Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

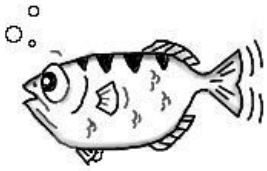
Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Recent Developments in GDB



Paul J. Gilliam
IBM Linux Technology Center
pgilliam@ibm.com

The views and opinions expressed in this paper are those of the author, not of his employer, nor of his colleagues who comprise the GDB community. Any errors of fact or judgment are his alone.

Abstract

Many programmers on many platforms depend on GDB to help them find and fix bugs in their programs. Some of these programmers use GDB directly and others indirectly through one of several available graphical front-ends.

This paper summarizes major changes in GDB over the last few years, starting with the release of GDB 6.0, which was released 2003-10-06.

These changes were made in a number of areas, including:

- Support for additional architectures.
- Internal changes and reorganizations aimed at better supporting current and future architectures.
- New features to help programmers debug their programs.
- Enhancements to make GDB easier to use, both for those who use it directly and for those who write and maintain front ends.

Also included is an overview of the GDB community and how it operates, touching on a recent overhaul in the way the community interrelates to improve GDB.

1 Introduction

Many programmers on many platforms depend on GDB to help them find and fix bugs in their programs. Some of these programmers use GDB directly, while other use it indirectly through one of several available graphical front-ends, such as DDD from GNU or Xcode from Apple.

These programmers have one thing in common with respect to GDB: they all wish it were better. Of course, each of them has a different idea of what constitutes *better*, and what's good for

one platform may not be good for *all* platforms. These basic conflicts could have led to a warring and bitter group of users. Instead, it has led to a vibrant community that maintains and extends GDB.

One measure of the vibrancy of the GDB community is the topic of this paper. What recent developments have been made? The answer is "lots!" There will be more about the GDB user community later in this paper.

For the purposes of this paper, GDB activity will be divided into five parts:

1. **Platforms** This part covers target architectures and host environments (operating system and processor). When these two are connected as part of GDB, it is called a *native* debugger. Otherwise, the two are connected by some communication protocol and GDB is called a *remote* debugger.
2. **User-Level Features** These are features that are primarily aimed at the user. These will directly help the user debug their program. An example is support for a particular computer language.
3. **Under-the-Hood Features** These are features that are primarily aimed at some part of GDB itself. Hopefully, these will help the user, but in a less direct way. For example, doing a better job of understanding a particular format for debug information will help GDB do a better job, but may not be reflected in GDB's user interfaces.
4. **Deleted Features** These are features that are no more. They are ex features. Sometimes they are features that were meant to ease a migration from one internal scheme to another and the migration has been completed. Sometimes they are components of GDB that have lost their usefulness. The most natural (in the Darwinian sense) reason for a feature to be deleted is a lack of interest in the GDB community to maintain it.
5. **Commands** These are the most visible part of GDB to most users. If GDB is accessed through a GUI, these may not be visible at all. In either case, they are the basic building blocks of GDB's functionality. Commands are usually added to GDB, but occasionally they are deleted.

2 Platforms

It's kind of hard to know what a *platform* is when talking about GDB. This comes from the fact that GDB is used in different ways by different communities. For example, the imbedded community sees GDB as running on a particular host, debugging their code on a particular target. A typical developer will see GDB as running in the same environment as the program they are developing on and for.

To keep things simple for this section, I'm going to ignore all this and hope that the description used is enough to figure out these details.

When indicating the affected version of GDB, >6.4 means it will be in the 6.5 release of GDB, which had not yet happened when this paper was written, but *should* have happened by the time it is presented.

2.1 Added

Table 1 shows the platforms that have been added to the list supported by GDB.

