Lecture 11 — Memory-Mapped I/O

Jeff Zarnett & Mike Cooper-Stachowsky
jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 6, 2024

Take Me There

Memory-mapped I/O – to communicate with a HW device, an area that looks like main memory really represents the HW device.

This simplifies some operations!

Kind of like how UNIX treats everything as a file…

Port Mapped

This contrasts with the older-style port-mapped I/O

There are different CPU instructions and separate registers used to communicate with the device.

It's not that this does not work, but it's just more difficult to work with.

I/O or Control

Maybe a better name for this is “Memory Mapped Control”.

The device is controlled via setting some voltages (1s, 0s) on the chip.

If we map those to memory locations, we know how to read write the 1s and 0s to the device just as if it were memory.

Peripheral

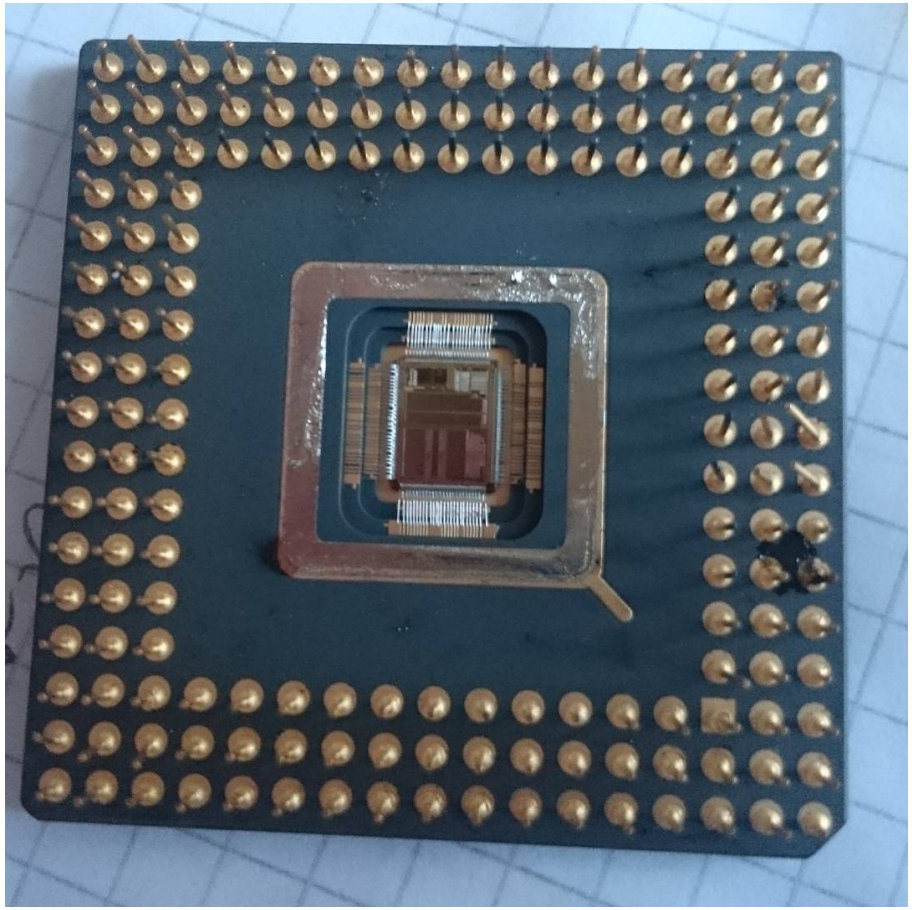
Hardware devices outside the CPU and memory are sometimes referred to as peripheral devices.

A peripheral is an often optional circuit that interfaces with the CPU.

Examples: UART, I2C, GPIO...

Circuit Problem

The physical connectors that link one device to the motherboard are called pins.



Pin Management

The chip only has a fixed number; some pins do multiple things… But only ever one at a time.

This implies a need to configure things!

