Operációs Rendszerek

AMP Laboratórium

VIMIAB00

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with you using PREEMPT_RT."

Linus Torvalds

Bajczi Levente

AMP on Consumer SoCs

A System on a Chip (SoC) is a tightly integrated hardware element that contains most of the components necessary for the computer to function. Nowadays these incorporate mostly multi-core processors, as we will see during this laboratory exercise, where we will uncover the reasons why a general-purpose operating system is not the best idea for tasks where either exact timing or low and stable latencies are necessities.

Supported Hardware

Theoretically, any multicore Pi supports this version of setting up AMP. These are the following:

- Raspberry Pi 2
- Raspberry Pi 3 *****
- Raspberry Pi 3B+
- Raspberry Pi 3 Model A+

The other versions (The original Raspberry Pi, the RPi Zero and some compute models) have only **one** core, and therefore they are not suitable for this laboratory! An exception to this rule is the Raspberry Pi 4 – it won't be compatible with *all* the steps, due to the modifications the engineers made while developing it. You can follow along, but some steps won't be compatible – there will always be a note stating this incompatibility.
Furthermore, you will need to have access to suitable cables to connect the Pi's GPIO pins together. For example, dupont[1] cables can be used for this, but lacking that, a paperclip can be used as well. Only neighboring pins will be connected.
For all tasks it is expected for the user to have access to a micro SD card and an SD card reader.

[1] https://www.amazon.co.uk/Dupont-cable-color-1p-1p-connector/dp/B0116IZ0UO

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Necessary Knowledge

General purpose tools

- SSH - SimpleWiki
- SCP - Wikipedia
- Makefiles - Wikipedia
- Mount - Wikipedia

Specific to this lab exercise (read through these before doing the lab!):

- **AMP -** Wikipedia
- **PREEMPT_RT -** LinuxFoundationWiki
- **Linux Kernel Build System -** Linuxjournal
- **Linux Priorities -** IBM Tutorials

Useful commands

For some of these commands, *sudo* or being root is necessary.

- Make a file executable: `chmod +x <file>`
- Copy a file: `cp -r <source> <dest> # -r is for directories`
- Move a file: `mv <source> <dest>`
- Delete a file: `rm -rf <file(s)> # -r is for directories`
- Find absolute path: `echo $PWD/<relative path of file>`
- Copy a file to the pi: `scp <local-file> <user>@<ip>:<remote-file>`
  Log in to the pi using ssh: `ssh pi@<ip>`
- Launch a program: `./<program>`
  - Modify its 'nice' value: `nice -N ./<program> # for -20: nice -20`[2]
  - Modify its real-time priority: `chrt N ./<program>`[3]
- Extracting a *.tar.xz* file: `tar xvf <tar.xz file>`
- Check system information: `uname -a`
- List devices of a live system: `dtc -I fs /sys/firmware/devicetree/base`
- Print out the system log: `dmesg`
- Clear the system log: `dmesg -C`
- Flash an SD card with an image:
  `dd if=<in> of=<out> bs=4M status=progress conv=sync,noerror`
  - out: */dev/sdc* or something similar
- Stress the CPUs: `stress-ng --cpu N -t Xm # N threads for X minutes`
  - Add *--aggressive* to crank the stress up
  - Add *sudo* to the beginning to make it even more aggressive (might lock up!)

Terminal tips

- Normal crtl+c/v does not work in the terminal. Use ctrl+shift+c/v instead.
- Use *tab* once to complete the current command, twice to see the possibilities.

[2] https://askubuntu.com/a/656787/613336
[3] https://stackoverflow.com/a/52501811/4564311

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Discovering the Cross-Compilation Toolchains

The computer you will be working on most likely contains an X86 (Intel or AMD) processor. If you compile a program using the ordinary `gcc` compiler, the generated machine code will run on any other X86 processor (running the same operating system). For our purposes though, this will not suffice: the Pi contains an ARM CPU, and is therefore incompatible with code compiled for X86. Two solutions exist for this problem: one, you could write all code on the Pi itself, and compile *natively* – or you could use a so called *cross compiler*. This is a collection of binaries that are compiled for a host architecture (in our case, X86/Linux), and targets another (in our case, ARM).

