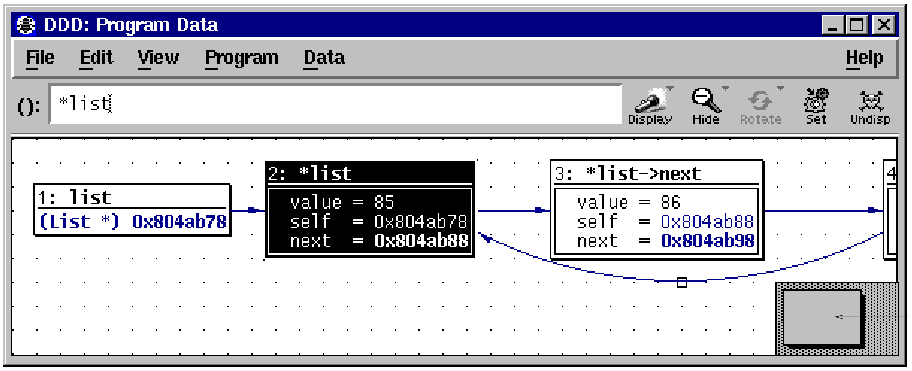
The value is printed in the debugger console

Displaying Simple Values in the Debugger Console

Displaying Complex Values in the Data Window

To explore complex data structures, you can use the *graphical data display* in the *data window*. The data window holds *displays* showing names and the values of variables. The display is updated each time the program stops.

To create a new display, select the desired 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.


DDD: Program Data
File Edit View Program Data Help
(): *list
Display Hide Rotate Set Undisp
4

```
1: list
(List *) 0x804ab78

2: *list
value = 85
self  = 0x804ab78
next  = 0x804ab88

3: *list->next
value = 86
self  = 0x804ab88
next  = 0x804ab98
```

Scroll the data display

Displaying Data

As a shorter alternative, you can simply press *mouse button 3* on the variable name and select the ‘**Display**’ item 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 enter

graph display *expr* [**clustered**] [**at (***x*, *y***)**] [**dependent on** *display*] [[**now or**] **when in** *scope*]

at the debugger prompt. The options have the following meaning:

- If the suffix '**clustered**' is specified, the new data display is created in a cluster. See '**Clustering Displays**', below, for a discussion.
- If the suffix '**at (***x*, *y***)**' is specified, the new data display is created at the position (*x*, *y*). Otherwise, a default position is assigned.
- If the suffix '**dependent on** *display*' is given, an edge from the display numbered or named *display* to the new display is created. Otherwise, no edge is created.
- If the suffix '**when in** *scope*' is given, display creation is *deferred* until execution reaches the given *scope* (a function name, as in the backtrace output).
- If the suffix '**now or when in** *scope*' is given, DDD attempts to create the display immediately. If display creation fails, it is *deferred* until execution reaches the given *scope* (a function name, as in the backtrace output).
- If no '**when in**' suffix or '**now or when in**' suffix is given, the display is created immediately.

If you created a display by mistake, use '**Edit**→**Undo**' to undisplay it.

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 individual displays by clicking on them with *mouse button 1*. The resulting expression is shown in the *argument field*, below.

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.

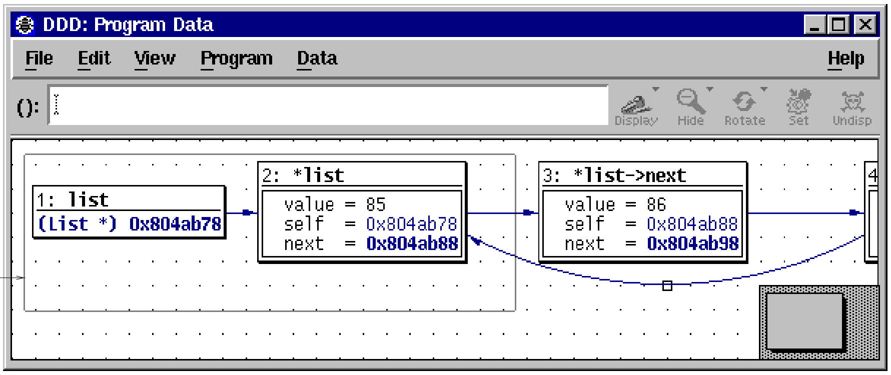
Single displays may also be selected by using the arrow keys.

Selecting Multiple Displays

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.



```
1: list
(List *) 0x804ab78

2: *list
value = 85
self  = 0x804ab78
next  = 0x804ab88

3: *list->next
value = 86
self  = 0x804ab88
next  = 0x804ab98
```

Selection rectangle

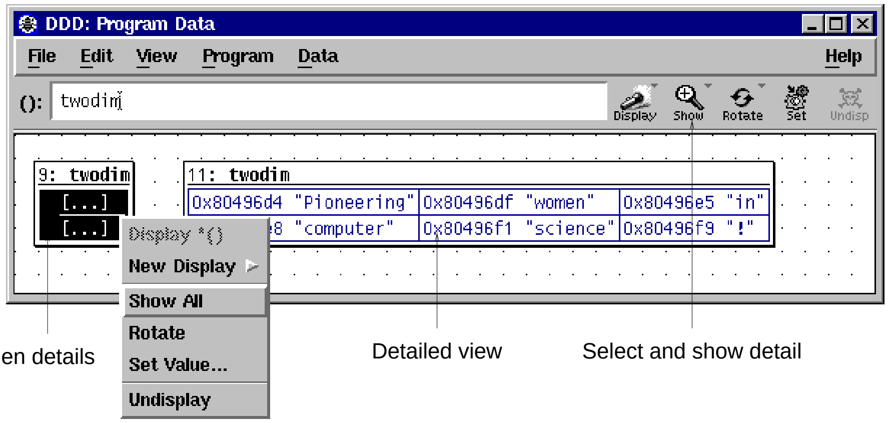
Selecting Multiple Displays

Showing and Hiding Values

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.



```
(): twodim

9: twodim
[...]
[...]

11: twodim
0x80496d4 "Pioneering"  0x80496df "women"    0x80496e5 "in"
8 "computer"            0x80496f1 "science"  0x80496f9 "!"
```

Hidden details

Detailed view

Select and show detail

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 the DDD commands

graph disable display *displays...*

and

graph enable display *displays...*

at the debugger prompt. *displays...* is either

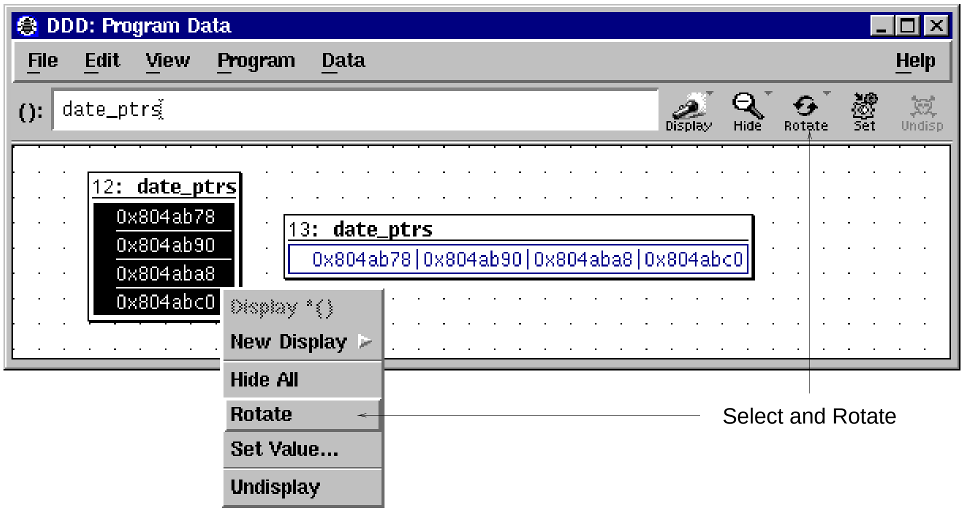
- a space-separated list of display numbers to disable or enable, or
- a single display name. If you specify a display by name, all displays with this name will be affected.

Use '**Edit→Undo**' to undo disabling or enabling displays.

Rotating Arrays

Arrays can be aligned horizontally or vertically. To change the alignment of an array, 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 the '**Rotate**' menu item.



(): date_ptrs

```
12: date_ptrs
 0x804ab78
 0x804ab90
 0x804aba8
 0x804abc0

13: date_ptrs
 0x804ab78|0x804ab90|0x804aba8|0x804abc0
```

Select and Rotate

Rotating an Array

Displaying Dependent Values

Dependent displays are created from an existing display. The dependency is indicated by arrows 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 ‘**Examining Shared Data Structures**’, below, for details on how to examine such structures.

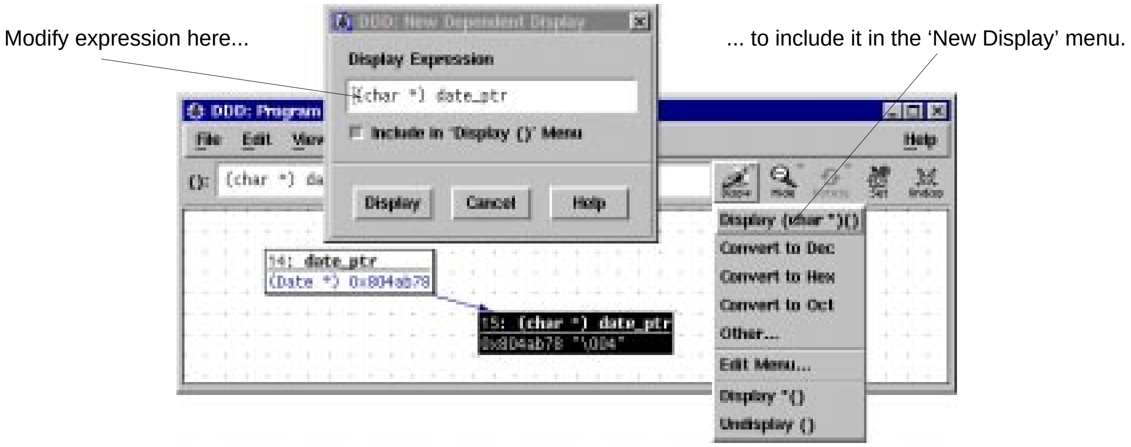
Display Shortcuts

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.



Modify expression here...

... to include it in the ‘New Display’ menu.

Display Expression: (char *) date_ptr

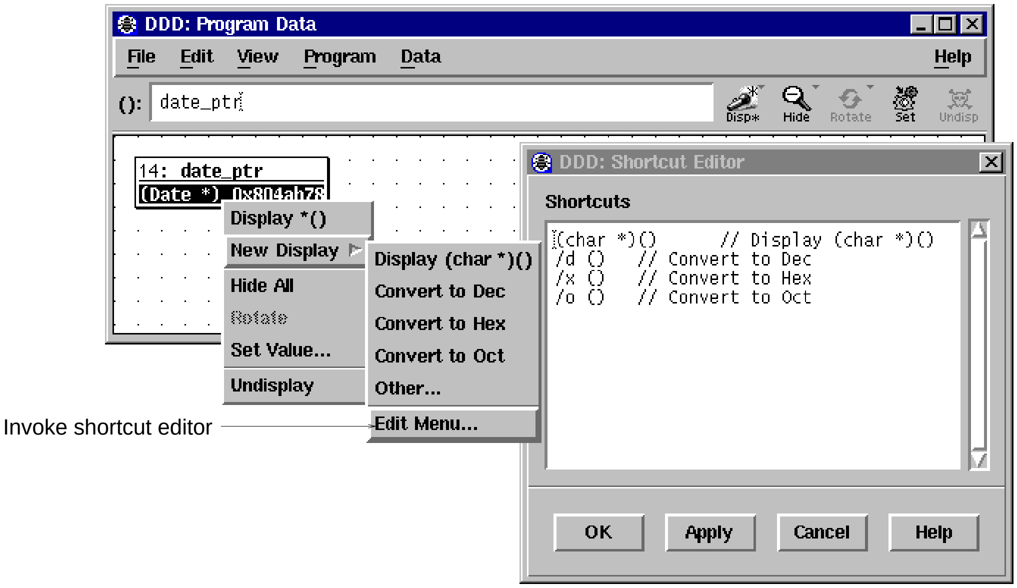
(): (char *) da

```
14: date_ptr
(Date *) 0x804ab78
```

```
15: (char *) date_ptr
0x804ab78 "\004"
```

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.



```
(): date_ptr
```

```
14: date_ptr
(Date *) 0x804ab78
```

```
(char *)()      // Display (char *)()
/d ()   // Convert to Dec
/x ()   // Convert to Hex
/o ()   // Convert to Oct
```

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.

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 '**Display *()**' 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.

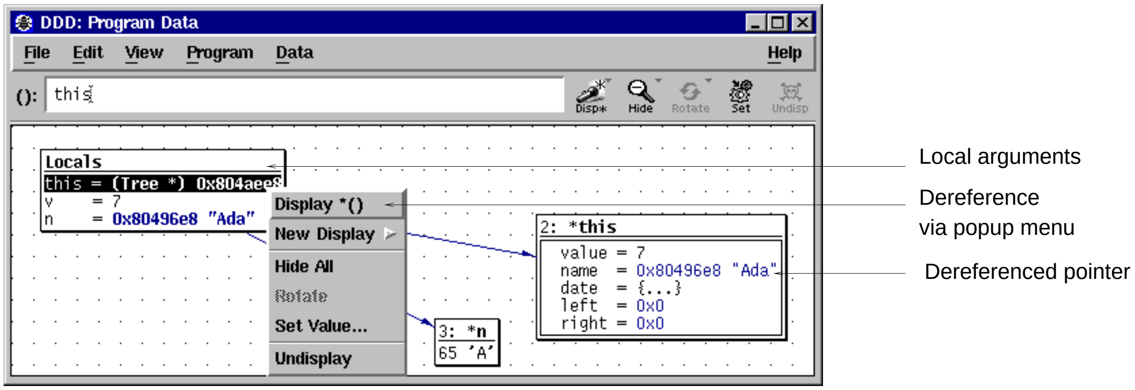
As an even faster alternative, you can also double-click *mouse button 1* on the originating pointer value or name. If you press **Ctrl** while double-clicking, the display will be dereferenced *in place*--that is, it will be replaced by the dereferenced display.

The '**Display *()**' function is also accessible by pressing and holding the '**Display ()**' button.

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 '**Display Arguments**'.

The display showing the local variables can be manipulated just like any other data display. Individual variables can be selected and dereferenced.



```
(): this

Locals
this = (Tree *) 0x804aee8
v    = 7
n    = 0x80496e8 "Ada"

2: *this
value = 7
name  = 0x80496e8 "Ada"
date  = {...}
left  = 0x0
right = 0x0

3: *n
65 'A'
```

Dereferencing a Local Variable

Displaying Program Status

You can create a display from the output of an arbitrary debugger command. By entering

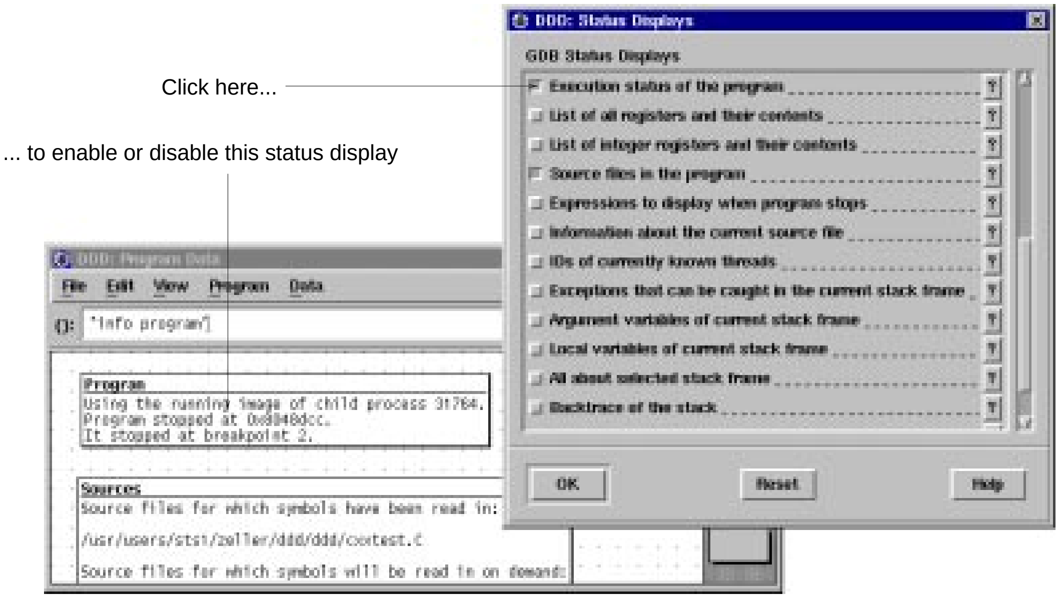
graph display ‘*command***‘**

the output of *command* is turned into a *status display* updated each time the program stops. For instance,

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→More Status Displays**’ and pick your choice from the list.



