Debugging Embedded Devices with GDB

Workbook

Version v 3.0

Embedded Linux Conference Europe, October 2020

1 Debugging embedded devices using GDB

Objectives

These exercises were written as an extra resource to my tutorial at the Embedded Linux Conference Europe 2020. They are intended to be a gentle introduction to remote debugging on the Raspberry Pi using GDB

1.1 Initial set up

At this point, you should have:

- A Raspberry Pi model 3B or 3B+ (*)
- A micro SD card
- An Ethernet cable
- A serial to USB cable, such as this one from Adafruit: (https://www.adafruit.com/product/954)
- A PC or laptop running a Linux distro, such as Ubuntu 18.04

(*) These are the only versions of the Raspberry Pi that these instructions will work with. Of course, the exercises themselves are not specific to the 3B/3B+, and they could be adapted to work on other Pis without much trouble. But, I repeat, the exercises only work verbatim on a Raspberry Pi 3B/3B+

1.2 (Optional) Build Yocto Project

You don't have to do this part, you can just skip to the next section and download my prebuilt image and SDK. If you do go ahead, note that building the core image and the SDK will take up about 70 GiB of disk space and may take over an hour to run

Get the meta layers for Yocto Project and Raspberry Pi. I am using the dunfell (3.1) release of Yocto Project, which is the latest version at the time of writing this workbook:

```
$ mkdir elce-gdb-totorial; cd elce-gdb-totorial
$ git clone -b dunfell git://git.yoctoproject.org/poky.git
$ git clone -b dunfell git://git.yoctoproject.org/meta-raspberrypi
```

Set up the build environment in directory `build-rpi`

```
$ source oe-init-build-env build-rpi
```

Add the Raspberry Pi layer

```
$ bitbake-layers add-layer ../meta-raspberrypi
```

Edit `conf/local.conf`. This is the entire content of my local.conf:

```
MACHINE = "raspberrypi3"
ENABLE_UART = "1"
DISTRO ?= "poky"
```

```
PACKAGE_CLASSES ?= "package_rpm"
EXTRA_IMAGE_FEATURES ?= "debug-tweaks tools-debug"

# enable tui interface in gdb
PACKAGECONFIG_append_pn-gdb = " tui"
PACKAGECONFIG_append_pn-gdb-cross-canadian-arm = " tui"

# Add the dropbear ssh server
CORE_IMAGE_EXTRA_INSTALL += "dropbear"

USER_CLASSES ?= "buildstats image-mklibs image-prelink"
PATCHRESOLVE = "noop"
BB_DISKMON_DIRS ??= "\
    STOPTASKS,${TMPDIR},1G,100K \
    STOPTASKS,${DL_DIR},1G,100K \
    STOPTASKS,${SSTATE_DIR},1G,100K \
    STOPTASKS,/tmp,100M,100K \
    ABORT,${TMPDIR},100M,1K \
    ABORT,${DL_DIR},100M,1K \
    ABORT,${SSTATE_DIR},100M,1K \
    ABORT,/tmp,10M,1K"

PACKAGECONFIG_append_pn-qemu-system-native = " sdl"
CONF_VERSION = "1"
```

Build base image

```
$ bitbake core-image-base
```

You will find the image here:
`tmp/deploy/images/raspberrypi3/core-image-base-raspberrypi3.wic.bz2`

Build the SDK

```
$ bitbake -c populate_sdk core-image-base
```

You will find the SDK here:
`tmp/deploy/sdk/poky-glibc-x86_64-core-image-base-cortexa7t2hf-neon-vfpv4-raspberrypi3-toolchain-3.1.3.sh`

1.3 Write the image to the SD card

If you did not build the image from source you can download it from here

https://2net.co.uk/downloads/core-image-base-raspberrypi3.wic.bz2

Plug a micro SD card into the reader on your computer. Run the command lsblk to find which device it has been assigned to. In this case I am using an 8 GB card, which shows up with an indicated size of 7.2 GB:

```
$ lsblk
NAME                    MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
[...]
mmcblk0                 179:0    0   7.2G  0 disk
|-mmcblk0p1             179:1    0   7.2G  0 part  /media/chris/6A44-4A1B
[...]
```