Start the VM, log in (user: *meres*, pwd: *LaborImage*) and start a terminal. Check that the following commands are available:

- `arm-linux-gnueabihf-gcc`
- `arm-none-eabi-gcc`

If everything works correctly, both of these commands return an error message that no source files were provided. So provide them with one:

```
#include <stdio.h>
int main()
{
   printf("Hello World!\n");
   return 0;
}
```

Write this into a file (say *program1.c*) and try to compile it using both toolchains.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Compile an RT-patched kernel

1. Get the sources: `git clone https://github.com/raspberrypi/linux.git --depth 1 --branch rpi-4.19.y-rt`
2. Depending on the Pi's model, issue either:
   a. `make bcm2711_defconfig ARCH=arm CROSS_COMPILE= arm-linux-gnueabihf-gcc #RPi4`
   b. `make bcm2709_defconfig ARCH=arm CROSS_COMPILE= arm-linux-gnueabihf-gcc #other`
3. `make -j2 zImage modules dtbs ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-gcc`
   This will take a long time. Take a short break while it compiles, maybe read up on AMP systems[4] or the PREEMPT_RT patch[5]!
4. Mount the *root* partition of the image file to */mnt/rootfs!*
5. `make modules_install INSTALL_MOD_PATH=/mnt/rootfs ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-gcc`
6. Unmount */mnt/rootfs* and mount the *boot* partition of the image file to */mnt/boot!*
7. Delete the following files in */mnt/boot*: **.dtb, kernel*, overlays/**
8. `cp arch/arm/boot/zImage /mnt/boot/`
9. `cp arch/arm/boot/dts/<name>.dtb /mnt/boot/`
   The <name> possibilities are (choose your Pi's version):
   a. `bcm2711-rpi-4-b`
   b. `bcm2710-rpi-3-b-plus`
   c. `bcm2709-rpi-2-b`
10. `cp arch/arm/boot/dts/overlays/*.dtb* /mnt/boot/overlays/`

Put `kernel=zImage` at the end of /mnt/boot/config.txt

[4] https://en.wikipedia.org/wiki/Asymmetric_multiprocessing
[5] https://wiki.linuxfoundation.org/realtime/documentation/start

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Why did we need to disable the LED? Normally the activity LED (yellow LED on the board near the red power LED) displays the “heartbeat” of the system – meaning it is constantly switched on and off based on the load the system is under. However, if you wanted to use it for something else (in our case, the examples are blinking the LED, which is **only** visible if this heartbeat display is disabled), you need to take the control away from the kernel.

To finish up preparing the images, we will need to configure the most special one – *amp.img*. The default bootloader (which runs on the GPU of the Pi) does not respect all kernel parameters that are passed to it. Therefore, we need to insert another 2nd stage bootloader into the boot sequence, which is called *Das U-Boot*[6]. **This is not compatible with the RPi 4! Skip this step.**

1. Clone the repo: `git clone git://git.denx.de/u-boot.git`
2. Configure: `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-gcc rpi_2_defconfig`
3. Build: `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-gcc u-boot.bin -j2`
4. Mount the *boot* partition of the image being modified to */mnt/boot*
5. Copy the created *u-boot.bin* file to the boot partition
6. Create a file called *boot.cmd* in the boot partition containing the following code:

```
setenv bootargs earlyprintk console=tty0 console=ttyAM0
   root=/dev/mmcblk0p2 rootfstype=ext4 rootwait noinitrd
fatload mmc 0:1 ${fdt_addr_r} bcm2710-rpi-3-b.dtb #your version
fatload mmc 0:1 ${kernel_addr_r} zImage
bootz ${kernel_addr_r} - ${fdt_addr_r}
```

7. Issue `mkimage -C none -A arm -T script -d boot.cmd boot.scr` to compile *boot.cmd* to the u-boot file *boot.scr*.
8. Verify that a *boot.scr* has been created in /mnt/boot
9. Add the following line to *config.txt*: `kernel=u-boot.bin`

If done well, this process has inserted the U-Boot bootloader into the boot process seamlessly, meaning that the same kernel works as before but with an additional step when booting.