2.2 Removed

The GDB community uses a two-step process to remove a platform. First it is made obsolete,

platform	GDB
Morpho Technologies ms2 (target)	>6.4
OpenBSD arm	6.4
OpenBDS mips64	6.4
GNU/Linux m32r	6.3
GNU/Linux hppa	6.2
OpenBSD m68k	6.2
OpenBSD m88k	6.2
OpenBSD PowerPC	6.2
NetBSD VAX	6.2
OpenBSD VAX	6.2
NetBSD amd64	6.1
OpenBSD amd64	6.1
OpenBSD alpha	6.1
OpenBSD sparc	6.1
OpenBSD sparc64	6.1

Table 1: Added Platforms

which means the code to support that platform is 'commented out.' Then after time has passed and no one has come forward to maintain it, the platform is completely removed from the source tree. (Of course, it's still in the CVS repository if someone wants to resurrect it). For the purposes of this section, I'll list the version of GDB where the platform was removed, or if it hasn't been removed yet, where it was obsoleted.

Table 2 shows the platforms that have been removed.

3 New or Improved User-level Features

It seems that the life-blood of many commercial software packages is new features or newly improved ones. This is driven by the need for new sales of the same product to the same customers. GDB does not feel this force because it is Free (as in Freedom). The force that drives new or improved user-level features in GDB is provided by the programmers willing to do the work (or by people paying someone to do the work) to make GDB better.

A note of caution: Not all the features discussed in this section or the next are universally available. Some are limited to particular operating systems and/or particular processors and only native or remote. I have tried to indicate where that is the case, but I may not have been completely successful.[1]

3.1 Checkpoints

This is a really cool feature: it lets a user select a point in time during a debugging session and mark it as one that can be gone back to later. The concept of checkpoints has been around for a long time and still has the same underlaying motivation: "If things go bad, I don't want to have to repeat every step I took to get there." Ideally, we would like to halt on an error and 'undo' back to a good state, but we can't do that.[2]

When you reach a point in your debugging that you might want to go back to, use the command `checkpoint`. GDB will tell you the number of the new checkpoint, just like it tells you the number of a new breakpoint. Now, after some more debugging, you want to go back to the way things were: use the command `restart` *id*, where *id* is the id number GDB assigned when you used the `checkpoint` command. With the `restart` command, the state of the target program is restored to what it was at the time of the corresponding `checkpoint` command. Then you can try something else.

[1] See Section 7.

[2] Yet: see Section 7.

platform	GDB	platform	GDB
Motorola MCORE	6.4	National Semiconductor NS32000	6.4
VxWorks and XDR protocol	6.4		
Ah8300	6.3	mn10300	6.3
sh64	6.3	v850	6.3
Sun 2, running SunOS 3	6.2	Sun 2, running SunOS 4	6.2
Sun 3, running SunOS 3	6.2	Sun 3, running SunOS 4	6.2
T386BSD	6.1	AT&T 3b1/Unix pc	6.1
Bull DPX2 (68k, SVR3)	6.1	decstation	6.1
Fujitsu SPARClite	6.1	H8/500 simulator	6.1
HP/PA Pro target	6.1	HP/PA running BSD	6.1
HP/PA running OSF/1	6.1	LynxOS	6.1
Matsushita MN10200 w/simulator	6.1	Motorola 680x0 running LynxOS	6.1
PMAX (MIPS) running Mach 3.0	6.1	riscos	6.1
Sequent family	6.1	SGI Iris (MIPS) run. Irix V3:	6.1
SGI Irix-4.x	6.1	sonymips	6.1
SPARC running LynxOS	6.1	SPARC running SunOS 4	6.1
SunOS 4	6.1	sysv	6.1
sqware Sparclet	6.1	Z8000 simulator	6.1
Argonaut Risc Chip (ARC)	6.0	Fujitsu FR30	6.0
HP/Apollo 68k Family	6.0	i386 running Mach	6.0
i386 running Mach 3.0	6.0	i386 running OSF/1	6.0
I960 with MON960	6.0	IBM AIX PS/2	6.0
Mitsubishi D30V	6.0	Motorola Delta 88000 run. Sys V	6.0
OS/9000	6.0	V850EA ISA	6.0

Table 2: Deleted Platforms

Other commands allow you to manage checkpoints. The `infocheckpoints` command tells you what checkpoints have been saved, listing for each checkpoint: its number, `pid`, and source line number or label. The `delete-checkpoint` *id* command will delete the checkpoint numbered *id*.