GDB Status Displays

- ☑ Execution status of the program
- ☐ List of all registers and their contents
- ☐ List of integer registers and their contents
- ☑ Source files in the program
- ☐ Expressions to display when program stops
- ☐ Information about the current source file
- ☐ IDs of currently known threads
- ☐ Exceptions that can be caught in the current stack frame
- ☐ Argument variables of current stack frame
- ☐ Local variables of current stack frame
- ☐ All about selected stack frame
- ☐ Backtrace of the stack

```
(): 'info program'

Program
Using the running image of child process 31764.
Program stopped at 0x8048dcc.
It stopped at breakpoint 2.

Sources
Source files for which symbols have been read in:

/usr/users/st51/zeller/ddd/ddd/cxxtest.C

Source files for which symbols will be read in on demand:
```

Activating Status Displays

Status displays consume time; you should delete them as soon as you don’t need them any more.

Displaying Multiple Array Values

When debugging C or C++ programs, one often has to deal with pointers to arrays of dynamically determined size. Both DDD and GDB provide special support for such dynamic arrays.

To display several successive objects of the same type (a section of an array, or an array of dynamically determined size), use the notation **[***FROM***..***TO***]** in display expressions. *FROM* and *TO* denote the first and last array position to display. Thus,

graph display argv[0..9]

creates ten new display nodes for '**argv[0]**', '**argv[1]**', ..., '**argv[9]**'.

Using GDB as inferior debugger, you can use *artificial arrays*. Typing

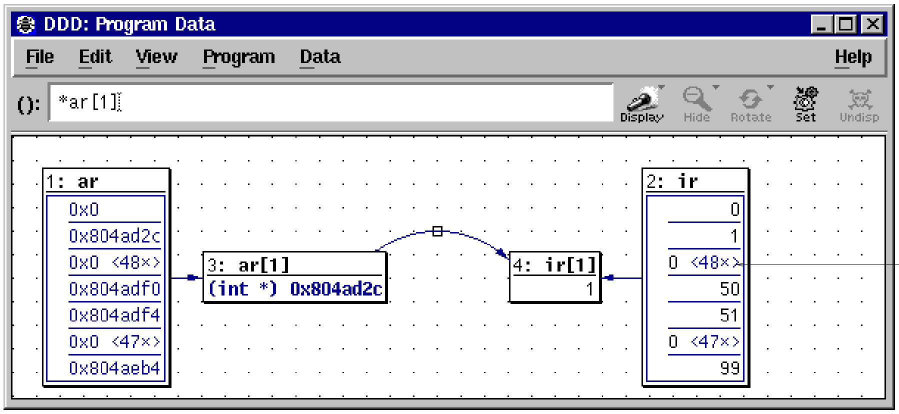
graph display argv[0] @ 10

creates a single array display node containing '**argv[0]**' up to '**argv[9]**'. Generally, by using the '**@**' operator, you can specify the number of array elements to be displayed.

For more details on artificial arrays, see the GDB documentation.

Repeated Array Values

Using GDB, an array value that is repeated 10 or more times is displayed only once. The value is shown with a '<*N*×>' postfix added, where *N* is the number of times the value is repeated. Thus, the display '**0x0 <30×>**' stands for 30 array elements, each with the value **0x0**. This saves a lot of display space, especially with homogeneous arrays.



Repeated Value

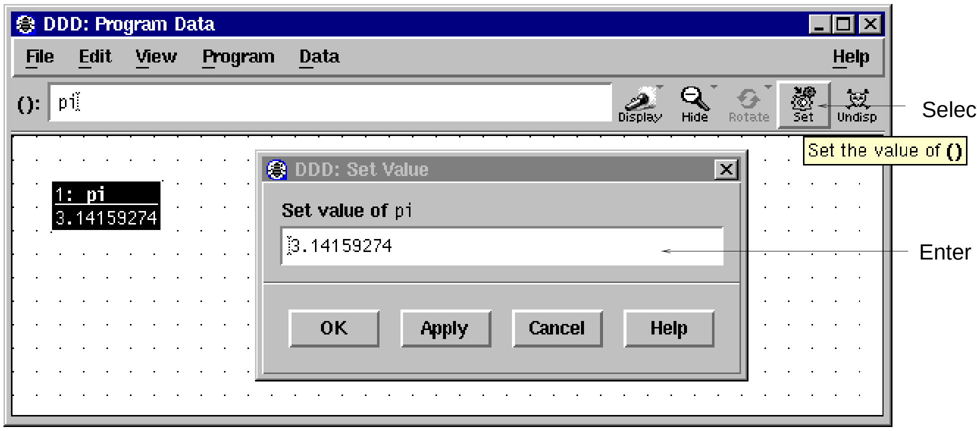
Displaying Repeated Array Values

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, regardless of GDB settings; see the '**expandRepeatedValues**' resource for details.

Altering Variable Values

Using the '**Set ()**' button or the '**Set Value**' menu item in the data popup menu, you can alter the value of the selected variable, to resume execution with the changed value. In a dialog, you can modify the variable value at will; clicking the '**OK**' or '**Apply**' button commits your change.



(): pi

1: pi
3.14159274

Set value of pi

3.14159274

Select to set

Enter new value here

Changing Variable Values

If you made a mistake, you can use ‘**Edit→Undo**’ to re-set the variable to its previous value.

Note: Altering variable values is not supported in JDB.

Refreshing the Data Window

The data window refreshes itself automatically each time the program stops. Values that have changed 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 Display**’ item.

Typing

graph refresh

at the debugger prompt has the same effect.

Deleting Displays

To delete a single display, select it and click on the ‘**Delete ()**’ button. As an alternative, you can also press *mouse button 3* on the display and select the ‘**Delete Display**’ item.

When a display is deleted, its immediate ancestors and descendants are automatically selected, so that you can easily delete entire graphs.

To delete several displays at once, select the ‘**Delete**’ item in the Display Editor (invoked via ‘**Data→Edit Displays**’). Select any number of display items in the usual way and delete them by pressing ‘**Delete**’.

As an alternative, you can also type

graph undisplay *displays...*

at the debugger prompt. *displays...* is either

- a space-separated list of display numbers to delete, or
- a single display name. If you specify a display by name, all displays with this name will be deleted.

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.

Examining Shared Data Structures

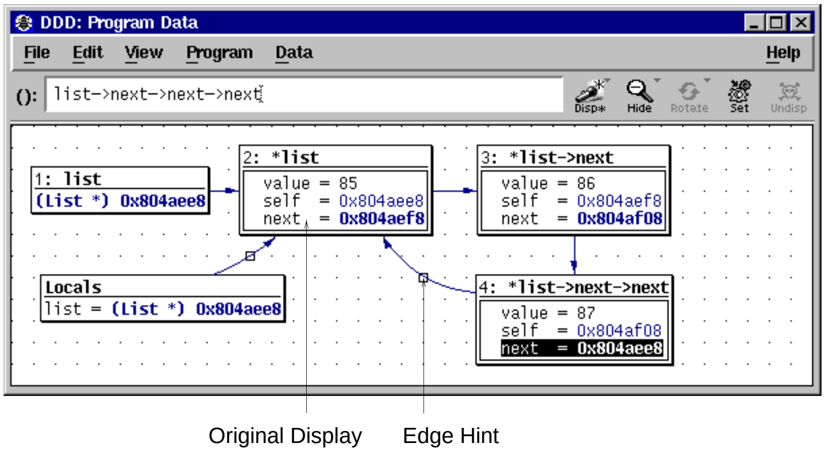
By default, DDD does not recognize shared data structures—that is, a data object referenced by multiple other data objects. For instance, if two pointers **p1** and **p2** point at the same data object **d**, the data displays **d**, ***p1**, and ***p2** will be separate, although they denote the same 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 the ‘**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 ‘**Undisplay ()**’.



```
list->next->next->next

1: list
(List *) 0x804aee8

2: *list
  value = 85
  self  = 0x804aee8
  next  = 0x804aef8

3: *list->next
  value = 86
  self  = 0x804aef8
  next  = 0x804af08

Locals
list = (List *) 0x804aee8

4: *list->next->next
  value = 87
  self  = 0x804af08
  next  = 0x804aee8
```

Examining Shared Data Structures

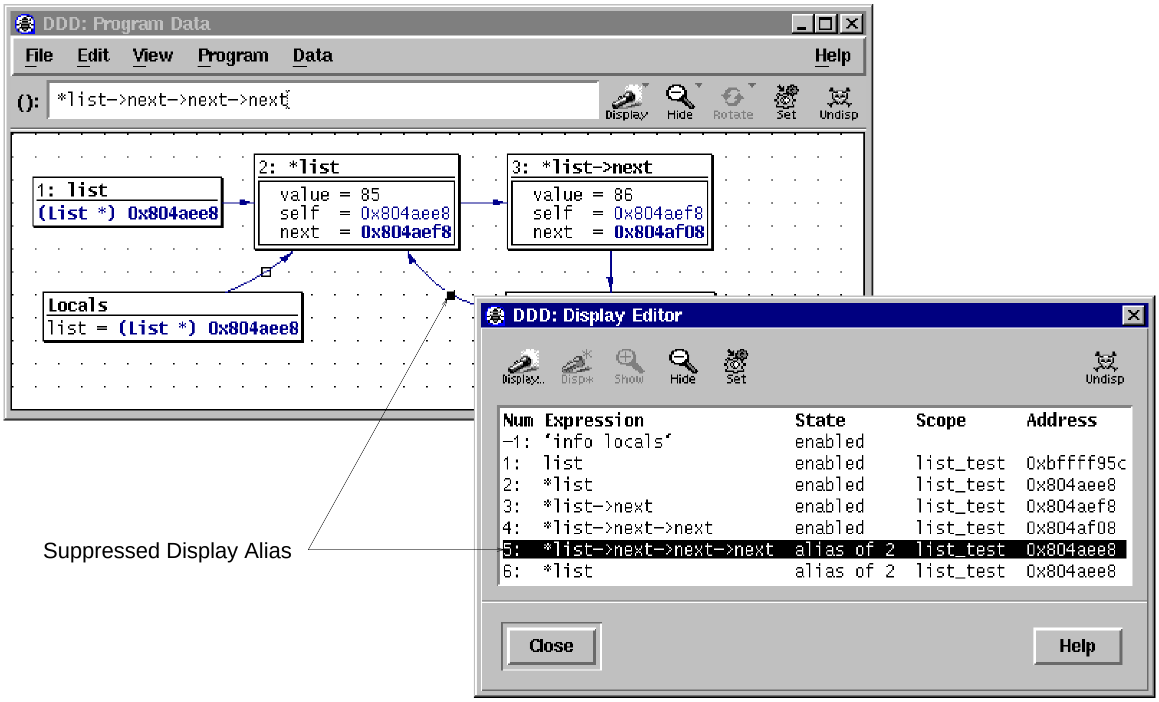
To access suppressed display aliases, you can also use the Display Editor. Suppressed displays are listed in the Display Editor as *aliases* of the original data display. Via the Display Editor, you can select, change, and delete suppressed displays.

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 and C++ 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 it is disabled by default. 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.



Num	Expression	State	Scope	Address
-1:	'info locals'	enabled		
1:	list	enabled	list_test	0xbffff95c
2:	*list	enabled	list_test	0x804aee8
3:	*list->next	enabled	list_test	0x804aef8
4:	*list->next->next	enabled	list_test	0x804af08
5:	*list->next->next->next	alias of 2	list_test	0x804aee8
6:	*list	alias of 2	list_test	0x804aee8

Suppressed Display Alias

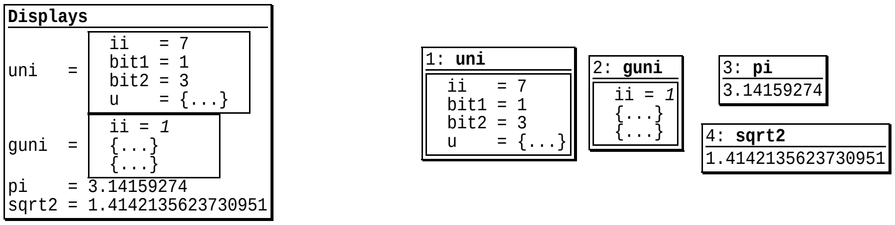
The Display Editor

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→Cluster Data Displays**'.



```
Displays
        ii   = 7
uni   = bit1 = 1
        bit2 = 3
        u    = {...}
        ii = 1
guni  = {...}
        {...}
pi    = 3.14159274
sqrt2 = 1.4142135623730951
```

```
1: uni              2: guni      3: pi
 ii   = 7            ii = 1      3.14159274
 bit1 = 1            {...}
 bit2 = 3            {...}      4: sqrt2
 u    = {...}                   1.4142135623730951
```

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→Cluster Data Displays**'.

Moving Displays Around

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.

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. See '**CUSTOMIZING DDD**', below, for details on how to set up scroll bars or panners.

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.

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.

Aligning Displays

You can align all displays on the nearest grid position by selecting '**Data→Align on Grid**'. This is useful for keeping edges 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.

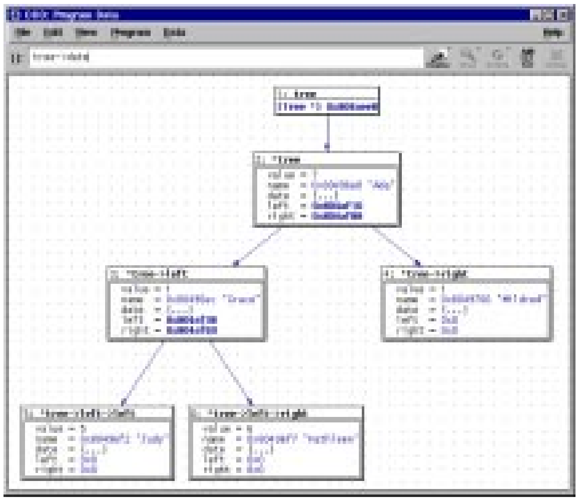
Layouting the Display Graph

You can layout the entire graph as a tree by selecting '**Data→Layout Graph**'.

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.



A Layouted Graph (with Compact Layout)

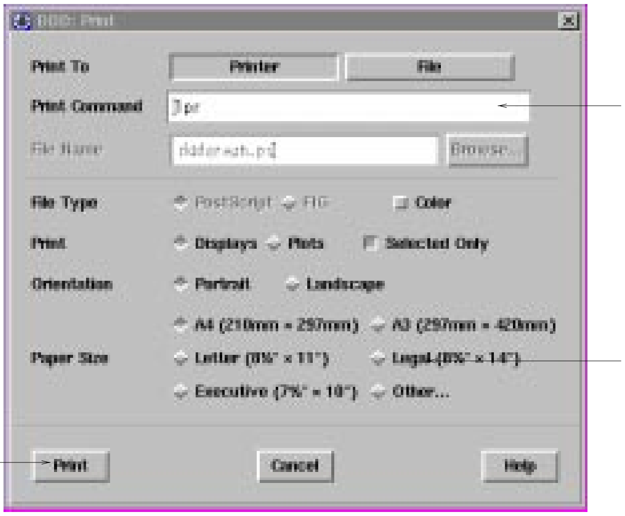
Rotating the Display Graph

You can rotate the entire graph clockwise by 90 degrees by selecting ‘**Data**→**Rotate Graph**’.

If the graph was previously layouted, you may need to layout it again. Subsequent layouts will respect the direction of the last rotation.

