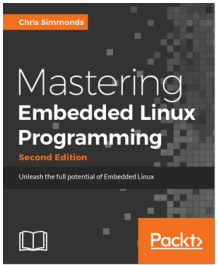
Debugging Embedded Devices using GDB

Chris Simmonds

Embedded Linux Conference Europe 2020

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 `https://2net.co.uk/`

@2net_software

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

Objectives

- Show how to use GDB to debug devices running embedded Linux
- How to attach to a running process
- How to look at core dumps
- Plus, we will look at graphical interfaces for GDB
- Reference: Mastering Embedded Linux programming, 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

Resources

- As part of this tutorial I will be running several live demos of the various techniques
- For the development machine I will be using a Linux laptop running Ubuntu 18.04
- And I will be using a Raspberry Pi 3B as the target, running a Yocto Project Linux build
- You can download a workbook containing full instructions for setting up and running the demos from
  https:/2net.co.uk/downloads/debugging-with-gdb-csimmonds-elce-2020-workbook.pdf
- I encourage you to follow along with the video :-)

The Raspberry Pi 3B

- Popular dev board
  `https://www.raspberrypi.org`
- Low cost ($35)
- BCM2837 Soc: 4 x Cortex-A53 ARMv8 64-bit @ 1.2GHz
- 1 GiB SDRAM
- Micro SD card slot
- 4 x full size USB 2.0 A host
- 100 Mbit Ethernet
- HDMI video output

Yocto Project

https://www.yoctoproject.org

- Yocto Project is a build system that creates packages from source code
- It is based on the **Bitbake** job scheduler and **OpenEmbedded** meta data
- They allow you to create your own tailor-made Linux distro
- Yocto Project and OpenEmbedded have been used to create the software running on many millions of devices
- Instructions for setting up Yocto Project are in the workbook

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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 (develop on target; run on target), e.g.

- PC
- Raspberry Pi running Raspbian

Cross (develop on host; run on target), e.g.

- Yocto Project/OpenEmbedded
- Buildroot

This tutorial uses cross compilation

Getting a toolchain

Your options are:

- Build from upstream source, e.g. using CrosstoolNG: `http://crosstool-ng.github.io`
- Download from a trusted third party, e.g. Linaro or Bootlin
- Use the one provided by your SoC/board vendor (check quality first)
- **Use an embedded build system (Yocto Project, OpenEmbedded, Buildroot) to generate one**

Toolchain prefix

- GNU toolchains are usually identified by a prefix of the form `arch-vendor-kernel-operating system`
- Example: `mipsel-unknown-linux-gnu-`
  - **arch**: mipsel (MIPS little endian)
  - **vendor**: unknown
  - **kernel**: linux
  - **operating system**: gnu
- So, the C compiler would be `mipsel-unknown-linux-gnu-gcc`

Toolchain prefix for 32-bit ARM toolchains

- 32-bit ARM has several incompatible ABIs (Application Binary Interface - the rules for function calls, parameter passing, etc.)
- Reflected in the **Operating system** part of the prefix
- Examples:
  - `arm-unknown-linux-gnu-`: Old ABI (obsolete)
  - `arm-unknown-linux-gnueabi-`: Extended ABI with soft floating point(*)
  - `arm-unknown-linux-gnueabihf-`: Extended ABI with hard floating point(*)

(*) Indicates how floating point arguments are passed: either in integer registers or hardware floating point registers

Toolchain sysroot

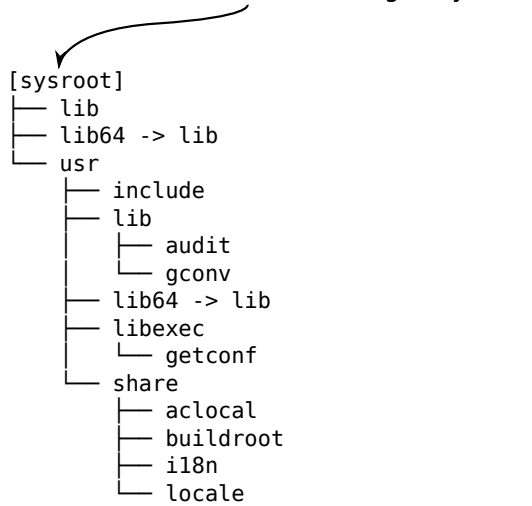
- The **sysroot** of the toolchain is the directory containing the supporting files
  - Header files; shared and static libraries, etc.
- Native toolchain: sysroot = '/'
- Cross toolchain: sysroot is usually inside the toolchain directory
- Find it using `-print-sysroot`
- Example:

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

You need to know the sysroot when cross-compiling and debugging

sysroot

sysroot = aarch64-buildroot-linux-gnu/sysroot



Getting to know your toolchain

Find out about GCC with these options

- `-print-sysroot`: print sysroot
- `--version`: version
- `-v`: configuration, look out for
  - `--enable-languages=` (example c,c++)
  - `--with-cpu=` (the default CPU)
  - `--enable-threads` (has POSIX threads library)

