NAME

ddd, xddd - the data display debugger (v1.4b)

SYNOPSIS

ddd [**−−help**] [**−−nw**] [**−−gdb**] [**−−dbx**] [**−−debugger** *gdb-name*] [**−−host** *hostname* [**−−login** *username*]] [**−−vsl−library** *library*] [**−−vsl−path** *path*] [**−−trace−dialog**] [**−−trace−shell**] [**−−exec−window**] [**−−no−exec−window**] [**−−attach−windows**] [**−−separate−windows**] [**−−scrolled−graph**] [**−−panned−graph**] [**−−tty**] [**−−fullname**] [**−−version**] [**−−configuration**] [**−−manual**] [*gdb-options*] [*x-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 *wrapper* around an inferior GDB or DBX debugger. In addition to the GDB or DBX command-line interface, DDD provides a *graphical user interface* to support common debugging tasks. The DDD *graphical data display* allows for interactive exploration of data structures.

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 uses GDB as inferior debugger. Use

ddd −−dbx program

to run DBX as inferior debugger for the program.

To learn more about DDD options, run

ddd −−help

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

GETTING HELP

You can get help on any visible DDD item by pressing the '**F1**' key and moving the question mark arrow on the item you want help for. Clicking on the item pops up a help text.

You can get help on GDB and DBX commands by entering the '**help**' command at the '**(gdb)**' or '**(dbx)**' prompt (see next section).

ENTERING COMMANDS

In the *command window*, you can interact with the GDB or DBX command interface. Enter GDB commands at the '**(gdb)**' prompt, and DBX commands at the '**(dbx)**' prompt. Arbitrary GDB and DBX commands can be used.

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

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

Here are some of the most frequently needed debugger commands:

break [*file***:**]*function*
: (GDB only) Set a breakpoint at *function* (in *file*).

stop in *function*
: (DBX only) Set a breakpoint at *function.*

run [*arglist*]
: Start your program (with *arglist*, if specified).

where
: Display the program stack.

print *expr*
: Display the value of an expression.

cont
: Continue running your program (after stopping, e.g. at a breakpoint).

next
: Execute next program line (after stopping); step *over* any function calls in the line.

step
: Execute next program line (after stopping); step *into* any function calls in the line.

help [*name*]
: Show information about the command *name*, or general usage information.

quit
: Exit DDD.

For full details on GDB commands, see *Using GDB: A Guide to the GNU Source-Level Debugger*, by Richard M. Stallman and Roland H. Pesch. The same text is available online as the **gdb** entry in the **info** program.

For full details on DBX commands, see the DBX documentation.

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.

To open a program to be debugged, select '**Open Program**' from the '**File**' menu.

To open a core dump for the program, select '**Open Core**' from the '**File**' menu.

To open an arbitrary source file, select '**Open Source**' from the '**File**' menu.

BREAKPOINTS

If the source of the debugged program is available, the *source window* displays its current source text. (If a source text cannot be found, use the GDB '**directory**' command to specify source text directories).

Using the source window, 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 the *left mouse button* on the source line number and then on the '**Break()**' button.

As an alternative, you can simply press the *right mouse button* on the source line number and select the '**Set Breakpoint**' item from the popup menu.

As another alternative, you can enter the line number in the argument field (indicated by '**():**') and click on the '**Break()**' button.

As yet another alternative, you can select '**Edit Breakpoints...**' in the '**Source**' menu, click on the '**New**' button and enter the line number.

And finally, you can also use the GDB '**break**' command or the DBX '**stop**' command. Type '**help break**' at the '**(gdb)**' prompt (or '**help stop**' at the '**(dbx)**' prompt) for details.

(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 GDB and DBX users* prefer the command line interface.)

Breakpoints are indicated by a plain stop sign, or as '**#***n***#**', where *n* is the breakpoint number.

Setting Breakpoints by Name

If the function name is visible, click with the *left mouse button*' on the function name. The function name is copied to the argument field. Click on the '**Break()**' button to set a breakpoint there.

As a shorter alternative, you can simply press the *right mouse button* on the function name and select the '**break**' item from the popup menu.

As another alternative, you can enter the function name in the argument field (possibly using name completion by pressing the '**Tab**' key) and click on the '**Break()**' button.