Printing the Display Graph

DDD allows for printing the graph picture on PostScript printers or into files. This is useful for documenting program states.



DDD: Print

Print To: Printer | File

Print Command: lpr

File Name: dddgraph.ps

File Type: ● PostScript ○ FIG □ Color

Print: ● Displays ○ Plots □ Selected Only

Orientation: ● Portrait ○ Landscape

Paper Size: ● A4 (210mm × 297mm) ○ A3 (297mm × 420mm)
○ Letter (8½" × 11") ○ Legal (8½" × 14")
○ Executive (7½" × 10") ○ Other...

Enter print command

Select paper size

Click to print

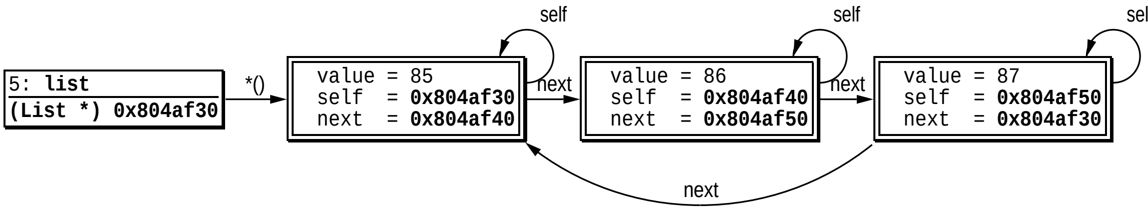
Printing displays

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, LaTeX, PIC), using the TRANSFIG or FIG2DEV package.



```
5: list
(List *) 0x804af30

value = 85
self  = 0x804af30
next  = 0x804af40

value = 86
self  = 0x804af40
next  = 0x804af50

value = 87
self  = 0x804af50
next  = 0x804af30
```

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.

PLOTTING DATA

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.

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 an array, select it 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.

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.

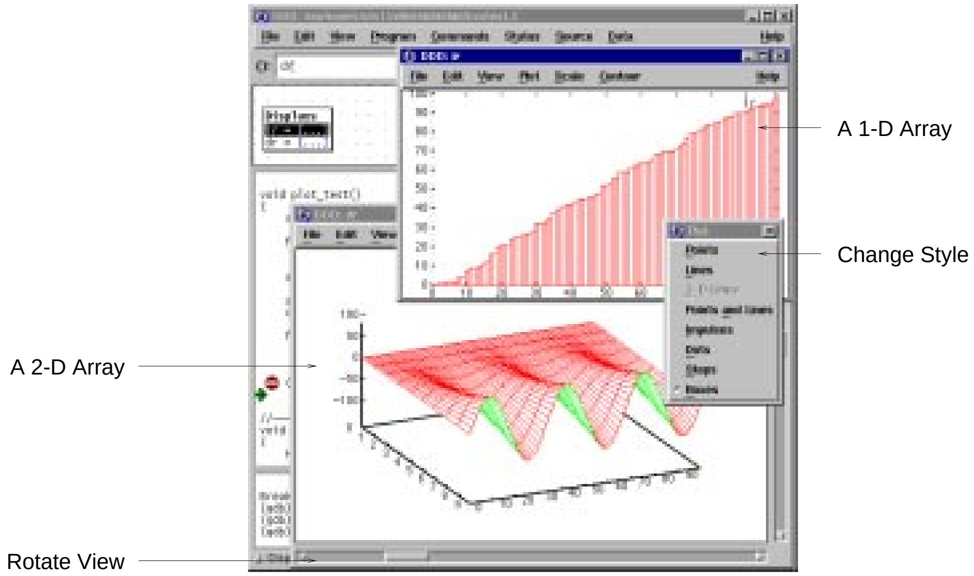
Changing the Plot Appearance

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.

You can also resize the plot window as desired.

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

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 '**Clustering Displays**', above, for details on clusters.) A common scenario is to plot a one-dimensional array together with the current index position. This is done in three steps:

- Display the array and the index, using '**Display ()**'.
- Cluster both displays: select them and choose '**Undisp→Cluster ()**'.
- 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 alignment by selecting the scalar display, followed by '**Rotate ()**'.

Plotting Display Histories

At each program stop, DDD records the values of all displayed variables. 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.

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 PostScript and X11 drivers 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.)

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]
```

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 LaTeX 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
```

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 ‘**$HOME/.ddd/log**’. The DDD ‘**−−trace**’ option logs this interaction on standard output.

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.

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. 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 topped for good, you can use ‘**Undo**’ and ‘Redo’ to redisplay and examine previous program states.

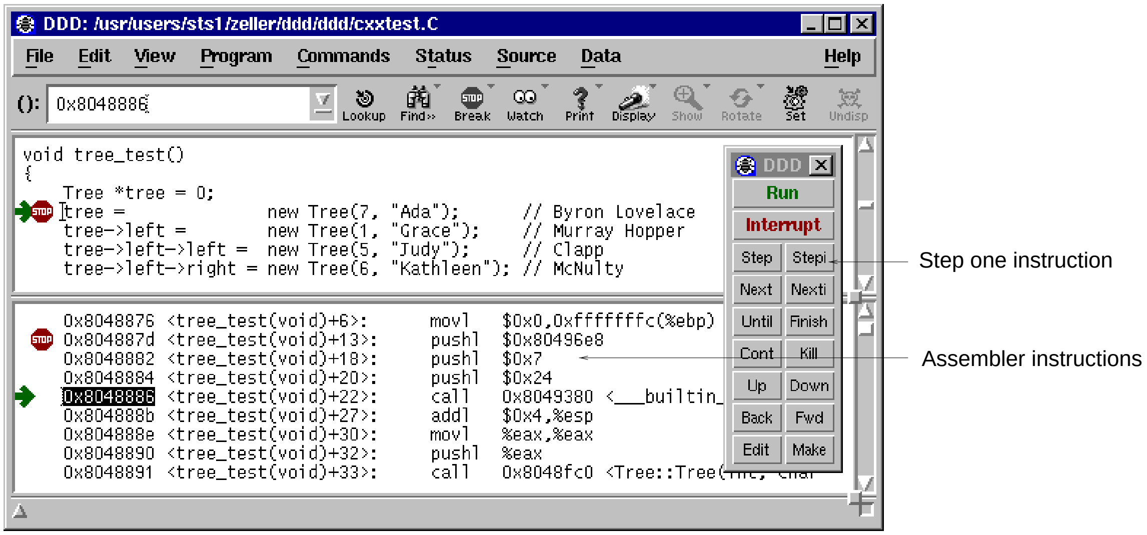
MACHINE-LEVEL DEBUGGING

Note: Machine-level support is available with GDB only.

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.

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. 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.



```
void tree_test()
{
    Tree *tree = 0;
    tree =                 new Tree(7, "Ada");      // Byron Lovelace
    tree->left =           new Tree(1, "Grace");    // Murray Hopper
    tree->left->left =     new Tree(5, "Judy");     // Clapp
    tree->left->right =    new Tree(6, "Kathleen"); // McNulty
```

```
0x8048876 <tree_test(void)+6>:     movl   $0x0,0xfffffffc(%ebp)
0x804887d <tree_test(void)+13>:    pushl  $0x80496e8
0x8048882 <tree_test(void)+18>:    pushl  $0x7
0x8048884 <tree_test(void)+20>:    pushl  $0x24
0x8048886 <tree_test(void)+22>:    call   0x8049380 <___builtin_
0x804888b <tree_test(void)+27>:    addl   $0x4,%esp
0x804888e <tree_test(void)+30>:    movl   %eax,%eax
0x8048890 <tree_test(void)+32>:    pushl  %eax
0x8048891 <tree_test(void)+33>:    call   0x8048fc0 <Tree::Tree(
```

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 at ()**’ or ‘**Clear at ()**’ 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.

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.

If source code is not available, only the machine code window is updated.

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:

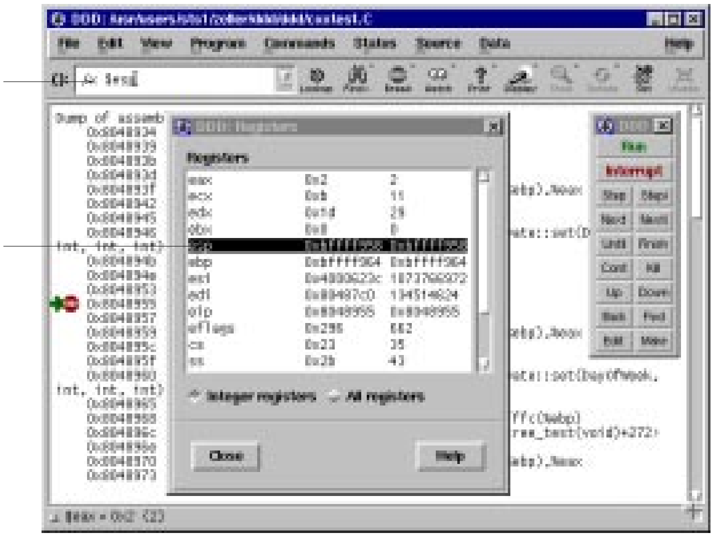
To execute just one machine instruction, click on the ‘**Stepi**’ button.

To continue to the next instruction in the current function, click on the ‘**Nexti**’ button. This is similar to ‘**Stepi**’, but any subroutine calls are executed without stopping.

Registers

DDD provides a *register window* showing the machine register values after each program stop. To enable the register window, select ‘**Status**→**Registers**’.

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.



The register name is copied to ()

Select register

```
eax       0x2          2
ecx       0xb          11
edx       0x1d         29
ebx       0x8          8
esp       0xbffff958   0xbffff958
ebp       0xbffff964   0xbffff964
esi       0x4000623c   1073766972
edi       0x80497c0    134514624
eip       0x8048955    0x8048955
eflags    0x296        662
cs        0x23         35
ss        0x2b         43
```

```
$eax = 0x2 (2)
```

Displaying Register Values

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**→**Examine Memory**’ pops up a panel where you can choose the format to be shown.

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.

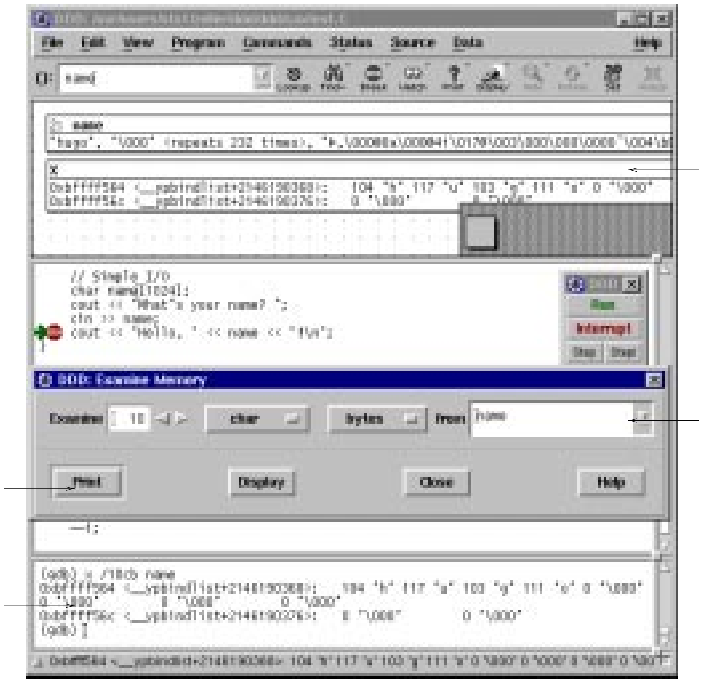
- a *unit size*—one of

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**, 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.


File Edit View Program Commands Status Source Data Help
(): name
Run
Interrupt
DDD: Examine Memory
Examine 10 char bytes from name
Print Display Close Help

```
name
"hugo", '\000' <repeats 232 times>, ...
```

```
x
0xbffff564 <__ypbindlist+2146190368>:  104 'h' 117 'u' 103 'g' 111 'o' 0 '\000'
0xbffff56c <__ypbindlist+2146190376>:  0 '\000'
```

```
// Simple I/O
char name[1024];
cout << "What's your name? ";
cin >> name;
cout << "Hello, " << name << "\n";
}
```

```
--i;
```

```
(gdb) x /10cb name
0xbffff564 <__ypbindlist+2146190368>:  104 'h' 117 'u' 103 'g' 111 'o' 0 '\000'
0 '\000'        0 '\000'        0 '\000'
0xbffff56c <__ypbindlist+2146190376>:  0 '\000'        0 '\000'
(gdb)
```

Memory Dump
as Status Display

Enter address here

Click here to print...

... in the GDB console

Examining Memory

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**'.

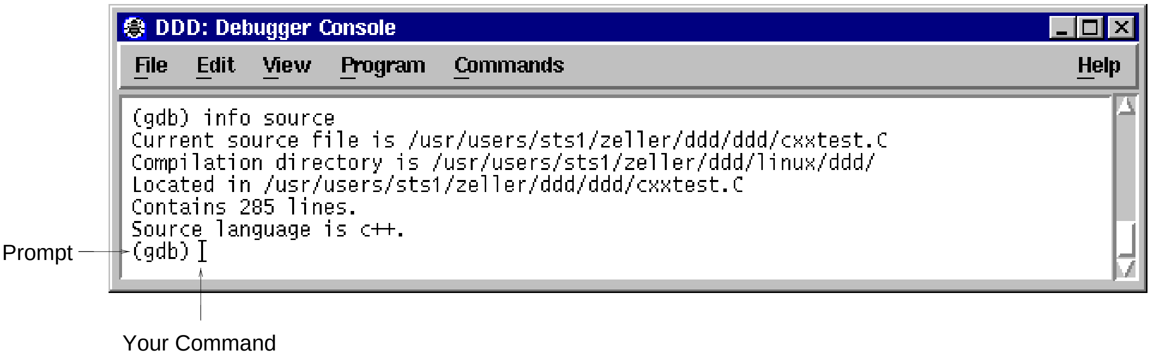
By default, DDD tries a number of common editors. You can customize DDD to use your favourite editor via '**Edit→Preferences→Helpers→Edit Sources**'.

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.

ENTERING COMMANDS

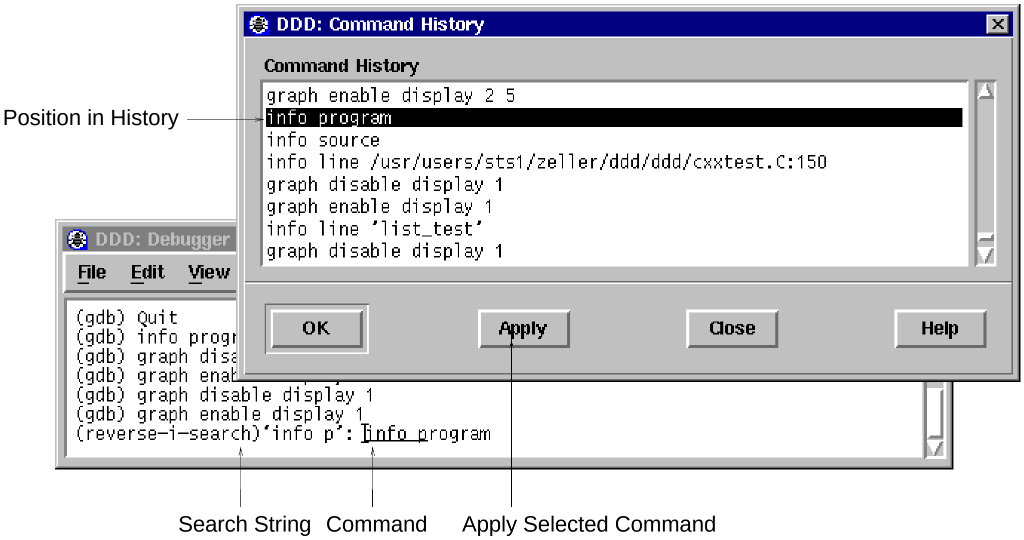
In the *debugger console*, you can interact with the command interface of the inferior debugger. Enter commands at the *debugger prompt*—that is, '**(gdb)**' for GDB, '**(dbx)**' for DBX, '**>**' for XDB, '**>**' and '*thread***[***depth***]**' for JDB, or '**(Pydb)**' for PYDB, or '**DB<>**' for Perl. You can use arbitrary debugger commands; use the **RETURN** key to enter them.



```
(gdb) info source
Current source file is /usr/users/sts1/zeller/ddd/ddd/cxxtest.C
Compilation directory is /usr/users/sts1/zeller/ddd/linux/ddd/
Located in /usr/users/sts1/zeller/ddd/ddd/cxxtest.C
Contains 285 lines.
Source language is c++.
(gdb) 
```

The Debugger Console

You can *repeat* previous and next commands by pressing the '**Up**' and '**Down**' arrow keys, respectively. If you enter an empty line, the last command is repeated as well. '**Commands→Command History**' shows the command history.



```
graph enable display 2 5
info program
info source
info line /usr/users/sts1/zeller/ddd/ddd/cxxtest.C:150
graph disable display 1
graph enable display 1
info line 'list_test'
graph disable display 1
```

```
(gdb) Quit
(gdb) info progr
(gdb) graph disa
(gdb) graph enab
(gdb) graph disable display 1
(gdb) graph enable display 1
(reverse-i-search)`info p': info program
```

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.

Using GDB and Perl, you can also *complete* commands and arguments by pressing the **TAB** key; pressing the **TAB** key multiple times shows one possible expansion after the other.

CUSTOMIZING DDD

You can set up your personal DDD preferences by using the ‘**Edit→Preferences**’ menu from the menu bar. These preferences affect your running DDD process only, unless you save these preferences for a later DDD invocation. Frequently used preferences can also be found in the individual menus.

Frequently Used Preferences

If you want to run your debugged process in a separate terminal emulator window, set ‘**Program→Run in 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.

By default, DDD finds only complete words. This is convenient for clicking on an identifier in the source text and search for exactly this identifier. If you want to find all occurrences, including word parts, unset ‘**Source→Find Words Only**’.

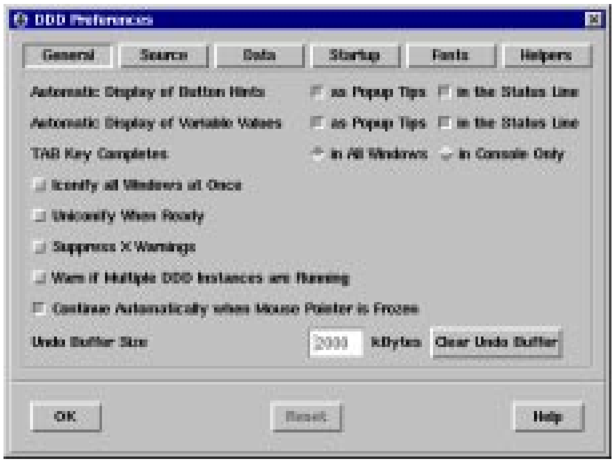
By default, DDD find is case-sensitive. This is convenient for case-sensitive programming languages. If you want to find all occurrences, regardless of case, unset ‘**Source→Find Case Sensitive**’.

If you wish to display machine code of selected functions, set ‘**Source→Display Machine Code**’. This makes DDD run a little slower, so it is disabled by default.

Through ‘**Edit→Preferences**’, you can set up more DDD preferences, which are discussed here.

General Preferences

By default, when you move the pointer over a button, DDD gives a hint on the button’s meaning in a small window. This feature is known as *button tips* (also known as *tool tips* or *balloon help*). Experienced users may find these hints disturbing; this is why you can disable them by unsetting the ‘**Automatic display of button hints as popup tips**’ option.



Automatic Display of Button Hints □ as Popup Tips □ in the Status Line

Automatic Display of Variable Values □ as Popup Tips □ in the Status Line

TAB Key Completes ○ in All Windows ○ in Console Only

□ Iconify all Windows at Once

□ Uniconify When Ready

□ Suppress X Warnings

□ Warn if Multiple DDD Instances are Running

□ Continue Automatically when Mouse Pointer is Frozen

Undo Buffer Size 2000 kBytes

General Preferences

The button hints are also displayed in the status line. Disabling hints in status line (by unsetting the ‘**Automatic display of button hints in the status line**’ option) and disabling button tips as well makes DDD run slightly faster.

By default, when you move the pointer over a variable in the source code, DDD displays the variable value in a small window. Users may find these *value tips* disturbing; this is why you can disable them by unsetting the ‘**Automatic display of variable values as popup tips**’ option.

The variable values are also displayed in the status line. Disabling variable values in status line (by unsetting the ‘**Automatic display of variable values in the status line**’ option) and disabling value tips as well will make DDD run slightly faster.

If you want to use **TAB** key completion in all text windows, set the ‘**TAB key completes in all windows**’ option. This is useful if you have pointer-driven keyboard focus (see below) and no special usage for the **TAB** key. Otherwise, the **TAB** key completes in the debugger console only.

If you frequently switch between DDD and other multi-window applications, you may like to set the ‘**Iconify all windows at once**’ option. This way, all DDD windows are iconified and deiconified as a group.

If you want to keep DDD off your desktop during a longer computation, you may like to set the ‘**Uniconify when ready**’ option. 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).

If you are bothered by X warnings, you can suppress them by setting the ‘**Suppress X warnings**’ option.

If you want to be warned about multiple DDD invocations sharing the same preferences and history files, enable ‘**Warn if Multiple DDD Instances are Running**’.

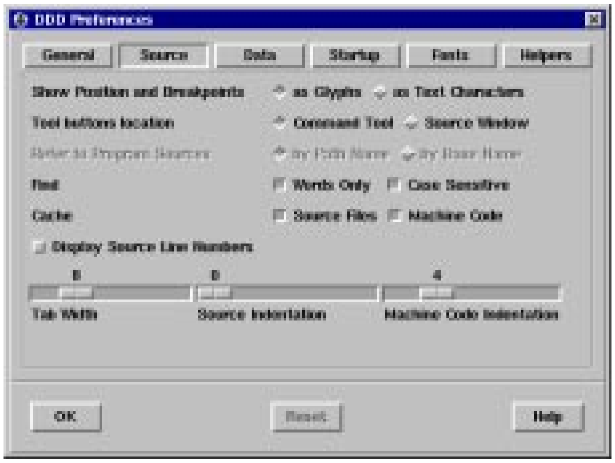
When debugging a modal X application, DDD may interrupt it while it has grabbed the pointer, making further interaction impossible. If the ‘**Continue automatically when mouse pointer is frozen**’ option is set, DDD will check after each interaction whether the pointer is grabbed. If this is so, DDD will continue the debugged program such that you can continue to use your display.

The *Undo Buffer* is the area where DDD stores old program states and commands in order to undo operations. When you are displaying lots of data, the undo buffer can quickly grow. In ‘**Undo Buffer Size**’, you can limit the size of the undo buffer. Setting this limit to **0** disables undo altogether. A negative value means to place no limit.

The ‘**Reset**’ button restores the most recently saved preferences.

Source Preferences

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 the ‘**As Text Characters**’ option. This also makes DDD run slightly faster, especially when scrolling.



Show Position and Breakpoints ● as Glyphs ○ as Text Characters

Tool buttons location ● Command Tool ○ Source Window

Refer to Program Sources ● by Path Name ○ by Base Name

Find □ Words Only □ Case Sensitive

Cache □ Source Files □ Machine Code

□ Display Source Line Numbers

Tab Width 8

Source Indentation 0

Machine Code Indentation 4

Source Preferences

DDD can locate the tool buttons in the command tool (‘**Command Tool**’) or in a *command tool bar* above the program source (‘**Source Window**’). Pick your choice.

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 ‘**Refer to sources by full path name**’ option.

By default, DDD finds only complete words. This is convenient for clicking on an identifier in the source text and search for exactly this identifier. If you want to find all occurrences, including word parts, unset ‘**Find words only**’.

By default, DDD find is case-sensitive. This is convenient for case-sensitive programming languages. If you want to find all occurrences, regardless of case, unset ‘**Find case sensitive**’.

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 ‘**Cache source files**’ option.

By default, DDD caches machine code in memory. This is bad for memory usage, but convenient for speed, since disassembling a function each time it is reached may take time. If you want to reduce memory usage, unset the ‘**Cache machine code**’ option.

If your source code uses a tab width different from **8** (the default), you can set an alternate width using the ‘**Tab width**’ slider.

You can instruct DDD to indent the source code, leaving more room for breakpoints and execution glyphs. This is done using the ‘**Source indentation**’ slider. The default value is **0** for no indentation at all. If the source indentation is **5** or higher, DDD will also show line numbers.

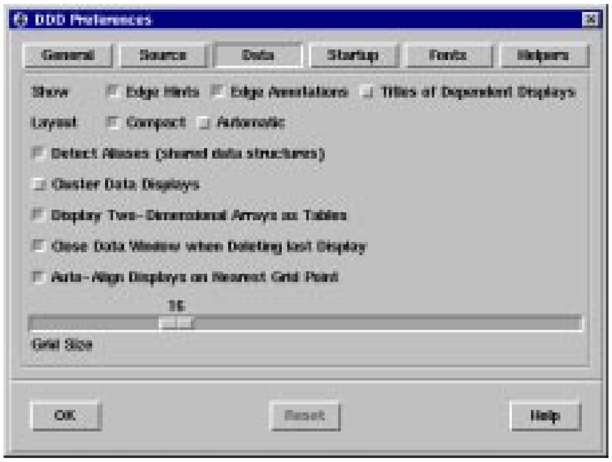
Finally, you can instruct DDD to indent the machine code, leaving room for breakpoints and execution glyphs. This is done using the ‘**Machine code indentation**’ slider. The default value is **4**.

The ‘**Reset**’ button restores the most recently saved preferences.

Data Preferences

You can control whether edge hints and edge annotations are displayed. Set or unset the ‘**Show Edge Hints**’ and ‘**Show Edge Annotations**’ option, respectively.

By default, DDD disables the title of a dependent display if the name can be deduced from edge annotations. If you want all dependent displays to have a title, set ‘**Show Titles of Dependent Displays**’.



Show ☑ Edge Hints ☑ Edge Annotations ☐ Titles of Dependent Displays

Layout ☑ Compact ☐ Automatic

☑ Detect Aliases (shared data structures)

☐ Cluster Data Displays

☑ Display Two-Dimensional Arrays as Tables

☑ Close Data Window when Deleting last Display

☑ Auto-Align Displays on Nearest Grid Point

16

Grid Size

Data Preferences

To enable a more compact layout, you can set the ‘**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.

To enforce layout, you can set the ‘**Re-layout graph automatically**’ option. If automatic layout is enabled, the graph is layouted after each change.

If you want DDD to detect aliases, set the ‘**Detect Aliases**’ option. Note that alias detection makes DDD run slower. See ‘**Examining Shared Data Structures**’, above, for details on alias detection.

By default, DDD displays two-dimensional arrays as tables, aligning the array elements in rows and columns. If you prefer viewing two-dimensional arrays as nested one-dimensional arrays, you can disable the ‘**Display two-dimensional arrays as tables**’ option.

To facilitate alignment of data displays, you can set the ‘**Auto-align displays**’ option. If auto-alignment is enabled, displays can be moved on grid positions only.

By default, the stacked data window is automatically closed when you delete the last data display. You can keep the data window open by unsetting ‘**Close data window when deleting last display**’.

In the ‘**Grid Size**’ scale, you can change the spacing of grid points. A spacing of 0 disables the grid. Default is 16.

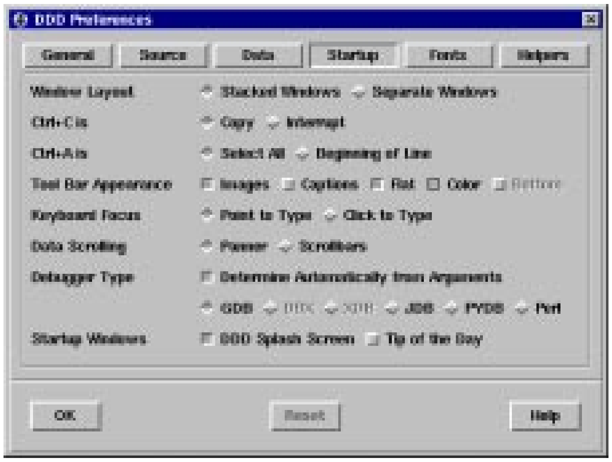
The ‘**Reset**’ button restores the most recently saved preferences.

Startup Preferences

If you change one of the resources in this panel, the change will not take effect immediately. Instead, you can

- save options (using ‘**Edit→Save Options**’) to make the change effective for future DDD sessions,
- or restart DDD (using ‘**File→Restart DDD**’) to make it effective for the restarted DDD session.

After having made changes in the panel, DDD will automatically offer you to restart itself, such that you can see the changes taking effect. Note that even after restarting, you still must save options to make the changes permanent.



Window Layout ● Stacked Windows ○ Separate Windows

Ctrl+C is ● Copy ○ Interrupt

Ctrl+A is ● Select All ○ Beginning of Line

Tool Bar Appearance ☑ Images □ Captions ☑ Flat □ Color □ Bottom

Keyboard Focus ● Point to Type ○ Click to Type

Data Scrolling ● Panner ○ Scrollbars

Debugger Type ☑ Determine Automatically from Arguments

● GDB ○ DBX ○ XDB ○ JDB ○ PYDB ○ Perl

Startup Windows ☑ DDD Splash Screen □ Tip of the Day

Startup Preferences

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 the ‘**Window Layout**’ option to ‘**Separate Windows**’. See also the ‘**−−attach-windows**’ and ‘**−−separate-windows**’ options, below.

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** tp 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.

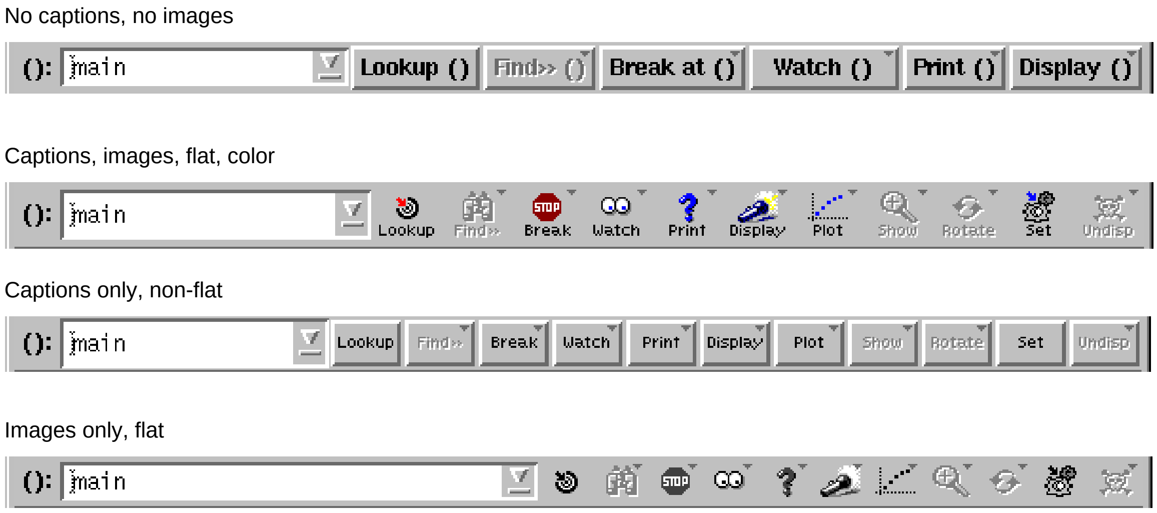
The DDD tool bar buttons can appear in a variety of styles:

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.


No captions, no images

Captions, images, flat, color

Captions only, non-flat

Images only, flat

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 Motif 2.0 and later, you can also choose a third setting, where buttons appear in color all the time.

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, enable the '**Bottom**' toggle. 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.

By default, DDD directs keyboard input to the item your mouse pointer points at. If you prefer a click-to-type keyboard focus (that is, click on an item to make it accept keyboard input), set the '**Keyboard Focus**' option on '**Click to Type**'.

