Digital electronics

J.-M Friedt

FEMTO-ST/time & frequency department

jmfriedt@femto-st.fr

slides at `jmfriedt.free.fr`

February 15, 2025

Plan

7 lessons/lab sessions (4-hour long schedules):

1. Executive environments: principles and introduction, getting started with FreeRTOS
2. FreeRTOS, RTEMS, Nuttx ... multitasking and associated methods to make sure shared data and resources are kept in known states (mutex & semaphore)
3. Using the scheduler, mutex and semaphores to solve the "philosopher problem"
4. Embedded systems with GNU/Linux – POSIX compatible operating system
   Architecture of an operating system, kernel v.s userspace
   Internet connectivity and networking
5. Accessing hardware resources from userspace – memory translation from physical to virtual address space (Memory Management Unit) – `/dev/mem`
6. Accessing hardware resources from a web server – internet connected instrument
7. From userspace to kernel space: character device (*char device*) for communicating between users and the kernel

Introduction

- Why use an operating system (OS) on an embedded board?
- Impact (memory, CPU workload)?
- Work method: programs developed on a personal computer (PC) will run on the embedded system with the same OS
- Added functionalities: networking, dynamic scheduler, filesystem, other (database? crypotography?)
- gcc: unified framework for developing on all platforms

Beyond C: an operating system

- adding yet another abstraction layer (assembly – C – kernel) and hence API
- Why an operating system? scheduler, memory management (multi-tasking), abstraction layer aimed at hiding the hardware low level description from the programmer, **filesystem** management ($>$ rawrite), **communication** (IP, TCP ...), console for interactive shell with user.
- But new additional constraints: understand a new set of protocols and programming methods (API) ...
- ... in order to properly handle resource sharing between multiple users.

Operating system basics

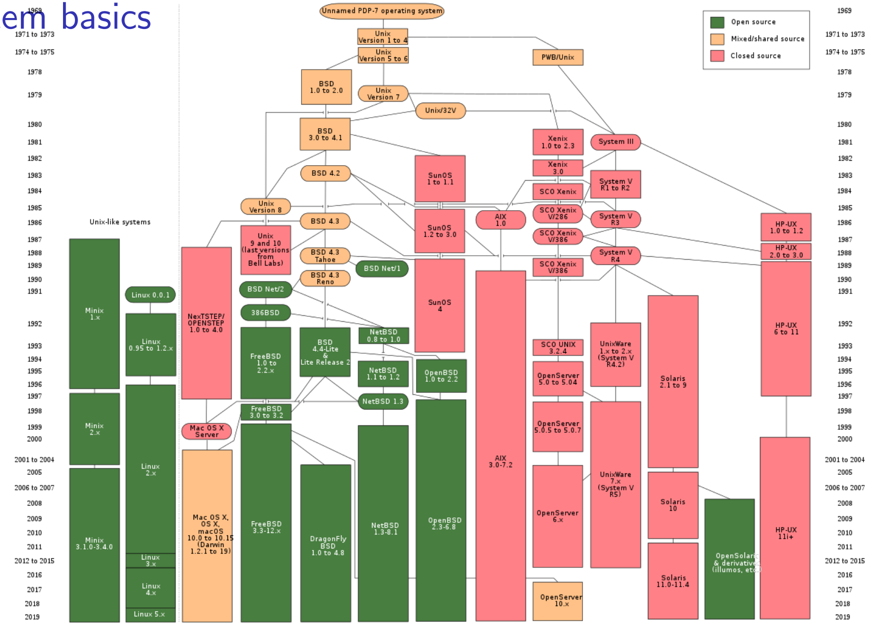
- What is GNU/Linux: operating system Unix clone aiming at POSIX compliance, **multiplatform** (for "better" POSIX compliance, check the BSDs).
- **Linux** is a kernel supporting free, opensource software tools developed in the framework of **GNU**).
- Various C libraries available with different memory footprints: glibc, uClibc, newlib ... and different functionalities.
- A distribution only wraps all these tools in packages.
- uClinux for systems with no MMU [1], Linux for systems with MMU, OpenWRT for routers ...
- other proprietary operating systems: ~~MS-Windows~~, macOS (BSD derivate) [2], LynxOS, QNX, vxWorks, iOS
- other opensource OS: *BSD (Free, Net, Open), Plan9, Inferno, Hurd
- Android: yet another layer (libraries, applications) provided by Google over Linux, executing in a Java Machine

[1] *Memory Management Unit*, a memory handler aimed at controlling access and converting virtual memory addresses to hardware physical addresses

[2] `http://www.bbc.com/news/technology-41551546`

Operating system basics

What is GNU/Linux: operating system Unix clone aiming at POSIX compliance, **multiplatform** (for "better" POSIX compliance, check the BSDs) [a]



