Access hardware from userspace with mmap

Introduction:- not me

I was working with a device that used Atmel SAMA5D3x MCU. Sometimes the devices needed to be re-flashed which required putting the MCU to programming mode. However, the device enclosure needed to be opened and also a jumper wire was needed to do this. Then I realized that it would be possible to enter the programming mode directly from Linux by manipulating boot sequence controller registers in the MCU. Writing a dedicated device driver for this seemed like an overkill so I wrote a simple utility application with mmap instead.

Why mmap is needed?

-- Usually peripherals and hardware blocks are accessed using memory mapped register interface.

-- The control and status registers are mapped to certain memory addresses, and the interaction with such peripherals is simply **reading and writing** to these memory locations.

-- Device drivers that are running on the operating system level can **easily access these physical addresses.**

However, processes running on the userspace are confined in their virtual memory and cannot access physical addresses directly.

Virtual memory is a memory management technique used by modern operating systems such as Linux in which the memory addresses used by applications are mapped to the physical memory of the device.

Basically an application sees a continuous memory, but in reality the memory pages may be spread around in the physical memory or not even loaded to it.

Besides the memory management, this scheme also has many additional benefits such as prohibiting processes from accessing each other's memory and prohibiting processes from accessing operating system resources or peripheral registers.

It also allows the operating system to swap memory pages in and out of the main memory and much more.

So basically, if a process tries to write to a physical memory address that is valid peripheral register, it will produce a **segmentation fault** because the **memory region was not mapped for that process**. In Linux these memory mappings can be requested from the operating system with mmap call.

Calling **mmap** creates new **virtual** memory mapping for the **calling process**. It is important to notice that even after the memory mapping the physical address and the mapped virtual memory address are most likely not the same.

But accessing the virtual memory addresses will access the requested physical address region, for instance peripheral register.

When should mmap be used?

More often than **not** accessing peripherals **directly** from a **userspace application** is a bad idea.

There are very good reasons why the hardware is accessed through device drivers.

To name a few the drivers provide abstraction between the hardware implementation details and the applications, the drivers can also serialize concurrent accesses and implement the necessary locking and mutual exclusion.

Nonetheless, in some situations accessing hardware directly can be the appropriate thing to do. Mostly such situations could be testing, debugging and problem solving related where investing the time to implement proper drivers might not be worth it. In production use, however, it is better to create proper drivers.

Mmap can also be **used to map regular files to memory**.

Example: Atmel SAMA5D3x programming mode

The Atmel SAMA5D3x has a special programming mode, SAM-BA monitor, that is entered if no valid boot media is found. This is the case with a blank device so it can easily be programmed. However, sometimes devices that are already programmed need to be re-flashed. The flashing is done using Atmel SAM-BA programming tool. In this case, if the device was already flashed it was still possible to boot to programming mode using a jumper wire. But as said this required opening the device enclosure which became, well, annoying at some point. Luckily it was possible to enter the SAM-BA monitor programmatically.

The boot sequence in this MCU is controlled by Boot Sequence Controller (BSC) peripheral which has a control register to select external boot memories. Setting this register to correct value also allows to bypass all media and enter the programming mode. This register retains its value on soft reboot (as long as power is not cut) so it is possible to write it from Linux and reboot directly to programming mode. The following figure shows the boot configuration options from the MCU datasheet.

sama5d3-boot-values

So, writing this configuration register to any value from 5 to 7 will make the MCU boot directly to programming mode and skip all other media. The following excerpt from the MCU datasheet shows the relevant control register.

sama5d3-bsc-cr

The register contains the BOOT value in lower bits and also a write-protect key in the higher bits to avoid accidental writes to the BOOT value. So in its simplicity value 0x66830007 needed to be written to physical memory address 0xFFFFFE54.

The following code example shows how to create memory mapping with mmap and then use the returned pointer to access the register.

```
1 int fd = open("/dev/mem", O_RDWR);
2
3 /* Map one page of memory that contains the BSC
4    control register */
5 void* ptr = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
6                   MAP_SHARED, fd, 0xFFFFF000);
7 close(fd);
8
9 if(ptr == MAP_FAILED)
10{
11    printf("mmap failed\n");
12    return -1;
13}
14
15volatile unsigned* bsc_cr = (unsigned*)(ptr + 0xE54);
16
17/* Write the new BOOT value */
18*bsc_cr = 0x66830007;
19printf("BOOT value set to %u\n", *bsc_cr);
20
21munmap(ptr, 4096);
```

In Linux the ***/dev/mem*** device file can be used to access the RAM memory. Even though access to only one register is needed, the mapping needs to be done for the whole memory page. In this case the page size is 4096 so the memory page in which the BSC register 0xFFFFFFE54 is located is at 0xFFFFF000. Line 5 shows how this page is mapped using mmap. It is important to note that the returned pointer ***ptr*** most likely does not point to 0xFFFFF000 but writing to the returned pointer will in fact write to that physical address.

In line 15 a pointer is constructed that is used to write to the BSC control register. The offset within the page, 0xE54, is added to the base pointer to get the correct register address. Now, as line 18 shows, the register value can easily be written using the pointer. Finally line 21 unmaps the page once it is not needed anymore.

So, running this small code snippet before reboot command boots the device directly to programming mode and avoids opening the device enclosure and fiddling with jumper wires.