By default, DDD uses Motif scroll bars to scroll the data window. Many people find this inconvenient, since you can scroll in the horizontal or vertical direction only. As an alternative, DDD provides a panner (a kind of two-dimensional scroll bar). This is much more comfortable, but may be incompatible with your Motif toolkit. To set up DDD such that it uses panners by default, set the '**Data Scrolling**' option to '**Panner**'. See also the '**−−panned-graph-editor**' and '**−−scrolled-graph-editor**' options, below.

By default, DDD determines the inferior debugger automatically. To change this default, unset '**Determine Automatically**' and set the '**Debugger Type**' option to a specific debugger. See also the '**−−gdb**', '**−−dbx**', '**−−xdb**', '**−−jdb**', '**−−pydb**', and '**−−perl**' options, below.

If you want the DDD splash screen shown upon startup, enable '**DDD Splash Screen**'.

If you want the DDD tips of the day displayed upon startup, enable '**Tip of the Day**'.

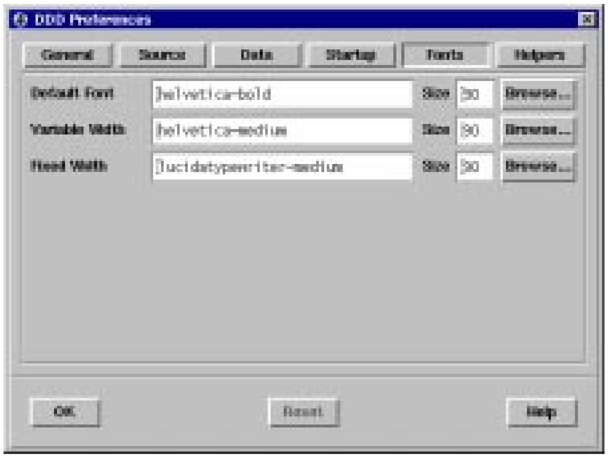
The '**Reset**' button restores the most recently saved preferences.

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 specification after '*Family*'. Thus, a pair '*family−weight*' usually suffices.
- The *font size* is given as (resolution-independent) 1⁄10 points.

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.



Default Font helvetica-bold Size 90

Variable Width helvetica-medium Size 90

Fixed Width lucidatypewriter-medium Size 90

Setting Font Preferences

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, data displays, and the execution window. Default is '**lucidatypewriter-medium**'.

Just like startup preferences, changes in this panel will not take effect immediately. Instead, you can

- save options (using ‘**Edit→Save Options**’) to make the change effective for future DDD sessions,
- or restart DDD (using ‘**File→Restart DDD**’) to make it effective for the restarted DDD session.

After having made changes in the panel, DDD will automatically offer you to restart itself, such that you can see the changes taking effect. Note that even after restarting, you still must save options to make the changes permanent.

The ‘**Reset**’ button restores the most recently saved preferences.

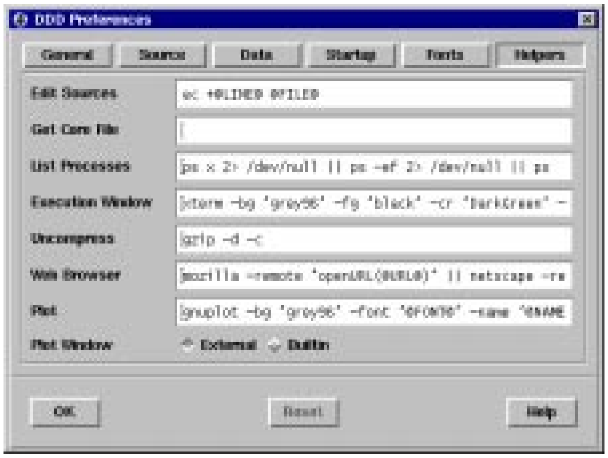
Helpers

DDD relies on some external applications (called *helpers*) for specific tasks. Through the ‘**Helpers**’ panel, you can choose and customize these applications.

In ‘**Edit Sources**’, you can select an X editor to be invoked via the DDD ‘**Edit**’ button. ‘**@FILE@**’ is replaced by the current file name; ‘**@LINE@**’ is replaced by the current line. Typical values include ‘**xedit @FILE@**’ or ‘**gnuclient +@LINE@ @FILE@**’. See also the ‘**editCommand**’ resource, below.

In ‘**Get Core File**’, you can enter a command to get a core file from a running process. ‘**@FILE@**’ is replaced by the name of the target core file; ‘**@PID@**’ is replaced by the process ID. A typical value is ‘**gcore -o @FILE@ @PID@**’. If you don’t have an appropriate command, leave this value empty: DDD will then kill the debuggee in order to get a core file. See also the ‘**getCoreCommand**’ resource, below.

‘**List Processes**’ is a command to get a list of processes, like ‘**ps**’. The output of this command is shown in the ‘**File→Attach to Process**’ dialog. See also the ‘**psCommand**’ resource, below.



Edit Sources: xc +@LINE@ @FILE@
Get Core File:
List Processes: ps x 2> /dev/null || ps -ef 2> /dev/null || ps
Execution Window: xterm -bg 'grey96' -fg 'black' -cr 'DarkGreen' -
Uncompress: gzip -d -c
Web Browser: mozilla -remote 'openURL(@URL@)' || netscape -re
Plot: gnuplot -bg 'grey96' -font '@FONT@' -name '@NAME
Plot Window: ● External ○ Builtin

Setting Helpers Preferences

In ‘**Execution Window**’, you can enter a command to start a terminal emulator. To this command, DDD appends Bourne shell commands to be executed within the execution window. A simple value is ‘**xterm −e /bin/sh −c**’. See also the ‘**termCommand**’ resource, below.

‘**Uncompress**’ is the uncompression command used by DDD to uncompress the DDD license and manual pages. The uncompression command should be invoked such that it reads from standard input and writes to standard output. A typical value is ‘**gunzip −c**’. See also the ‘**uncompressCommand**’ resource, below.

‘**Web Browser**’ is the command to invoke a WWW browser for the DDD WWW page. ‘**@URL@**’ is replaced by the URL (web page) to be shown. A simple value is ‘**netscape @URL@**’. See also the ‘**wwwCommand**’ resource, below.

‘**Plot**’ is the name of a Gnuplot program to invoke. DDD can run Gnuplot in two ways:

- DDD can use an **External Plot Window**, i.e. the plot window as supplied by Gnuplot. DDD “swallows” the Gnuplot output window into its own user interface. Unfortunately, some window managers, notably MWM, have trouble with swallowing techniques.
- DDD can supply a **Builtin Plot Window** instead. This works with all window managers, but plots are less customizable (Gnuplot resources are not understood).

Pick your choice from the menu. See also the ‘**plotCommand**’ and ‘**plotTermType**’ resources, below.

Saving Options

You can save the current option settings by selecting ‘**Edit→Save Options**’. Options are saved in a file named ‘**.ddd/init**’ in your home directory. If a session *session* is active, options will be saved in ‘**$HOME/.ddd/sessions/***session***/init**’ instead.

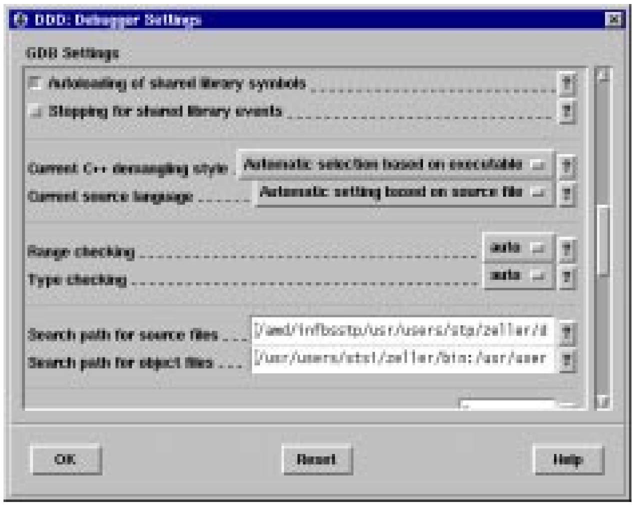
Other Customizations

Other personal DDD resources can also be set in your ‘**.ddd/init**’ file. See the ‘**RESOURCES**’ section, below.

The inferior debugger can be customized via ‘**Edit→Settings**’. See the ‘**DEBUGGER SETTINGS**’ section, below.

DEBUGGER SETTINGS

For most inferior debuggers, you can change its 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.



□ Autoloading of shared library symbols

□ Stopping for shared library events

Current C++ demangling style: Automatic selection based on executable

Current source language: Automatic setting based on source file

Range checking: auto

Type checking: auto

Search path for source files: /amd/infbsstp/usr/users/stp/zeller/d

Search path for object files: /usr/users/stst/zeller/bin:/usr/user

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.

USER-DEFINED ACTIONS

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**

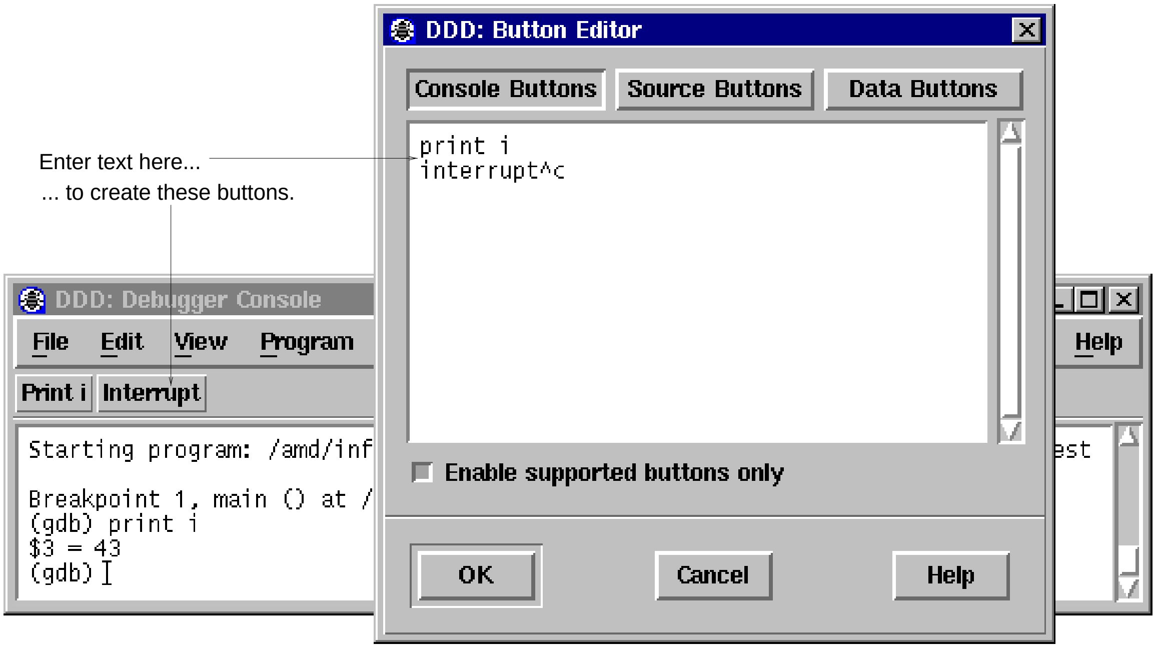
Buttons’).

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.



Enter text here...

... to create these buttons.

```
print i
interrupt^c
```

☐ Enable supported buttons only

```
Starting program: /amd/inf
Breakpoint 1, main () at /
(gdb) print i
$3 = 43
(gdb) 
```

Defining individual buttons

DDD also allows you to specify control sequences and special labels for user-defined buttons. See the examples in ‘**User-defined Buttons**’ in the ‘**RESOURCES**’ section, below.

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**’.

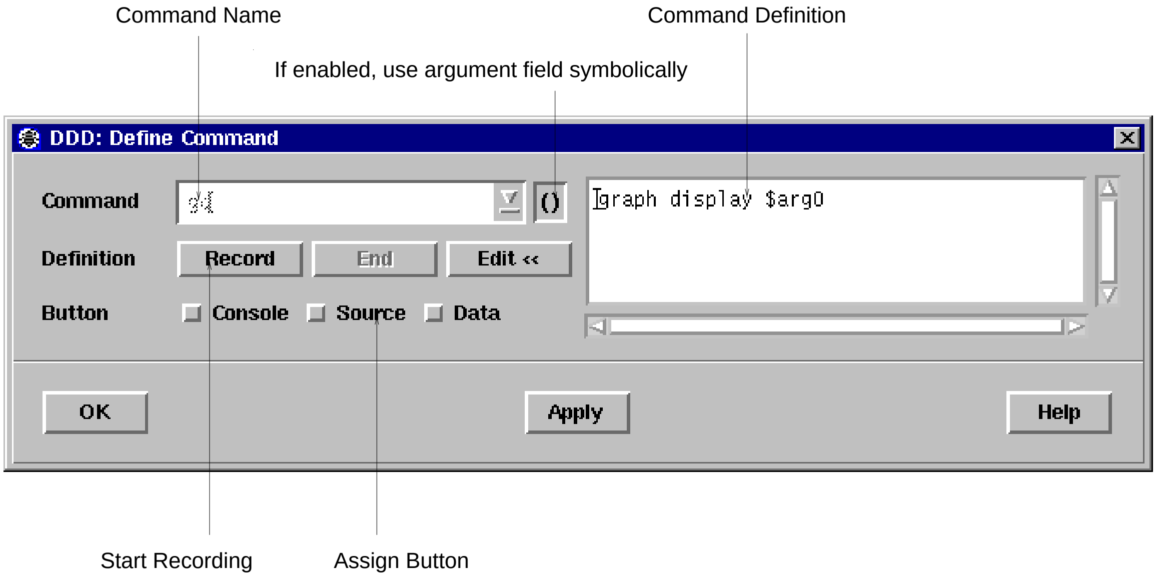
A GDB command is created in five steps:

- Enter the name of the command in the ‘**Command**’ field. Use the drop-down list on the right to select from already defined commands.
- Click on ‘**Record**’ to begin the recording of the command sequence.

- 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.
- To stop the recording, click on ‘**End**’ or enter ‘**end**’ at the GDB prompt. To *cancel* the recording, click on ‘**Interrupt**’ or press **ESC**.
- 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 ‘**Apply**’ to apply 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 also ‘**Defining Buttons**’, above.



Command: gd
()
□ Console □ Source □ Data

```
graph display $arg0
```

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.

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.

To save all command definitions, use ‘**Edit**→**Save Options**’.

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.

- Enter '**contuntil**' in the '**Definition**' field.
- Enable the '**()**' toggle button.
- Now click on '**Record**' to start recording. Note that the contents of the argument field change to '**$arg0**'.
- Click on '**Break at ()**' to create a breakpoint. Note that the recorded breakpoint command refers to '**$arg0**'.
- Click on '**Cont**' to continue execution.
- Click on '**End**' to end recording. Note that the argument field is restored to its original value.
- 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.
- 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.

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 **$HOME/.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 '**$HOME/.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 $HOME**' when being debugged! As a consequence, be sure to choose your own *prefix*; it must be at least three characters long.

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 QUIT signal to the inferior debugger. Use this in emergencies only; the inferior debugger may be left inconsistent or even exit after a QUIT 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.

PERSISTENT SESSIONS

Note: Persistent sessions are supported with GDB running on the local machine only. Support for other DBX, XDB, and JDB is partially implemented; your mileage may vary.

If you want to interrupt your current DDD session, you can save its entire DDD state in a file and restore it later.

Saving Sessions

To save a session, select '**File→Save Session As**'. You will be asked for

- a symbolic session name *session* and
- whether to include a core dump of the debugged program. Including a core dump is necessary for restoring memory contents and the current execution position.

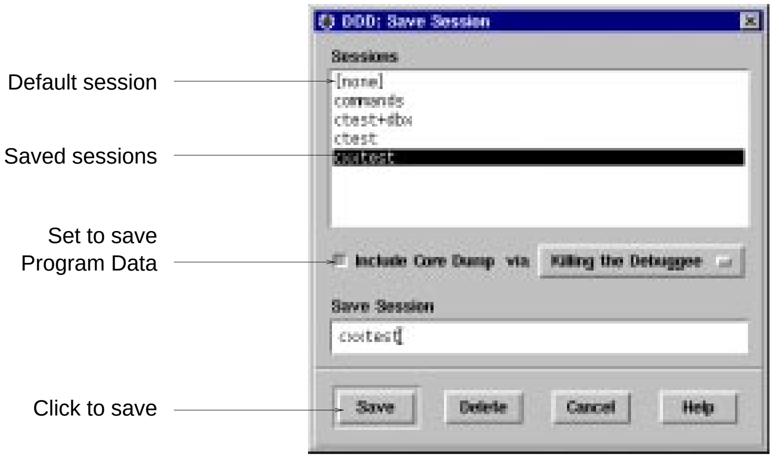
After clicking on '**Save**', the session is saved in '**$HOME/.ddd/sessions/***session*'.