Write the image to SD card:

```
$ sudo bmaptool copy core-image-base-raspberrypi3.wic.bz2 /dev/mmcblk0
```

1.4 Boot the Raspberry Pi

Plug the SD card into the Raspberry Pi

Plug in the serial cable and run a terminal emulator program such as minicom

```
$ minicom -D /dev/ttyUSB0 -w
```

Power on the Raspberry Pi. You should see something like this on the serial console:

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Aug 13 2017, 15:25:34.
Port /dev/ttyUSB0, 15:29:16

Press CTRL-A Z for help on special keys

[    0.000000] Booting Linux on physical CPU 0x0
[...]
[   11.348058] NET: Registered protocol family 39
umount: can't unmount /mnt/.psplash: No such file or directory

Poky (Yocto Project Reference Distro) 3.1.3 raspberrypi3 /dev/ttyS0

raspberrypi3 login:
```

Log in as user root, no password

eth0 will use DHCP to get an IP address - use ifconfig to find out what it is

```
root@raspberrypi3:~# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr B8:27:EB:DB:3A:91
          inet addr:192.168.42.2  Bcast:192.168.42.255  Mask:255.255.255.0
[...]
```

In this case it is 192.168.42.2

1.5 Install the SDK

If you built the SDK from source in section 1.2, then you can install it like this:

```
\filePath{poky-glibc-x86_64-core-image-base-cortexa7t2hf-neon-vfpv4-raspberrypi3-}
\filePath{toolchain-3.1.3.sh}
```

Alternatively, if you did not build the SDK from source you can download a copy from

https://2net.co.uk/downloads/sdk-raspberrypi3-toolchain-3.1.3.sh (Note: you don't want to open the file, you want to save it)

Install it like so:

```
$ ./sdk-raspberrypi3-toolchain-3.1.3.sh
```

1.6 Get the sample code

Download the sample code from

https://2net.co.uk/downloads/debug-samples-elce2020.tar.gz

Extract the files:

```
$ cd
$ mkdir -p elce-gdb-totorial
$ cd elce-gdb-totorial
$ tar xf Downloads/debug-samples-elce2020.tar.gz
$ tree
.
|-- sample-code
    |-- gdb.init
    |-- helloworld
    | |-- helloworld.c
    | |-- Makefile
    |-- may-crash
    | |-- Makefile
    | |-- may-crash.c
    |-- usb-demo
    | |-- Makefile
    | |-- usb-demo.c
    |-- word-count
        |-- Makefile
        |-- test.txt
        |-- word-count.c
```

2 Remote debugging with gdbserver

2.1 Building the helloworld sample program

Set up the Yocto Project SDK

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

Compile the sample helloworld program:

```
$ cd
$ cd elce-gdb-totorial
$ cp -a sample-code/helloworld .
$ cd helloworld
$ make
```

Verify that it has been compiled for an ARM target

```
$ file helloworld
helloworld: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-armhf.so.3, BuildID[sha1]=d938ecdd080deec08ff56e33783b309193d2d8
16, for GNU/Linux 3.2.0, with debug_info, not stripped
```

Copy it to the Raspberry Pi:

```
$ scp helloworld root@192.168.42.2:/usr/bin
```

Log in to the Raspberry Pi using ssh

```
$ ssh root@192.168.42.2
```

Run the program on the Raspberry Pi. You should see that it prints out four lines of text:

```
# helloworld
0 Hello world
1 Hello world
2 Hello world
3 Hello world
```

2.2 Debugging helloworld

The next task is to run helloworld in a GDB session.

On the Raspberry Pi, launch the program with gdbserver:

```
# gdbserver :2001 /usr/bin/helloworld
Process /usr/bin/helloworld created; pid = NNN
Listening on port 2001
```

On you computer, start gdb

```
cd helloworld
$ arm-poky-linux-gnueabi-gdb helloworld
[...]
Reading symbols from helloworld...
(gdb)
```

Connect to the Raspberry Pi

```
(gdb) set sysroot /opt/poky/3.1.3/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi
(gdb) target remote 192.168.42.2:2001
```

Set a breakpoint on `main`

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

List the breakpoints

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

Continue to the breakpoint

```
(gdb) continue
Continuing.

