Linux for Embedded Systems

Lab 2 - Report

By Noman Noor (id: 302343)

Current Date: 01/05/2022 (dd/mm/yy)

PROBLEM DESCRIPTION

Task: Making a program that uses the LEDs and Buttons of the Lab Raspberry Pi4 extension board, setting it up as a package, and demonstrating GDB cross-debugging (with debouncing).

Program Description: I made a game of Whackamole. There are a number of rounds (7 in the current source code) in which one random light will light up, and you have to hit the switch corresponding/next to it within a set amount of time. Missing any leads to getting 0 points added to the score. Pressing the correct one leads to getting some points (3 in the current version) added to the score. Pressing any wrong buttons results in getting 1 point subtracted from the score (for each specific button that was pressed wrongly at least once). After each round, the results are printed (including any button presses). After the final round, the program prints the final score and quits.

The program also has thorough error checking.

Procedure to recreate the design from the attached archive:

In the directory where the contents of the archive were extracted, run the script build.sh:

```
$ chmod +x ./build.sh
$ ./build.sh
```

Step 1: Writing the program and dealing with libgpiod

I built a C program using the “libgpiod” library. (#include “gpiod.h”) and including a “-l gpiod” flag in the Makefile for compilation. This is a library for interacting with GPIOD devices. However, I do regret not using WiringPi from the beginning because even though it would affect portability, it would improve the program by virtue of enabling the developer to use various tools not present in libgpiod. Also, it would be way easier to use and find help for.

The main functions I used are ones for waiting for an event in bulk, and reading multiple (16 = max number of events) events from the lines which did get an event using the multi variant of the event reading function because that doesn’t cause blocking (the single event reading one does). Then I have debouncing for each line (doesn’t work fully well, needed changes are in the top of the whackamole.c file in form of comments). Then I run the logic of each of those lines. Of course, I do properly initialize the chip and all the needed lines and request for output/events as needed in the beginning of the program. At the end of the program I release all of the resources.

libgpiod lacks any proper documentation or resources even from third-parties, therefore even figuring out which functions to use when and which order (or which functions need to be run before or after the given function) and so on has been extremely laborious and hard. The “documentation” that libgpiod has is in the form of JsDoc comments for their functions, but even my own JsDoc comments in the whackamole.c program are better qualified to be called “documentation” than the libgpiod ones.

Step 2: Setting up Whackamole as a package

We write an appropriate “whackamole.mk” and “Config.in” and put it in the {Buildroot Directory}/package/whackamole path. Then, we edit {Buildroot Directory}/package/Config.in and go to the games section (because our package is a game). There, we add source "package/whackamole/Config.in" to it in alphabetical order of package names (maintaining the order isn’t mandatory but generally agreed upon to be the best way to do it). The way the configurations have been done so far, we have our sourcecode directory set to be directly outside of the Buildroot directory (the directory containing the main Buildroot folder).

As such, relative to Buildroot, we have to have the source directory ../whackamole in which we have an appropriate makefile and the whackamole.c program source code.

Step 3: Setting up Buildroot

I mostly use the same basic configuration from the previous lab without too much modifications (except now I don’t use overlays for the script,etc.).

The configuration for this lab specifically included:
Turning on “build packages with debugging symbols” and setting “gcc debug level (debug level 3)” so as to get more debugging info.

Then we select to include the GDB server as a package, as well as the “whackamole” package.

Also, select “Build cross gdb for the host” in case it is needed in your case. (for lab machines, the default cross-debugger in the /output/build(or /output/host?) directory works.

Then we build the image, and flash it to our RPi4 as described in the previous lab report.

Step 4: Remote debugging the program

In order to run the program, we simply type “whackamole” in our shell and press enter (new line).

Now, to debug it:

1. **On RPi4:** We run the “gdbserver {ipaddress}:{port} whackamole” where {ipaddress} and {port} are the IP of the RPi4 and the port for GDB Server to run on respectively.
2. **On your PC:** run **“**gdb {Buildroot Directory}/build/whackamole” [Note: make sure you don’t rebuild the package before doing so, buildroot requires the same program to be run without any changes (maybe even the same build)]
3. **On your PC:** You’ll enter GDB’s interactive mode on the terminal. There, run “target remote {rpi4_ipaddress}:{rpi4_port}”.
4. **On your PC:** You can set up breakpoints now, or after running the program by pausing it again. To set breakpoints, use: “break whackamole:{line_num}” or simply “b whackamole:{line_num}” in short. In fact, if the program is currently running in the same file (whackamole), and not one from a different library or something, you can just do “break {line_num}” or “b {line_num}”.
5. **On your PC:** use “run” to start the program.

Note: you can use the following cheatsheets to learn to do more:
https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf