Embedded systems architectures and Linux devicetrees

Contents

Chapter 1

Introduction

1.1 What is an embedded system?

Generally speaking, an **embedded system** is an information-processing element designed to accomplish one or several tasks, depending on its complexity. What sets it apart from a consumer-grade computer, for example, is the range of applications and the constraints that come with it. Typically, embedded systems are seen in smartwatches, smartphones, IoT, and automotive[1], where constraints such as size, cost, power consumption, and security[2] can highly influence the system's design. On the other hand, consumer-grade computers are more general-purpose, therefore their constraints will most likely end up being more relaxed, thus allowing more freedom in their design choices.

1.2 Terminology - processor, MCU, SoC, and SoM

Before going further, it's important to understand the meaning of some terms that are commonly used in the embedded systems area.

The **processor** is the element that performs various operations such as basic logic or arithmetic operations based on the instructions that are being fed into it. To work, it needs to be connected to one or more memories (so that it can fetch instructions and process data) and to I/O blocks (so that it can interact with a user).

Figure 1.1 shows the microarchitecture of a processor. Note that the diagram also includes the instruction and data memories which are not part of the processor itself. Among the components we can identify:

- The PC (program counter) register. This holds the value of the program counter, which is used to address the instruction memory.
- The register file. This is a (sort of) memory used to store the contents of the processor registers. Usually, this will have a lower access time compared to the data/instruction memory.[3]
- The ALU (Arithmetic Logic Unit). This is the "heart" of the processor and the component that performs the arithmetic and logic operations.

[1] List is not exhaustive.
[2] List is not exhaustive.
[3] https://intra.ece.ucr.edu/ stan/courses/ee120a/ee120a_10fall/labs/Lab_7_reg_file_design.pdf provides a nice table with a few differences between a register file and a SRAM memory.

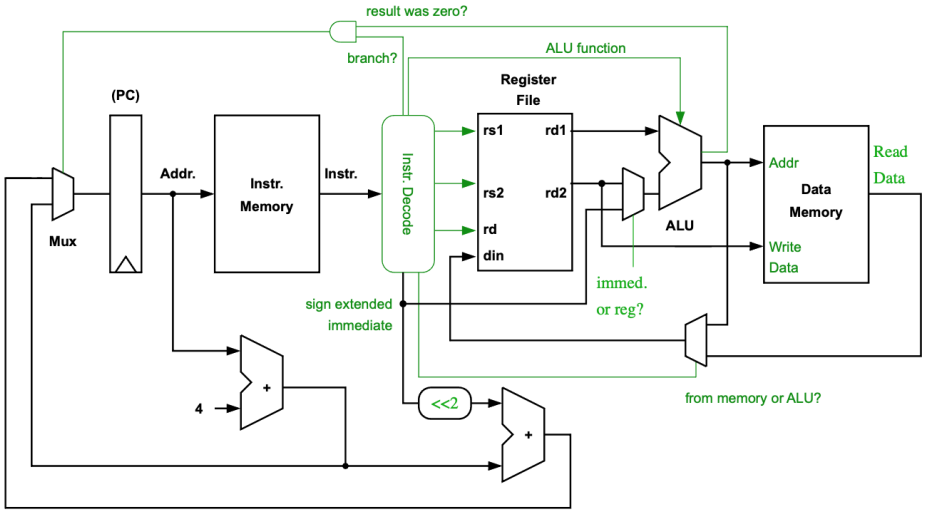

Figure 1.1: Microarchitecture of a generic processor

Source: ARM University Program. Introduction to Computer Architecture

An **MCU (microcontroller unit)** can be seen as a small computer that contains the processor, some memory and basic I/O blocks, all of which is integrated in a single chip.

Figure 1.2 shows an example of a microcontroller. Of course, the complexity of a microcontroller's architecture can vary, ranging from relatively simple architectures such as PIC's to more complex ones such as NXP's LPC840[4].

The **SoC (System-on-Chip)** can be thought of as a **significantly** more complex MCU. This can contain one or more processors, accelerators (for instance, GPUs, FPGAs, DSPs), specialized I/O blocks etc...All of this is integrated in a single chip. Thanks to its complexity, a SoC is usually able to run some sort of rich-OS like Linux.

In contrast, a **SoM (System-on-Module)** is made up of multiple chips, all of which being integrated on a PCB. Optionally, one of these chips can be an SoC.

Figure 1.3 shows the top and bottom views of the PICO-IMX8M SoM. The components are as follows:

1. i.MX8MQ SoC
2. Memory (LPDDR4)
3. PMIC (Power Management IC)
4. Wi-FI/BT module
5. eMMC IC
6. Connector E1
7. Connector X1
8. Connector X2

[4]https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc800-arm-cortex-m0-plus-/lpc840-32-bit-arm-cortex-m0-plus-based-low-cost-mcu:LPC84X