To achieve our goal of not having all the cores under linux’s scheduler, we have to configure the kernel parameters the bootloader passes to the booting kernel. This is specified in the *bootargs* environment variable in the *boot.cmd* file. A (somewhat complete) list of parameters can be found in the kernel’s documentation[7].

[6] https://www.denx.de/wiki/U-Boot

[7] https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Mapping Physical Addresses to Virtual Addresses

Even though while running in kernel mode, the I/O is reachable as a simple I/O instruction, user mode is not so lucky. Instead of that, the use of the *mmap* function is necessary. The arguments to be passed are the following:

1. Addr - NULL, because we don't care where it will end up. The kernel knows better.
2. Prot - `PROT_WRITE`
3. Flags - `MAP_SHARED`
4. Fd - Returned file descriptor of the file "**/dev/gpiomem**" using the open system call.
5. Offset - 0, because the file above is aligned to the start of the GPIO block
6. Size - calculate it using the value of the GPIO register at the highest address

Running Bare-Metal Programs on the Pi

Bare-metal in this context means that there is no operating system we could rely on while developing the program we are trying to run. This means that anything we want to achieve has to be done completely from scratch, there is no abstraction of the hardware we could use. As already uncovered in Section 1., not even *printf* has a default implementation - the compiler does not know *how* it could give us an output.

Another difference arises from the memory usage - unless we explicitly say otherwise, the program has direct access to the entirety of the *physical* addresses. This makes it easier to develop applications, but at the same time, makes it a security vulnerability to do so. In our use-case, the cores running Linux will see 512MBytes of the RAM and the rest (however large it is) will belong to the bare-metal program. It could, however, potentially see and even modify the RAM contents of the processes running on top of Linux.

Furthermore, the core is *uninitialized* the first time we make it jump to the program's address, and therefore we need to set up a few things before we could un anything else. To do so, we will need the following file structure:

- **Makefile** - to facilitate compilation
- **main.c** - contains our program's code
- **start.S** - contains necessary code to bootstrap the CPU and provide basic LCM[8]

Starting a Core on the Raspberry Pi

Normally, the boot process takes care of starting all cores a given system has, but in this case, we have specifically made it so that it only has access to 3 of the 4 available cores. We *could* start it from under Linux, but then it would be assigned tasks from the scheduler, and we want to refrain from that. The easiest way of figuring out how to wake up a core is to look at the kernel's source and see how it does it[9]. In this case, the cores are in a *Wait-for-Instruction* (WFI) loop, meaning they are monitoring a register which will contain a memory location to jump to. In the case of the Raspberry Pi, this register is the *Mailbox* of the core, more specifically the 3rd one.

[8] Life Control Management - Describing what should happen before/after the program's end/start
[9] https://github.com/raspberrypi/linux/blob/rpi-4.1.y/arch/arm/mach-bcm2709/bcm2709.c#L1248

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Linux Kernel Modules

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.
In the scope of this project we will create two kernel modules, as Tasks #2 and #8 outlined.
The Linux Kernel Build System[10] makes the development and compilation of a kernel module very convenient, but we must abide by a few rules:

- The running kernel's source (more specifically, the headers) must be used for compiling the module
- To build the module (from the same directory as the source.c):

```
make obj-m+=<source.c without .c >.o ARCH=arm \
CROSS_COMPILE=$CROSS_LINUX -C <linux-source-root> M=$PWD modules
```

- To clean the project (from the same directory):

```
make -C <linux-source-root> M=$PWD modules
```

This can easily be incorporated into a Makefile of its own:

```
obj-m += kernel-module.o
all:
   make ARCH=arm -C ../../linux M=$(PWD) modules
clean:
   make ARCH=arm -C ../../linux M=$(PWD) clean
```