There are some restrictions, of course. I/O can not be 'taken back.' Data written to the disk will not be erased, for example. File pointers are, however, 'rewound' to their values at the time the checkpoint was taken. Another potential problem is that each checkpoint will have its own `pid`. When the `restart` command is used, the restored program will not have the same `pid` as before.

This only works for a native GDB on a GNU/Linux system.[3]

3.2 Internationalization

When supported, GDB will be built with internationalization (libintl). Not all the necessary mark up is complete, but it's getting done and such mark up is now the way *things are done*. So all that's needed now are translations: any volunteers?

[3]For now. See Section 7.

3.3 Ada

Support has been added for debugging `ada` programs compiled with the GNAT compiler. Currently, support is limited to expression evaluation, but it's a start.

3.4 Pending Breakpoints

A *pending* breakpoint is one for which a valid address can't be found right now, but will be in the future. A straightforward example would be setting a breakpoint in a shared library that has yet to be loaded. When the library *is* loaded, GDB will be able to find the valid address for the breakpoint. Then the pending breakpoint is removed and a *real* breakpoint is created.

3.5 Objective-C

GDB fully supports the Objective-C language. This should be no surprise because the 'Next' system, from back in the late 1980's, used the GNU toolchain to build its software, most of which was in Objective-C. Why then is this a 'recent' development? Back then there was no GDB community as we know it today. 'Next' had their own GDB, Tektronix had their own GDB (I ported it to the M88000/SysV environment while at Tek), and the FSF had the 'real' version.

3.6 Processes

GDB did not deal with multiple processes very well: if your program did a fork, you had to make sure that the new process would sleep long enough to get to another terminal, start another GDB and `attach` to the new process. The next step was to introduce the `set|show follow-fork` commands so that the user could decide which process, the parent or the child, would be the one GDB continued to debug. The one not being debugged would simply keep running. This only works on a couple of operating systems: HP-UX and GNU/Linux.

Now the fate of the fork not being debugged can be decided using the new `set detach-on-fork` command. If set to `yes`, then the fork will detach from GDB and run independently. But if set to `no`, then the fork will stay in the stopped state and join the list of forks being debugged. The command `info forks` shows you the `id`s of the forks currently being debugged. The command `fork` *id* tells GDB to stop debugging the current fork, leaving it stopped, and start debugging form *id*. To remove a fork from the list being debugged, either use `detach-fork` to let the fork run on its own after removing it or use `delete-fork` to kill the fork after removing it. This only works on GNU/Linux.[4]

3.7 Text-mode User Interface

The GDB Text User Interface, TUI for short, is a terminal interface which uses the `curses` library to show the source file, the assembly output, the program registers and GDB commands in separate text windows.[5] What's new is that this is now a run-time option where it used to be a build-time option.

3.8 Convenience Variables/User Defined Functions

Convenience variables are used in GDB either for the user to save things they are interested in or for GDB to make some value available.

[4] So far, see Section 7.

[5] Cut-n-pasted from `gdb.textinfo`.

For example, the user may want to remember a particular index in an array while stepping through a loop: `set $foo = bar` where `$foo` is the convenience variable and `bar` is the index variable for the user's program. You could use a value in a processor register in an expression like this `x12/g $r2*8+base` which tells GDB to print the twelve 64-bit integers in the array `base` indexed by register `r2`.

So what's new? Two things are new, both designed to help users who write their own GDB commands.

The first one is a new convenience variable, `$argc`. If you guessed that this variable contains the number of arguments to the user-defined command, you would be correct. Using this, it is now possible (easier?) to write user-defined commands that take a variable number of arguments.

The other new thing provides a way to initialize a convenience variable if it does not already have a value. Normally, the first time a convenience value is used, it has a special 'void' value. Using the `init-if-undefined` *$variable = expression* command, you can now create a convenience variable that starts with a value of your choice.

4 New or Improved Under-the-Hood Features

The features in this section are not the ones the sales force cares about. They are not glaringly visible to users, even though they may have a big impact. From a user's point of view, these features are more behind the scenes.

