Embedded Linux

LXE22109

Practical Labs

bootlin

https://bootlin.com

August 24, 2022

About this document

Updates to this document can be found on https://bootlin.com/doc/training/lxe22109-06.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

Copying this document

Corrections, suggestions, contributions and translations are welcome!

Application development with Buildroot

Objectives:

- *Build and run your own application*
- *Remote debug your application*
- *Create a package for your application*

Build and run your own application

Let's create your own little application that we will use for demonstration in this lab. Create a folder `$HOME/lxe22109-06-labs/myapp`, and inside this folder a single C file called `myapp.c` with the following contents:

```
#include <stdio.h>

int main(void) {
        printf("Hello World\n");
        return 0;
}
```

To build this application, we'll use the cross-compiler generated by Buildroot. To make this easy, let's add the Buildroot host directory into our PATH:

```
export PATH=$HOME/lxe22109-06-labs/buildroot/output/host/bin:$PATH
```

Now you can build your application easily:

```
arm-none-linux-gnueabihf-gcc -o myapp myapp.c
```

Copy the myapp binary to your target using scp:

```
scp myapp root@192.168.0.2:
```

And run the `myapp` application on your target.

Now, let's extend the application a little bit more to use a library, the `libconfig` library we've already used in a previous lab. Change the source code of the application to the one provided in this lab data directory, `myapp.c`.

If you try to build this application with just:

```
arm-none-linux-gnueabihf-gcc -o myapp myapp.c
```

It fails to build because it does not link with `libconfig`. So you can manually do:

```
arm-none-linux-gnueabihf-gcc -o myapp myapp.c -lconfig
```

Since libconfig.so is in output/staging/usr/lib and the compiler is configured to automatically look in output/staging as its *sysroot*, it works fine.

However, there's a better solution: using *pkg-config*. Buildroot has installed a special version of pkg-config in output/host/bin, which you can query for libraries available for the target. Run:

```
pkg-config --list-all
```

And check you have libconfig mentionned. You can query the compiler and linker flags for libconfig:

```
pkg-config --cflags --libs libconfig
```

And use that to build your application:

```
arm-none-linux-gnueabihf-gcc -o myapp myapp.c $(pkg-config --cflags --libs \
    libconfig)
```

In the case of libconfig, it doesn't simplify a lot because the compiler and linker flags are simple, but for some other libraries, they are more complicated.

Copy the new version of myapp to your target, and run it. Create a myapp.cfg config file, and run your application again.

Remote debug your application

Our application is simple and works, but what if you need to debug it? So let's set up remote debugging.

The *ARM* toolchain is provided with a pre-compiled *gdbserver*, so we'll simply use it. Enable the option BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY, and then force the re-installation of the toolchain using:

```
make toolchain-external-arm-arm-reinstall
```

Reflash your system, or alternatively, just copy output/target/usr/bin/gdbserver to the target /usr/bin/ directory using scp.

To do some appropriate debugging, we need to have debugging symbols available. So we need to do two things:

1. Rebuild our application with the -g flag.
2. Rebuild the Buildroot system with debugging symbols, so that shared libraries have debugging symbols. However, since we don't want to rebuild the entire Buildroot system now, we'll use a trick and rebuild only the library we need to have the debugging symbols for: libconfig. To achieve this, first go to Buildroot menuconfig, and in Build options, enable build packages with debugging symbols. Then, do make libconfig-dirclean all to force the rebuild of just libconfig.

Now, on your target, start *gdbserver* in multi-process mode, listening on TCP port 2345:

```
gdbserver --multi localhost:2345
```

Back on the host, run the cross-gdb with the myapp application as argument:

```
arm-none-linux-gnueabihf-gdb myapp
```

We need to tell `gdb` where the libraries can be found:

```
(gdb) set sysroot output/staging
```

And then connect to the target:

```
(gdb) target extended-remote 192.168.0.2:2345
```

Define which program we want to run on the target:

```
(gdb) set remote exec-file myapp
```

Let's put a breakpoint on the `main` function, and start the program:

```
(gdb) break main
(gdb) run
```

It stops on the first line of the `main` function, which is the call to `config_init`, implemented by the `libconfig` library. If you do the *gdb* instruction `step`, *gdb* will step into the function, so you can follow what happens. After having done `step` once, you can do `backtrace` to see that you are in the function `config_init` called by `main`:

```
(gdb) backtrace
#0  config_init (config=0xbefffc3c) at libconfig.c:725
#1  0x000106f0 in main () at myapp.c:11
```

Note that if you want `gdbserver` to stop on the target, you need to run the *gdb* command `monitor exit`.

Create a package for your application

Building manually your own application is not desirable, we obviously want to create a Buildroot package for it. A useful mechanism to package your own applications is to use the `local` *site method*, which tells Buildroot that the source code of your application is available locally.

Create a new package called `myapp` in your `BR2_EXTERNAL` tree, and by using the `local` *site method*, make it use directly the `myapp` source code from `$HOME/lxe22109-06-labs/myapp`. Remember that you can use `$(TOPDIR)` to reference the top-level directory of the Buildroot sources.

For now, directly call `gcc` in the build commands. Of course, if your application becomes more complicated, you should start using a proper build system (Makefile, autotools, CMake, etc.).

When the package builds, you should see as the first step being done that the `myapp` source code gets *rsynced* from `$(HOME)/bootlin/myapp`:

```
>>> myapp custom Syncing from source dir /home/thomas/bootlin/myapp
```

The build should now proceed to the end. Now, make a stupid but visible change to the source code in `myapp.c`.

Restart the build of `myapp` using `make myapp-rebuild`, you will see that Buildroot automatically *rsyncs* again the source code. Then scp the file `output/target/usr/bin/myapp` to `192.168.0.2:/usr/bin` and run `myapp` again on the target.

As you can see you can now develop your applications and libraries, using your normal version control system and relying on Buildroot to do all the configure, build and install steps for you.