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 '**$HOME/.ddd/sessions/***session***/init**' instead of '**$HOME/.ddd/init**'; see '**Saving Options**', below, for details.
- The DDD command history will be saved in '**$HOME/.ddd/sessions/***session***/history**' instead of '**$HOME/.ddd/history**'; see '**Entering Commands**', above, for details.

To make the current session inactive, open the *default session* named '**[None]**'; see below for details on opening sessions.

If your program is running, or if you have opened a core file, DDD can 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.



Save Session: cxxtest

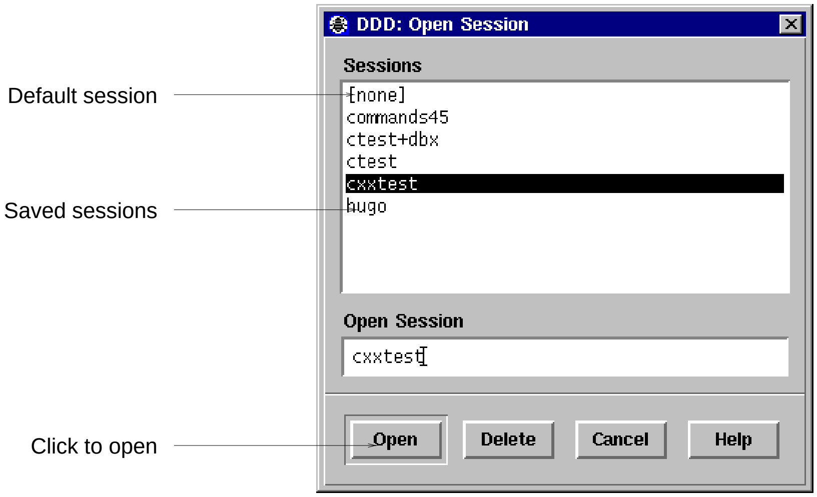
Saving a Session

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.

Opening 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, open the default session and save options; see '**Saving Options**' below for details.



Open Session: cxxtest

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 environment (open files, resources, etc.) will be lost. 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.

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 cannot be deleted.

Starting DDD with a Session

To start-up DDD with a given session named *session* instead of the default session, use

ddd −−session *session*

There is also a shortcut that opens the session *session* and also 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.

INTEGRATING DDD

You can run DDD as an inferior debugger in other debugger front-ends, combining their special abilities with those of DDD.

General Information

To have DDD run as an inferior debugger in other front-ends, 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.

You may also invoke '**ddd −−tty**' directly, entering DDD commands from your TTY, or use DDD as the end of a pipe, controlled by a remote program. Be aware, however, that the TTY interface does not support line editing and command completion and that DDD exits as soon as it detects an EOF condition on its standard input. Also, do not try to run DDD with DDD as inferior debugger.

Using DDD in TTY mode automatically disables some DDD windows, because it is assumed that their facilities are provided by the remote program:

- If DDD is invoked with the '**−−tty**' option, the debugger console is initially disabled, as its facilities are supposed to be provided by the integrating front-end.
- 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.

Using DDD with GNU Emacs

Use '**M-x gdb**' or '**M-x dbx**' 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.

Using DDD with XEmacs

Set the variable **gdb-command-name** to **"ddd"**, by inserting the following line in your **$HOME/.emacs** file or evaluating it by pressing **ESC :** (**ESC ESC** for XEmacs 19.13 and earlier):

```
(setq gdb-command-name "ddd")
```

Use '**M-x gdb**' or '**M-x gdbsrc**' to start a debugging session. Proceed as usual.

Using DDD with XXGDB

Invoke **xxgdb** as

```
xxgdb −db_name ddd −db_prompt ’(gdb) ’
```

USING DDD WITH LESSTIF

DDD 2.1.1 and later include a number of hacks that make DDD run with *LessTif*, a free Motif clone, 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**’ indicates the LessTif version against which DDD is linked. For LessTif version *x.y*, its value is *x* multiplied by 1000 plus *y*—for instance, the value **95** stands for LessTif 0.95 and the value **1000** stands for LessTif 1.0. To specify the version number of the LessTif library at DDD invocation, 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 0.85 dynamic library instead. Invoke DDD with ‘**−−lesstif-version 85**’.
- 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**’.

REMOTE DEBUGGING

It is possible to have the inferior debugger run on a remote UNIX host. This is useful when the remote host has a slow network connection or when DDD is available on the local host only.

Furthermore, the inferior debugger may support debugging a program on a remote host. This is useful when the inferior debugger is not available on the remote host—for instance, because the remote system does not have a general purpose operating system powerful enough to run a full-featured debugger.

Using DDD with a Remote 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 don't 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 also the '**rshCommand**' resource, below.

Using DDD with 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.

The section '**Remote debugging**' in the GDB documentation contains all the 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 '**file**' command.
- Attach to the remote '**gdbserver**' using the '**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.

ROOT DEBUGGING

Sometimes, you may require to debug programs with root privileges, but without actually logging in as root. This is usually done by installing the debugger *setuid root*, that is, having the debugger run with root privileges. For security reasons, you cannot install DDD as a setuid program; DDD invokes shell commands and even shell scripts, such that all known problems of setuid shell scripts apply. Instead, you should invoke DDD such that a *setuid* copy of the inferior debugger is used.

Here is an example. Have a *setuid root* copy of GDB installed as '**rootgdb**'. Then invoke

```
ddd −−debugger rootgdb
```

to debug programs with root privileges.

Since a program like '**rootgdb**' grants root privileges to any invoking user, you should give it very limited access.

RESOURCES

DDD understands all of the core X Toolkit resource names and classes. The following resources are specific to DDD.

Setting DDD Fonts

DDD uses the following resources to set up its fonts:

defaultFont (class Font)

The default DDD font to use for labels, menus, buttons, etc. The font is specified as an X font spec, where the initial 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 '**$HOME/.ddd/init**' file.

defaultFontSize (class FontSize)

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)

The variable width DDD font to use for help texts and messages. The font is specified as an X font spec, where the initial specification after '*Family*'. Defaults to '**helvetica-medium-r**'.

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

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

in your '**$HOME/.ddd/init**' file.

variableWidthFontSize (class FontSize)

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)

The fixed width DDD font to use for source code, the debugger console, text fields, data displays, and the execution window. The font is specified as an X font spec, where the initial specification after '*Family*'. Defaults to '**lucidatypewriter-medium**'.

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

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

in your '**$HOME/.ddd/init**' file.

fixedWidthFontSize (class FontSize)

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.

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 '**$HOME/.ddd/init**' file.

To find out your favorite font size, try '**−−fontsize** *SIZE*'. This also sets all font sizes to *SIZE*.

If you want to set the fonts of specific items, see the ‘**Ddd**’ application defaults file for instructions.

Setting DDD Colors

These are the most important color resources used in DDD:

Ddd*foreground:	**black**
Ddd*background:	**grey**
Ddd*XmText.background:	**grey96**
Ddd*XmTextField.background:	**grey96**
Ddd*GraphEdit.background:	**grey96**
Ddd*XmList.background:	**grey96**
Ddd*graph_edit.nodeColor:	**black**
Ddd*graph_edit.edgeColor:	**blue4**
Ddd*graph_edit.selectColor:	**black**
Ddd*graph_edit.gridColor:	**black**
Ddd*graph_edit.frameColor:	**grey50**
Ddd*graph_edit.outlineColor:	**grey50**

You can copy and modify the appropriate resources to your ‘**$HOME/.ddd/init**’ file. For colors within the data display, things are slightly more complicated—see the ‘**vslDefs**’ resource, below.

General Preferences

The following resources determine DDD general behavior.

buttonTips (class **Tips)**

Whether button tips are enabled (‘**on**’, default) or not (‘**off**’). Button tips are helpful for novices, but may be distracting for experienced users.

buttonDocs (class **Docs)**

Whether the display of button hints in the status line is enabled (‘**on**’, default) or not (‘**off**’).

checkGrabs (class **CheckGrabs)**

When debugging a modal X application, DDD may interrupt it while it has grabbed the pointer, making further interaction impossible. 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)**

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.

checkOptions (class **CheckOptions)**

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.

cutCopyPasteBindings (class **BindingStyle)**

Controls the key bindings for cut, copy, and paste operations.

- If this is ‘**Motif**’ (default), Cut/Copy/Paste is on **Shift+Del**/**Ctrl+Ins**/**Shift+Ins**. This is conformant to the Motif 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.

filterFiles (class **FilterFiles)**

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.

globalTabCompletion (class **GlobalTabCompletion)**

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.

grabAction (class **grabAction)**

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)**

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.

groupIconify (class **GroupIconify)**

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.

saveHistoryOnExit (class **SaveHistoryOnExit)**

If '**on**' (default), the command history is automatically saved when DDD exits.

selectAllBindings (class **BindingStyle)**

Controls the key bindings for the select all operation.

- If this is '**Motif**', Select All 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 the **HOME** key instead.

splashScreen (class **SplashScreen)**

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

splashScreenColorKey (class **ColorKey)**

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.

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.

startupTips (class **StartupTips)**

Whether a tip of the day is to be shown at startup ('**on**', default) or not ('**off**').

startupTipCount (class **StartupTipCount)**

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

suppressWarnings (class **SuppressWarnings)**

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

tip*n* (*class* **Tip)**

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

maxUndoDepth (class **MaxUndoDepth)**

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.

maxUndoSize (class **MaxUndoSize)**

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**.

uniconifyWhenReady (class **UniconifyWhenReady)**

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.

valueTips (class **Tips)**

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)**

Whether the display of variable values in the status line is enabled ('**on**', default) or not ('**off**').

warnIfLocked (class **WarnIfLocked)**

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

Source Window

The following resources determine the DDD source window.

cacheGlyphImages (class **CacheMachineCode)**

Whether to cache (share) glyph images ('**on**') or not ('**off**'). Caching glyph images requires less X resources, but has been reported to fail with Motif 2.1 on XFree86 servers. Default is '**off**' for Motif 2.1 or later on Linux machines, and '**on**' otherwise.

cacheMachineCode (class **CacheMachineCode)**

Whether to cache disassembled machine code ('**on**', default) or not ('**off**'). Caching machine code requires more memory, but makes DDD run faster.

cacheSourceFiles (class **CacheSourceFiles)**

Whether to cache source files ('**on**', default) or not ('**off**'). Caching source files requires more memory, but makes DDD run faster.

disassemble (class **Disassemble)**

If this is '**on**', the source code is automatically disassembled. The default is '**off**'. See also the '**−−disassemble**' and '**−−no-disassemble**' options, below.

displayGlyphs (class **DisplayGlyphs)**

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 also the '**−−glyphs**' and '**−−no-glyphs**' options, below.

displayLineNumbers (class **DisplayLineNumbers)**

If this is '**on**', lines in the source text are prefixed with their respective line number. The default is '**off**'.

findCaseSensitive (class **FindCaseSensitive)**

If this is '**on**' (default), the '**Find**' commands are case-sensitive. Otherwise, occurrences are found regardless of case.

findWordsOnly (class **FindWordsOnly)**

If this is '**on**' (default), the '**Find**' commands find complete words only. Otherwise, arbitrary occurrences are found.

glyphUpdateDelay (class **GlyphUpdateDelay)**

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**.

indentCode (class **Indent)**

The number of columns to indent the machine code, such that there is enough place to display breakpoint locations. Default: **4**.

indentSource (class **Indent)**

The number of columns to indent the source code, such that there is enough place to display breakpoint locations. Default: **0**.

indentScript (class **Indent)**

The minimum indentation for script languages, such as Perl and Python. Default: **4**.

lineNumberWidth (class **LineNumberWidth)**

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**.

linesAboveCursor (class **LinesAboveCursor)**

The minimum number of lines to show before the current location. Default is **2**.

linesBelowCursor (class **LinesBelowCursor)**

The minimum number of lines to show after the current location. Default is **3**.

maxDisassemble (class **MaxDisassemble)**

Maximum number of bytes to disassemble (default: **256**). If this is zero, the entire current function is disassembled.

maxGlyphs (class **MaxGlyphs)**

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.

sourceEditing (class **SourceEditing)**

If this is '**on**', the displayed source code becomes editable. This is an experimental feature and may become obsolete in future DDD releases. Default if '**off**'.

tabWidth (class **TabWidth)**

The tab width used in the source window (default: **8**)

useSourcePath (class **UseSourcePath)**

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.

Window Creation and Layout

The following resources determine DDD window creation and layout as well as the interaction with the X window manager.

autoRaiseTool (class **AutoRaiseTool)**

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**'.

autoRaiseMenu (class **AutoRaiseMenu)**

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**':

```
Ddd*autoRaiseMenu: off
```

colorWMIcons (class **ColorWMIcons)**

If '**on**' (default), DDD uses multi-color icons. If your window manager has trouble with multi-color icons, set this resource to '**off**' and DDD will use black-and-white icons instead.

decorateTool (class **Decorate)**

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.

openDataWindow (class **Window)**

If '**off**' (default), the data window is closed upon start-up.

openDebuggerConsole (class **Window)**

If '**off**', the debugger console is closed upon start-up.

openSourceWindow (class **Window)**

If '**off**', the source window is closed upon start-up.

separateDataWindow (class **Separate)**

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. See also the '**−−attach-windows**' and '**−−attach-data-window**' options, below.

separateExecWindow (class **Separate)**

If '**on**', the debugged program is executed in a separate execution window. If '**off**' (default), the debugged program is executed in the console window. See also the '**−−exec−window**' and '**−−no−exec−window**' options, below.

separateSourceWindow (class **Separate)**

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. See also the '**−−attach-windows**' and '**−−attach-source-window**' options, below.

statusAtBottom (class **StatusAtBottom)**

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 also the '**−−status-at-bottom**' and '**−−status-at-top**' options, below.

stickyTool (class **StickyTool)**

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.

transientDialogs (class **TransientDialogs)**

If '**on**' (default), all dialogs are created as transient windows—that is, they always stay on top of the main DDD windows, and they iconify with it. If '**off**', the important selection dialogs, such as the breakpoint and display editors, are created as top-level windows on their own, and may be obscured by the DDD main windows.

