OPENSTEP 4.2 Release Notes: The GNU Source-Level Debugger

This file contains information about GDB, the GNU Debugger. For more information, see the debugging chapter in the *OPENSTEP Development Tools Reference* manual. On Mach, you may also refer to the **gdb**(1) manual page.

Notes Specific to GDB on Mach:

New GDB Version

The GDB debugger in OpenStep 4.0 for Mach (and later versions) is based on the version 4.14 release from GNU/FSF. This brings with it many bug fixes and new features, many of which are mentioned in the debugging chapter of the *OPENSTEP Development Tools Reference* manual.

New Features

Dynamic Link-Editor Support

GDB supports debugging of dynamic shared libraries (sometimes known as Frameworks or Bundles in the OPENSTEP world). The presence of dynamic shared libraries has some impact on debugging, which is described in this section.

Debugging symbols for dynamic shared libraries are not present in the program itself. GDB obtains them when the running program attaches and links to the shared library. This means

that before the program is actually running, GDB has no information about the contents of the dynamic shared libraries that it uses.

Because of this, it is not possible to set ordinary breakpoints in a shared library before that library has been attached. GDB provides a new command for this purpose, **future-break** or **fb**. If GDB cannot find the necessary symbols to resolve a **future-break** command, it defers the breakpoint and attempts to resolve it later, when new symbols from a shared library become available. Caveat: since the **future-break** command deals with names and symbols that are as yet unknown to the debugger, it cannot check spelling for you; if you make a spelling mistake, it will never be detected and the breakpoint will never take effect.

There is also an environment variable, **DYLD_LIBRARY_PATH**, which tells the dynamic link-editor where to search for dynamic libraries. This variable can be used to cause a library with debugging symbols to be linked, even though the library on the default path has no symbols. This environment can be set from within GDB by using the **setenv** command. In order to affect the program being debugged, it should be set before running the program.

The "view" interface

In prior releases, GDB supported a GUI interface that used the NEXTSTEP Edit application as a source file viewer (invoked by the **view** command). Edit has been replaced by Project Builder as the source file viewer for GDB, and the **view** command now connects GDB to Project Builder. You must start Project Builder yourself before giving GDB the **view** command (GDB will not start Project Builder automatically). Project Builder has its own user interface for interacting with GDB (see the Project Builder documentation).

Methods with Variable Number of Arguments

GDB now understands the syntax for calling a method with a variable number of arguments (for example, **[MyClass myMethod: 1, 2, 3, 4]**).

Known Problems

Debugging Apps that Use the Sybase Client Library

GDB hangs (actually, the new Sybase CT-Lib adaptor blocks) when you use the **next** command to step over a line of code which eventually causes a call to the Sybase client library.

More generally, when debugging a program that uses multiple threads it's possible to create a situation in which a deadlock will occur. Whenever the **next** or **step** commands are issued, GDB lets only the thread being debugged to execute. All other threads are suspended until the command is completed. Therefore, if you attempt to step over a line of code which tries to communicate with another thread, the program will deadlock.

To deal with problems like this, a "run-all-threads" option has been added to GDB. This controls whether or not all of the threads should execute while single-stepping. The default value for this option is "off", meaning the behavior is the same as in OPENSTEP 4.1. In order to prevent a deadlock like that described above, issue the following command in GDB:

set run-all-threads on

We recommend that you use this option only if you're experiencing a deadlock. Allowing other threads to execute while stepping through code can produce confusing results if, for example, the other threads may be changing the values of global data.

Known Problems with Dynamic Link-Editor Support

It has been observed that GDB sometimes hangs or crashes if you run a program that uses a dynamic shared library (Framework or Bundle), then recompile the dynamic shared library, and run the program again. If this happens to you, we recommend that you quit GDB and start a new debugging session every time you rebuild the library. You can use the **.gdbinit** file to help re-establish things such as breakpoints that you need in your debugging session.

