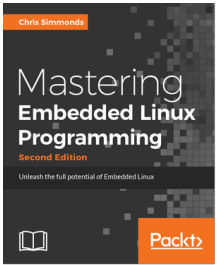
Debugging with GDB

Chris Simmonds

E-ALE @ Embedded Linux Conference Europe 2019

License

About Chris Simmonds



- Consultant and trainer
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at `http://2net.co.uk/`

@2net_software

`https://uk.linkedin.com/in/chrisdsimmonds/`

Objectives

- Show how to use GDB to debug applications
- How to attach to a running process
- How to look at core dumps
- Plus, we will look at graphical interfaces for GDB
- Reference: MELP2 Chapter 14

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it"
- Brian W. Kernighan

Toolchains

GNU toolchain = GCC + binutils + C library + GDB

GCC	GNU Compiler Collection - C, C++, Objective-C, Go and other languages
binutils	assembler, linker and utilities to manipulate object code
C library	the POSIX API and interface to the Linux kernel
GDB	the GNU debugger

Native vs cross compiling

Native (on target)

- PocketBeagle running Debian or Raspberry Pi running Raspbian
- PC

Cross (on host)

- Traditional embedded development
  - OpenEmbedded/Yocto Project
  - Buildroot

More about debugging cross-compiled code later...

Toolchain sysroot

- The **sysroot** of the toolchain is the directory containing the supporting files
  - Header files; shared and static libraries, etc.
- Native toolchain: sysroot = '/'
- Cross toolchain: sysroot is usually inside the toolchain directory
- Find it using `-print-sysroot`
- Example:

```
$ aarch64-buildroot-linux-gnu-gcc -print-sysroot
/home/traning/aarch64--glibc--stable/bin/../
aarch64-buildroot-linux-gnu/sysroot
```

You will need to know the sysroot when cross-compiling

Preparing to debug 1/2

Compile with the right level of debug information

```
gcc -gN myprog.c -o myprog
```

where N is from 0 to 3:

Level	Description
0	no debug information (equivalent to omitting -g)
1	minimal information, just enough to generate a backtrace
2	**(default) source-level debugging and single-stepping**
3	information about macros

You can replace **-gN** with **-ggdbN** to generate GDB specific debug info instead of generic DWARF format

Preparing to debug 2/2

- Code optimization can be a problem
  - especially if you plan to do a lot of single-stepping
- Consider turning off optimization with compiler flag `-O0`
- Or enable just GDB compatible optimizations with compiler flag `-Og`

A debug session

- Launch your program with gdb

```
$ gdb helloworld
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://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 "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from helloworld...done.
```

Breakpoints

Add a breakpoint

`break [line|function]`, example

```
(gdb) break main
Breakpoint 1 at 0x400535: file helloworld.c, line 7.
```

List breakpoints

`info break`:

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00400535 in main at helloworld.c:7
```

Delete a breakpoint

`delete break`:

```
(gdb) delete break 1
```

Controlling execution

Run the program from the start

```
run
```

Continue executing the program from a breakpoint

```
continue
```

Step one line of code, stepping into functions

```
step
```

Step one line of code, stepping over functions

```
next
```

Run to the end of the current function

```
finish
```

Displaying and changing variables

Display a variable

```
print some_var
```

```
(gdb) print i
$1 = 1
```

Change a variable

```
set some_var=new_value
```

```
(gdb) set var i=99
```

watchpoints

Break when a variable changes

`watch some_var`

```
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.
0 Hello world

Hardware watchpoint 2: i

Old value = 0
New value = 1
0x0000000000400556 in main (argc=1, argv=0x7fffffffde28) at helloworld.c:7
7       for (i = 0; i < 4; i++)
```

Conditional watch

`watch some_var if condition`

```
(gdb) watch i if i == 3
```

stack frames and back trace

Each function has a **stack frame** which contains the local (auto) variables

Show stack frames