Debugger Settings

The following resources determine the inferior debugger.

autoCommands (class **AutoCommands)**

If this is ‘**on**’ (default), each line output by the inferior debugger beginning with the value of the ‘**autoCommandPrefix**’ resource (see below) will be interpreted as DDD command and executed. Useful for user-defined commands; see ‘**USER-DEFINED COMMANDS**’, above.

autoCommandPrefix (class **AutoCommandPrefix)**

The prefix for auto-commands. By default, an empty string, meaning to generate a new prefix for each DDD session. If this is set to ‘**ddd:** ’, for example, each GDB output in the form ‘**ddd:** *command*’ will cause DDD to execute *command*.

autoDebugger (class **AutoDebugger)**

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.

blockTTYInput (class **BlockTTYInput)**

Whether DDD should block when reading data from the inferior debugger via the pseudo-tty interface. Some systems *require* this, such as Linux with libc 5.4.33 and earlier; set it to ‘**on**’. Some other systems *prohibit* this, such as Linux with GNU libc 6 and later; set it to ‘**off**’. The value ‘**auto**’ (default) will always select the “best” choice (that is, the best choice known to the DDD developers).

dbxInitCommands (class **InitCommands)**

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 ‘**$HOME/.dbxinit**’ or ‘**$HOME/.dbxrc**’ file. See your DBX documentation for details.

dbxSettings (class **Settings)**

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

debugger (class **Debugger)**

The type of the inferior debugger to invoke (‘**gdb**’, ‘**dbx**’, ‘**xdb**’, ‘**jdb**’, ‘**pydb**’, or ‘**perl**’). This resource is usually set through the ‘**−−gdb**’, ‘**−−dbx**’, ‘**−−xdb**’, ‘**−−jdb**’, ‘**−−pydb**’, and ‘**−−perl**’, options; see below for details.

debuggerCommand (class **DebuggerCommand)**

The name under which the inferior debugger is to be invoked. If this string is empty, the debugger type (‘**debugger**’ resource) is used. This resource is usually set through the ‘**−−debugger**’ option; see below for details.

debuggerHost (class **DebuggerHost)**

The host where the inferior debugger is to be executed; an empty string (default) means the local host. See the ‘**−−host**’ option, below, and ‘**REMOTE DEBUGGING**’, above.

debuggerHostLogin (class **DebuggerHostLogin)**

The login user name on the remote host; an empty string (default) means using the local user name. See the ‘**−−login**’ option, below, and ‘**REMOTE DEBUGGING**’, above.

debuggerRHost (class **DebuggerRHost)**

The host where the inferior debugger is to be executed; an empty string (default) means to use the ‘**debuggerHost**’ resource. In contrast to ‘**debuggerHost**’, using this resource causes DDD to login interactively to the remote host and invoke the inferior debugger from the remote shell. See also the ‘**−−rhost**’ option, below, and ‘**REMOTE DEBUGGING**’, above.

fullNameMode (class **TTYMode)**

If this is ‘**on**’, DDD reports the current source position on standard output in GDB ‘**−fullname**’ format. As a side effect, the source window is disabled by default. See also the ‘**−−fullname**’ option, below.

gdbInitCommands (class **InitCommands)**

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 '**$HOME/.gdbinit**' file. See your GDB documentation for details.

gdbSettings (class **Settings)**

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.

jdbInitCommands (class **InitCommands)**

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)**

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.

openSelection (class **OpenSelection)**

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.

perlInitCommands (class **InitCommands)**

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.

pydbSettings (class **Settings)**

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.

pydbInitCommands (class **InitCommands)**

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)**

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.

questionTimeout (class **QuestionTimeout)**

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

rHostInitCommands (class **RHostInitCommands)**

These commands are initially executed in a remote interactive session, using the '**−−rhost**' option. By default, it sets up the remote terminal such that it suits DDD:

```
Ddd*rHostInitCommands: stty −echo −onlcr
```

You may add other commands here—for instance, to set the executable path or to invoke a suitable shell.

sourceInitCommands (class **SourceInitCommands)**

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.

synchronousDebugger (class **SynchronousDebugger)**

If '**on**', X events are not processed while the debugger is busy. This may result in slightly better performance on single-processor systems. See also the '**−−sync-debugger**' option, below.

terminateOnEOF (class **TerminateOnEOF)**

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.

ttyMode (class **TTYMode)**

If '**on**', enable TTY interface, taking additional debugger commands from standard input and forwarding debugger output on standard output. As a side effect, the debugger console is disabled by default. See also the '**−−tty**' and '**−−fullname**' options, below.

useTTYCommand (class **UseTTYCommand)**

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.

xdbInitCommands (class **InitCommands)**

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 '**$HOME/.xdbrc**' file. See your XDB documentation for details.

xdbSettings (class **Settings)**

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

User-defined Buttons

The following resources can be used to create and control tool bars and user-defined buttons.

activeButtonColorKey (class **ColorKey)**

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)**

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)**

The geometry of the caption subimage within the button icons. Default is ‘**29**×**7**+**0**−**0**’.

buttonImages (class **ButtonImages)**

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)**

The geometry of the image within the button icon. Default is ‘**25**×**21**+**2**+**0**’.

buttonColorKey (class **ColorKey)**

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.

commandToolBar (class **ToolBar)**

Whether the tool buttons (see the ‘**toolButtons**’ resource, below) 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.

commonToolBar (class **ToolBar)**

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**’).

consoleButtons (class **Buttons)**

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 ‘**$HOME/.ddd/init**’ file. These are the settings of DDD 1.x:

```
Ddd*consoleButtons: Yes\nNo\nbreakˆC
```

This setting creates some more buttons:

```
Ddd*consoleButtons: \
Yes\nNo\nrun\nClear\nPrev\nNext\nApply\nbreakˆC
```

See also the ‘**dataButtons**’, ‘**sourceButtons**’ and ‘**toolButtons**’ resources, below.

dataButtons (class **Buttons)**

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.

flatToolbarButtons (class **FlatButtons)**

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)**

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.

labelDelimiter (class **LabelDelimiter)**

The string used to separate labels from commands and shortcuts. Default is ‘//’.

sourceButtons (class **Buttons)**

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 ‘**$HOME/.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\ninterruptˆC
```

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.

toolbarsAtBottom (class **ToolbarsAtBottom)**

Whether source and data tool bars should be placed above source and data, respectively (‘**off**’, default), or below, as in DDD 2.x (‘**on**’). See also the ‘**−−toolbars-at-bottom**’ and ‘**−−toolbars-at-top**’ options, below.

toolButtons (class **Buttons)**

A newline-separated list of buttons to be included in the command tool or the command tool bar (see the ‘**commandToolBar**’ resource, above). Each button issues the command given by its name. See the ‘**consoleButtons**’ resource, above, for details on button syntax.

The default resource value is

```
Ddd*toolButtons: \
run\nbreakˆC\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.

toolRightOffset (class **Offset)**

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

toolTopOffset (class **Offset)**

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

verifyButtons (class **VerifyButtons)**

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”.

User-Defined New Display Menu

The following resources control the user-defined '**New Display**' menu.

dbxDisplayShortcuts (class **DisplayShortcuts)**

A newline-separated list of display expressions to be included in the '**New Display**' menu for DBX. If a line contains a label delimiter (the string '//'; can be changed via the '**labelDelimiter**' resource), 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)**

A newline-separated list of display expressions to be included in the '**New Display**' menu for GDB. See the description of '**dbxDisplayShortcuts**', above.

jdbDisplayShortcuts (class **DisplayShortcuts)**

A newline-separated list of display expressions to be included in the '**New Display**' menu for JDB. See the description of '**dbxDisplayShortcuts**', above.

labelDelimiter (class **LabelDelimiter)**

The string used to separate labels from commands and shortcuts. Default is '//'.

perlDisplayShortcuts (class **DisplayShortcuts)**

A newline-separated list of display expressions to be included in the '**New Display**' menu for Perl. See the description of '**dbxDisplayShortcuts**', above.

pydbDisplayShortcuts (class **DisplayShortcuts)**

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)**

A newline-separated list of display expressions to be included in the '**New Display**' menu for XDB. See the description of '**dbxDisplayShortcuts**', above.

Data Display

The following resources control the data display.

align2dArrays (class **Align2dArrays)**

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.

autoCloseDataWindow (class **AutoClose)**

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)**

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)**

If '**on**', new independent data displays will automatically be clustered. Default is '**off**', meaning to leave new displays unclustered.

deleteAliasDisplays (class **DeleteAliasDisplays)**

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)**

If '**on**', DDD attempts to recognize shared data structures. See '**Examining shared data structures**', above, for a discussion. The default is '**off**', meaning that shared data structures are not recognized.

expandRepeatedValues (class **ExpandRepeatedValues)**

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*×**>' appended to indicate the repetition.**

hideInactiveDisplays (class HideInactiveDisplays)

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.

pannedGraphEditor (class **PannedGraphEditor)**

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 Motif scrollbars are used.

See also the '**−−scrolled-graph-editor**' and '**−−panned-graph-editor**' options, below.

paperSize (class **PaperSize)**

The paper size used for printing, in format *width* x *height*. The default is A4 format, or '**210mm** × **297mm**'.

showBaseDisplayTitles (class **ShowDisplayTitles)**

Whether to assign titles to base (independent) displays or not. Default is '**on**'.

showDependentDisplayTitles (class **ShowDisplayTitles)**

Whether to assign titles to dependent displays or not. Default is '**off**'.

typedAliases (class **TypedAliases)**

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.

vslBaseDefs (class **VSLDefs)**

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)**

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 general pattern to replace a function definition *function* with a new definition *new_def* is:

#pragma replace *function*
*function***(***args...***) =** *new_def***;**

The following VSL functions are frequently used:

color(*box***,** *foreground* **[,** *background* **])**

Set the *foreground* and *background* colors of *box*.

display_color(*box***)**

The color used in data displays. Default: **color(***box***, "black", "white")**

title_color(*box***)**

The color used in the title bar. Default: **color(***box***, "black")**

disabled_color(*box***)**

The color used for disabled boxes. Default: **color(***box***, "white", "grey50")**

simple_color(*box*)
The color used for simple values. Default: **color(*box*, "black")**

pointer_color(*box*)
The color used for pointers. Default: **color(*box*, "blue4")**

struct_color(*box*)
The color used for structures. Default: **color(*box*, "black")**

array_color(*box*)
The color used for arrays. Default: **color(*box*, "blue4")**

reference_color(*box*)
The color used for references. Default: **color(*box*, "blue4")**

changed_color(*box*)
The color used for changed values. Default: **color(*box*, "black", "#ffffcc")**

stdfontfamily()
The font family used. One of **family_times()**, **family_courier()**, **family_helvetica()**, **family_new_century()**, or **family_typewriter()** (default).

stdfontsize()
The font size used (in pixels). **0** (default) means to use **stdfontpoints()** instead.

stdfontpoints()
The font size used (in 1⁄10 points). **0** means to use **stdfontsize()** instead. Default value: **90**.

stdfontweight()
The font weight used. Either **weight_medium()** (default) or **weight_bold()**.

To set the pointer color to "red4", use

```
Ddd*vslDefs: \
#pragma replace pointer_color\n\
pointer_color(box) = color(box, "red4");\n
```

To set the default font size to resolution-independent 10.0 points, use

```
Ddd*vslDefs: \
#pragma replace stdfontsize\n\
#pragma replace stdfontpoints\n\
stdfontsize() = 0;\n
stdfontpoints() = 100;\n
```

To set the default font to 12-pixel courier, use

```
Ddd*vslDefs: \
#pragma replace stdfontsize\n\
#pragma replace stdfontfamily\n\
stdfontsize() = 12;\n\
stdfontfamily() = family_courier();\n
```

See the file ‘**ddd.vsl**’ for further definitions to override using the ‘**vslDefs**’ resource.

vslLibrary (class **VSLLibrary)**
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)**

A colon-separated list of directories to search for VSL include files. Default is '**.**', 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:  \
.:/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 '**/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 the '**vslDefs**' resource, above, for details.

Plot Window

The following resources control the plot window.

plotTermType (class **PlotTermType)**

The Gnuplot terminal type. Can have one of two values:

- If this is '**x11**', DDD "swallows" the 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).

plotCommand (class **PlotCommand)**

The name of a Gnuplot executable. Default is '**gnuplot**', followed by some options to set up colors and the initial geometry.

plotWindowClass (class **PlotWindowClass)**

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)**

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**.

plotInitCommands (class **PlotInitCommands)**

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.

plot2dSettings (class **PlotSettings)**

Additional initial settings for 2-D plots. Default is '**set noborder**'. Feel free to customize these settings as desired.

plot3dSettings (class **PlotSettings)**

Additional initial settings for 3-D plots. Default is '**set border**'. Feel free to customize these settings as desired.

Debugger Console

The following resources control the debugger console.

lineBufferedConsole (class **LineBuffered)**

If this is '**on**' (default), each line from the inferior 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.

Value Histories

The following resources control the pop-down value histories associated with various text fields.

popdownHistorySize (class **HistorySize)**

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)**

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

Customizing Helpers

The following resources determine external programs invoked by DDD.

editCommand (class **EditCommand)**

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@
```

fontSelectCommand (class **FontSelectCommand)**

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
```

getCoreCommand (class **GetCoreCommand)**

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 this entry empty if you have no '**gcore**' or similar command.

lessTifVersion (class **LessTifVersion)**

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.

listCoreCommand (class **listCoreCommand)**

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)**

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)**

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)**

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
```

printCommand (class **PrintCommand)**

The command to print a postscript file. Usually '**lp**' or '**lpr**'.

psCommand (class **PsCommand)**

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.

rshCommand (class **RshCommand)**

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

termCommand (class **TermCommand)**

The command to invoke a separate TTY for showing 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
```

termType (class **TermType)**

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**’.

uncompressCommand (class **UncompressCommand)**

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**’.

wwwCommand (class **WWWCommand)**

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, then **$WWWBROWSER**, then to invoke a new Netscape process, then to let a running Emacs do the job, then to invoke Mosaic, then to invoke Lynx in an xterm.

To specify ‘**netscape-4.0**’ as browser, use the setting:

```
Ddd*wwwCommand: \
   netscape-4.0 −remote ’openURL(@URL@)’ \
|| netscape-4.0 ’@URL@’
```

This command first tries to connect to a running **netscape-4.0** browser; if this fails, it starts a new **netscape-4.0** process.

wwwPage (class **WWWPage)**

The DDD WWW page. Value:

```
Ddd*wwwPage: http://www.cs.tu-bs.de/softech/ddd/
```

Obtaining Diagnostics

The following resources are used for debugging DDD and to obtain specific DDD information.

appDefaultsVersion (class **Version)**

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)**

If ‘**on**’, check the DDD environment (in particular, the X configuration), report any possible problem causes and exit. See also the ‘**−−check-configuration**’ option, below.

dddinitVersion (class **Version)**

The version of the DDD executable that last wrote the ‘**$HOME/.ddd/init**’ file. If this string does not match the version of the current DDD executable, DDD issues a warning.

debugCoreDumps (class **DebugCoreDumps)**

If ‘**on**’, DDD invokes a debugger on itself when receiving a fatal signal.

dumpCore (class **DumpCore)**

If ‘**on**’ (default), DDD dumps core when receiving a fatal signal.

maintenance (class **Maintenance)**

If '**on**', enables a top-level '**Maintenance**' menu with additional options. See also the '**−−maintenance**' option, below.

showConfiguration (class **ShowConfiguration)**

If '**on**', show the DDD configuration on standard output and exit. See also the '**−−configuration**' option, below.

showFonts (class **ShowFonts)**

If '**on**', show the DDD font definitions on standard output and exit. See also the '**−−fonts**' option, below.

showInvocation (class **ShowInvocation)**

If '**on**', show the DDD invocation options on standard output and exit. See also the '**−−help**' option, below.

showLicense (class **ShowLicense)**

If '**on**', show the DDD license on standard output and exit. See also the '**−−license**' option, below.

showManual (class **ShowManual)**

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 also the '**−−manual**' option, below.

showNews (class **ShowNews)**

If '**on**', show the DDD news on standard output and exit. See also the '**−−news**' option, below.

showVersion (class **ShowVersion)**

If '**on**', show the DDD version on standard output and exit. See also the '**−−version**' option, below.

trace (class **Trace)**

If '**on**', show the dialog between DDD and the inferior debugger on standard output. Default is '**off**'.

More Resources

The '**Ddd**' application defaults file contains even more information about setting DDD resources. The '**Ddd**' file comes with the DDD distribution.

OPTIONS

You can use the following options when starting DDD. All options may be abbreviated, as long as they are unambiguous; single dashes may also be used. DDD also understands the usual X options such as '**−display**' or '**−geometry**'; see **X(1)** for details.

All other arguments and options are passed to the inferior debugger. 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, below.

−−attach-windows

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

−−attach-source-window

Attaches only the source window to the debugger console.

−−attach-data-window

Attaches only the source window to the debugger console.

−−automatic-debugger

Determine the inferior debugger automatically.

−−button-tips

Enable button tips.

−−configuration
Show the DDD configuration settings and exit.

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

−−data-window
Create the data window upon start-up.

−−dbx Run the DBX debugger as inferior debugger.

−−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 as '**gdb**', '**dbx**', '**xdb**', '**jdb**', '**pydb**', or '**perl**' respectively.
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).

−−debugger-console
Create the debugger console upon start-up.

−−disassemble
Disassemble the source code. See also the '**−−no-disassemble**' option, below.

−−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.

−−fonts
Show the font definitions used by DDD on standard output.

−−fontsize *size*
Set the default font size to *size* 1⁄10 points. To use 12-point fonts, say '**−−fontsize 120**'.

−−fullname
Enable 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.

−−gdb Run the GDB debugger as inferior debugger.