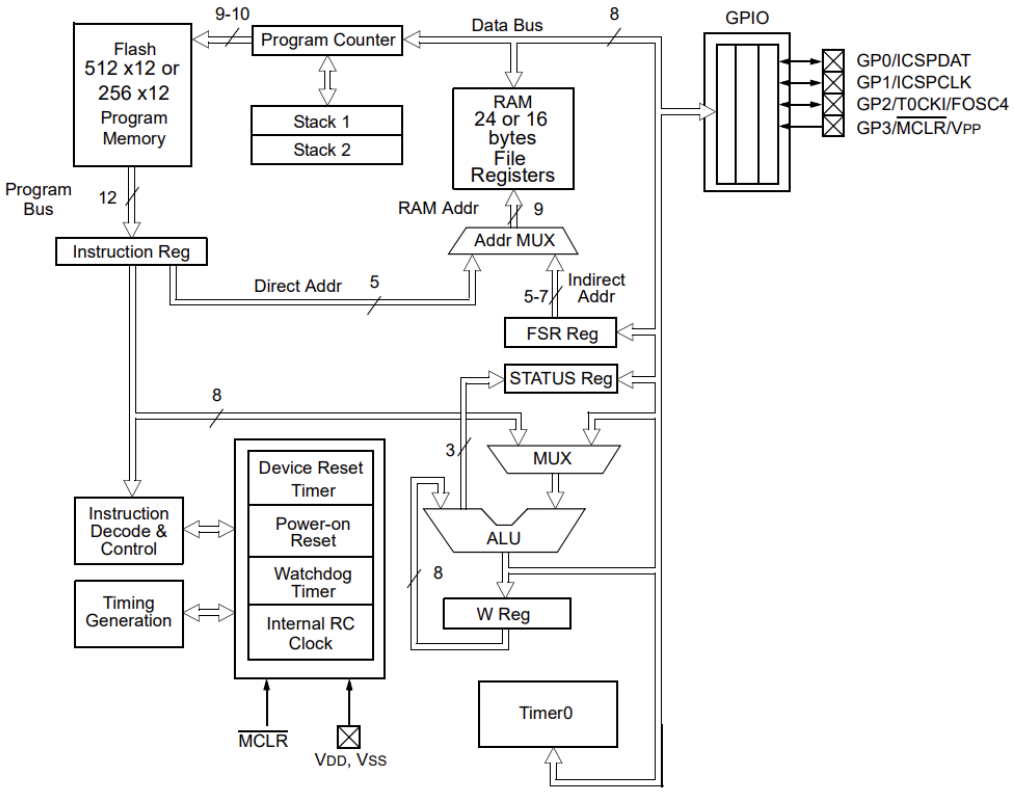

Figure 1.2: Architecture of Microchip's PIC MCU

Source: PIC10F200/202/204/206 datasheet

Figure 4 – PICO-IMX8M Top View

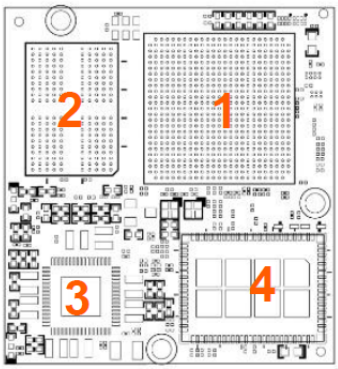

Figure 5 – PICO-IMX8M Bottom View

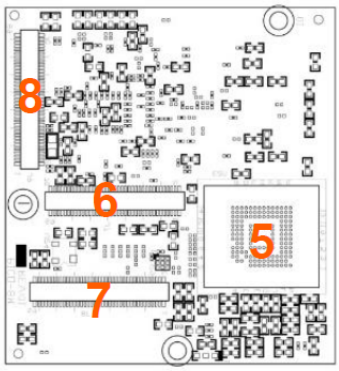

Figure 1.3: Top and bottom view of PICO-IMX8M

Source: PICO-IMX8M System on Module product manual

Chapter 2

ARM-based SoC architecture

Chapter 1 established what an SoC is and gave some examples of components that may be present in it. Of course, this might not be enough for an embedded engineer who needs to have a somewhat good understanding of the SoC's architecture to be able to write software for it. As such, this chapter shall go a bit more into detail regarding the SoC's architecture.

Figure 2.1 proposes a generic architecture of an SoC. As it can be seen, there are four components:

- The masters (from M0 to M4).
- The slaves (from S1 to S5).
- The interconnects.
- The bridge.

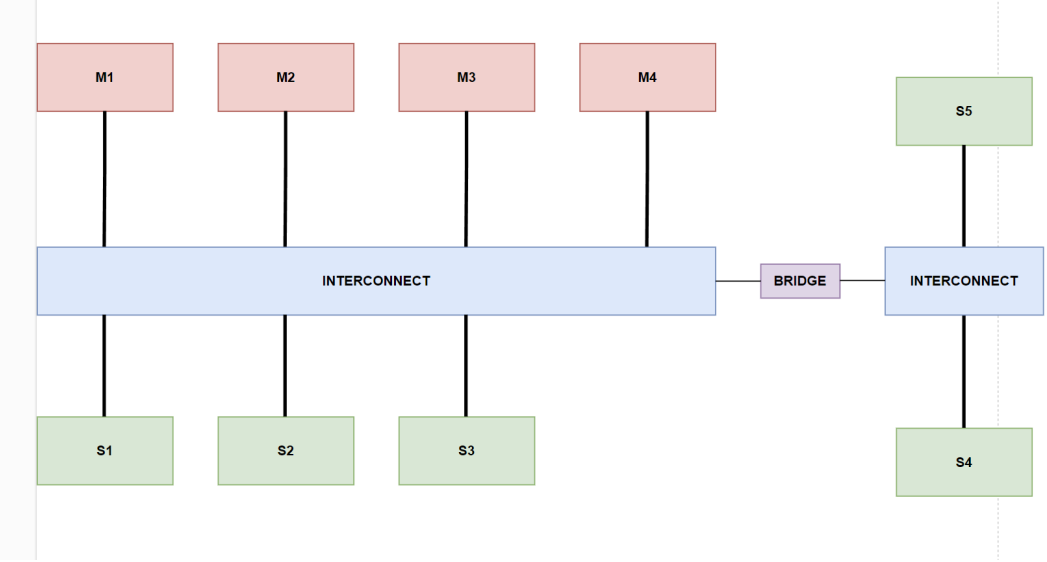

Figure 2.1: Generic architecture of an SoC

2.1 Masters and slaves

Usually, in an ARM-based SoC, components communicate with each other for different purposes. For instance, a processor might want to communicate with a memory so that it can read or write some data to it. Another typical example involves a processor communicating with an UART module in order to send some data over the serial interface.

In this communication, one of the two components involved will be the initiator (i.e: the one that starts the "conversation"), while the other will simply "reply" and/or comply with the initiator's request (note: the entity that replies may "refuse" the initiator's request based on several factors). In this model, the initiator is called a **master**, while the other component is called a **slave**. Examples of master components include: processors, accelerators, DMA (Direct Memory Access) engines, etc..., while examples of slave components include: memories, peripherals, etc...

Please note that masters cannot communicate with each other directly[1].

2.2 Interconnects

Normally, if there were only two components included in a SoC, we would be able to just connect them directly via a wire. This can be seen in figure 2.2.

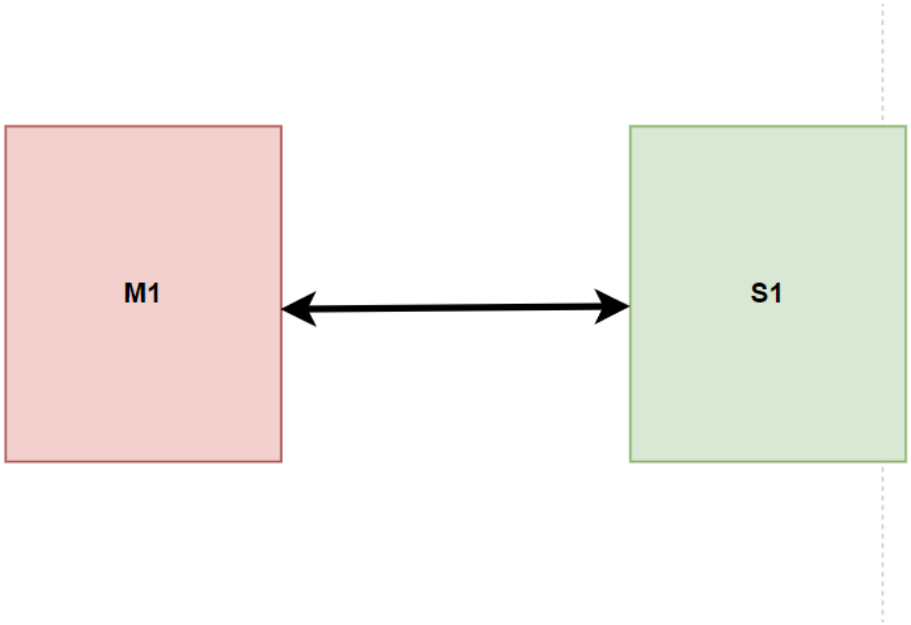

Figure 2.2: Wiring of two components

Since the SoC is made up of multiple components, we need to allow a way for them to "directly" communicate with each other. This is where an **interconnect** comes into play. The main purpose of the interconnect is to allow one component to communicate with another in a system that contains more than two components.