Alright, but let’s think about how to control a device…

We need to send data out to those pins.

Okay, but where are they?

The data sheet for the device will tell you where the registers are.

One option: absolute address (which is nice).

Other option: base and offset locations.

Why base and offset? Support multiple devices.

Register Types

Configuration: Set up the device.

Data: Read/Write data locations.

Control: Tell the device what to do!

STM Did What?

The chip has multiple GPIO ports so it’s base + offset.

Base locations are given in the memory map; offset in GPIO’s documentation.

Addresses are the byte address, but registers are often 32 bits.

Global Enable

Some chips require global settings or signals to be sent to them.

Example: anything that uses the clock signal needs it sent to it!

Let’s talk RCC

RCC: Reset and Clock Control

These are registers used to manage power.

If we don’t need a peripheral, don’t send it the clock signal!

If it doesn’t get the clock signal, it does nothing and saves power.

Consult the Map

Altenative: `mmap()`

An alternative use for memory-mapping in UNIX involves the use of `mmap()`, a function nominally used to map a file into memory.

Then, interact with the file just like memory!

Mapping

```
void* mmap( void* address, size_t length, int protection, int flag,
   int fd, off_t offset );
```

`address`: where you want the mapped region to go; use `NULL`.

`length`: how many bytes to map.

`protection`: rules for how memory can be used.

`flag`: mode for mapping.

`fd`: file descriptor of the file to map.

`offset`: how far from the start of the file mapping begins.

Protection Flags

Valid values are `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and `PROT_EXECUTE`.

They can be combined with the bitwise OR operator.

Whatever flags you choose have to be consistent with how the file was opened with open.

What's the point of PROT_NONE, if all things are forbidden?