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

−−help Give a list of frequently used options. Show options of the inferior debugger as well.

−−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 '**REMOTE DEBUGGING**', above.

−−jdb Run JDB as inferior debugger.

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

−−lesstif-version *version*
Enable some hacks to make DDD run properly with LessTif. See the '**lessTifVersion**' resource and '**USING DDD WITH LESSTIF**', above, for a discussion.

−−license
Show the DDD license and exit.

−−login *username*
Use *username* as remote user name. See '**REMOTE DEBUGGING**', above.

−−maintenance
Enable the top-level '**Maintenance**' menu with options for debugging DDD.

−−manual
Show this manual page and exit.

−−news
Show the DDD news and exit.

−−no-button-tips
Disable button tips.

−−no-data-window
Do not create the data window upon start-up.

−−no-debugger-console
Do not create the debugger console upon start-up.

−−no-disassemble
Do not disassemble the source code.

−−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.

−−no-glyphs
Display the current execution position and breakpoints as text characters. Do not use glyphs.

−−no-lesstif-hacks
Equivalent to '**−−lesstif-version 1000**'. Deprecated.

−−no-source-window
Do not create the source window upon start-up.

−−no-value-tips
Disable value tips.

−−nw Do not use the X window interface. Start the inferior debugger on the local host.

−−perl Run Perl as inferior debugger.

−−pydb
Run PYDB as inferior debugger.

−−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 Motif implementations do not work well with Athena widgets. See also **−−scrolled-graph-editor**, below.

−−play *log-file*
Recapitulate a previous DDD session. Invoke 'ddd **−−PLAY** *log-file*' as inferior debugger, simulating the inferior debugger given in *log-file* (see below). This is useful for debugging DDD.

−−PLAY *log-file*
Simulate an inferior debugger. *log-file* is a '**$HOME/.ddd/log**' file as generated by some previous DDD session. When a command is entered, scan the *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 [*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 '**REMOTE DEBUGGING**', above.

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

−−scrolled-graph-editor
Use Motif scroll bars to scroll the data window. This is the default in most DDD configurations. See also **−−panned-graph-editor**, above.

−−source-window
Create the source window upon start-up.

−−status-at-bottom
Place the status line at the bottom of the source window.

−−status-at-top
Place the status line at the top of the source window.

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

−−toolbars-at-bottom
Place the toolbars the bottom of the window.

−−toolbars-at-top
Place the toolbars at the top of the window.

−−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 '**$HOME/.ddd/log**', such that you can also do a post-mortem debugging.

−−tty 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.

−−value-tips
Enable value tips.

−−version
Show the DDD version and exit.

−−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.

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

−−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.

−−xdb Run XDB as inferior debugger.

ACTIONS

The following DDD actions may be used in translation tables.

General Actions

These actions are used to assign the keyboard focus.

ddd-get-focus ()
Assign focus to the element that just received input.

ddd-next-tab-group ()
Assign focus to the next tab group.

ddd-prev-tab-group ()
Assign focus to the previous tab group.

ddd-previous-tab-group ()
Assign focus to the previous tab group.

Data Display Actions

These actions are used in the DDD graph editor.

end () End the action initiated by **select**. Bound to a button up event.

extend ()
Extend the current selection. Bound to a button down event.

extend-or-move ()
Extend the current selection. Bound to a button down event. If the pointer is dragged, move the selection.

follow ()
Continue the action initiated by **select**. Bound to a pointer motion event.

graph-select ()
Equivalent to **select**, but also updates the current argument.

graph-select-or-move ()
Equivalent to **select-or-move**, but also updates the current argument.

graph-extend ()
Equivalent to **extend**, but also updates the current argument.

graph-extend-or-move ()
Equivalent to **extend-or-move**, but also updates the current argument.

graph-toggle ()
Equivalent to **toggle**, but also updates the current argument.

graph-toggle-or-move ()
Equivalent to **toggle-or-move**, but also updates the current argument.

graph-popup-menu ([graph|node|shortcut])
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 ()
Dereference the selected display.

graph-detail ()
Show or hide detail of the selected display.

graph-rotate ()

Rotate the selected display.

graph-dependent ()

Pop up a dialog to create a dependent display.

hide-edges ([**any**|**both**|**from**|**to**]**)**

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*]]**)**

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***)**

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 ()

Place all nodes on their positions and redraw the graph.

rotate ([[+|−]*degrees*]**)**

Rotate the graph around *degrees* degrees. *degrees* must be a multiple of 90. Default is **+90**.

select ()

Select the node pointed at. Clear all other selections. Bound to a button down event.

select-all ()

Select all nodes in the graph.

select-first ()

Select the first node in the graph.

select-next ()

Select the next node in the graph.

select-or-move ()

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 ()

Select the previous node in the graph.

show-edges ([**any**|**both**|**from**|**to**]**)**

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 ()

Place all nodes on the nearest grid position.

toggle ()

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 ()

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 ()
Clear the selection.

Debugger Console Actions

These actions are used in the debugger console and other text fields.

gdb-backward-character ()
Move one character to the left. Bound to **Left**.

gdb-beginning-of-line ()
Move cursor to the beginning of the current line, after the prompt. Bound to **HOME**.

gdb-control (*control-character***)**
Send the given *control-character* to the inferior debugger. The *control-character* must be specified in the form '**^***X*', where *X* is an upper-case letter or '**?**'.

gdb-command (*command***)**
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***)**
Complete current argument as if *command* was prepended. Bound to **Ctrl+T**.

gdb-complete-command ()
Complete current command line in the debugger console. Bound to **TAB**.

gdb-complete-tab (*command***)**
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***)**
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 ()
Move cursor to the end of the current line. Bound to **End**.

gdb-forward-character ()
Move one character to the right. Bound to **Right**.

gdb-insert-graph-arg ()
Insert the contents of the data display argument field '**()**'.

gdb-insert-source-arg ()
Insert the contents of the source argument field '**()**'.

gdb-interrupt ()
If DDD is in incremental search mode, exit it; otherwise call **gdb-control(^C)**.

gdb-isearch-prev ()
Enter reverse incremental search mode. Bound to **Ctrl+B**.

gdb-isearch-next ()
Enter incremental search mode. Bound to **Ctrl+F**.

gdb-isearch-exit ()
Exit incremental search mode. Bound to **ESC**.

gdb-next-history ()
Recall next command from history. Bound to **Down**.

gdb-prev-history ()

Recall previous command from history. Bound to **Up**.

gdb-previous-history ()

Recall previous command from history. Bound to **Up**.

gdb-process ([*action* **[,** *args...* **]])**

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 ()

If the '**selectAllBindings**' resource is set to **Motif**, perform *beginning-of-line*. Otherwise, perform *select-all*. Bound to **Ctrl+A**.

gdb-set-line (*value***)**

Set the current line to *value*. Bound to **Ctrl+U**.

Source Window Actions

These actions are used in the source and code windows.

source-delete-glyph ()

Delete the breakpoint related to the glyph at cursor position.

source-double-click ([*text-action* [,*line-action* [,*function-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 a '**(**', 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 ()

Initiate a drag on the glyph at cursor position.

source-drop-glyph ([*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 ()

End selecting a word.

source-follow-glyph ()

Continue a drag on the glyph at cursor position. Usually bound to some motion event.

source-popup-menu ()

Pop up a menu, depending on the location.

source-set-arg ()

Set the argument field to the current selection. Typically bound to some selection operation.

source-start-select-word ()

Start selecting a word.

source-update-glyphs ()

Update all visible glyphs. Usually invoked after a scrolling operation.

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
‘**Break at ()**’ button.

clear_at
‘**Clear at ()**’ button.

ddd
DDD icon.

delete
‘**Delete ()**’ button.

disable
‘**Disable**’ button.

dispref
‘**Display * ()**’ button.

display
‘**Display ()**’ button.

drag_arrow
The execution pointer (being dragged).

drag_cond
A conditional breakpoint (being dragged).

drag_stop
A breakpoint (being dragged).

drag_temp
A temporary breakpoint (being dragged).

enable
‘**Enable**’ button.

find_forward
‘**Find**>> **()**’ button.

find_backward
‘**Find**<< **()**’ button.

grey_arrow
The execution pointer (not in lowest frame).

grey_cond
A conditional breakpoint (disabled).

grey_stop
A breakpoint (disabled).

grey_temp
A temporary breakpoint (disabled).

hide
‘**Hide ()**’ button.

lookup
‘**Lookup ()**’ button.

maketemp
‘**Make Temporary**’ button.

new_break
‘**New Breakpoint**’ button.

new_display
‘**New Display**’ button.

new_watch
‘**New Watchpoint**’ button.

plain_arrow
The execution pointer.

plain_cond
A conditional breakpoint (enabled).

plain_stop
A breakpoint (enabled).

plain_temp
A temporary breakpoint (enabled).

print
‘**Print ()**’ button.

properties
‘**Properties**’ button.

rotate
‘**Rotate ()**’ button.

set
‘**Set ()**’ button.

show
‘**Show ()**’ button.

signal_arrow
The execution pointer (stopped by signal).

undisplay
‘**Undisplay ()**’ button.

unwatch
‘**Unwatch ()**’ button.

watch
‘**Watch ()**’ button.

ENVIRONMENT

DDD is controlled by the following environment variables:

DDD_NO_SIGNAL_HANDLERS
If set, DDD does not catch fatal errors. This is sometimes useful when debugging DDD.

DDD_STATE
Root of DDD state directory. Default is ‘**$HOME/.ddd/**’.

DDD_SESSION
If set, indicates a session to start, overriding all options. This is used by DDD when restarting itself.

DDD_SESSIONS DDD session directory. Default is ‘**$DDD_STATE/sessions/**’.

EDITOR The text editor to invoke for editing source code. See the ‘**editCommand**’ resource, above.

VSL_INCLUDE Where to search for VSL include files. Default is the current directory.

WWWBROWSER The WWW browser to invoke for viewing the DDD WWW page. See the ‘**wwwCommand**’ resource, above.

XEDITOR The X editor to invoke for editing source code. See the ‘**editCommand**’ resource, above.

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. If the debuggee runs in a separate execution window, the debuggee’s **TERM** value is set according to the ‘**termType**’ resource (see ‘**RESOURCES**’, above).

TERMCAP Set to ‘’ (none), the DDD terminal capabilities.

PAGER Set to ‘**cat**’, the preferred DDD pager.

FILES

$HOME/.ddd/ DDD state directory.
$HOME/.ddd/init Individual DDD resource file. DDD options are saved here.
$HOME/.ddd/history Default DDD command history file.
$HOME/.ddd/lock DDD lock file; indicates that a DDD is running.
$HOME/.ddd/log Trace of the current interaction between DDD and the inferior debugger.
$HOME/.ddd/sessions/
DDD session directory. One subdirectory per session.
$HOME/.ddd/sessions/*session***/dddcore**
DDD core file for *session*.
$HOME/.ddd/sessions/*session***/init**
DDD resource file for *session*.
$HOME/.ddd/sessions/*session***/history**
DDD command history for *session*.
$HOME/.ddd/sessions/.ddd/
The DDD ‘restart’ session.
$HOME/.ddd/tips DDD tips resource file. Contains the number of the next tip of the day.
$HOME/.gdbinit GDB initialization file.
$HOME/.dbxinit DBX initialization file.
$HOME/.dbxrc Alternate DBX initialization file.
$HOME/.xdbrc XDB initialization file.
$HOME/.gnuplot Gnuplot initialization file.
$HOME/.dddinit Old-style DDD initialization file; used only if **$HOME/.ddd/init** does not exist.

SEE ALSO

X(1), **gdb**(1), **dbx**(1), **xdb**(1), **perldebug**(1), **remsh**(1), **rsh**(1), **gnuplot**(1),

‘ **gdb** ’ entry in **info**.

Using GDB: A Guide to the GNU Source-Level Debugger, by Richard M. Stallman and Roland H. Pesch.

jdb—The Java Debugger, at **http://java.sun.com/** (and its mirrors) in **/products/jdk/1.1/docs/tooldocs/solaris/jdb.html**

Java Language Debugging, at **http://java.sun.com/** (and its mirrors) in **/products/jdk/1.1/debugging/**

The Python Language, at **http://www.python.org/** and its mirrors.

DDD—A Free Graphical Front-End for UNIX Debuggers, by Andreas Zeller and Dorothea Lütkehaus, Computer Science Report 95-07, Technische Universität Braunschweig, 1995.

DDD – ein Debugger mit graphischer Datendarstellung, by Dorothea Lütkehaus, Diploma Thesis, Technische Universität Braunschweig, 1994.

The DDD *FTP site,*

ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/ddd/

The DDD *WWW page,*

http://www.cs.tu-bs.de/softech/ddd/

The DDD *Mailing List,*

ddd-users@ips.cs.tu-bs.de

For more information on this list, send a mail to

ddd-users-request@ips.cs.tu-bs.de .

LIMITATIONS

General Limitations

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. This problem can be avoided by running the program in the separate execution window.

Output that confuses DDD includes:

- Primary debugger prompts (e.g. '**(gdb)** ' or '**(dbx)** ')
- 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 should run it in the separate execution window.

If the inferior debugger changes the default TTY settings, for instance through a '**stty**' command in its initialization file, DDD will likely become confused. The same applies to debugged programs which change the default TTY settings.

Limitations using 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 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 $pc**' command to ensure the program counter value is updated correctly at each stop. You may also enter the command in **$HOME/.gdbinit** or (better yet) upgrade to the most recent GDB version.

Limitations using 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.

Limitations using 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.

Limitations using 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.

The JDB '**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.

Limitations using 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.

REPORTING BUGS

If you find a bug in DDD, please send us a bug report. We will either attempt to fix the bug—or include the bug description in the DDD '**BUGS**' file, such that others can attempt to fix it. (Instead of sending bug reports, you may also send *fixes*; DDD is an excellent tool for debugging itself :-)

Where to Send Bug Reports

We recommend that you send bug reports for DDD via electronic mail to

ddd-bugs@ips.cs.tu-bs.de

As a last resort, send bug reports on paper to:

Technische Universität Braunschweig
Abteilung Softwaretechnologie
DDD-Bugs
Bültenweg 88
D-38092 Braunschweig
GERMANY

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 Motif installations, for instance, or bugs in the X server or Motif 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 '**$HOME/.ddd/log**'.) Compare the debugger output to the output of DDD and determine which one is wrong.

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.

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 '**$HOME/.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.

HISTORY

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: Persistent 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.

EXTENDING DDD

If you have any contributions to be incorporated into DDD, please send them to '**ddd@ips.cs.tu-bs.de**'. For suggestions on what might be done, see the file '**TODO**' in the DDD distribution.

DDD NEEDS YOUR SUPPORT!

DDD needs your support! If you have any success stories related to DDD, please write them down on a picture postcard and send them to us:

Technische Universität Braunschweig
Abteilung Softwaretechnologie
Bültenweg 88
D-38092 Braunschweig
GERMANY

You may also leave a message in the *DDD Guestbook*. It is accessible via the DDD WWW page,

http://www.cs.tu-bs.de/softech/ddd/ .

PRINTING THIS MANUAL

Invoke DDD with the '**−−manual**' option to show this manual page on standard output. This text output is suitable for installation as formatted manual page (as '**/usr/local/man/cat1/ddd.1**' or similar) on UNIX systems.

A PostScript copy of this manual page, including several DDD screen shots and diagrams, is included in the DDD source distribution and available separately as '**ddd.man.ps.gz**' in

ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/ddd/doc/

This directory also contains other documentation related to DDD.

A ROFF copy of this manual page, suitable for installation as manual page on UNIX systems (as '**/usr/local/man/man1/ddd.1**' or similar), is included in the DDD source distribution.

COPYRIGHT

DDD

DDD is Copyright © 1995, 1996, 1997, 1998 Technische Universität Braunschweig, Germany.

DDD 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.

DDD 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 License for more details.

You should have received a copy of the License along with DDD. If not, invoke DDD with the '**−−license**' option; this will print a copy on standard output. To read the License from within DDD, use '**Help→DDD License**'.

DDD Manual

This DDD manual is Copyright © 1995, 1996, 1997, 1998 Technische Universität Braunschweig, Germany.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be included in translations approved by the Free Software Foundation instead of in the original English.