4.1 Threads

Support for threads has gotten a lot better. For example, GDB used to get confused by programs that do a lot of thread creation and deletions. Several flavors of threads are now supported: NPTL threads, linux threads (on GNU/Linux), and BSD user-level threads (on OpenBSD and FreeBSD). Per-thread variables (aka thread-local storage) is now supported on GNU/Linux.

4.2 GDB/mi

The GDB/mi interface (interpreter in GDB parlance) is the new way for GDB to be used as part of a larger system, such as a GUI or IDE, DDD and XCode. The GDB/mi was introduced with GDB 5.0, so it's not new. But it is vastly improved and as more experience is gained using it, it gets better and better. There have been three versions of this interface, but only the two most recent are still available. MI3, the most recent, is now the default.

4.3 BSD libkvm Interface

Using `set target kvm`, when running native on a BSD flavored OS, allows debugging of kernel core dumps and even live kernels, though only on a few processors: i386, amd64, m68k, and sparc. For GNU/linux, another approach was taken. There, a stand-alone variant of GDB, called KDB, is used for kernel debugging.

4.4 Windows Host Support[6]

GDB runs on MS Windows, either with Cygwin or MinGW. GDB support for both of these environments has had improvements. For Cygwin, see Section 4.6 below.

[6] Lifted whole from GDB's NEWS file.

GDB now builds as a cross debugger hosted on i686-mingw32, including native console support, and remote communications using either network sockets or serial ports

4.5 Remote Debugging

GDB has had a number of improvements in this area. Below are a few examples.

It has been possible for a while to set the communication time-out using the `set remotetimeout` *n* command, where *n* is the number of seconds to wait before giving up on reading from a target. Now, the time-out value can also be set when GDB is started using the new `-l` *n* command line option.

Before the new `p` packet was introduced as part of the remote protocol, target registers had to be read from the target in groups. Now using the `p` packet, GDB can read a single register from the target. Combine this improvement with the register cache and communication with the target is much more efficient and hence faster.

Another change to the protocol allows hosted file I/O. This is where target programs access files in GDB's filesystem via the remote protocol.

4.6 Improvements in Dwarf Support

The DWARF 2 standard for debugging information has had a profound impact on GDB, but it didn't happen all at once. In fact, GDB's support for DWARF 2 is getting better all the time. One example is that GDB built for Cygwin now supports DWARF 2.

Another improvement is support for DWARF 2 location expressions. These are used by DWARF to tell the debugger where to find the value of a given variable. This used to be easy, but now with different optimizations performed by the compiler, the location of a variable's value can change during the execution of the target program. Location expressions allow the compiler to tell the debugger how to keep track.

One complaint lodged against DWARF is that even though great care was taken by the DWARF committee to keep DWARF's 'footprint' small, DWARF can add tremendously to the size of an executable. One answer to this is the new `-feliminate-dwarf2-dups` flag to GCC and GDB's new support for it. Using this, GCC will try and pare down the duplicate information it puts out in the DWARF sections. GDB was modified to deal with the new DIEs[7] GCC uses under this flag.

Another answer to the "debugger information bloat" problem works with other debugger information formats as well. In conjunction with BINUTILS, GDB now supports debug information in separate files. Now, for example, library packages can be distributed without debug info, making them much smaller. The debug information can be in a separate package which is only installed if needed.

4.7 Improved C++ Support

Support for debugging programs written in C++ has been improved in a couple of areas. For one, GDB has a new C++ name demangler. Not only does it do a better job demangling the names produced by newer versions of GCC, but it does so faster than the old one. This can substantially reduce the start-up time when debugging a large C++ program.

Another improvement is with support for C++ nested types and namespaces. GDB now understands that the namespace and/or outer type

[7] Debug Information Entries

must be included, using the scope operator, in the type or function's name.

4.8 Strictly Internal Improvements

In order to increase GDB's reliability, maintainability, and all the other goodabilities, major sections of the code have been overhauled or re-architected.

The code that deals with signal trampolines has been overhauled. This has fixed many problems GDB was having in this area. A couple of these problems were that GDB had trouble showing a correct backtrace from inside a signal handler and had trouble single stepping through a signal trampolines.

