Linux on embedded systems

Author: Mateusz Urbaniak
Class: Intermediate Project
Supervisor: Ph.D. Witold Paluszyński
Date: January 22, 2022
Embedded Robotics,
Chair of Cybernetics and Robotics,
Faculty of Electronics, Photonics and Microsystems,
Wrocław University of Technology

Abstract

One of the project goals was to get familiar with building Linux system using different tools(Buildroot, Yocto Project) for multiple embedded devices(like Raspberry pi 4B and STM). This goal was slightly changed after talks with the supervisor. It was agreed that it is better to focus on one board and build a simple application that would use, for example, I2C communication to get some information from some peripherals.

The main objective of the project was achieved. Linux distributes were successfully built and ran using both Buildroot and Yocto Project. Custom application that read temperature of the room using I2C temperature sensor was also successfully built and ran.

1 Introduction

Linux on embedded system is standard in industrial applications. Linux is the most widely used operating system on embedded devices.It is also worth to emphasize that the issues raised here are not limited to hobby projects and academic research, but they are essential for industrial and multimedia applications.

1.1 Tools

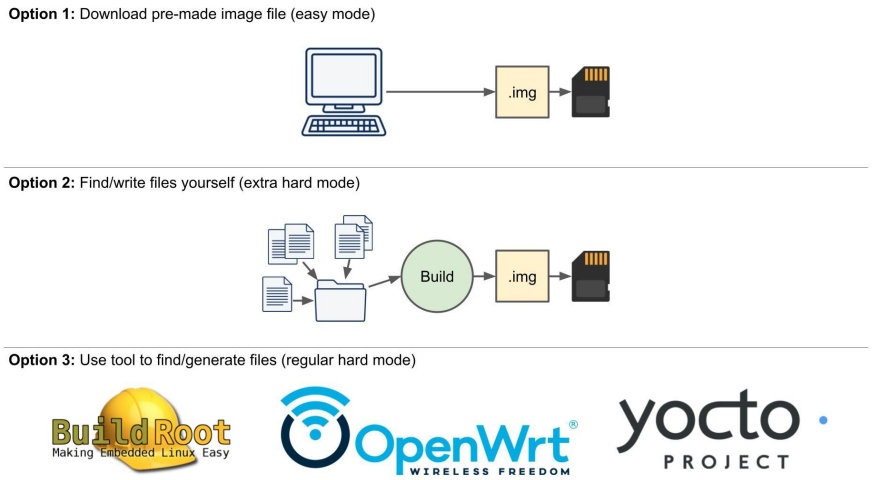

Figure 1: Different options for building Linux distribution

Many users are familiar with a single board computer (e.g., Raspberry Pi) and they are probably familiar with the easiest method of loading an operating system onto the board: downloading a pre-made image (usually a .img file) from the Internet and copy it to an SD card.

Using option 2 or 3 allows for customization of the Linux distribution to the exact needs of the user(such as disabling a user interface and enabling networking). This customized distribution can reduce boot time, save on power, and reduce possible attack vectors for hackers. Then the manufacturer can copy the image to entire production line of single board computers or systems on a module (SOMs) during manufacturing.

Second option for creating such a customized distribution includes finding or writing the necessary drivers and libraries by the user, compiling them, and creating an image. This is a difficult and time-consuming task.

The final option is to use a tool like Buildroot, OpenWRT, or the Yocto Project to help script the entire process of finding the required source code and building an image. These are likely the three most popular embedded Linux build tools.

2 Goals of the project

The project goal is to get familiar with building Linux system using different tools for multiple embedded devices.
Must do:

- build Linux on Raspberry Pi 4 using Buildroot and Yocto project,
- build Linux on one different board (example: BeagleBone, Nvidia, Odyssey, STM),
- compare building process (how easy it is, how quick it is),
- show examples of usage.

3 Results

3.1 Building with Buildroot

Building a custom Linux distribution with Buildroot is much simpler than with Yocto Project. It can be recommended for new users, but with its simplicity comes two significant cons. The first is that user loses out some customization options. The second is that user have to build the image from scratch every time you make a change and it can take hours.