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

Finally, you can use the GDB '**break**' command or the DBX '**stop**' command. Type '**help break**' at the '**(gdb)**' prompt (or '**help stop**' at the '**(dbx)**' prompt) for details.

Disabling and Enabling Breakpoints

Note: DBX does not support breakpoint disabling.

Press the *right mouse button* on the breakpoint name and select the '**Disable Breakpoint**' item (or the '**Enable Breakpoint**' item, respectively) from the popup menu.

As an alternative, you can select the breakpoint and click on '**Disable**' or '**Enable**' in the Breakpoint editor (invoked through '**Edit Breakpoints...**' in the '**Source**' menu).

Disabled breakpoints are indicated by a grey stop sign, or '**_***n***_**', where *n* is the breakpoint number.

Finally, you can use the GDB '**disable**' command. Type '**help disable**' at the '**(gdb)**' prompt for details.

Temporary Breakpoints

A *temporary breakpoint* is immediately deleted as soon as it is reached. To set a temporary breakpoint, press the *right mouse button* on the source line number and select the '**Set Temporary Breakpoint**' item from the popup menu.

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 source line number and immediately continues execution. In GDB, execution stops when a source line greater than the source line is reached, or when the current stack frame is exited. In DBX, execution stops when the temporary breakpoint is reached.

Deleting Breakpoints

If the breakpoint is visible, click with the *left mouse button* on the breakpoint. The breakpoint location is copied to the argument field. Click on the '**Clear()**' button to delete all breakpoints there.

If the function name is visible, click with the *left mouse button* on the function name. The function name is copied to the argument field. Click on the '**Clear()**' button to set a breakpoint there.

As a shorter alternative, you can simply press the *right mouse button* on the function name and select the '**clear**' item from the popup menu.

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

Finally, you can use the GDB '**clear**' and '**delete**' commands. Type '**help clear**' or '**help delete**' at the '**(gdb)**' prompt for details.

More Breakpoint Features

Using the Breakpoint editor (invoked through '**Edit Breakpoints...**' in the '**Source**' menu), you can specify breakpoint conditions and ignore counts. Click on the '**Condition...**' and '**Ignore Count...**' buttons and click on the '**Help**' button.

More breakpoint features can be invoked through the command window. Enter '**help breakpoints**' at the '**(gdb)**' prompt.

LOOKING UP ITEMS

Searching Definitions

If you wish to lookup a specific function or variable definition whose name is visible in the source text, click with the *left mouse button* on the function or variable name. The name is copied to the argument field. Click on the '**Lookup()**' button to find its definition.

As a shorter alternative, you can simply press the *right mouse button* on the function name and select the '**lookup**' item from the popup menu.

As another alternative, you can enter the function name in the argument field and click on the '**Lookup()**' button to find its definition.

Finally, you can use the GDB '**info line**' command. Type '**help info line**' at the '**(gdb)**' prompt for details.

Searching Items in the Source Text

If the item you wish to search is visible in the source text, click with the *left mouse button* 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 '**Find Words Only**' option in the '**Source Options**' menu.

Finding Breakpoints

If you wish to lookup a specific breakpoint, select the '**Lookup**' item from the '**Breakpoints**' menu. 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.

Looking up Previous Locations

Use the '**Back**' and '**Forward**' buttons to lookup previous or next looked-up locations. The location found is underlined.

RUNNING THE PROGRAM

Starting Program Execution

Select '**Run**' from the '**Program**' menu to start execution of the debugged program. You will then be prompted for the arguments to give 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 '**OK**' button to start execution with the selected arguments.

To run your program again, with the same arguments, select '**Run Again**' from the '**Program**' menu or press the '**Run**' button below. You may also enter '**run**', followed by arguments at the '**(gdb)**' or '**(dbx)**' prompt instead.

When the program is first started, DDD invokes an *execution window*, where the program terminal input and output is shown. (You can disable this feature by starting DDD with the '**−−no−exec−window**' option.)

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 **Ctrl+C** 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 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 '**Temporary Breakpoints**', above, for a discussion.

Examining the Stack

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

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 type the GDB and DBX '**up**' and '**down**' commands at the '**(gdb)**' or '**(dbx)**' prompt.