Interrupting with ^C during Dynamic Symbol Loading

Immediately after you start your program running under GDB, the program will start to load

dynamic shared libraries, and GDB will begin reading symbols from these libraries. If you attempt to interrupt GDB by typing ^C (control-C) during this process, the debugger will be left in a confused internal state from which the only known recovery is to quit the debugger and start over.

Notes Specific to GDB on Windows:

GDB Version

The GDB debugger for OPENSTEP for Windows is based on the version 4.15.1 release from GNU. This brings with it many, if not most of the features of debugging on UNIX and Mach, although there are inevitably some differences.

New Features

Dynamically Loaded Library (DLL) Support

GDB supports debugging of dynamically loaded libraries (DLLs). In OPENSTEP for Windows, Frameworks and Bundles are implemented as DLL's. The presence of DLLs has some impact on debugging, which is described in this section.

Debugging symbols for dynamically-loaded libraries are not present in the program itself. GDB obtains them when the running program attaches and links to the DLL. This means that before the program is actually running, GDB has no information about the contents of the DLLs that the program uses.

Because of this, it's not possible to set breakpoints or access data in a DLL before the DLL has been attached by the program. GDB provides a new command for this purpose, **future-break** or **fb**. If GDB cannot find the necessary symbols to resolve a **future-break** command, it defers the breakpoint and attempts to resolve it later, when new symbols from a shared library become available. Caveat: since the **future-break** command deals with names and symbols that are as

yet unknown to the debugger, it cannot check spelling for you; if you make a spelling mistake, it will never be detected and the breakpoint will never take effect.

Attach and Detach

GDB's **attach** command works pretty much as it does on Mach. The process ID to attach to can be obtained from PVIEW, or by having the attachee call **getpid()** and output the result. However, when GDB attaches to an already-running process, it won't learn about symbols from DLLs that the process has already linked to.

The **detach** command is only partially useful at this time. **detach** will let the attached process continue to run, but a parent/child relationship continues to exist between the debugger and the detached process. Because of this, if you then quit the debugger, the detached process will also die.

When you attach to a running process, you will frequently find yourself in a non-debuggable area of code from which you cannot even get a backtrace. See the section on known problems, "Non-Debuggable Code" below for more information on this topic.

Add-Symbol-File

When GDB attaches to a running process, it does not read any symbols from DLLs that the process has previously linked itself to. If you need those symbols for debugging, you can explicitly cause GDB to load them by using the **add-symbol-file** command. This command takes two arguments: the fully-qualified filename of the DLL file, and a base address. For a DLL that you build, this base address will usually be the base address at which you linked the DLL plus 0x1000. However, if the address of a DLL that you build conflicts with another DLL in the process, it may automatically be assigned a new address. If you run the program under GDB (instead of attaching to it), GDB displays the address at which each DLL actually landed.

Here are the base addresses to give to the **add-symbol-file** command for the major OPENSTEP DLLs:

- 0x30001000 System
- 0x31001000 nextpdo
- 0x32011000 Foundation
- 0x34021000 AppKit
- 0x38031000 NeXTApps
- 0x3A041000 DevKit
- 0x3C051000 ProjectBuilder
- 0x3C051000 InterfaceBuilder
- 0x40001000 Message
- 0x42011000 WebObjects
- 0x44021000 EOControl
- 0x46031000 EOAccess
- 0x48041000 EOInterface
- 0x4A051000 Sybase
- 0x4C061000 Oracle
- 0x4E071000 Informix
- 0x50081000 nextorb
- 0x52091000 EOModeler

Command Editing

GDB's command line history can be accessed by using the up and down arrow keys, or by using the EMACS key bindings (^P and ^N to scroll thru previous commands, ^B, ^A, ^E to move around on a line, and so on). Also, **set history expansion on** enables C-shell-like command history within GDB (**!!**, **!print** and so on). As usual, an empty newline repeats the previous command (except where specifically disabled, as with the **run** command).

