Debugging with GDB

Workbook

Version v 2.0

October 2019

1 Debugging embedded devices using GDB

Objectives

This exercise is a gentle introduction to debugging on the PocketBeagle

While you are running these labs, keep a copy of the GDB Quick Reference card open. It's in the file `refcard.pdf`.

1.1 Initial set up

At this point, you should have:

- A PocketBeagle with TechLab cape
- A micro SD card containing the PocketBeagle system image
- A micro USB to full size USB cable
- Your laptop!

Download https://cm.e-ale.org/2019/ELCE2019/debugging-with-gdb/debugging-samples.tar.gz and copy it to the micro SD card

Plug the micro SD card into the PocketBeagle and boot it up by inserting **both** USB cables into the PocketBeagle.

Log on to the PocketBeagle as debian (password temppwd)

Extract the sample code into the home directory:

```
cd
tar xf debug-samples.tar.gz
```

Install gdb:

```
cd
cd debug-sample-code/debs
sudo ./install-gdb.sh
```

1.2 Building helloworld

Compile the sample helloworld program:

```
cd
cd debug-sample-code/helloworld
gcc helloworld.c -o helloworld -ggdb
```

Verify that it has been compiled for an ARM target

```
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, inte
rpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0, BuildID[sha1]=c1ff6dbe460c9ae4fea4121
80558497b8cbf459f, not stripped
```

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

```
./helloworld
0 Hello world
1 Hello world
2 Hello world
3 Hello world
```

1.3 Debugging helloworld

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

```
cd
cd debug-sample-code/helloworld
gdb helloworld
[...]
Reading symbols from helloworld...done.
(gdb)
```

Set a breakpoint on `main`

```
(gdb) break main
Breakpoint 1 at 0xNNNNN: 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
```

Run the program from the beginning

```
(gdb) run
Starting program: /home/root/debug-sample-code/helloworld/helloworld

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

Note that it has stopped at the breakpoint

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 **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**)

1.4 A better user interface

Pure command-line GDB is not so easy. Instead, try the Text User Interface by adding `-tui` when starting gdb:

```
cd
cd debug-sample-code/helloworld
gdb -tui helloworld
```

Repeat the previous session using this interface

Hint: when helloworld prints to stdout it causes the top (source code) window to scroll, which corrupts the display. When this happens, press **Ctrl-l**, or type **refresh**, to redraw the screen

There is another popular text interface to GDB called **cgdb** which uses colour highlighting to make the display easier to read. If you have it installed, try using it like this:

```
cd
cd debug-sample-code/helloworld
cgdb helloworld
```

1.5 Looking at variables

Start helloworld as before. 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

Restart the program and try setting **i** to a negative number

1.6 Shared libraries

Compile a library that controls the two 7-segment LED displays

```
cd
cd debug-sample-code/seven-seg/libsevenseg
make
sudo make install
```

Compile a program that uses libsevenseg

```
cd
cd debug-sample-code/seven-seg/rotate
make
```

The program is meant to make one illuminated segment rotate around on of the displays. Use GDB to debug the program and make it work properly. You will need to step into the code for libsevenseg

1.7 (Optional) core dump

Build the test program

```
cd
cd debug-sample-code/may-crash/
make
```

Run the program

```
./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:

```
sudo mkdir /corefiles
sudo chmod 777 /corefiles
sudo sh -c 'echo "/corefiles/%e-%p" > /proc/sys/kernel/core_pattern'
```

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

Use GDB to look at the state of the program when is crashed.

```
$ gdb <program name> <core file name>
```

1.8 (Optional) JIT debugging

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

Use the attach option of GDB to attach to init, which always has PID 1

```
sudo gdb /sbin/init
(gdb) attach 1
```

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.