EXAMINING DATA

In DDD, the usual way to examine data is using 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.

Displaying Variable Values

To create a new display, select the variable by clicking the *left mouse button* 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.

As a shorter alternative, you can simply press the *right mouse button* on the variable name and select the '**display**' item from the popup menu.

As another alternative, you may also enter the variable name in the argument field and press the ‘**Display()**’ button.

If the data window is visible, you may invoke the ‘**Edit Displays...**’ item from the ‘**Data**’ menu and click on the ‘**New...**’ button. Enter the variable name in the resulting prompt dialog.

Finally, you may enter

graph display *expr*

at the ‘**(gdb)**’ or ‘**(dbx)**’ prompt.

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 the *left mouse button*. The resulting expression is shown in the *argument field*, below.

You can *extend* an existing selection by pressing the **Shift** *Key while selecting.*

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

Selecting Multiple Displays

Multiple displays are selected by pressing and holding the *left mouse button* 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 the mouse button is released.

By double-clicking on a display, the display itself and all connected displays are automatically selected.

Selecting Display Parts

If a display is composed from several values (that is, a C/C++ *class*, *struct*, *union*, or *array*; or a Pascal/Modula *RECORD* or *ARRAY*), you can select individual parts by clicking on their names or values. The resulting expression is shown in the *argument field*, below.

Selection of multiple display parts is not supported.

Manipulating Aggregate Values

Aggregate Values 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 the *left mouse button* on its name or value and click on the ‘**Show()**’ button.

To hide details about an aggregate, select the aggregate by clicking the *left mouse button* on its name or value and click on the ‘**Hide()**’ button.

As a faster alternative, you can also press the *right mouse button* on the aggregate and select the appropriate menu item.

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 the *right mouse button* on the array and select the ‘**Rotate**’ menu item.

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 click on the ‘**Dependent()**’ button. A prompt dialog appears where you can enter a new expression *EXPR* in the argument field. By clicking on ‘**OK**’, the new display showing *EXPR* is created.

As a faster alternative, you can also press the *right mouse button* on the originating display part and select the ‘**Dependent Display...**’ menu item.

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.

Dereferencing Pointers

Since so many data structures are realized using pointers, there is a shortcut for creating dependent displays showing the value of a dereferenced pointer.

To dereference a pointer, select the originating pointer value or name and click on the ‘**Dereference()**’ button. A new display showing the dereferenced pointer value is created.

As a faster alternative, you can also press the *right mouse button* on the originating pointer value or name and select the ‘**Dereference**’ menu item.

Displaying Multiple Array Items

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]**’.

As an alternative, you can use GDB *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.

Deleting Displays

To delete a single display, select it and click on the ‘**Delete()**’ button. As an alternative, you can also press the *right mouse button* 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 through the ‘**Edit Displays...**’ item in the ‘**Data**’ menu). Select any number of display items in the usual way and delete them by pressing ‘**Delete**’.

As an alternative, you can also type

undisplay *NUM...*

at the ‘**(gdb)**’ or ‘**(dbx)**’ prompt, where *NUM...* is a space-separated list of the displays to delete. You may need to refresh the data window afterwards (see below).

Disabling or Enabling Displays

Note: DBX does not support display disabling.

Unlike a *deleted* display item, a *disabled* display item remains in the display, but its value is not shown until it is enabled again.

Displays are enabled and disabled the same way they are deleted, using the ‘**Disable()**’/‘**Enable()**’ button, the ‘**Disable**’/‘**Enable Display**’ popup menu item, or the ‘**Disable**’/‘**Enable**’ buttons in the Display Editor.

The corresponding GDB commands are

disable display *NUM*...

and

enable display *NUM*...

respectively.

Refreshing the Data Window

Although the data window refreshes itself automatically each time the program stops, there are some situations where you should refresh it explicitly.

The data window should be refreshed whenever:

- you disabled, enabled or deleted displays items at the '**(gdb)**' or '**(dbx)**' prompt,
- the display stays unchanged although all items should be deleted (e.g. because you chose another file to debug),
- you think it could be useful.

You can refresh the data window by selecting the '**Refresh Displays**' item in the '**Data**' menu.

As an alternative, you can press the *right mouse button* on the background of the data window and select the '**Refresh Display**' item.

Typing

graph refresh

at the '**(gdb)**' or '**(dbx)**' prompt has the same effect.

GRAPH EDITING

The DDD data window offers some basic functionality to manipulate the display graph.

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 the *left mouse button* 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.**

Aligning Displays

You can align all displays on the nearest grid position by selecting '**Align Displays**' from the '**Graph**' menu. This is useful for keeping edges horizontal or vertical.

You can enforce alignment by selecting '**Snap to Grid**' in the '**Data Options**' menu. If '**Snap to Grid**' 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 '**Layout Graph**' from the '**Data**' menu.

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. You can make edge hints visible by selecting '**Show Edge Hints**' in the '**Data Options**' menu. Edge hints can be moved around like arbitrary displays.

To enable a more compact layout, you can set the '**Compact Layout**' option in the '**Data Options**' menu.

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 the '**Automatic Layout**' option in the '**Data Options**' menu. If '**Automatic Layout**' is enabled, the graph is layouted after each change.

Rotating the Display Graph

You can rotate the entire graph clockwise by 90 degrees by selecting '**Rotate Graph**' from the '**Data**' menu.

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.

To print the graph on a PostScript printer, select the '**Print Graph...**' item from the '**File**' menu. Enter the printing command in the '**Printer Command**' field. Click on the '**Print**' button to start printing.

To re-print the display graph using the previous settings, select the '**Print Again**' item from the '**File**' menu.

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 it in another document;
- XFIG - suitable for editing, using the XFIG graphic editor, or for conversion into other formats using the TRANSFIG package.

MACHINE-LEVEL DEBUGGING

Note: Machine-level support is not available with DBX.

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 '**Display Machine Code**' from the '**Source Options**' menu. With '**Display 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.

The machine code window works very much like the source window. You can set, clear, and change breakpoints by selecting the address and pressing a '**Break()**' or '**Clear()**' button; the usual popup menus are also available. Breakpoints and the current execution position are displayed simultaneously in both source and machine code.

The '**Lookup()**' button can be used to look up the machine code for a specific function—or the function for a specific address. Just click on the location in one window and press '**Lookup()**' to see the corresponding code in the other window.

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

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 '**Show Registers**' from the '**Stack**' menu.

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.

EDITING SOURCE CODE

In DDD itself, you cannot change the source file currently displayed. DDD allows you to invoke a *text editor* instead. To invoke a text editor for the current source file, select the '**Edit**' button or the '**Edit Source...**' item from the '**Source**' menu.

The editor command is specified in the '**editCommand**' resource. By default, DDD tries to invoke the first available editor from the following list:

- The editor specified in the '**XEDITOR**' environment variable;
- The '**gnuclient**' program, invoking a separate window from a running **Emacs** editor;
- The '**emacsclient**' program, invoking a separate window from a running **Emacs** editor;
- A '**xterm**' terminal emulator running the editor specified in the '**EDITOR**' environment variable;
- A '**xterm**' terminal emulator running the '**vi**' editor.

After the editor has exited, the source code shown is automatically brought up-to-date.

If you have DDD and an editor running in parallel, you can also update the source code manually by selecting the '**Reload Source...**' item from the '**Source**' menu. This reloads the source code shown from the source file. As DDD automatically reloads the source code if the debugged program has been recompiled, this should seldom be necessary.

QUITTING DDD

To exit DDD, select '**Exit**' from the '**File**' menu. You may also type the '**quit**' command at the '**(gdb)**' or '**(dbx)**' prompt. GDB also accepts the '**q**' command or an end-of-file character (usually **Ctrl+D**).

An interrupt (**Ctrl+C**) does not exit from DDD, but rather terminates the action of any GDB or DBX 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.

You may also interrupt DDD using an interrupt signal (SIGINT)—for instance by pressing **Ctrl+C** in the shell DDD was started from, or by using the UNIX '**kill**' command. This interrupts any DDD action; the inferior GDB or DBX debugger is interrupted as well. Use this as a last resort in emergencies only—if DDD hangs, for example.

REMOTE DEBUGGING

It is possible to have GDB or DBX 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.

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*

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.
- 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 command window.
- 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−shell−commands**' option. This prints the shell commands issued by DDD on standard error.