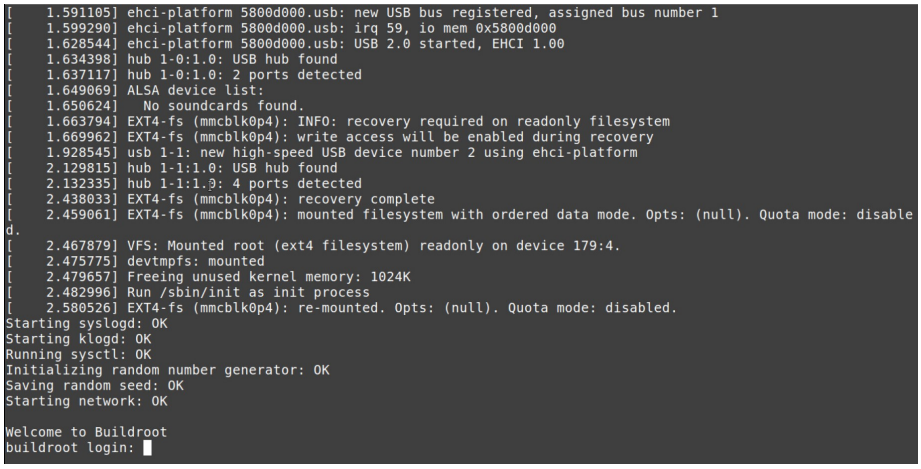

Figure 2: Boot process and a login prompt

3.2 Building with Yocto Project

Yocto project is a powerful tool for building customized Linux images, but it has a steep learning curve. Terminology in the Yocto Project can be confusing. These definitions are the most important:

- **OpenEmbedded**– build system and community
- **The Yocto Project**– umbrella project and community

- **Metadata**– files containing information about how to build an image
- **Recipe**– file with instructions to build one or more packages
- **Layer**– directory containing grouped metadata (start with "meta-")
- **Board support package (BSP)**– layer that defines how to build for board (usually maintained by vendor)
- **Distribution**– specific implementation of Linux (kernel version, rootfs, etc.)
- **Machine**– defines the architecture, pins, buses, BSP, etc.
- **Image**– output of build process (bootable and executable Linux OS)

3.3 Custom application to get temperature

To get the temperature from TMP102[2] sensor, the simple program that reads the correct memory addresses was written. Next step was to add it to custom image Linux and rebuild this image.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <fcntl.h>

int main()
{
    // Settings
    const unsigned char tmp102_addr = 0x48; // I2C address of the TMP102
    const unsigned char reg_temp = 0x00;     // Address of temperature register
    const char filename[] = "/dev/i2c-1";    // Location of I2C device file
    int file;
    char buf[5];
    int16_t temp_buf;
    float temp_c;
    // Open the device file for read/write
    if ((file = open(filename, O_RDWR)) < 0)
    {
        printf("Failed to open the bus.\n");
        exit(1);
    }
    // Change to I2C address of TMP102
    if (ioctl(file, I2C_SLAVE, tmp102_addr) < 0) {
        printf("Failed to acquire bus access or talk to device.\n");
        exit(1);
    }
```

```
    // Start read by writing location of temperature register
    buf[0] = 0x00;
    if (write(file, buf, 1) != 1)
    {
        printf("Could not write to I2C device.\n");
        exit(1);
    }
    // Read temperature
    if (read(file, buf, 2) != 2)
    {
        printf("Could not read from I2C device.\n");
        exit(1);
    }
    // Combine received bytes to single 16-bit value
    temp_buf = (buf[0] << 4) | (buf[1] >> 4);
    // If value is negative (2s complement), pad empty 4 bits with 1s
    if (temp_buf > 0x7FF)
    {
        temp_buf |= 0xF000;
    }
    // Convert sensor reading to temperature (Celsius)
    temp_c = temp_buf * 0.0625;
    // Print results
    printf("%.2f deg C\n", temp_c);