---

[a] en.wikipedia.org/wiki/File:Unix_history-simple.svg

Development methods

OS on the target platform:

- Development cycle: the code is **validated on a PC** before being transfered to the **embedded platform** (OS common to both platforms).
- Compliance with POSIX system calls ⇒ code usable on any platform **as long as** hardware access is separated from software/algorithm (be aware of **endianness** issues)
- on two systems with **memory management units**, the code developed on PC is immediately usable
- lack of memory management unit ⇒ some functionalities might be missing and should be avoided (`fork`, `malloc`) or replaced
- NFS (*Network File System*) to quickly test applications by sharing a common storage medium between host and target.

```
mount -o nolock 192.168.2.1:/home/etudiant/nfs /mnt
```

- the target is hardly ever x86 ⇒ cross-compile
- `output/host/usr/bin/` for the toolchain (host = PC – set `$PATH` accordingly)

Reminder: libraries

Implementation of `libc` providing access to Linux system calls (orders from userspace to the kernel).

On the target: impact of an inconsistent toolchain

```
# ldd gpio_sleep
        libc.so.0 => /lib/libc.so.0 (0xb6f56000)
        ld-uClibc.so.0 => /lib/ld-uClibc.so.0 (0xb6fac000)
```

- Here, uClibc provides these functionalities
- Lacking the dynamically loaded libraries, a cryptic (at first) error message

  ```
  # ./gpio_sleep
  -sh: ./gpio_sleep: not found
  ```

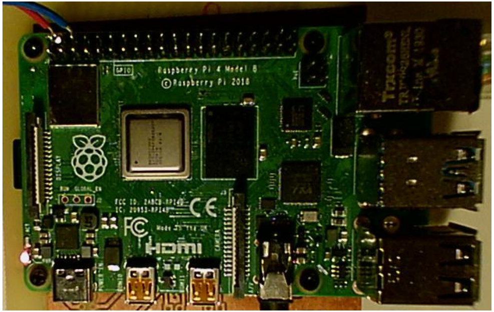
  here due to using the wrong compiler (missing dynamically loaded libraries) as shown by `strace`

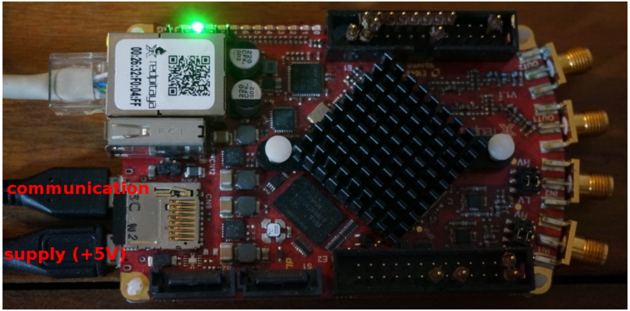
  ```
  # ldd gpio_sleep
  checking sub-depends for 'not found'
          libc.so.6 => not found (0x00000000)
          /lib/ld-linux.so.3 => /lib/ld-linux.so.3 (0x00000000)
  # ls -l /lib | grep ld-l
  #
  ```

Development environment

Complex development framework since it must provide a consistent set of tools for

- compiling the Linux kernel
- compiling the libraries needed for the userspace applications
- compiling applications themselves ("packages")
- compiling the *bootloader* using the platform configuration files
- using a compilation toolchain dedicated and optimized towards the targeted platform.





Redpitaya: Zynq processor (dual ARM Cortex A9 @ 800 MHz), OS and bootloader on SD card (+FPGA)
Raspberry Pi4: BCM2711 (quad ARM Cortex A72 @ 1.5 GHz), OS and bootloader on SD card

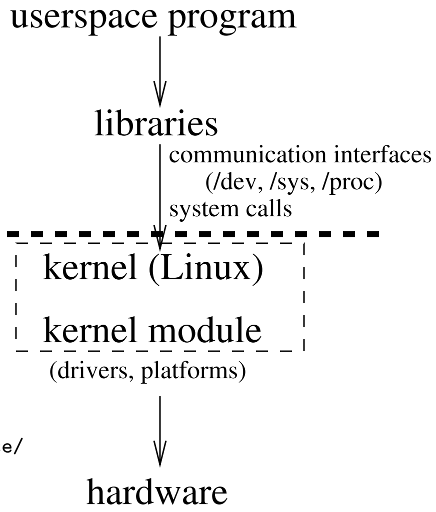
Objectives

A consistent framework [a] to generate

1. a cross compilation toolchain
2. a bootloader (uboot: initializes the CPU + loads the kernel)
3. operating system kernel (Linux)
4. rootfs (userspace applications and libraries)