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 GDB resp. DBX. 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 receives a EOF on its standard input. Also, do not try to run DDD with DDD as inferior debugger.

Using DDD with FSF 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 via **ESC ESC**:

(setq gdb-command-name "ddd")

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

Using DDD with XXGDB

Invoke **xxgdb** as

xxgdb −db_name ddd -db_prompt '(gdb) '

CUSTOMIZING DDD

You can set up your personal DDD preferences by using the '**Options**' menu from the menu bar. These options affect your running DDD process only, unless you save these options for a later DDD invocation.

General Options

These items are found in the '**Options**' menu:

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

If you want to use **TAB** key completion in all text windows, set the '**Global Tab Completion**' option. This is useful if you have pointer-driven keyboard focus (see below) and no special usage for the **TAB** key.

If you want to run your debugged process in a separate terminal emulator window, set the '**Run in Execution Window**' option. This is useful for programs that have special terminal requirements not provided by the debugger window, as raw keyboard processing or terminal control sequences.

If you want to save changed options automatically for the next DDD invocation, set the '**Save Options on Exit**' option.

If you want to save the command history for the next DDD invocation, set the '**Save Options on Exit**' option. Note: This option is always set if your GDB is set up to save its command history.

Source Options

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 the '**Find Words Only**' option.

If you wish to display machine code of selected funtions, set the '**Display Machine Code**' option. This makes DDD run a little slower, so it is disabled by default.

As of this DDD release, the current execution position and breakpoints are indicated by symbols (“glyphs”) in the source text. Previous releases used special texts for this purpose. If you wish to disable glyphs, unset the '**Display Glyphs**' option. This also makes DDD run slightly faster, especially when scrolling.

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 Sources**' 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.

Data Options

By default, DDD displays a grid in the data window to facilitate display alignment. If you prefer not to have a grid, unset the '**Show Grid**' option.

Layouting the display 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. These edge hints can be selected and moved around like displays. You can make edge hints visible by setting the '**Show Edge Hints**' option.

To enforce alignment, you can set the '**Snap to Grid**' option. If '**Snap to Grid**' is enabled, displays can be moved on grid positions only.

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 '**Automatic Layout**' option. If '**Automatic Layout**' is enabled, the graph is layouted after each change.

Startup Options

By default, DDD uses three separate windows for commands, source, and data. To attach the source and data windows to the command window, creating one single big DDD window, set the '**Windows**' option to '**One Single Window**'. This change takes only effect after you have saved options and restarted DDD. See also the '**−−attach−windows**' and '**−−separate−windows**' options, below.

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**'. This change takes only effect after you have saved options and restarted DDD. See also the '**−−panned−graph**' and '**−−scrolled−graph**' options, below.

By default, DDD runs with GDB as inferior debugger. To change this default, set the '**Debugger Type**' option to another debugger. This change takes only effect after you have saved options and restarted DDD. See also the '**−−gdb**' and '**−−dbx**' options, below.

Saving Options

You can save the current option settings by selecting '**Save Options**' in the '**Options**' menu. Options are saved in a file named '**.dddinit**' in your home directory.

Other Customizations

Other personal DDD resources can also be set in your '**.dddinit**' file. To find out which resources are available, and how they are set by default, look at the '**Ddd**' application defaults file, which is shipped with the DDD source. For individual customization, copy the lines of resources you want to change into your '**.dddinit**' file and modify them as you like.

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 '**PROJECTS**' in the DDD distribution.

If you wish something done in DDD, but cannot do it yourself, you can *finance* specific DDD extensions. For details, contact '**ddd@ips.cs.tu-bs.de**'; direct mail is also welcome.

Some of the people offering DDD support may also accept offers to extend DDD. For a list of people offering DDD support, see the file '**SERVICE**' in the DDD distribution.

IF YOU LIKE DDD...

If you appreciate this software, or have any suggestions, please send us a picture postcard. Our address is:

Technische Universitaet Braunschweig
Institut fuer Programmiersprachen und Informationssysteme
Abteilung Softwaretechnologie
Gaussstrasse 17
D-38092 Braunschweig
GERMANY

You may also leave a message in the DDD guestbook (accessible via the DDD WWW page, '**http://www.cs.tu-bs.de/softech/ddd/**')

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. Options not listed here are passed to the inferior debugger.