Although not entirely accurate, you can think of an interconnect as a network switch. All it does is forward data from one component to another. For instance, if M1 wishes to communicate with S2, it will send the data to the interconnect and the interconnect will send the data directly to S2. The same is applicable to S2 if it wishes to reply to M1. Note that multiple masters

[1] Unless they also have a slave interface alongside the master interface.

and slaves may communicate with each other at the same time depending on the architecture of the interconnect (i.e: if it allows this).

Figure 2.3 shows the communication between M1 and S2 (denoted by the red lines) happening at the same time as the one between M4 and S3 (denoted by the yellow lines).

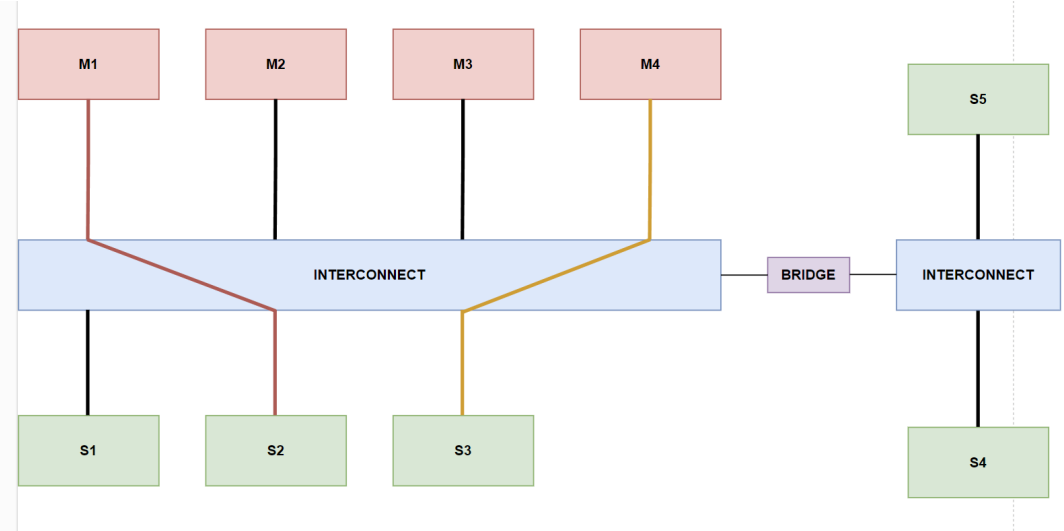

Figure 2.3: Communication through an interconnect

2.3 Communication between components

For the communication to happen, the initiator and the interconnect need to know who the recipient is. Intuitively, this is the same as two computers trying to communicate over the network. For this to work, the computer sending data needs to know who to send it to. This is where addresses come into play. In our case, the master will use the slave's address to communicate with it. Based on the same address, the interconnect will know who the recipient is.

Unlike IP or MAC addresses which will most likely be unique, the slaves have a range of addresses that's assigned to them.

Figure 2.4 shows an example of how addresses may be assigned to the slave components from figure 2.1. In this scenario, S1 is assigned the addresses from 0x0 to 0x1000, S2 is assigned the addresses from 0x2000 to 0x3000 and so on. What this means is that if a master wishes to communicate with S1 for instance, it will use an address from the 0x0 to 0x1000 space.

Usually, the addresses in a slave's address space have a special meaning and a register behind it. Figure 2.5 shows an example of this. Assuming the same address space as S2, this means that at address 0x2000 we have the **VERID** register, at address 0x2004 the **PARAM** register and so on.

Let's use an example to further understand the idea of address spaces. Imagine M1 wishes to communicate with S2. This tells us that M1 will have to use an address from the 0x2000 - 0x3000 address space (since this is the space that's assigned to S2). Specifically, M1 wishes to read from the **PARAM** register and write some value to the **TCSR** register. First, we need to identify the start of the address space and the offsets of the registers. In our case, the start of the address space is 0x2000. According to figure 2.5, the offset of **PARAM** is 0x4 and the offset of **TCSR** is 0x8. As such, to read from **PARAM**, M1 will have to use address 0x2004 (computed as address space base address + the offset of the register) and address 0x200C.

ADDRESS SPACE ASSIGNMENT	
SLAVE NAME	**ADDRESS RANGE**
S1	0x0 - 0x1000
S2	0x2000 - 0x3000
S3	0x4000 - 0x5000
S4	0x6000 - 0x7000
S5	0x8000 - 0x9000

Figure 2.4: Address space assignment

Offset (hex)	Register	Width (In bits)	Access	Reset value (hex)
0	Version ID (VERID)	32	R	0302_0002
4	Parameter (PARAM)	32	R	See section
8	Transmit Control (TCSR)	32	RW	0000_0000
C	Transmit Configuration 1 (TCR1)	32	RW	0000_0000
10	Transmit Configuration 2 (TCR2)	32	RW	0000_0000
14	Transmit Configuration 3 (TCR3)	32	RW	0000_0000

Figure 2.5: Registers from a slave's address space

Source: i.MX93 Technical Reference Manual

In figure 2.6, M1 tries to read from 0x2004. The interconnect will know to send the request to S2 based on the fact that it belongs to S2's address space (0x2000-0x3000). In figure 2.7 we can see that S2 replies with an OK and sends M1 the data from the **PARAM** register (which is **0xdeadbeef**).

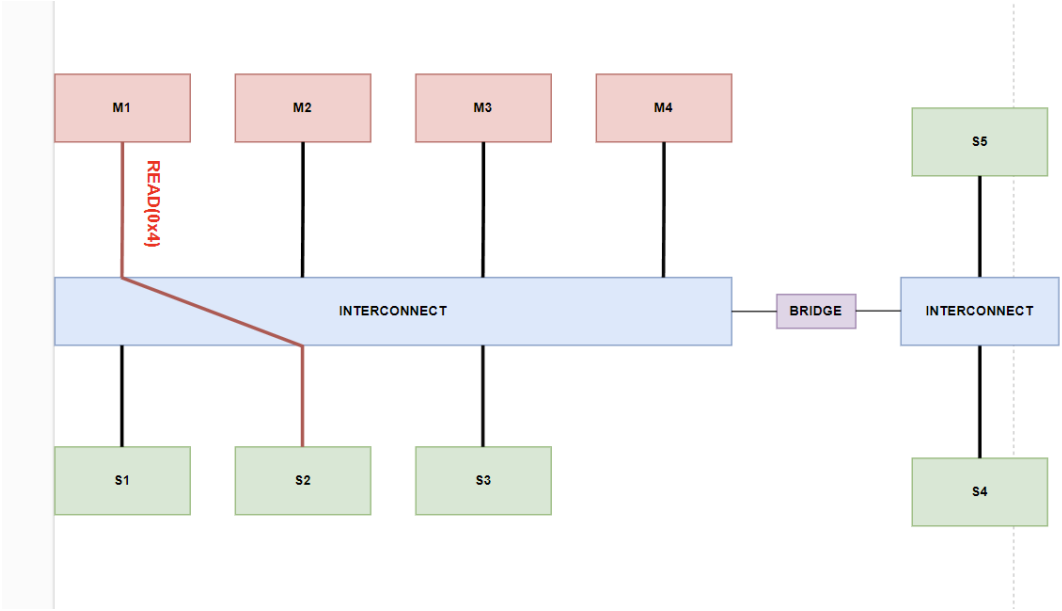

Figure 2.6: Sending a read request to S2

Figure 2.8 shows a snippet of a SoC's memory map.