    return 0;
}
```

```
root@stm32mp1:~# gettemp
21.81 deg C
root@stm32mp1:~# gettemp
22.88 deg C
root@stm32mp1:~# gettemp
23.06 deg C
root@stm32mp1:~# gettemp
25.00 deg C
root@stm32mp1:~# gettemp
24.44 deg C
root@stm32mp1:~# gettemp
22.31 deg C
root@stm32mp1:~#
```

Figure 3: Result from the gettemp application

4 Summary

Creating a custom distribution is not an easy task. To configure a kernel for custom needs, user needs to have good knowledge about various topics(e.g., power management, drivers, buses) to properly setup needed functions without enabling functions that one does not need. Also working with custom distributions might be hard for some people because it is common to work many hours, checking the documentation[4] and not get visible results.

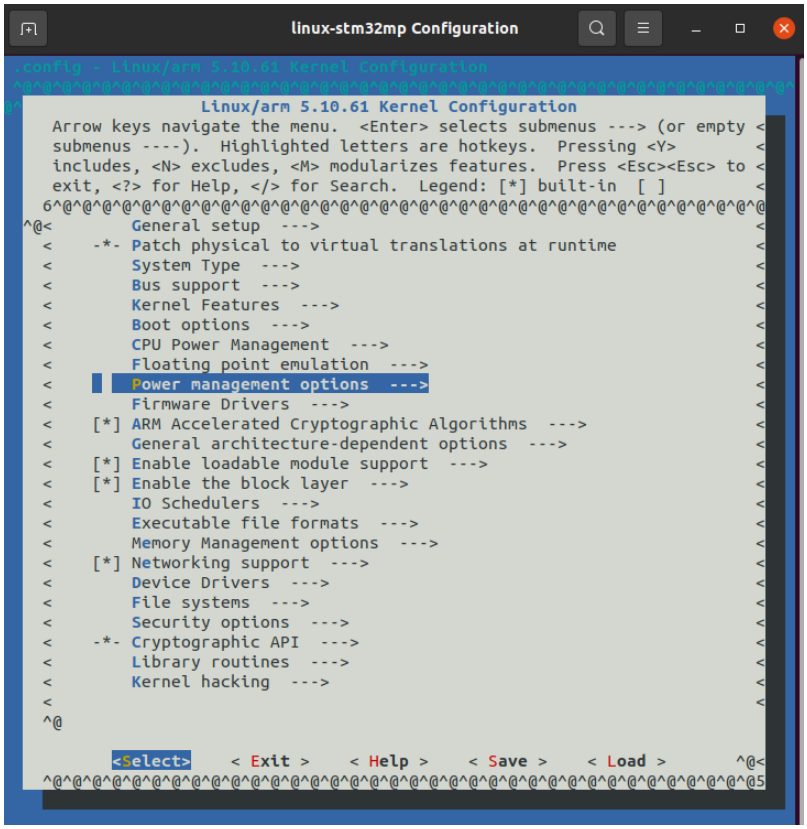

Figure 4: Main window of kernel configuration

Overall, the most important goals were achieved and the project was a success.

5 References

References

[1] Digi-Key. *Intro to Embedded Linux*. https://www.digikey.com/en/maker/projects/intro-to-embedded-linux-part-1-buildroot/a73a56de62444610a2187cd9e681c3f2. Access: January 2022.

[2] Texas Instruments. *TMP102 Temperature Sensor*. https://www.ti.com/lit/ds/symlink/tmp102.pdf?ts=1642845287112&ref_url=https%253A%252F%252Fwww.google.de%252F. Access: January 2022.

[3] Jay Carlson. *So you want to build an embedded Linux system?* https://jaycarlson.net/embedded-linux/#1602627646244-26484bfd-5515. Access: January 2022.

[4] ST. *STM32MP157 reference manual*. https://www.st.com/resource/en/reference_manual/dm00327659-stm32mp157-advanced-arm-based-32-bit-mpus-stmicroelectronics.pdf. Access: January 2022.