The Yocto Project SDK

- If using Yocto Project/OpenEmbedded, you can create an SDK that includes a toolchain with

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

- Generates self-installing shell script with a name like `poky-glibc-x86_64-core-image-base-cortexa7t2hf-neon-vfpv4-raspberrypi3-toolchain-3.1.3.sh`
- Default install path for this SDK is `/opt/poky/3.1.3`
- To use the SDK, you must first source a script, e.g.

```
$ source /opt/poky/3.1.3/environment-setup-cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi
```

Finding the sysroot of a Yocto Project toolchain

- The sysroot is reported as `/not/exist`

```
$ arm-poky-linux-gnueabi-gcc -print-sysroot
/not/exist
```

- Instead, the sysroot is set by shell variables CC, CXX and LD
- For example, CC contains

```
$ echo $CC
arm-poky-linux-gnueabi-gcc -mthumb -mfpu=neon-vfpv4 -mfloat-abi=hard -mcpu=cortex-a7
-fstack-protector-strong -D_FORTIFY_SOURCE=2 -Wformat -Wformat-security -Werror=
format-security --sysroot=/opt/poky/3.1.3/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi
```

- So, compile code using:

```
$ $CC helloworld.c -o helloworld
```

The tools

addr2line	Converts program addresses into file and line no.
ar	archive utility is used to create static libraries
as	GNU assembler
cpp	C preprocessor, expands #define, #include etc
g++	C++ front end, (assumes source is C++ code)
gcc	C front end, (assumes source is C code)
gcov	code coverage tool
gdb	GNU debugger
gprof	program profiling tool
ld	GNU linker
nm	lists symbols from object files
objcopy	copy and translate object files
objdump	display information from object files
readelf	displays information about files in ELF object format
size	lists section sizes and the total size
strings	displays strings of printable characters in files
strip	strip object file of debug symbol tables

Demo time (1)

Boot the Raspberry Pi

Log on

Cross compile helloworld and run on the Raspberry Pi

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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:

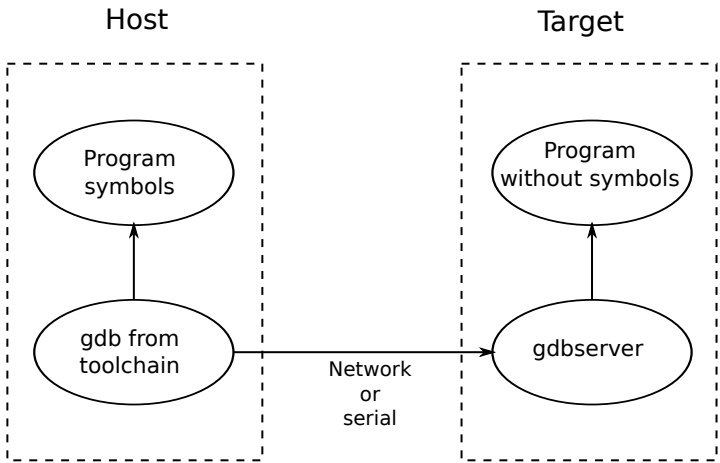
Level	Description
0	no debug information (equivalent to omitting -g)
1	minimal information, just enough to generate a backtrace
2	**(default) source-level debugging and single-stepping**
3	information about macros

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`

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)

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/3.1.3/...
(gdb) target remote 192.168.42.2:2001
```

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

Embedded target

```
# gdbserver :2001 helloworld
```

```
“Remote debugging from host 192.168.42.1”
```

```
{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

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

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
```

Run the program from the start (**does not work with remote debugging**)

```
run
```

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
```

Demo time

Debug helloworld

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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
```

Defining a new command

- You can define a new command like this:

```
define bmain
  break main
  info break
end
```

- Then run it just like any other gdb command:

```
(gdb) bmain
Breakpoint 1 at 0x400516: file helloworld.c, line 7.
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00400516 in main at helloworld.c:7
```

- Useful for sequences that you use many times
- You can put the code into a command file

Demo time

Create a command file to speed things up

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

TUI

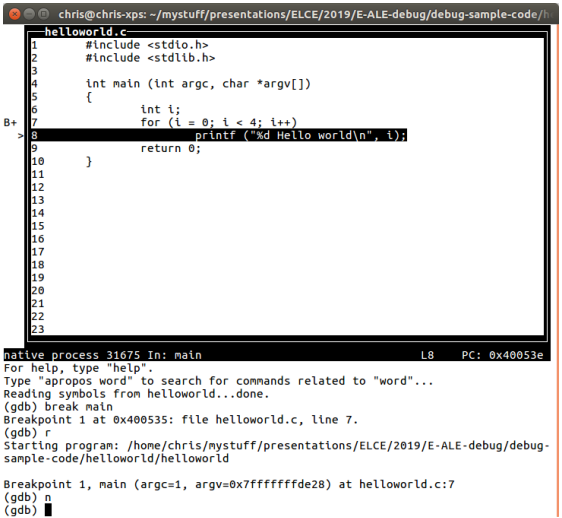
The Terminal User Interface, TUI, is an optional component of GDB

Just add -tui to the gdb command (assuming tui is enabled), for example

