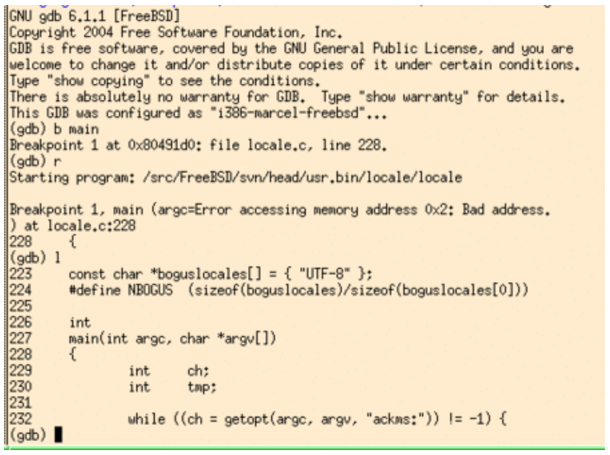
GNU Debugger

From Wikipedia, the free encyclopedia

The **GNU Debugger (GDB)** is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Java[1] and partially others.[2]

GNU Debugger



```
GNU gdb 6.1.1 [FreeBSD]
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-marcel-freebsd"...
(gdb) b main
Breakpoint 1 at 0x80491d0: file locale.c, line 228.
(gdb) r
Starting program: /src/FreeBSD/svn/head/usr.bin/locale/locale

Breakpoint 1, main (argc=Error accessing memory address 0x2: Bad address.
) at locale.c:228
228     {
(gdb) l
223     const char *boguslocales[] = { "UTF-8" };
224     #define NBOGUS  (sizeof(boguslocales)/sizeof(boguslocales[0]))
225
226     int
227     main(int argc, char *argv[])
228     {
229             int     ch;
230             int     tmp;
231
232             while ((ch = getopt(argc, argv, "ackms:")) != -1) {
(gdb) ▮
```

Developer(s)	GNU Project
Initial release	1986
Stable release	7.12 / October 7, 2016
Repository	sourceware.org/git/gitweb.cgi?p=binutils-gdb.git (http://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git)
Written in	C
Operating system	Unix-like, Windows
Type	Debugger
License	GNU GPL
Website	www.gnu.org/software/gdb (https://www.gnu.org/software/gdb)

Contents

History

GDB was first written by Richard Stallman in 1986 as part of his GNU system, after his GNU Emacs was "reasonably stable".[3] GDB is free software released under the GNU General Public License (GPL). It was modeled after the DBX debugger, which came with Berkeley Unix distributions.[3]

From 1990 to 1993 it was maintained by John Gilmore. Now it is maintained by the GDB Steering Committee which is appointed by the Free Software Foundation.[4]

Technical details

Features

GDB offers extensive facilities for tracing and altering the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.

GDB target processors (as of 2003) include: Alpha, ARM, AVR, H8/300, Altera Nios/Nios II, System/370, System 390, X86 and its 64-bit extension X86-64, IA-64 "Itanium", Motorola 68000, MIPS, PA-RISC, PowerPC, SuperH, SPARC, and VAX. Lesser-known target processors supported in the standard release have included A29K, ARC, ETRAX CRIS, D10V, D30V, FR-30, FR-V, Intel i960, 68HC11, Motorola 88000, MCORE, MN10200, MN10300, NS32K, Stormy16, and Z8000. (Newer releases will likely not support some of these.) GDB has compiled-in simulators for even lesser-known target processors such like M32R or V850.[5]

GDB is still actively developed. As of version 7.0 new features include support for Python scripting[6] and as of version 7.8 GNU Guile scripting as well.[7] Since version 7.0, support for "reversible debugging" — allowing a debugging session to step backward, much like rewinding a crashed program to see what happened — is available.[8]

Remote debugging

GDB offers a 'remote' mode often used when debugging embedded systems. Remote operation is when GDB runs on one machine and the program being debugged runs on another. GDB can communicate to the remote 'stub' which understands GDB protocol via Serial or TCP/IP.[9] A stub program can be created by linking to the appropriate stub files provided with GDB, which implement the target side of the communication protocol.[10] Alternatively, gdbserver can be used to remotely debug the program without needing to change it in any way.

The same mode is also used by KGDB for debugging a running Linux kernel on the source level with gdb. With KGDB, kernel developers can debug a kernel in much the same way as they debug application programs. It makes it possible to place breakpoints in kernel code, step through the code and observe variables. On architectures where hardware debugging registers are available, watchpoints can be set which trigger breakpoints when specified memory addresses are executed or accessed. KGDB requires an additional machine which is connected to the machine to be debugged using a serial cable or Ethernet. On FreeBSD, it is also possible to debug using Firewire direct memory access (DMA).[11]

Graphical user interface

The debugger does not contain its own graphical user interface, and defaults to a command-line interface. Several front-ends have been built for it, such as UltraGDB, Xxgdb, Data Display Debugger (DDD), Nemiver, KDbg, Xcode debugger, GDBtk/Insight and the HP Wildebeest Debugger GUI (WDB GUI). IDEs such as Codelite, Code::Blocks, Dev-C++, Geany, GNAT Programming Studio (GPS), KDevelop, Qt Creator, Lazarus, MonoDevelop, Eclipse, NetBeans and VisualStudio can interface with GDB. GNU Emacs has a "GUD mode" and tools for VIM exist (e.g. clewn.) These offer facilities similar to debuggers found in IDEs.

Some other debugging tools have been designed to work with GDB, such as memory leak detectors.

