

```
>>> Operating Systems And Applications For Embedded Systems  
>>> Debugging
```

```
Name: Mariusz Naumowicz  
Date: 27 sierpnia 2018
```

>>> Plan

## 1. Connecting GDB and gdbserver

- TCP

- Serial

- Preparing to debug

- Remote debugging

- Breakpoints

- Running and stepping

- Information commands

- Running to a breakpoint

## 2. Debugging shared libraries

- The Yocto Project

- Buildroot

- Just-in-time debugging

## 3. Debugging kernel code

- Debugging kernel code with kgdb

- Additional reading

>>> TCP

On server side:

gdbserver :10000 ./hello-world

Process hello-world created; pid = 103

Listening on port 10000

On client side:

arm-poky-linux-gnueabi-gdb hello-world

(gdb) target remote 192.168.1.101:10000

On server side:

Remote debugging from host 192.168.1.1

```
>>> Serial
```

```
On server side:
```

```
gdbserver /dev/tty00 ./hello-world
```

```
On client side:
```

```
stty -F /dev/tty01 115200
```

```
(gdb) set remotebaud 115200
```

```
(gdb) target remote /dev/ttyUSB0
```

## >>> Preparing to debug

0: This produces no debug information at all and is equivalent to omitting the `-g` or `-ggdb` switch

1: This produces little information but which includes function names and external variables which is enough to generate a back trace

2: This is the default and includes information about local variables and line numbers so that you can do source level debugging and a single step through the code

3: This includes extra information which, among other things, means that GDB handles macro expansions correctly

## >>> Remote debugging

- \* At the start of a debug session you need to load the program you want to debug on the target using gdbserver and then separately load GDB from your cross toolchain on the host.
- \* GDB and gdbserver need to connect to each other before a debug session can begin.
- \* GDB, running onto the host, needs to be told where to look for debug symbols and source code, especially for shared libraries.
- \* The GDB run command does not work as expected.
- \* gdbserver will terminate when the debug session ends and you will need to restart it if you want another debug session.
- \* You need debug symbols and source code for the binaries you want to debug on the host, but not necessarily on the target. Often there is not enough storage space for them on the target and they will need to be stripped before deploying to the target.
- \* The GDB/gdbserver combination does not have all the features of GDB running natively: for example, gdbserver cannot follow the child after fork() whereas native GDB can.
- \* Odd things can happen if GDB and gdbserver are different versions or are the same version but configured differently. Ideally they should be built from the same source using your favorite build tool.

>>> **Breakpoints**

Command	Use
<code>break &lt;location&gt;</code> <code>b &lt;location&gt;</code>	Set a breakpoint on a function name, line number or line. Examples are: "main", "5", and <code>sortbug.c:42</code>
<code>info break</code> <code>i b</code>	List breakpoints
<code>delete break &lt;N&gt;</code> <code>d b &lt;N&gt;</code>	Delete breakpoint N

>>> Running and stepping

Command	Use
run r	Load a fresh copy of the program into memory and start it running. This does not work for remote debug using gdbserver
continue c	Continue execution from a breakpoint Ctrl-C Stop the program being debugged
step s	Step one line of code, stepping into any function that is called
next n	Step one line of code, stepping over a function call finish Run until the current function returns

>>> Information commands

Command	Use
backtrace bt	List the call stack
info threads	Continue execution from a breakpoint
Info libs	Stop the program
print <variable> p <variable>	Print the value of a variable, e.g. print foo
list	List lines of code around the current program counter

>>> Running to a breakpoint

(gdb) break main

Breakpoint 1, main (argc=1, argv=0xbeffffe24) at helloworld.c:8

8 printf("Hello, world!  
n");

## >>> The Yocto Project

You can add these debug packages selectively to your target image by adding <package name-dbg> to your target recipe. For glibc, the package is named glibc-dbg. Alternatively, you can simply tell the Yocto Project to install all debug packages by adding dbg-pkgs to EXTRA\_IMAGE\_FEATURES.

>>> Buildroot

1. BR2\_ENABLE\_DEBUG in the menu Build options | build packages with debugging symbols
2. setting Build options | strip command for binaries on target to none

>>> Just-in-time debugging

```
gdbserver -attach :10000 109
```

```
Attached; pid = 109
```

```
Listening on port 10000
```

```
(gdb) detach
```

```
Detaching from program: /home/chris/MELP/helloworld/helloworld,  
process 109
```

```
Ending remote debugging.
```

## >>> Debugging kernel code with kgdb

1. CONFIG\_DEBUG\_INFO is in the Kernel hacking | Compile-time checks and compiler options | Compile the kernel with debug info menu
2. CONFIG\_FRAME\_POINTER may be an option for your architecture, and is in the Kernel hacking | Compile-time checks and compiler options | Compile the kernel with frame pointers menu
3. CONFIG\_KGDB is in the Kernel hacking | KGDB: kernel debugger menu
4. CONFIG\_KGDB\_SERIAL\_CONSOLE is in the Kernel hacking | KGDB: kernel debugger | KGDB: use kgdb over the serial console menu

>>> Additional reading

1. The Art of Debugging with GDB, DDD, and Eclipse, by Norman Matloff and Peter Jay Salzman, No Starch Press; 1 edition (28 Sept. 2008), ISBN 978-1593271749
2. GDB Pocket Reference by Arnold Robbins, O'Reilly Media; 1st edition (12 May 2005), ISBN 978-0596100278
3. Getting to grips with Eclipse: cross compiling, <http://2net.co.uk/tutorial/eclipse-cross-compile>
4. Getting to grips with Eclipse: remote access and debugging, <http://2net.co.uk/tutorial/eclipse-rse>

## >>> References



C. Simmonds.

*Mastering Embedded Linux Programming.*

Packt Publishing, 2015.