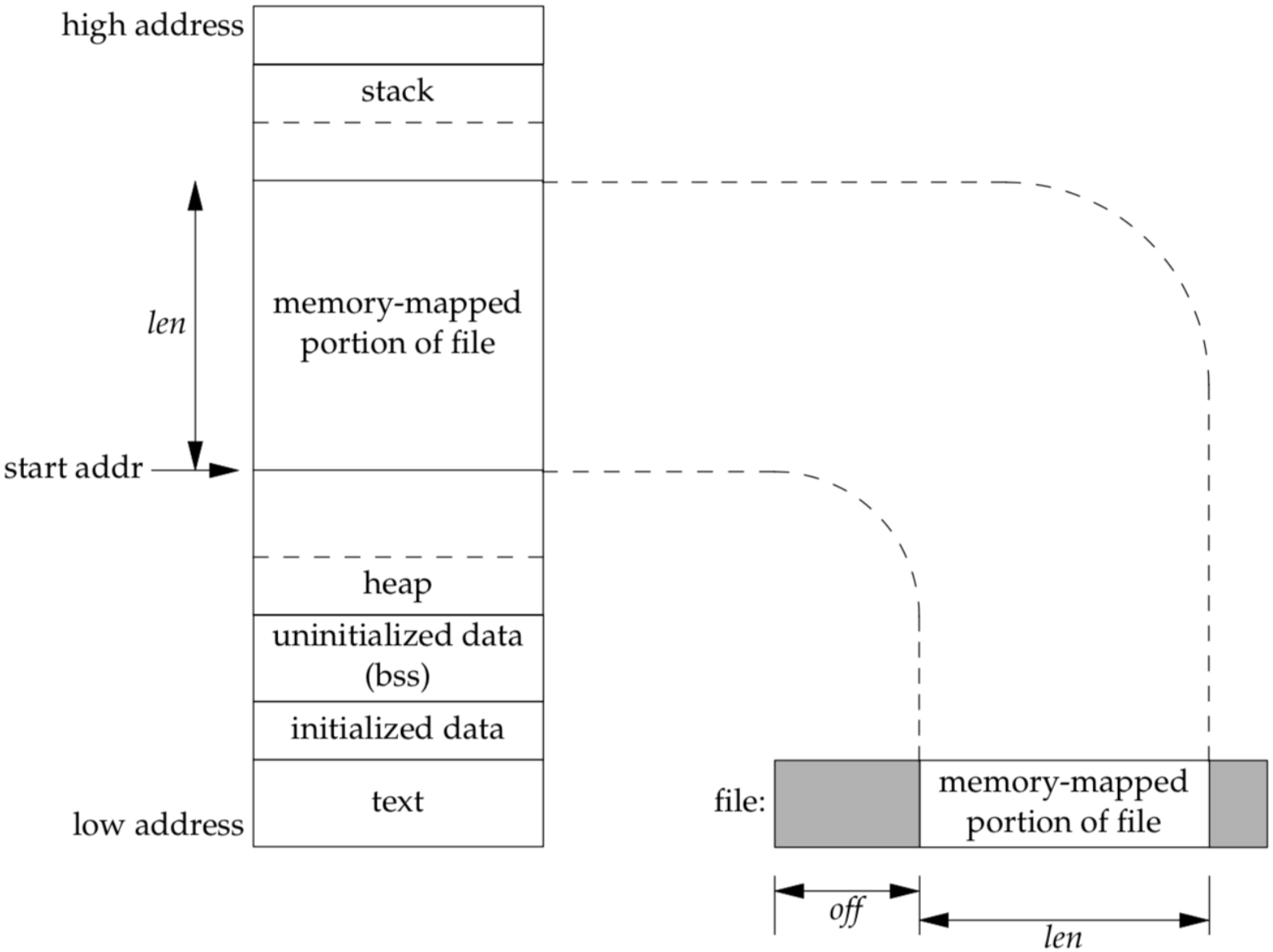
Flags

Flags can be one of two options: `MAP_PRIVATE` or `MAP_SHARED`.

Private: modifications are not visible to other processes mapping the same file and not written out to the underlying file.

Shared: modifications are visible to other processes and written out to the file… but maybe not instantly.

Memory Mapped File



Protection

If we wish to change the protection rules for a section, we use `mprotect`.

```
int mprotect( void* address, size_t length, int prot );
```

`address`: the memory to modify protection of.

`length`: the size of said memory.

`prot`: the new protection rules.

Synchronize

```
int msync( void* address, size_t length, int flags );
```

`address`: the memory to synchronize.

`length`: how many bytes to synchronize.

`flags`: mode for synchronization; use `MS_SYNC` (blocking).

Unmap

```
int munmap( void* address, size_t length );
```

`address`: the memory to unmap.

`length`: how many bytes to unmap.

A segment would be unmapped automatically when a process exits, but as always it is polite to unmap it as soon as you know that you are done with it.

Memory Mapping Example

```c
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main( int argc, char** argv ) {

    int fd = open( "example.txt", O_RDWR );

    struct stat st;
    stat( "example.txt", &st );
    ssize_t size = st.st_size;
    void* mapped = mmap( NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0 );
```

Memory Mapping Example

```
    int pid = fork();
    if ( pid > 0 ) { /* Parent */
        waitpid( pid, NULL, 0 );
        printf("The new content of the file is: %s.\n", (char*) mapped);
        munmap( mapped, size );
    } else if ( pid == 0 ) { /* Child */
       memset( mapped, 0, size ); /* Erase what's there */
       sprintf( mapped, "It is now Overwritten");
       /* Ensure data is synchronized */
       msync( mapped, size, MS_SYNC );
       munmap( mapped, size );
    }
    close( fd );
    return 0;
}
```

The Example is... Flawed

The example works acceptably in the sense that we successfully overwrite the data with the new data and the parent process sees the change.

But things get weird if we tried to write fewer bytes than the original message.

In general, the mapped area size cannot change.

Linux has `mremap` but this is not portable...

Memory-Mapped Files

How does it work, though?

Basic premise: disk blocks mapped to main memory.

First access is a page fault; subsequent accesses are reads/writes.