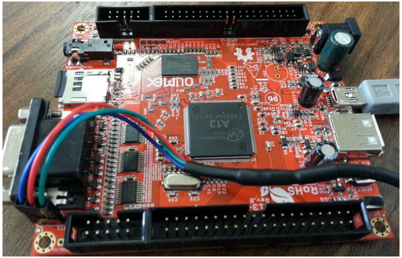
all stored in various partitions and directories of the SD card.
Most embedded boards are non-x86 based[3] ⇒ cross-compile programs from the host to the target

[a] Why not use a distribution? See F. Dolcini, *"Vanilla" Debian On An Industrial Embedded Device*, FOSDEM (2024) https://fosdem.org/2024/schedule/event/fosdem-2024-2572--vanilla-debian-on-an-industrial-embedded-device/



[3] http://iqjar.com/jar/an-overview-and-comparison-of-todays-single-board-micro-computers/

Development environments



Consistent framework to target many platforms:

1. OpenEmbedded
2. Yocto
3. **Buildroot**

For Olinuxino A13-micro (26 euros [4]) :

- https://github.com/trabucayre/buildroot
- 6 GB hard disk space for a 200+ MB image in output/images/a13_olinuxino.sdimg
- Check configs/ for supported platforms: a13_olinuxino_micro_defconfig so this target is configured [5] with: make a13_olinuxino_micro_defconfig && make

For Redpitaya:

- https://github.com/trabucayre/redpitaya.git complements the official Buildroot version (source sourceme.ggm to load the BR2_EXTERNAL variable):
- make redpitaya_defconfig && make

For Raspberry Pi4

- make raspberrypi4_64_defconfig && make

[4] https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware
[5] http://jmfriedt.free.fr/A13_v2.pdf

Buildroot

`make menuconfig`: configure Buildroot (userspace packages, toolchain)

`make linux-menuconfig` to configure the kernel (USB support e.g. – the kernel source code is found in `output/build/linux*` after compiling Buildroot)

Tree structure organization:

1. `configs/*defconfig`: Buildroot configurations (e.g `make raspberrypi4_64_defconfig`)
2. `board/raspberrypi4-64/`: board specific scripts
3. `output` holds all Buildroot compilation results
4. `output/host/bin` holds all files related to the (x86) host – toolchain and other tools (e.g. `dtc`)
5. `output/target/` holds all files related to the ARM target
6. `output/target/lib` holds the dynamically loaded libraries for the embedded target board
7. `output/build/linux-*`: Linux source code
8. `output/build/linux-*/arch/arm/boot`: compiled Linux kernel
9. `output/images/*img`: image to be transfered on the non-valatile storage medium (`dd`)

Buildroot (custom [6] [7] [8])

- Adding custom functionalities (board, packages): BR2_EXTERNAL
- See https://github.com/oscimp/oscimp_br2_external for GNU Radio 3.9 support, temporary packages prior to mainline support
- See https://github.com/trabucayre/redpitaya for new board support
- Handling new packages: add in packages https://buildroot.org/downloads/manual/manual.html#adding-packages

---

[6] G. Goavec-Merou, J.-M Friedt, *"On ne compile jamais sur la cible embarquée" : Buildroot propose GNU Radio sur Raspberry Pi (et autres)*, Hackable **37** (Avril-Mai-Juin 2021)

[7] G. Goavec-Merou, *GNURadio running on embedded boards: porting to buildroot*, GNU Radio Conference (2018) at https://pubs.gnuradio.org/index.php/grcon/article/view/86

[8] G. Goavec-Merou & J.-M Friedt, *Never compile on the target ! GNU Radio on embedded systems using Buildroot*, FOSDEM 2021, at https://archive.fosdem.org/2021/schedule/event/fsr_gnu_radio_on_embedded_using_buildroot/

Linux filesystem

Two partitions (sda1 and sda2) on the hardware non-volatile memory (sda)

- first small FAT filesystem including the bootloader, Linux kernel image and devicetree hardware configuration description
- large Unix filessystem (EXT) with rootfs including libraries and applications
    - /etc configuration files
    - /sbin system binaries
    - /bin unix binaries
    - /usr user specific files (/usr/bin and /usr/lib)
    - /root administrator home directory
    - /home users home directories
    - /proc pseudo-files representing the Linux kernel state and processes
    - /dev and /sys pseudo-files for communicating with hardware through kernel drivers

`dd if=output/images/sdcard.img of=/dev/sda bs=8M` to transfer the Buildroot generated image to the SD card (**will definitely erase the mass storage medium content**)

`minicom -D /dev/ttyUSB0` to access console (root/root)