Another aspect of the communication between two components is the **protocol**. Roughly, this tells the components involved in the communication when to send the data and how to encode it. Most commonly, ARM-based SoCs will use one or more protocols from ARM's AMBA (Advanced Microcontroller Bus Architecture)[1] specification (e.g: AXI4-Lite, APB, etc...) but the SoC vendor may also choose to use their own protocol. It's important to note that multiple protocols may be used in the same SoC. For instance, M1 can use AXI4-Lite, while S4 may use APB.

[1] https://developer.arm.com/Architectures/AMBA

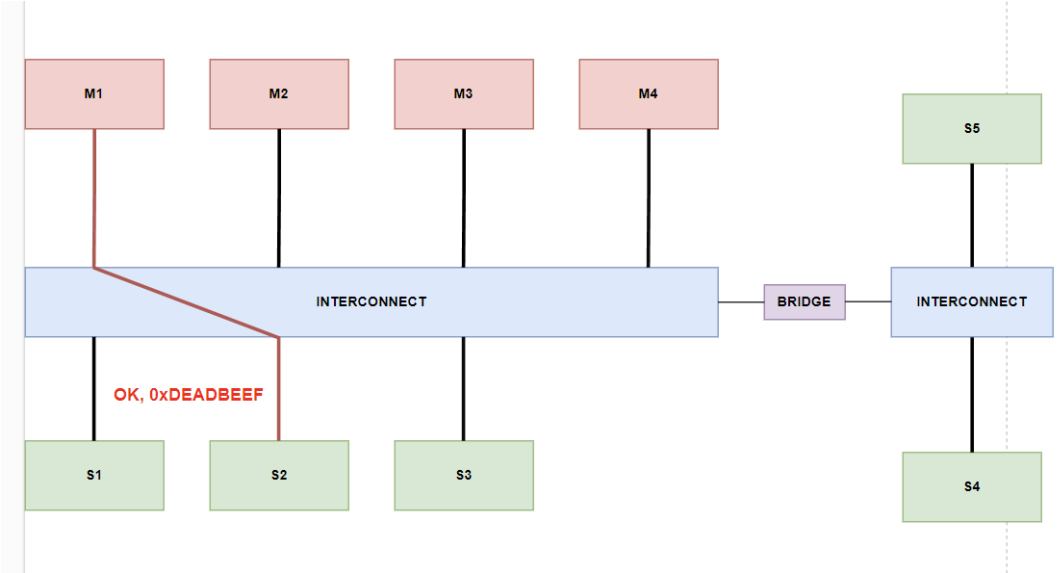

Figure 2.7: Receiving the reply from S2

Base	Slot	Module	Alias
DBG Modules			
2808_0000	0	Debug Access Port ROM 0	DAPROM
2808_1000	1	CoreSight Funnel 0	CS-FNL
2808_2000	2	CoreSight Embedded Trace FIFO 0	CS-ETF
2808_3000	3	CoreSight Trace Port Interface Unit 0	CS-TPIU
2808_4000	4	CoreSight Embedded Trace Router 0	CS-ETR
2808_5000	5	CoreSight Cross Trigger Interface 0	CS-CTI0
2808_6000	6	CoreSight Single Wire Output 0	CS-SWO
2808_7000	7	CoreSight Timestamp Generator 0	CS-TSGEN
CORE Modules			
2808_8000	8	Cortex-A35 Debug ROM 0	A35-DBGROM
2808_9000	9	Cortex-A35 CPU Debug 0	A35-DBG0
2808_A000	10	Cortex-A35 Performance Monitoring Unit 0	A35-PMU0
2808_B000	11	Cortex-A35 Cross Trigger Interface 0	A35-CTI0
2808_C000	12	Cortex-A35 Embedded Trace Module 0	A35-ETM0
2808_D000	13	Cortex-A35 CPU Debug 1	A35-DBG1
2808_E000	14	Cortex-A35 Performance Monitoring Unit 1	A35-PMU1
2808_F000	15	Cortex-A35 Cross Trigger Interface 1	A35-CTI1
2809_0000	16	Cortex-A35 Embedded Trace Module 1	A35-ETM1

Figure 2.8: Snippet from i.MX8ULP's memory map

Source: i.MX8ULP Technical Reference Manual

2.4 The bridge

For this to work, the SoC may need a bridge. What this does (amongst other things) is it converts a protocol to another. This is needed because in order to understand each other, a master and a slave need to use the same protocol. This is exactly the same for humans which need to use the same language to understand each other. As such, assuming the first interconnect only uses AXI4-Lite, the bridge will convert AXI4-Lite-encoded data coming from the left interconnect to APB. What the second interconnect does is it takes the APB-encoded data and "forwards" it to one of the slaves it's connected to. Note that the bridge will also do the conversion from APB to AXI4-Lite.

Chapter 3

Introduction to devicetrees

3.1 What is a devicetree?

The **devicetree** is a set of nodes arranged in a tree-like structure that describes the hardware of a system. Each node has exactly one parent (except for the root node, which has no parent) and contains a set of properties that describe the characteristics of the device represented by it[1][2]. Figure 3.1 shows an example of how components of a system may be organized using the devicetree structure.

There are a couple of notes to be made here regarding figure 3.1:

- The root node is depicted as */*.
- The *CPUS* node refers to the CPU cluster present in the system. Its children are the CPU cores that make up the cluster. Here, the cluster contains a single core depicted as *CPU_0*.
- The *MEMORY* node refers to the system's RAM.
- The *SOC* node can be viewed as a bus that's connected to the CPU cluster. Usually, in Linux, this is the node in which peripherals will be placed. In our case, this "bus" (or virtual bus) has two devices: *DEVICE 1* and *DEVICE 2*.

3.2 Devicetree source (DTS) and devicetree blob (DTB)

As mentioned in section 3.1, the main purpose of the devicetree is to describe hardware. This is needed by software (e.g: an OS) running on platforms for which the hardware components are not dynamically discoverable (like it's the case for PCI-based systems). As such, we need to represent the devicetree in two formats:

- One format that's human-readable, used by developers that wish to describe the hardware of their platforms. This is known as the **devicetree source** format.
- Another format which is recognizable by the software. This is known as the **devicetree blob** or **flattened devicetree (FDT)**. The naming comes from the fact that the devicetree is encoded as a binary linear data structure instead of the tree-like structure mentioned in 3.1.