Examples of commands

`gdb program`	debug "program" (from the shell)
`run -v`	run the loaded program with the parameters
`bt`	backtrace (in case the program crashed)
`info registers`	dump all registers
`disas $pc-32, $pc+32`	disassemble

An example session

Consider the following source-code written in C:

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

size_t foo_len (const char *s)
{
  return strlen (s);
}

int main (int argc, char *argv[])
{
  const char *a = NULL;

  printf ("size of a = %d\n", foo_len (a));

  exit (0);
}
```

Using the GCC compiler on Linux, the code above must be compiled using the `-g` flag in order to include appropriate debug information on the binary generated, thus making it possible to inspect it using GDB. Assuming that the file containing the code above is named `example.c`, the command for the compilation could be:

```
$ gcc example.c -g -o example
```

And the binary can now be run:

```
$ ./example
Segmentation fault
```

Since the example code, when executed, generates a segmentation fault, GDB can be used to inspect the problem.

```
$ gdb ./example
GNU gdb (GDB) Fedora (7.3.50.20110722-13.fc16)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /path/example...done.
(gdb) run
Starting program: /path/example

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400527 in foo_len (s=0x0) at example.c:8
8     return strlen (s);
(gdb) print s
$1 = 0x0
```

The problem is present in line 8, and occurs when calling the function `strlen` (because its argument, `s`, is `NULL`). Depending on the implementation of strlen (inline or not), the output can be different, e.g.:

```
GNU gdb (GDB) 7.3.1
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
```

```
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /tmp/gdb/example...done.
(gdb) run
Starting program: /tmp/gdb/example

Program received signal SIGSEGV, Segmentation fault.
0xb7ee94f3 in strlen () from /lib/i686/cmov/libc.so.6
(gdb) bt
#0  0xb7ee94f3 in strlen () from /lib/i686/cmov/libc.so.6
#1  0x08048435 in foo_len (s=0x0) at example.c:8
#2  0x0804845a in main (argc=<optimized out>, argv=<optimized out>) at example.c:16
```

To fix the problem, the variable `a` (in the function `main`) must contain a valid string. Here is a fixed version of the code:

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

size_t foo_len (const char *s)
{
  return strlen (s);
}

int main (int argc, char *argv[])
{
  const char *a = "This is a test string";

  printf ("size of a = %d\n", foo_len (a));

  exit (0);
}
```

Recompiling and running the executable again inside GDB now gives a correct result:

```
GNU gdb (GDB) Fedora (7.3.50.20110722-13.fc16)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /path/example...done.
(gdb) run
Starting program: /path/example
size of a = 21
[Inferior 1 (process 14290) exited normally]
```

GDB prints the output of `printf` in the screen, and then informs the user that the program exited normally.

See also

- Binary File Descriptor library (libbfd)
- dbx
- ddd, a GUI for gdb and other debuggers
- gdbserver

References

1. "GDB Documentation - Supported Languages". Retrieved 2011-11-28.
2. "GDB Documentation - Summary". Retrieved 2011-11-28.
3. "Richard Stallman lecture at the Royal Institute of Technology, Sweden (1986-10-30)". Retrieved 2006-09-21. "Then after GNU Emacs was reasonably stable, which took all in all about a year and a half, I started getting back to other parts of the system. I developed a debugger which I called GDB which is a symbolic debugger for C code, which recently entered distribution. Now this debugger is to a large extent in the spirit of DBX, which is a debugger that comes with Berkeley Unix."
4. "GDB Steering Committee". Retrieved 2008-05-11.
5. "GDB Documentation - Summary - Contributors". Retrieved 2011-12-01.
6. "GDB 7.0 Release Notes". Retrieved 2011-11-28.
7. Joel Brobecker (2014-07-29). "GDB 7.8 released!". Retrieved 2014-07-30.
8. "Reverse Debugging with GDB". Retrieved 2014-01-20.
9. "Howto: GDB Remote Serial Protocol: Writing a RSP Server" (PDF).
10. "Implementing a remote stub".
11. "Kernel debugging with Dcons".

External links

- Official website (https://www.gnu.org/software/gdb)
- UltraGDB: Visual C/C++ Debugging with GDB on Windows and Linux (http://www.ultragdb.com/)
- KGDB: Linux Kernel Source Level Debugger (http://linsyssoft.com/product_kgdb.php)
- The website for "MyGDB: GDB Frontend" in the Korean language (https://web.archive.org/web/20100829051021/http://kldp.net:80/projects/mygdb/)
- A Visual Studio plugin for debugging with GDB (http://visualgdb.com/)
- Comparison of GDB front-ends, 2013 (http://www.drdobbs.com/testing/13-linux-debuggers-for-c-reviewed/240156817)
- Using Eclipse as a Front-End to the GDB Debugger (https://developer.palm.com/content/api/dev-guide/pdk/eclipse-gdb-debugging.html)

Documentation

- Richard M. Stallman, Roland Pesch, Stan Shebs, et al., *Debugging with GDB* (http://sourceware.org/gdb/current/onlinedocs/gdb.html) (Free Software Foundation, 2011) ISBN 978-0-9831592-3-0
- GDB Internals (http://sourceware.org/gdb/current/onlinedocs/gdbint.html)

Tutorials

- *RMS's gdb Tutorial* (http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html) (Ryan Schmidt)

Retrieved from "https://en.wikipedia.org/w/index.php?title=GNU_Debugger&oldid=765085965"

Categories: Debuggers | GNU Project software | Unix programming tools | Video game development software for Linux