−−attach−windows
Attach the source and data windows to the command window, creating one single big DDD window. This gives the good ole '**xxgdb**' or '**xdbx**' feeling.

−−attach−source−window
Attaches only the source window to the command window.

−−attach−data−window
Attaches only the source window to the command window.

−−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.

−−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 GDB and DBX cannot be invoked as '**gdb**' or '**dbx**', respectively.

−−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.

−−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.

−−gdb Run the GDB debugger as inferior debugger.

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

−−host *hostname*
Run the inferior debugger on the remote host *hostname*. See **REMOTE MODE**, above.

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

−−manual
Show this manual page and exit.

−−no−exec−window
Do not run the debugged program in a specially created execution window; use the command window instead. Useful for programs that have little terminal input/output, or for remote debugging.

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

−−panned−graph
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**, below.

−−separate−windows
Separate the command, source and data windows. This is the default. See also the '**−−attach**' options, above.

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

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

−−trace−dialog
Show the interaction between DDD and the inferior debugger on standard error. This is useful for debugging DDD.

−−trace−shell
Show the shell commands issued by DDD on standard error. This is useful for debugging DDD.

−−trace
Show both interaction and shell commands.

−−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.

−−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).

RESOURCES

See the '**Ddd**' application defaults file for full information about customizing DDD. The '**Ddd**' file comes with the DDD distribution.

FILES

$HOME/.dddinit	Individual DDD resource file. DDD options are saved here.
$HOME/.gdbinit	GDB initialization file.
$HOME/.dbxinit	DBX initialization file.

SEE ALSO

X(1), **gdb**(1), **dbx**(1), **remsh**(1), **rsh**(1)

' **gdb** ' entry in **info.**

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

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

DDD — ein Debugger mit graphischer Datendarstellung, by Dorothea Luetkehaus, Diploma Thesis, Technische Universitaet Braunschweig, 1994.

The DDD *FTP site*,

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

The DDD *World-Wide-Web 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

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 print pretty
set print repeats 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. If you want to display C strings (i.e., '**char ***'-fields), it might be useful to change the settings of '**print repeats**' and '**print elements**'. But to recognize array structures correctly, you should have '**print repeats**' set to 0. Also, do not change the setting of '**print elements**' while there are array structures 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 Modula-2 or Pascal programs, DDD always numerates array elements starting with zero, instead of using correct array subscripts. This shall be fixed in a future DDD release.
Using DBX 3.x or later as inferior debugger, DDD strips C-style and C++-style comments from the DBX2+2 output in order to operate properly. This also affects the output of the debugged program when sent to the command window. Using the separate execution window avoids these problems.

General Limitations

If command output is sent to the command window, it is impossible for DDD to distinguish between the output of the debugged program and the output of the inferior debugger. For instance, if your program output includes debugger prompts, DDD will become confused. Using the separate execution window avoids these problems.
Paper sizes larger than 2,147,483,648pt (757km, 470 miles) can not be handled.

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 Universitaet Braunschweig
Institut fuer Programmiersprachen und Informationssysteme
Abteilung Softwaretechnologie
DDD-Bugs
Gaussstrasse 17
D-38092 Braunschweig
GERMANY
Fax: +49 531 391-8140

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 GDB or DBX debugger; although unlikely, they may have bugs. To find out whether a bug was caused by GDB or DBX, run DDD as

ddd −−trace−dialog

This shows the dialog between DDD and the inferior debugger on standard error while DDD is running. Compare the debugger output to the DDD output 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-4.14**' or '**dbx as shipped with Solaris 2.4**').
- The compiler you used to compile DDD and its version (e.g., '**gcc-2.6.3**').

- 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".
- If possible, include a *dialog trace* showing the interaction between DDD and the inferior debugger. Such a dialog trace is created by invoking DDD with the '**--trace**' option. Include all trace output from the DDD invocation up to the first bug occurrence.
- 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 Luetkehaus* 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, many people have contributed to the success of DDD; see the file '**CONTRIBUTORS**' in the DDD distribution for details.

DDD 1.0 was released in May, 1995.

COPYING

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, suitable for printing, is included in the DDD source distribution and available separately as

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

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.