[1] https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.4
[2] The node may not necessarily represent a hardware device but ideally it should.

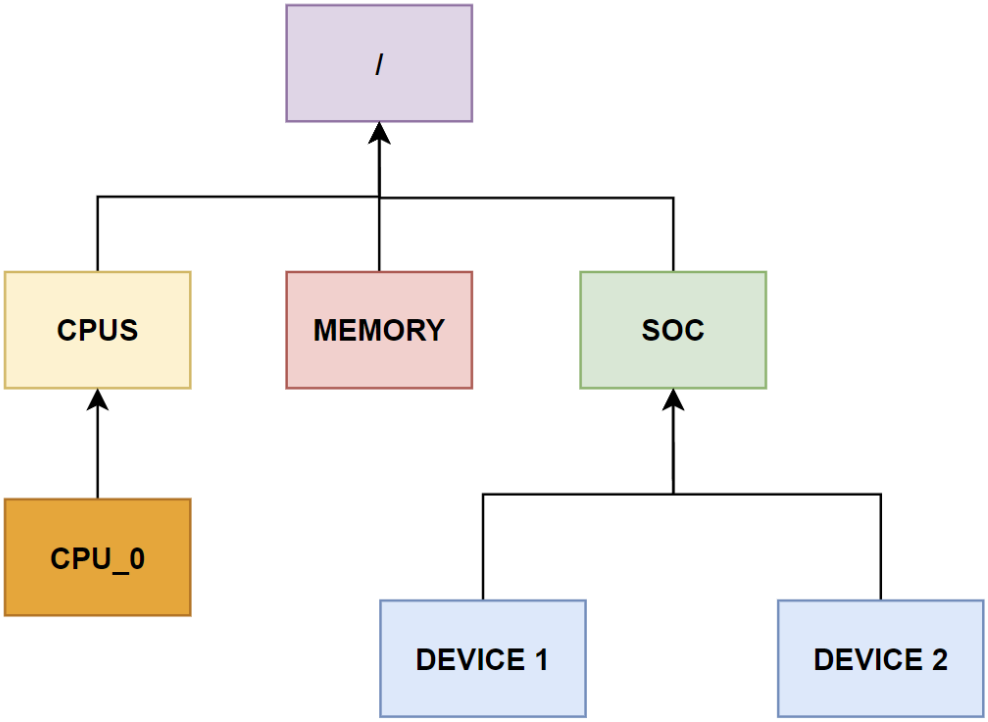

Figure 3.1: Generic devicetree example

The entity that does the conversion between a DTS and a DTB is called the **devicetree compiler (DTC)**.

Before attempting to write our own DTS, we need to talk about a few of the most important properties.

3.2.1 The *#address-cells* and *#size-cells* properties

These properties have the following format

```
#size-cells = <u32>;
#address-cells = <u32>;
```

where *u32* is an unsigned 32-bit integer (big-endian).

They are used to help the software (e.g: an OS) interpret the value of the *reg* property. It's important to note that:

- If missing, the values that should be assumed by the software are:

  ```
  #size-cells = <1>;
  #address-cells = <2>;
  ```

- These properties should be specified for each node with children. This is because they are not inherited from ancestors (i.e: grandparent and so on).

3.2.2 The *reg* property

The *reg* property is used to specify the address space of a node. This is an array of *<u32>* values (big-endian) with the following format:

```
reg = <value1 value2 ... valueN>;
```

The meaning of these values is described via the *#address-cells* and *#size-cells* properties. First N values from *reg* shall be used as the base address, while the next M values shall be used as the size of the address space. It is assumed that

```
#address-cells = <N>;
#size-cells = <M>;
```

Let's take the following snippet as an example:

```
#size-cells = <1>;
#address-cells = <1>;
reg = <0x2000 0x1000>;
```

Here, the address space is 0x2000 - 0x3000 (base is 0x2000 and the size is 0x1000).

Alternatively, according to:

```
#address-cells = <2>;
#size-cells = <1>;
reg = <0x0 0x80000000 0x1000>;
```

The address space is 0x80000000 - 0x80001000. The base address here is obtained by concatenating the first two cells from the *reg* property.

3.2.3 The *compatible* property

This property has the following format:

```
compatible = "string_1", "string_2", ..., "string_N";
```

where *string_ i* should ideally be of the form:

```
string_i = manufacturer,model
```

The *manufacturer* bit is usually some sort of identifier for the manufacturer. For instance, NXP uses *fsl* or *nxp* (lately).

If the *compatible* property has multiple string values, these should be sorted from the most specific device to the most generic one. In such cases, this can be interpreted as the device being compatible with other, similar devices.

3.2.4 The *status* property

The format of this property is:

```
status = "string";
```

where *string* can have one of the following values (not exhaustive):

- okay - device is operational
- disabled - the device is currently not operational (or it shouldn't be used)

3.2.5 The format of a DTS node

A devicetree node has the following format

```
[node-label]: node-name[@unit-address] {
    [node-properties-go-here]
 };
```

where

- *node-label* denotes the node's label. This is useful when we want to reference a node. For instance:

  ```
  my-property = <&label-of-some-other-node>;
  ```

- *node-name* denotes the node's name. Ideally this should be generic. For instance: *interrupt-controller*, *serial*, *ethernet*, etc...
- *unit-address* usually represents the base address of the device. It needs to match the address from the *reg* property.

All fields encased in the square brackets are optional. Every node from the devicetree source needs to use this format, the only exception being the root (/) node which has the following format:

```
/ {
    [node-properties-go-here]
 };
```

3.2.6 Translating to DTS format

All that being said, assuming the address space from figure 3.2, figure 3.3 shows how the devicetree presented in figure 3.1 can be translated to the DTS format.

ADDRESS SPACE ASSIGNMENT	
SLAVE NAME	**ADDRESS RANGE**
MEMORY	0xdead0000 - 0xdead1000
DEVICE 1	0x0 - 0x1000
DEVICE 2	0x1000 - 0x2000

Figure 3.2: Address space assignment of components depicted in figure 3.1

There are a few things worth mentioning regarding figure 3.1:

- The *device_type* property has been deprecated so you should generally avoid using it. The only exception to this are the *memory* and the *cpu* nodes. For these, the property is **required** and should have the value **cpu** for the *cpu* nodes and **memory** for the *memory* nodes.
- The *ranges* property is used to provide a translation between the bus address space and the bus parent's address space. In this case, what the property says is: *For a region of size 0x2000, bus child address 0x0 can be translated to bus parent address 0x0.*. For now though, we should ignore this property which was added for the sake of completeness.
- The *simple-bus* compatible value from the *soc* node is used to tell Linux that it should create devices for the child nodes of *soc*.

```
/dts-v1/;

/ {
        #address-cells = <2>;
        #size-cells = <2>;

        cpus {
                #address-cells = <1>;
                #size-cells = <0>;

                cpu_0: cpu@0 {
                        device_type = "cpu";
                        compatible = "vendor,cpu-name";
                        reg = <0>;
                };
        };

        /* RAM - spanning from 0xdead0000 to 0xdead1000 */
        memory@dead0000 {
                device_type = "memory";
                reg = <0x0 0xdead0000 0x0 0x1000>;
        };

        soc@0 {
                compatible = "simple-bus";
                #address-cells = <1>;
                #size-cells = <1>;
                ranges = <0x0 0x0 0x0 0x2000>;

                /* spans from 0x0 to 0x1000 */
                device1: device1-functionality@0 {
                        compatible = "manufacturer1,model1";
                        reg = <0x0 0x1000>;
                        status = "okay";
                };

                /* spans from 0x1000 to 0x2000 */
                device2: device2-functionality@1000 {
                        compatible = "manufacturer2,model2";
                        reg = <0x1000 0x1000>;
                        status = "disabled";
                };
        };
};
```

Figure 3.3: DTS translation for figure 3.1

- The */dts-v1/;* statement tells DTC which version of DTS you're using. If the statement is not present, DTC will assume the DTS uses version 0, which is obsolete.

Chapter 4

Linux kernel devices and drivers

The term **driver** is used to refer to a piece of software that's responsible for managing a resource (usually a hardware component). In Linux, this resource is known as a **device**. Usually, the relationship between drivers and devices is as follows:

- A device can have at most one driver.
- A driver can control up to N devices.

Figure 4.1 exemplifies this relationship. As you can see, **DEVICE_1** and **DEVICE_2** are assigned to **DRIVER_1**, **DEVICE_3** is assigned to **DRIVER_2** and **DEVICE_4** has no driver.

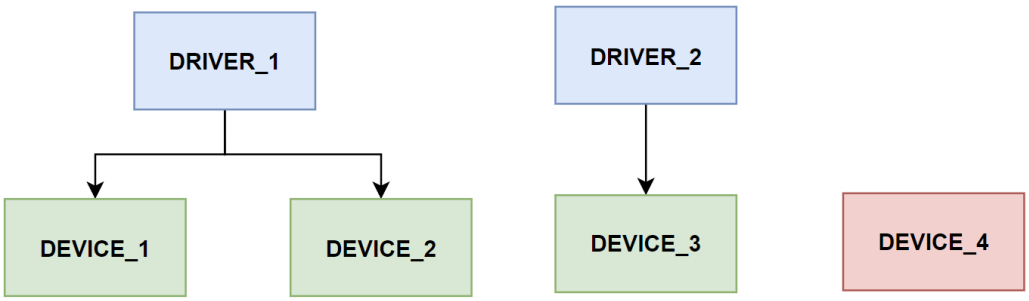

Figure 4.1: Example of relationship between devices and drivers

In Linux, a device is represented by a *struct device* and a driver is represented by a *struct device_driver*. In our case though, this isn't really interesting as we won't be using these structures directly.

4.1 Platform devices and drivers

A **platform device** is simply a more "specialized" type of device. This is used to refer to a class of devices that are not dynamically discoverable (like it's the case for devices sitting on

PCI buses). In the embedded world, this is usually the type of devices you will end up having to deal with. Intuitively, a **platform driver** is just a driver written for a platform device.

A platform device is represented by a *struct platform_ device*, while a platform driver is represented by a *struct platform_ driver*. First, let's look at the definition of *struct platform_ device*:

```
/* definition taken from include/linux/platform_device.h */
struct platform_device {
    struct device dev;
    struct resource *resource;
    u32 num_resources;
    // some fields intentionally omitted here
};
```

Unsurprisingly enough, the *platform_ device* structure contains a *struct device*. This is because, as mentioned, a platform device is just a type or variant of a device. The *resource* field is an array of resources assigned to the device. For a devicetree-based device, this can be one or more memory regions (described via the *reg* property), interrupt lines, etc....

The definition of *struct platform_ driver* is as follows:

```
/* definition taken from include/linux/platform_device.h */
struct platform_driver {
    int (*probe)(struct platform_device *);
    struct device_driver driver;
    // some fields intentionally omitted here
};
```

As expected, the *platform_ driver* structure is a container of the *device_ driver* structure. A very interesting field of the structure is *probe*. Whenever a platform device is added to Linux's infrastructure, the core will try to look for a platform driver for it based on some name matching which will be exemplified later on. If there's a match between the two, the core will try to probe the platform driver. What this function does is it tells the core if the driver is able to handle the device. In turn, from the platform driver's point of view, what this function usually does is it performs some sort of initialization. If everything goes well, the function will return 0, thus letting the core know that it can bind the device to the driver.

4.2 Matching a platform driver with a device

For DT-based systems, a platform driver needs to give the core a list of platform device names it supports so that it can be used in the matching process. This list is represented as an array of *struct of_ device_ id*, which has the following definition:

```
/* definition taken from include/linux/mod_devicetable.h */
struct of_device_id {
    char compatible[128];
    const void *data;
    // some fields are intentionally omitted here
};
```

The *compatible* field contains the name of the device the driver is, well, compatible with and is what's used by the core for the matching. Since the driver may be compatible with multiple devices, the *data* field can be used to associate some data with a certain device[1].

[1]Note that this data is only relevant to the platform driver and is optional

From the platform device's side, the values of the *compatible* property (described in 3.2.3) are the ones used in the matching process.

As such, assuming the following devicetree node:

```
my_awesome_device: device {
    compatible = "v1,m1", "v2,m2", "m3";
    status = "okay";
};
```

and the following snippet from our platform driver:

```
/* taken from my_awesome_driver.c */

static const struct of_device_id my_awesome_device_of_match[] = {
    { .compatible = "v1,m1" },
    { /* sentinel */ }
};
```

the core will detect a match between the platform device associated with the *my_awesome_device* node and the *my_awesome_driver* platform driver. Note that this can only happen if the node's *status* property is set to "okay". If missing (i.e: *status* property is not specified), the core will assume the status is "okay".

Please be very careful here with respect to what compatible strings you choose to use. Linux doesn't seem to have a score-based mechanism when doing the driver-device binding. If you have more than one compatible string what could end up happening is the device will be associated with the driver corresponding with the more generic compatible string, which might not be a desired outcome.

4.3 Writing our first platform driver

Assuming we have the following devicetree node:

```
my_awesome_node: my-awesome-device {
    compatible = "nss,my-awesome-model";
    status = "okay";
};
```

we want to create a platform driver for it. All the driver has to do is print some sort of message during its *probe()* function. As such, the following snippet shows a possible implementation of our platform driver:

```
#include <linux/of.h>
#include <linux/platform_device.h>

static int my_awesome_driver_probe(struct platform_device *pdev)
{
    pr_info("Hello, world from my awesome driver!\n");

    return 0;
}

static const struct of_device_id my_awesome_driver_of_match[] = {
```

```
    { .compatible = "nss,my-awesome-model", },
    { /* sentinel */ },
};

static struct platform_driver my_awesome_driver_platform_driver = {
    .probe = my_awesome_driver_probe,
    .driver = {
        .name = "my_awesome_driver",
        .of_match_table = my_awesome_driver_of_match,
    }
};

/* note: what this statement does is it creates two functions:
 * my_awesome_driver_platform_driver_init() and
 * my_awesome_driver_platform_driver_exit(). These functions will call
 * platfrom_driver_register() and platform_driver_unregister().
 */
module_platform_driver(my_awesome_driver_platform_driver);
```

Chapter 5

GPIO basics

5.1 What is a GPIO?

A GPIO (General Purpose Input Output) (pin) is a digital pin, which can be controlled from the software. Usually, these pins can operate as *intput* or as *output* pins.