The generic outline of any kernel module (my_module.c) is the following:

```
#include <linux/module.h> // Necessary include for kernel modules
MODULE_LICENSE("GPL");    // Could be any other license as well
MODULE_AUTHOR("Author Name");         // Your name
MODULE_DESCRIPTION("Short description"); // Concise summary
MODULE_VERSION("0.01");               // Version number
static int __init lkm_example_init(void) // This will run when loaded
{ printk(KERN_INFO "Hello World!"); return 0; }
static void __exit lkm_example_exit(void)// This will run when unloaded
{ printk(KERN_INFO "Goodbye World!"); }
module_init(lkm_example_init); // To make the correct function run
module_exit(lkm_example_exit); // To make the correct function run
```

After compiling the module, a *.ko* will appear in the directory of the source, along with a plethora of other (for us) uninteresting files. After copying it to the device, you can load and unload it with *insmod <file>.ko* and *rmmod <file>.ko*.

[10] https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Kthreads

In the previous section, the basic skeleton of a kernel module was introduced. It did something when it was loaded and something else when it was unloaded - but what if some long-running functionality was necessary? In our case, this is the situation, as we want to respond to GPIO signals right away *at any time* - meaning it should not be restricted to the time we load it. For this purpose, two solution exist:

1. Making the long-running functionality time-constrained, i.e. limiting how long it should be allowed to run. Then it can be invoked from the init function.
2. Creating a long-running background thread that is allowed to exist until we unload the module.

Option #2 is a lot more elegant, and therefore we will explore that one.
As we are running in kernel mode, there is no way to just simply invoke a *fork* syscall and use it to run a function asynchronously. Instead, since kernel version 2.6, there are new primitives to be used for this purpose (until then, this functionality was always implemented in a custom way), called kthreads, which are lightweight processes running in kernel mode.
Its use is fairly simple and well detailed in the source (https://lwn.net/Articles/65178/).

Character Devices[11]

So far, we have seen how kernel modules are built - but how can we actually use them, besides loading and unloading them? The UNIX philosophy (and by extension, that of Linux) says that *everything is a file*[12] - and this is how most device drivers are built. For example, there are files under */dev/* representing physical disks such as HDDs and SSDs (e.g. /dev/sda) and virtual terminals (e.g. /dev/tty1). In this example, the former is called a *block special* and the latter a *character special* file (confirm these by issuing *file /dev/{sda,tty1}*).
To achieve the goals of our loader, we are going to use a *character special* file which will provide the following services:

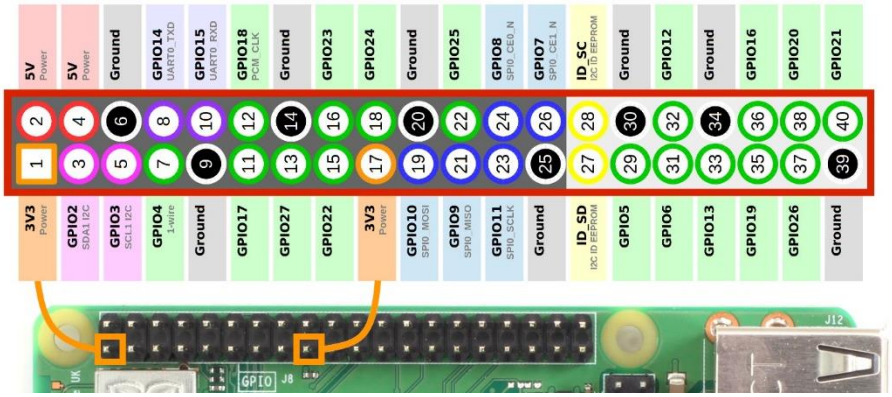
- When being read, it prints a message to the kernel log stating it is write only
- When being written to, it treats it as a program to be run - if the last program has terminated and is currently waiting for the next, the module will load the new program into memory and then make the core jump to it.

Linux provides us with a well-build toolset to deal with character devices[13].

[11] https://linux-kernel-labs.github.io/master/labs/device_drivers.html
[12] https://en.wikipedia.org/wiki/Everything_is_a_file
[13] http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Raspberry Pi pinout



Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.