Speaking of backtracing, a whole new mechanism was created to help GDB do that. One major improvement that this made possible was to use DWARF 2's call frame information. It also makes it easier to separate out target dependent heuristics and makes the whole approach to backtracing more robust and modular.

In the same vain, a new framework for supporting different architectures was added to GDB. This helps GDB keep track of all the different target architectures it supports. It also paves the way for two trends GDB is taking: object oriented design and muli-arch support.[8]

5 Deleted Features

The features in this section, whether user-level or under-the-hood, are no more. They are ex features. All that's left to say about them is what they were and why they got the axe.

[8] See Section 7.

ARM `rdi-share` module[9] RDI is an ABI standard for ARM hardware debuggers and simulators, giving you access to the full processor state. In rdi-share, there was support for building a gdb that could talk to an RDI DLL and use that as a target.

IIRC it was removed because it was outdated and its legal status was unclear, given that ARM ltd. gives you the necessary headers under NDA only.

Netware NLM debug server It's sometimes hard to remember that Novell had the corner on networking PCs: Netware was the best (only?) way to do it. Now other networking code has pretty much taken over and the need to debug Netware Loadable Modules (NLM's) has gone away.

command line options `-async` and `-noasync|` Once upon a time, GDB had no "event loop": it was totally synchronous, you typed a command and GDB would go away and do it. These command options were an attempt to be able to specify that behavior, even after GDB became more event driven. But now, that's just not possible any more and these options have been removed.

registers and frame compatibility modules When there is a major internal change to the way GDB does something, like access registers, or deal with the stack, a 'bridge' is provided so that all the different configurations that GDB supports don't have to all change at once. But there comes a time to burn the bridge and move on.

[9] Taken from an e-mail from Simon Richter. Thanks Simon, I didn't have a clue.

6 Commands

Some of these commands have been discussed above: they are included here as well for completeness.

6.1 New Commands

checkpoint This command creates a checkpoint and tells you its id. At some later point you can use that id to restore your program's state back to what it was when you issued this command.

restart *id* Restore your program's state back to what it was when the checkpoint with the given *id* was created.

info checkpoints Show information, including id, about all the checkpoints that have been taken and not deleted or detached.

delete-checkpoint *id* Forget about *id*.

set detach-on-fork [on or off] If set to `on` then GDB will detach from the parent or child after a fork, depending on what **follow-fork** is set to.

show detach-on-fork Show if `detach-on-fork` is set to `on` or `off`.

info forks Tell what forks are available for debugging.

fork *id* Switch from debugging the current fork, leaving it in the stopped state, and start debugging the fork with the given *id*.

delete-fork *id* Delete the given fork and kill the associated process.

detach-fork *n* Detach from the given fork, letting its associated process run independently.

init-if-undefined *$variable* **=** *expression* Set the given variable to the given value, but only if the variable has not been used before.

set print array-indexes When array element values are displayed, their index will also be shown.

set logging [on or off] If logging is set `on`, then GDB's output will be written to a log file, as well as displayed.

set logging file *name* Set the logging file name to *name*. the default is `gdb.txt`.

set logging redirect [on or off] Normally, when logging is on, GDB output will both be displayed and written to the log file. If this is set to `on`, then the output is sent only to the log file.

set logging overwrite [on or off] When logging is turned `on`, the log file will be be overwritten if this value is `on`, otherwise, it is appended to.

show logging Show if logging is set `on` or `off`. Also show if the log file is, or would be, overwritten or append to. Also show if GDB's output is to be displayed or just written to the log file. And throw in the name of the log file too.

start *arguments* This is the same as setting a temporary breakpoint at "main" and then issuing the "run *arguments*" command.

disconnect When `detach` is used to stop debugging and disconnect from the target, the target is allowed to run independent of GDB. `disconnect` works the same way except that the target is not allowed to run. It just sits there, waiting for someone to attach to it, and start debugging it. This would let you debug the child and parent with different GDBs after a fork. Or

maybe even debug with two altogether different debuggers.

`maint set profile [on or off]`
GDB has code built into it so that it can be profiled. This is used to turn `on` or `off` that code and is used to study the behavior of GDB itself and has nothing to do with the target.