5.2 Arduino example

Let's consider the following snippet, which can be run on an Arduino board:

```
#define MY_PIN_NUMBER 2

void setup()
{
    pinMode(MY_PIN_NUMBER, OUTPUT);
}

void loop()
{
    digitalWrite(MY_PIN_NUMBER, HIGH);
    delay(1000);
    digitalWrite(MY_PIN_NUMBER, LOW);
    delay(1000);
}
```

What this does is it first configures the GPIO pin number 2 as output (in the *setup()* call which is invoked only once during the start of the program). It then proceeds to drive the pin low (logic 0) and high (logic 1) in the *loop()* function, which is called endlessly.

More importantly, to toggle the pin output level, this example follows three steps:

1. Identify the targeted GPIO pin.
2. Set the GPIO pin direction.
3. Write a logic value to the GPIO pin.

5.3 Linux GPIO on PICO-PI-IMX8M

These steps can be followed to achieve the same effect on Linux. Since some of the information below is somewhat vendor-specific, we shall refer to the PICO-PI-IMX8M board and its SoC, i.MX8MQ.

5.3.1 i.MX8MQ GPIO controllers and pins

i.MX8MQ uses the concept of a GPIO controller. In this context, a GPIO controller is a module in charge of configuring and driving the GPIO pins. The SoC contains 5 GPIO controllers (indexed from 0 to 4), each of them managing 32 GPIO pins (indexed from 0 to 31). As such, working with a GPIO pin means we need to identify the GPIO controller that manages said pin and its corresponding index.

Let's assume one wishes to use the *GPIO_P30_3V3* pin. Sometimes, the pin name is intuitive enough to deduce the GPIO controller and pin index from it but this is not the case. As such, we need to have a look at the PICO-PI-IMX8M board schematic and the PICO-IMX8M reference manual.

According to the board schematic (see page 2, *BOARD TO BOARD*), the *GPIO_P30_3V3* pin is connected to the E1 connection header (pin/pad 30). Looking at the PICO-IMX8M reference manual (*Table 7 - PICO Compute Module Pin Assignment*), we can see that the signal connected to E1's pin 30 is called *GPIO3_IO5*. From this, we can deduce that the index of the controller is 3, while the index of the GPIO pin is 5.

With this information in mind, we can now follow the steps presented in the previous section.

5.3.2 Identify the targeted GPIO pin

At this point, just knowing the GPIO controller and pin index is not enough. This information needs to be encoded in the devicetree, which is fairly straightforward and can be achieved using the following property

```
[name]-gpios = <&[gpio_controller_node] [gpio_pin_index] [flags]>;
```

where

- *name* is the name of the GPIO as given by the user.
- *gpio_controller_node* is the devicetree node of the GPIO controller.
- *gpio_pin_index* is the GPIO pin index.
- *flags* refers to various configuration options. For now, only the *GPIO_ACTIVE_HIGH* flag will be taken into account.

With this in mind, using the *GPIO_P30_3V3* pin would mean having to add the following property to the consumer node (i.e: node corresponding to the device that wishes to use said GPIO pin):

```
led-gpios = <&gpio3 5 GPIO_ACTIVE_HIGH>;
```

5.3.3 Set the GPIO pin direction

Naturally, the next step is to configure the pin direction in our platform driver. Unlike Arduino though, Linux doesn't work directly with the pin index. Instead, it uses a *struct gpio_desc* to refer to a GPIO pin. As such, the platform driver needs to obtain a reference to said structure

so that it can use its assigned GPIO pin. The following snippet exemplifies how this can be done:

```
static int my_platform_probe(struct platform_device *pdev)
{
    struct gpio_desc *gpio_handle;

    gpio_handle = devm_gpiod_get(&pdev->dev, "led", GPIOD_OUT_LOW);
    if (IS_ERR(gpio_handle)) {
        // do some error handling here
    }

    return 0;
}
```

The *devm_ gpiod_ get()* function has the following prototype

```
struct gpio_desc *__must_check
    devm_gpiod_get(struct device *dev,
                   const char *con_id,
                   enum gpiod_flags flags);
```

where

- *con_ id* is the name of the GPIO. It needs to match the [name] bit of the devicetree property.
- *flags* refers to various configuration options. It can be used to set pin direction and initial value (low or high). For now, only *GPIOD_ OUT_ LOW* shall be taken into account.

5.3.4 Write a logic value to the GPIO pin

The final step is to toggle the GPIO pin. To do so, use the previously obtained reference to *struct gpio_ desc* in conjunction with the *gpiod_ set_ value()* function. Its prototype is

```
void gpiod_set_value(struct gpio_desc *desc, int value);
```

Chapter 6

Exercises

Before starting the exercises, make sure you enable the following configurations:

- *CONFIG_NSS_DRIVERS*
- *CONFIG_NSS_DRIVERS_LAB02*

6.1 Devicetree warm-up

Starting from the skeleton found at

```
drivers/nss/lab02/ex1/imx8mq-pico-pi.dts
```

write a simple DTS for the *PICO-PI-IMX8M* board based on the following requirements:

- The DTS needs to include the *cpus* node.

  ```
  Hints:
      1) What SoC is the board based on?
      2) What and how many ARM cores does the SoC have?
      (only include the A cores)
      3) What compatible should you use for a core?
  ```

- The DTS needs to describe the available RAM. Assume the 1GB model.
- The DTS needs to include the *UART1*, *SAI6*, and *GIC400* components. Additionally:
  - The node names should be taken from: https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.4

    ```
    Hint: what's the purpose of each of these
    components?
    ```

  - *UART1* is most compatible with the *fsl,imx8mq-lpuart* driver, but can also work with the *fsl,imx8-lpuart* driver.

    ```
    Hint: which compatible(s) to use here?
    ```

- *GIC400* can only work with the *arm,gic400* driver.
- *SAI6* is most compatible with the *fsl,imx8mq-sai* driver, but can also work with the *fsl,imx6-sai* driver. The component is controlled by another OS so it shouldn't be used by Linux.

```
Hint: what status should we put here?
```

To compile a devicetree source, use the following command:

```
dtc -I dts -O dtb -o <output.dtb> <input.dts>
```

Information regarding what the components are and their address spaces can be found in the SoC reference manual.