```
bt
```

```
(gdb) bt
#0  addtree (p=0x0, w=0xffffdcd0 "quick") at word-count.c:39
#1  0x004008b4 in addtree (p=0x603250, w=0xffffdcd0 "quick") at word-count.c:53
#2  0x004009fd in main (argc=1, argv=0xffffde28) at word-count.c:92
```

Display local variables

```
info local
```

Change current stack frame

```
frame N
```

```
(gdb) frame 2
```

GDB command files

- At start-up GDB reads commands from
  - `$HOME/.gdbinit`
  - `.gdbinit` in current directory
  - Files named by gdb command line option **-x [file name]**
- Note: auto-load safe-path
  - Recent versions of GDB ignore `.gdbinit` unless you enable it in `$HOME/.gdbinit`

```
add-auto-load-safe-path /home/myname/myproject/.gdbinit
```

Debugging library code

- By default GDB searches for source code in
  - $cdir: the compile directory (which is encoded in the ELF header)
  - $cwd: the current working directory

```
(gdb) show dir
Source directories searched: $cdir:$cwdv
```

- You can extend the search path with the **directory** command:

```
(gdb) dir /home/chris/src/mylib
Source directories searched: /home/chris/src/mylib:$cdir:$cwd
```

Just-in-time debugging

- Both gdb and gdbserver can **attach** to a running process and debug it, you just need to know the PID
- With gdbserver, you attach like this (PID 999 is an example)

```
# gdbserver --attach :2001 999
```

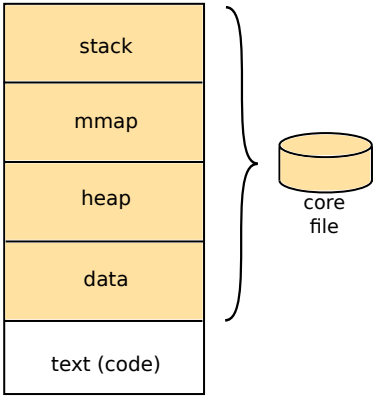
- If debugging natively using GDB, use the attach command:

```
(gdb) attach 999
```

- In either case, to detach and allow the process to run freely again:

```
(gdb) detach
```

Core dump



A core file is created if:

- size is < `RLIMIT_CORE`
- the program has write permissions to create a file
- not running with set-user-ID
- Set `RLIMIT_CORE` to un-limited using command: `ulimit -c unlimited`

Using gdb to analyse a core dump

- Command-line gdb

```
gdb sort-debug ~/rootdir/usr/bin/core
...
Core was generated by `/sort-debug /etc/protocols'.
Program terminated with signal 11, Segmentation fault.
#0  0x00008570 in addtree (p=0x0, w=0xbeaf4c68 "Internet") at
sort-debug.c:45
45          p->word = strdup (w);
(gdb) back
#0  0x00008570 in addtree (p=0x0, w=0xbeaf4c68 "Internet") at
sort-debug.c:45
#1  0x00008764 in main (argc=2, argv=0xbeaf4e34) at sort-debug.c:95
(gdb)
```

Core pattern

- By default, core files are called `core` and placed in the working directory of the program
- Or, core file names are constructed according to `/proc/sys/kernel/core_pattern`
- See man core(5) for details

Example: `/corefiles/%e-%p`

%e executable name
%p PID

GUI front ends

- There are many front-ends, including
  - TUI: Terminal User Interface
  - cgdb: an improved version of TUI
  - DDD: Data Display Debugger

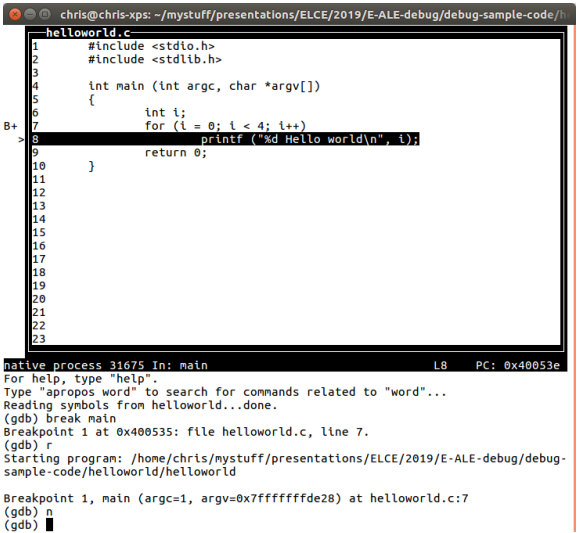
TUI

Part of GDB

Launch like this:

```
gdb -tui myprog
```

Or toggle on and off with **Ctrl-x a**

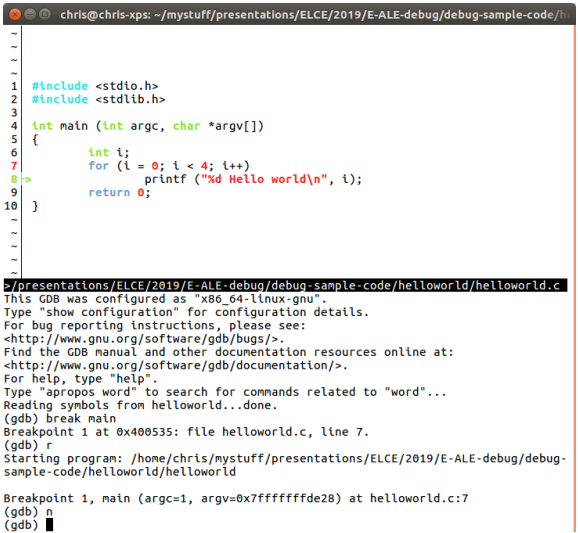


cgdb

A wrapper for TUI, adds syntax highlighting. Better than TUI by itself

Launch like this:

```
cgdb myprog
```

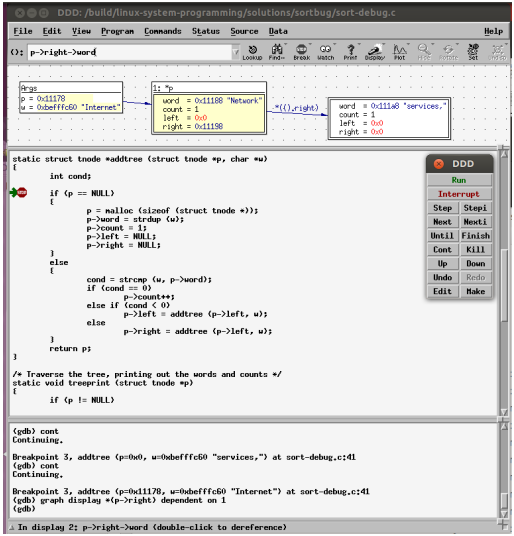


DDD: Data Display Debugger

A graphical front-end to GDB

Launch like this:

```
ddd myprog
```



Lab time...

Get the slides and sample code from
https://cm.e-ale.org/2019/ELCE2019/debugging-with-gdb

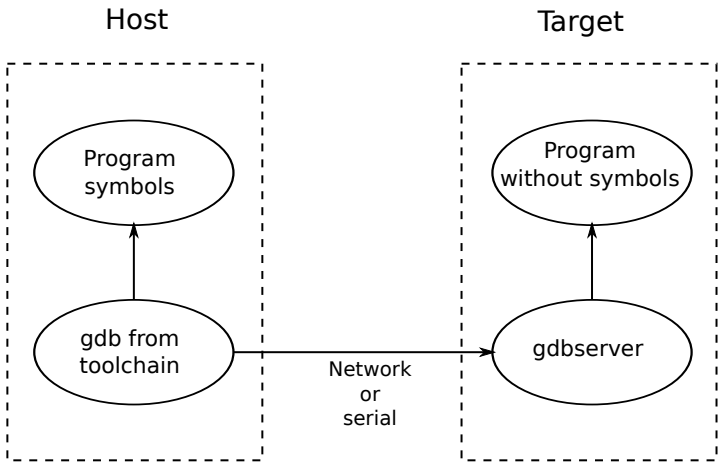
Follow the notes in
debugging-EALE-2019-csimmonds-workbook.pdf

Call me or one of the helpers if you encounter problems

Debugging with cross compiled code

- Native compiling is not so common in embedded Linux
- Most embedded projects are developed using cross-compilers
- Tools such as OpenEmbedded/Yocto Project and Buildroot work this way
- I will be talking about this in my talk on Wednesday
  - Debian or Yocto Project? Which is the Best for your Embedded Linux Project?
  - https://sched.co/TLJZ

Remote debugging



Debug info

- Need debug info **on the host** for the applications and libraries you want to debug
  - It's OK for the files on the target to be stripped: gdbserver does not use debug info
- Debug info may be included in the binary (the Buildroot way)
- Or placed in a sub-directory named `.debug/` (the Yocto Project/OpenEmbedded way)

Debug build - Yocto Project

- You need to add debug tools for the target: add this to your `conf/local.conf`

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-debug"
```