6.2 Deleted Commands

Table 3 shows commands that have been deleted along with the commands that replace them.

7 The GDB Community and Future Trends

Why does this section cover two different things? Because they are so tightly linked. The future of GDB depends on the community of software freedom fighters that are involved with it. In a simple view: no community, no GDB development.

A more complex view: the impetus for change comes from two different sources. One source is the individual who wants something fixed, changed, or added and is willing to do the work, (or pay someone else to). The attitude of the community is that if you want something and are willing to do the work to get it, then go for it. Of course there are limits. Those limits are determined and enforced by a set of official 'maintainers.'

There are two types of maintainers and just recently, a third type was added. There are 'global' maintainers who are free to OK patches to any part of GDB. There are also what might be called 'areas of interest' maintainers. These maintainers can reject or OK patches to particular areas of GDB. The new type of maintainers are patch champions. Their role is to make sure that no patch gets lost. Patches can be rejected, but thanks to the patch champions, not by default.

So what if you have a good idea for some specific, but can't do the work yourself, or you see a need for a bigger change then one person could do? This is where the "group mind" aspect of the GDB community comes into play.

Someone tries to start a dialog in the GDB mailing list, maybe just to call attention to something that needs some. If the message catches someone else's interest, then they reply to the mailing list, and the discussion has begun.

The discussion can end in one of three ways:

1. the community reaches consensus that it should be done, now who has the time to do the work?
2. the community reaches consensus that it should not be done. Sorry.
3. the community is leaning toward option one above, but before they reach consensus, someone gets tired of all the discussion and says "I'll just do it, OK?" and the community replies "Fine, but if we don't like it when you're done, it won't become a part of GDB."

Sometimes an idea or request can stay in option one for a long time before someone has time for it. Or sometimes it is such a big or far reaching idea that it needs to be done in steps. It is mostly by this path that the community develops trends for the future.

So what are some of the future trends?

deleted command	replacement
`set|show arm dissembly-favor`	`set|show arm disassembler`
`othernames`	`set arm disassembler`
`set|show remotedebug`	`set|show debug remote`
`set|show archdebug`	`set|show debug arch`
`set|show eventdebug`	`set|show debug event`
`regs`	`info registers`
`set prompt-escape-char`	- none -

Table 3: Deleted Commands

propagate Several really cool things only work on one OS or with a small number of processors and need to be propagated to *all* configurations. This will not always be possible due to hardware and/or software incompatibilities. But things like kernel bugs can be fixed and should be if that's the problem.

separate Some features that could logically be independent, like the MI and the CLI, are not quite. They should be separated and made independent. The next trend should help with that.

objectify GDB is slowly evolving into an object oriented design. Thanks to the "observer" mechanism, "Catch and throw" and a bunch of other stuff Cagney and others did, GDB looks a lot like a "coarse grain" object oriented design. More is needed. A more complete object orientation will make GDB much easier to deal with as a **large** piece of software.[10] Generalizations like "multi-arch" will be easier to do. A "multi-arch" GDB will work with any architecture GDB knows about, not just the one it was built for.

it's a given There will be new processors, new operating systems, new languages (natural and computer), new compilers, etc.

optimized code GDB has been getting better in dealing with optimized code but still has a ways to go. The mailing list often gets messages asking "why does the line number keep jumping around when I step through my code?"

blue sky Some ideas start life as "blue sky" ideas. They are real cool, but they will never be practical. Then some new piece of technology comes along and, just like that, the idea isn't so far fetched anymore.

A current Blue Sky idea for GDB is "reverse execution." This would be like having a target with a reverse gear. Every instruction could go forward or backward, performing a computation or undoing the results of one. Some believe that a processor is coming "soon" that will make this possible. In the meantime, we can talk about how GDB would deal with such a processor and maybe even hack one of the simulators so that the idea can be experimented with.

And if we can't go back instruction by instruction, maybe we can go back by some larger amount: maybe back to some checkpoint. Oh right: already did that. So now, what's the next step? Better send a query to the GDB mailing list and see if it catches anyone's eye.

[10] About 1.75 million lines of code