6.2 Platform drivers warm-up

Starting from the skeleton found at

```
drivers/nss/lab02/ex2/my-awesome-driver.c
```

write a platform driver that prints a *Hello, world* message in its *probe()* function.

The devicetree node you should use is

```
my-awesome-device {
    compatible = "nss,my-awesome-device";
}
```

You'll have to place this node inside the *nss-bus* node. Before starting, make sure you enable *CONFIG_NSS_DRIVERS_LAB02_EX02*.

6.3 LED warm-up

Starting from the skeleton found at

```
drivers/nss/lab02/ex3/led.c
```

write a platform driver that exposes to the user space a character device interface through which a user can toggle the state (on/off) of an LED. The LED should be connected to the *GPIO_P26_3V3* pin as shown in figure 6.1.

Assuming the name of the character device is *led-chardev*, issuing

```
echo "on" > /dev/led-chardev
```

should turn on the LED. In contrast, doing

```
echo "off" > /dev/led-chardev
```

should turn off the LED.

Before starting, make sure you enable the *nss-led* devicetree node (by setting its status to *okay*) and the *CONFIG_NSS_DRIVERS_LAB02_EX3* configuration.

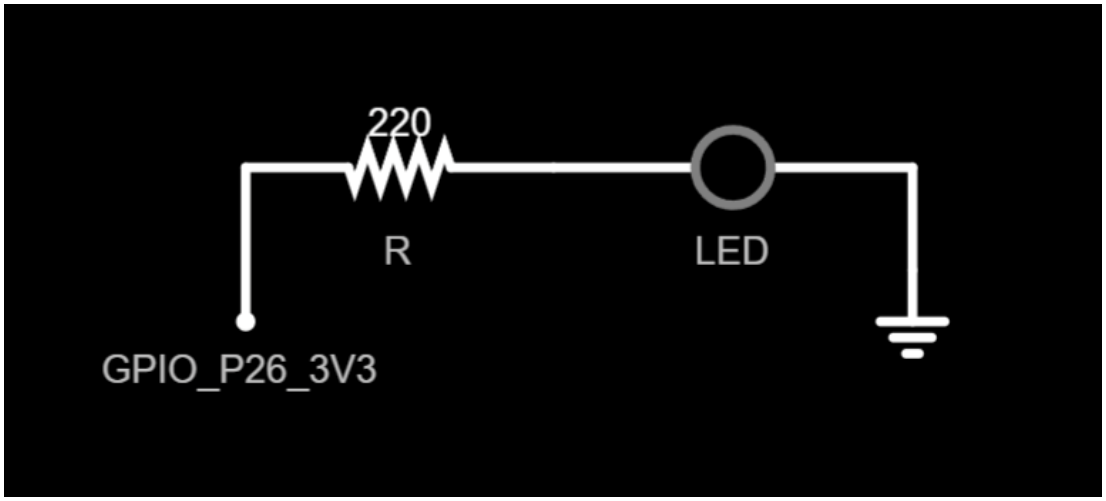

Figure 6.1: LED warm-up circuit

6.4 LED math

Starting from the skeleton found at

```
drivers/nss/lab02/ex4/led-math.c
```

write a platform driver that will act as a very simple calculator. The input will be received through a character device interface and will have the following format

```
[digit_1][op][digit_2]
```

digit_1 and *digit_2* can have any value from the 0-9 range, while *op* has two valid values:

- *"+"* - addition
- *"-"* - subtraction (second number is subtracted from first)

The result will be "written" to three LEDs, each of them signifying a bit of the resulting number. If the LED is lit that means its corresponding bit is set.

The pins you should use and their corresponding bit indexes are:

- *GPIO_P26_3V3* - bit 0
- *GPIO_P28_3V3* - bit 1
- *GPIO_P30_3V3* - bit 2

Figure 6.2 shows how the LEDs should be connected to these pins.

Examples of inputs and their results:

- *1+2* - result is 3, LED states are:

```
GPIO_P26_3V3 LED => on
```

```
GPIO_P28_3V3 => on
GPIO_P30_3V3 => off
```

- *3-1* - result is 2, LED states are:

```
GPIO_P26_3V3 LED => off
GPIO_P28_3V3 => on
GPIO_P30_3V3 => off
```

- *9-3* - result is 6, LED states are:

```
GPIO_P26_3V3 LED => off
GPIO_P28_3V3 => on
GPIO_P30_3V3 => on
```

- *9+9* - invalid, LEDs keep their previous state
- *3-6* - invalid, LEDs keep their previous state

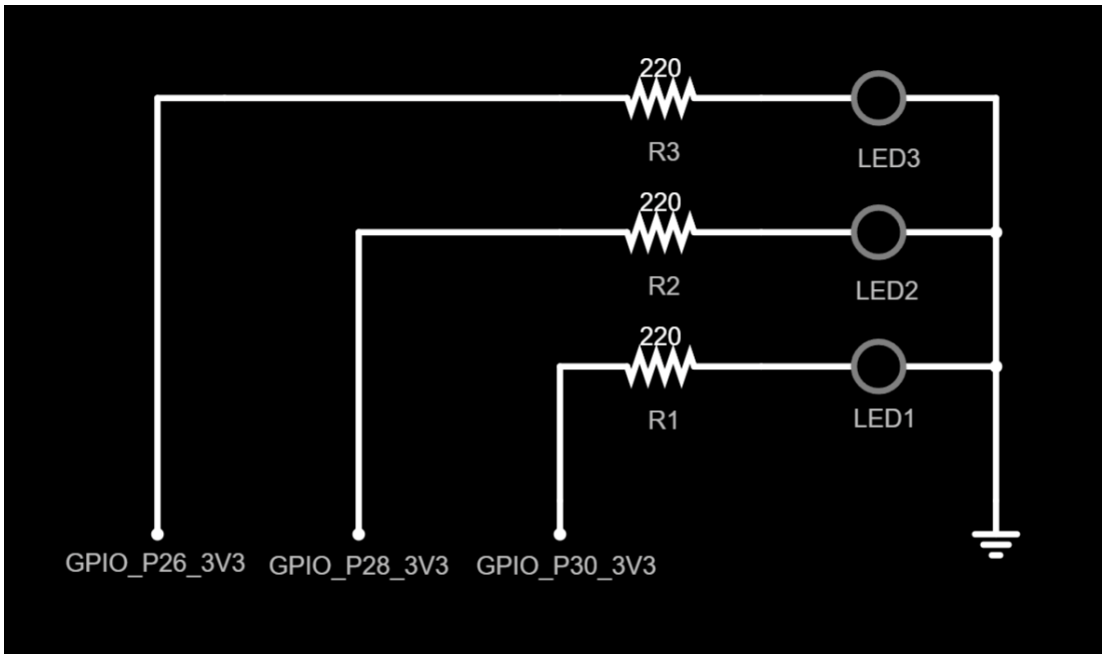

Figure 6.2: LED math circuit

Before starting, make sure you:

- Enable the *nss-led-math* node.
- Disable the *nss-led* node.
- Enable *CONFIG_NSS_DRIVERS_LAB02_EX04*