Breakpoint 1, main (argc=1, argv=0x7efffdb4) at helloworld.c:7
7               for (i = 0; i < 4; i++)
```

List the lines of code around the breakpoint (which is on line 7)

```
(gdb) list
2       #include <stdlib.h>
3
4       int main (int argc, char *argv[])
5       {
6               int i;
7               for (i = 0; i < 4; i++)
8                       printf ("%d Hello world\n", i);
9               return 0;
10
```

Run the program one step at a time using the **next** instruction, which you can abbreviate to **n**, until it has been round the loop once.

Type **continue** (abbreviation **c**) to allow the program to continue executing

When the program has finished, quit GDB by typing **quit** (abbreviation **q**)

2.3 Looking at variables

Launch helloworld with gdbserver on the Raspberry Pi and `arm-poky-linux-gnueabi-gdb helloworld` on your development host, as in the precious exefcrise Set a breakpoint on main and run the program

Use the **print** command (abbreviation **p**) to display variable **i**:

```
(gdb) print i
```

Step through the program and see that i changes on each iteration

Try setting i to a different number **just before** the printf:

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

Note that the program prints out 99

3 GDB command files

To reduce the amount of typing, create a GDB command file. Call it `gdb.init` and put these lines into it

```
set sysroot /opt/poky/3.1.3/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi

define rpi3
  target remote 192.168.42.2:2001
end
```

Launch helloworld with gdbserver as before, then launch GDB like this

```
$ arm-poky-linux-gnueabi-gdb -x gdb.init helloworld
```

To connect to the Raspberry Pi, type

```
(gdb) rpi3
```

You will find a copy of gdb.init in the `sample-code` directory. I will be using that for the remainder of this tutorial

4 GDB front ends

In this exercise we will be using **cgdb**

Install cgdb for your Linux distro, e.g. for Ubuntu you would type:

```
$ sudo apt install cgdb
```

Alternatively, you can install it from source as shown on the slides:

```
$ 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
```

To use the cgdb front end to GDB:

```
$ cgdb -d arm-poky-linux-gnueabi-gdb -x gdb.init helloworld
```

Repeat the previous session using this interface

5 Stack frames

For this demo, we will use the word-count program. Cross-compile it like so:

```
$ cd
$ cd elce-gdb-totorial
$ cp -a sample-code/word-count .
$ cd word-count
$ make
```

Copy word-count and the sample data file to the Raspberry Pi:

```
$ scp word-count root@192.168.42.2:/usr/bin
$ scp test.txt root@192.168.42.2:
```

Now check that it works by logging on to the Raspberry Pi and running the program:

```
# word-count test.txt
   1 brown
   1 dog
   1 fox
   1 jumps
   1 lazy
   1 over
   1 quick
   2 the
```

Start word-count with gdbserver

```
# gdbserver :2001 /usr/bin/word-count test.txt
Process /usr/bin/helloworld created; pid = NNN
Listening on port 2001
```

Launch gdb

```
$ cgdb -d arm-poky-linux-gnueabi-gdb -x gdb.init word-count
```

Use the rpi3 command to connect to the Raspberry Pi

Set a breakpoint on function addtree, then type 'c'

You will see that addtree is called recursively. At each break, you can use the backtrace command (abbreviated bt) to show where you are in the call sequence. You can use the `frame` command to switch to another stack frame

6 Debugging libraries

This exercise is based on a program called usb-demo, which prints a list of usb devices. The objective is to be able to step into and see the code for the libraries it uses, which are libc and libusb

Compile the program and test it:

```
$ cd
$ cd elce-gdb-totorial
$ cp -a sample-code/usb-demo .
$ cd usb-demo
$ make
```

Copy it the Raspberry Pi:

```
$ scp usb-demo root@192.168.42.2:/usr/bin
```

Run it on the Raspberry Pi to see what it does:

```
# usb-demo
USB demo program
USB device 0 0424:7800
USB device 1 0424:2514
USB device 2 0424:2514
USB device 3 1d6b:0002
```

To set breakpoints and step through library code you need to tell GDB where to find the source code.

GDB reads the executable to find the compiled source directory ($cdir). You can see what that is using this command:

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

But that path does not lead to the place where the source code is stored. In the case of the Yocto Project SDK, the source code is actually in `usr/src/debug` **relative to the sysroot**. So you need tell GDB to substitute the paths like so:

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

With this substitution in place you will find that you can step into C library functions such as printf

But, when you try to step into functions such as `libusb_init`, you find that GDB reports an error:

```
12              int ret = libusb_init(NULL);
(gdb) s
libusb_init (context=0x0) at ../../libusb-1.0.22/libusb/core.c:2121
2121    ../../libusb-1.0.22/libusb/core.c: No such file or directory.
```

This path does not work with the substitute-path set earlier. One simple solution is to use the **dir** command to point explicitly to the directory containing the code for libusb. The code should be in the sysroot, so:

```
$ find /opt/poky/3.1.3/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/ -name core.c
/opt/poky/3.1.3/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/src/debug/libusb1/1.
0.22-r0/libusb-1.0.22/libusb/core.c
```

Now run GDB again and add this directory to the search with

```
dir /opt/poky/3.1.3/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/src/debug/libusb
1/1.0.22-r0/libusb-1.0.22/libusb
```

Then check that you can step into the `libusb_init` function

7 Attaching to a running program

Let's take a look at what the init program is doing.

First, you need to find the full path to the executable you want to debug. Bearing in mind that the init program always has PID 1, you can look into the proc file system to find the path to the executable like this:

```
# ls -l /proc/1/exe
lrwxrwxrwx    1 root     root             0 Jan  1  1970 /proc/1/exe -> /sbin/init.sysvinit
```

Use the attach option of gdbserver to attach to init, with PID 1

```
# gdbserver --attach :2001 1
```

Now use GDB to load the symbol table for init and connect to the target

```
$ cgdb -d arm-poky-linux-gnueabi-gdb -x ../gdb.init \
/opt/poky/3.1.3/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/sbin/init.sysvinit
```

In gdb, type next. After a while, init will wake up and run the next line of code. Now you have control of init and can debug it in the normal way

Try the **backtrace** command to show the call stack to the current location

Try stepping through init to see how it works

When you are done, detach from init so that it can continue without intervention

```
(gdb) detach
```

8 Core dumps

Build the test program Compile the program and test it

```
$ cd
$ cd elce-gdb-totorial
$ cp -a sample-code/may-crash .
$ cd may-crash
$ make
```

Copy it the Raspberry Pi:

```
$ scp may-crash root@192.168.42.2:/usr/bin
```

Run it on the Raspberry Pi to see what it does:

```
# may-crash
0 Hello world
Segmentation fault
```

But, there is no core file, because the ulimit is not set.

```
# ulimit -c
0
```

So, go ahead and set the limit for core files to "unlimited":

```
# ulimit -c unlimited
```

Now run the program and it will generate a core file in the current directory.

This is usually inconvenient, so try creating a directory for core files and set a core pattern that references it:

```
# mkdir /corefiles
# chmod 777 /corefiles
# echo "/corefiles/%e-%p" > /proc/sys/kernel/core_pattern
```

Run the program again - a core file is written to `/corefiles`

Now, you can copy the core file to your PC and use GDB to look at the state of the program when is crashed.

```
$ scp root@192.168.42.2:/corefiles/* .
```

```
$ cgdb -d arm-poky-linux-gnueabi-gdb -x gdb.init may-crash may-crash-907
[...]
Core was generated by ‘may-crash’.
Program terminated with signal SIGSEGV, Segmentation fault.
(gdb)
```