Debugging Objective-C: Differences from Mach GDB

The Windows version of GDB has separate features for many different languages, including Objective-C. It attempts to guess the source language by looking at the extension of the source file name ("**.m**" or "**.M**" for Objective-C). By default, GDB's ªcurrent languageº is Objective-C. At any time, you can find out what GDB's ªcurrent languageº is with **show language**. To force the

current language to Objective-C, type **set language objective-c**.

Calling Methods from GDB

To call a method in your program from GDB, use the **print**, **set**, or **call** commands with an argument that looks just like a method call in Objective C, as shown here:

```
(gdb) print [myClass showValue: 12]
```

If the method comes from a Category, you must include the category name, like this:

```
(gdb) print [myClass(myCategory) showValue: 12]
```

Listing and Setting Breakpoints on Methods

To refer to a method in a **list** or **break** command, you can give the full class and method name, including a `+' or `-' to indicate a class method or instance method. If there is a category name, you must give that too:

```
(gdb) list +[myClass init]
(gdb) break -[myClass(myCategory) showValue]
```

You can also set breakpoints or list a method just by giving a selector. If the selector is implemented by more than one class, gdb will list the corresponding methods and ask you to choose one or more:

```
(gdb) break init
[0] cancel
[1] all
[2] -[Change init] at Change.m:20
[3] -[DrawApp init] at DrawApp.m:130
[4] -[Graphic init] at Graphic.m:139
>
```

You would then enter your choice or choices at the ">" prompt.

Getting Information about Classes and Methods

GDB for Windows now has the **info classes** and **info selectors** commands. These commands accept the same regular expression language as GDB's **info type** and **info function** commands (ie. the Unix style regular expression language). This is a change from the Mach gdb, where **info classes** and **info selectors** accept a slightly different regular expression language. For instance, to learn about class names beginning with NS (using the '^' character to designate ªbeginning withº):

```
(gdb) info classes ^NS
```

To learn about selectors, you can use the **info selectors** command. To find every selector containing the string "withObject:" you could enter:

```
(gdb) info selector withObject:
```

To learn about methods, you can use the **info function** command, which also takes a regular expression. Since the square bracket characters `[' and `]' are significant in regular expressions, you can quote them with a backward slash to prevent their being treated as special characters. To list all the methods of a class, you might say:

```
(gdb) info function \[MyClass
```

To list all the methods whose selector ends with ªcount:º, you might say:

```
(gdb) info function count:\]
```

If you want to know about a specific method of a specific class, but you are not sure if it belongs to a category, you could use the ª.*º wildcard sequence to stand for ªany number of any charactersº:

```
(gdb) info function MyClass.*mySelector:
```

Debugging Threads on Windows

The GDB for Windows is thread-aware. If you are debugging a multi-threaded program, you can use the **info threads** command to see a list of the currently active threads. Use the **thread** command to switch to one of the threads listed by **info threads**, giving it the number of the thread you want to debug (a small integer such as 1 or 2, not the thread ID).

When you switch threads, you will frequently find yourself in a non-debuggable area of code from which you cannot even get a backtrace. See the section on known problems, ªNon-Debuggable Codeº below.

Known Problems

Non-Debuggable Code

When you interrupt a program with ^C, attach to a running program, or switch threads in a running program, in very rare instances the program will be in the middle of executing Windows code that cannot be debugged. Occasionally you'll find that GDB cannot even give you a backtrace, because Windows has done something with the program stack. When this happens, if you don't want to simply let the program continue (for instance, you need to know exactly how you got to where you are), you can use the **stepi** command to step by single machine instructions until the Windows code cleans up the stack and returns, at which time you will suddenly be able to see symbols and backtraces again. GDB will notify you when you have returned to a symbolic region (say, a function or a method) that it knows about. Usually this does not take too long; on the order of a few dozen instruction steps.