- And you need to build an SDK which will contain the tools for the host, and the debug symbols

```
bitbake -c populate_sdk <image name>
```

Debug build - Buildroot

- You need to run menuconfig and enable these options

```
PACKAGE_HOST_GDB
PACKAGE_GDB
PACKAGE_GDB_SERVER
ENABLE_DEBUG
```

- Then re-build the image
- The executables with debug symbols are put in `output/host/usr/<arch>/sysroot`

Setting sysroot

- **sysroot** tells GDB where to find library debug info
- For Buildroot

```
set sysroot <toolchain sysroot>
```

- Using a Yocto Project SDK:

```
set sysroot /opt/poky/<version>/sysroots/<architecture>
```

Command-line debugging

Development host

```
$ arm-poky-linux-gnueabi-gdb helloworld
(gdb) set sysroot /opt/poky/2.5.1/...
(gdb) target remote 192.168.7.2:2001
```

```
(gdb) break main
(gdb) continue
```

Embedded target

```
# gdbserver :2001 helloworld
```

```
“Remote debugging from host 192.168.7.1”
```

```
{program runs to main()}
```

Notes

- GDB command **target remote** links gdb to gdbserver
- Usually a TCP connection, but can be UDP or serial
- gbdserver loads the program into memory and halts at the first instruction
- You can't use commands such as **step** or **next** until after the start of C code at `main()`
- **break main** followed by **continue** stops at `main()`, from which point you can single step

GDB front-ends with a cross toolchain

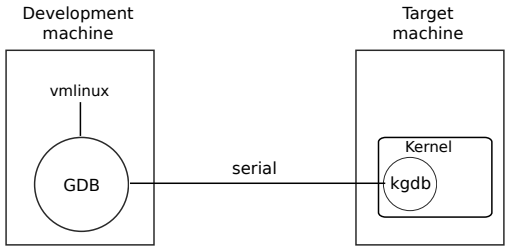
```
cgdb -d arm-poky-linux-gnueabi-gdb myprog
end{codeSmall}

\begin{codeSmall}
ddd --debugger arm-poky-linux-gnueabi-gdb myprog
```

Debugging kernel code

Outside the scope of this workshop, but ...

- Build kernel with KGDB - which is like gdbserver but integrated into the kernel
- Connect to serial port on target
- Read debug symbols from vmlinux file



Delving deeper

- This is an excerpt from my **Mastering embedded Linux** class
- If you would like to discover more about the power of embedded Linux, visit `http://www.2net.co.uk/training.html` and enquire about training classes for your company
  - 2net training is available world-wide
- Also, my book, *Mastering Embedded Linux Programming*, covers the topics discused here in much greater detail

Further reading

- The Art of Debugging with GDB, DDD, and Eclipse, by Norman Matloff and Peter Jay Salzman, No Starch Press; 1st edition (28 Sept, 2008)
- GDB Pocket Reference by Arnold Robbins, O'Reilly Media; 1st edition (12 May, 2005)