```
arm-poky-linux-gnueabi-gdb -tui helloworld
```

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

TUI



cgdb

https://cgdb.github.io/

Similar to TUI, but better

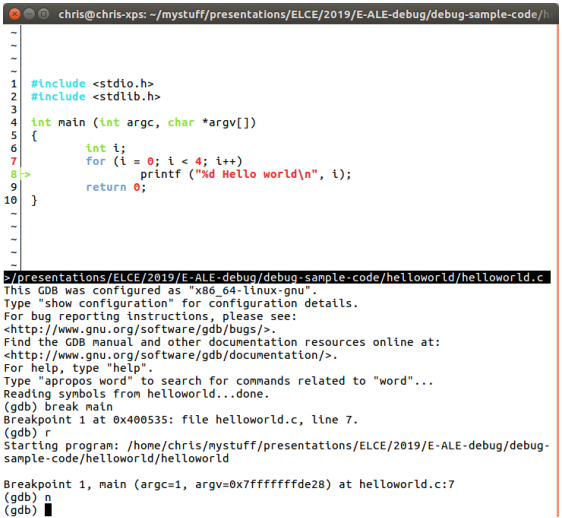
Not usually installable as a package, but you can instead can get it and install like this:

```
$ wget https://cgdb.me/files/cgdb-0.7.1.tar.gz
$ tar xz cgdb-0.7.1.tar.gz
$ cd cgdb-0.7.1
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

Then, launch it like this:

```
cgdb -d arm-poky-linux-gnueabi-gdb helloworld
```

cgdb



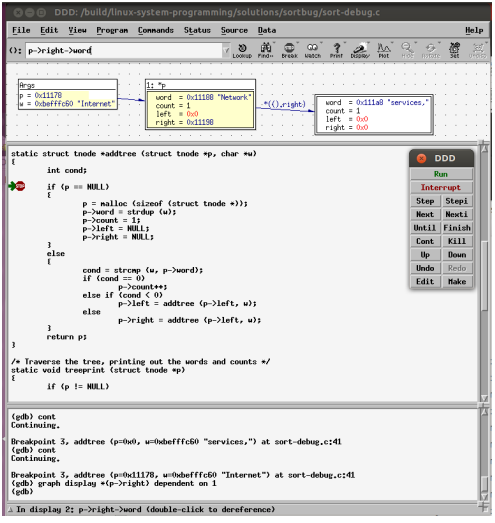
DDD: Data Display Debugger

A graphical front-end to GDB

Launch like this:

```
ddd --debugger arm-poky-linux-gnueabi-gdb helloworld
```

DDD: Data Display Debugger



Other front ends for GDB

- Eclipse CDT (C/C++ Development Toolkit)
- Microsoft Visual Studio Code
- KDevelop
- ... and others

Demo time

Debug using cgdb

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
```

Demo time

Set a watchpoint

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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
```

Demo time

Stack frames

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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: $cdir:$cwd
```

- 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
```

Coping with a relocated sysroot

- $cdir may be wrong if the library is copied to a different directory
  - For example, when installing an SDK
- You can find $cdir like this:

```
$ arm-poky-linux-gnueabi-objdump --dwarf helloworld | grep DW_AT_comp_dir
[...]
    <1c>   DW_AT_comp_dir    : (indirect string, offset: 0x8):
/usr/src/debug/glibc/2.31+gitAUTOINC+6fdf971c9d-r0/git/csu
[...]
```

- Then you can ask GDB to substitute the embedded path with the new one:

```
(gdb) set substitute-path /usr/src/debug /opt/poky/3.1.3/sysroots/cortexa7t2hf-
neon-vfpv4-poky-linux-gnueabi/usr/src/debug/
```

Demo time

Debug library code

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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
```

- To detach and allow the process to run freely again:

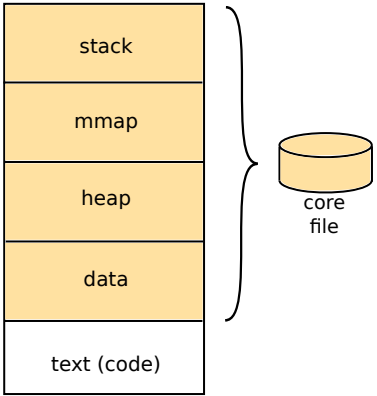
```
(gdb) detach
```

Demo time

Attaching to a running program

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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`

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

Using GDB to analyse a core dump

- Copy the core file from the target
- Then run `gdb <program executable> <core file>`

```
arm-poky-linux-gnueabi-gdb may-crash core
...
Core was generated by `may-crash'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0046851a in gratuitous_error () at may-crash.c:7
7       *p = 42;
```

Demo time

Core dumps

- Toolchains
- Remote debugging with gdbserver
- GDB command files
- GDB front ends
- Stack frames
- Debugging libraries
- Attaching to a running program
- Core dumps
- Final thoughts

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)
- Mastering Embedded Linux Programming by Chris Simmonds, Packt Publishing; 2nd edition

Any questions?

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

@2net_software

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