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 7, 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

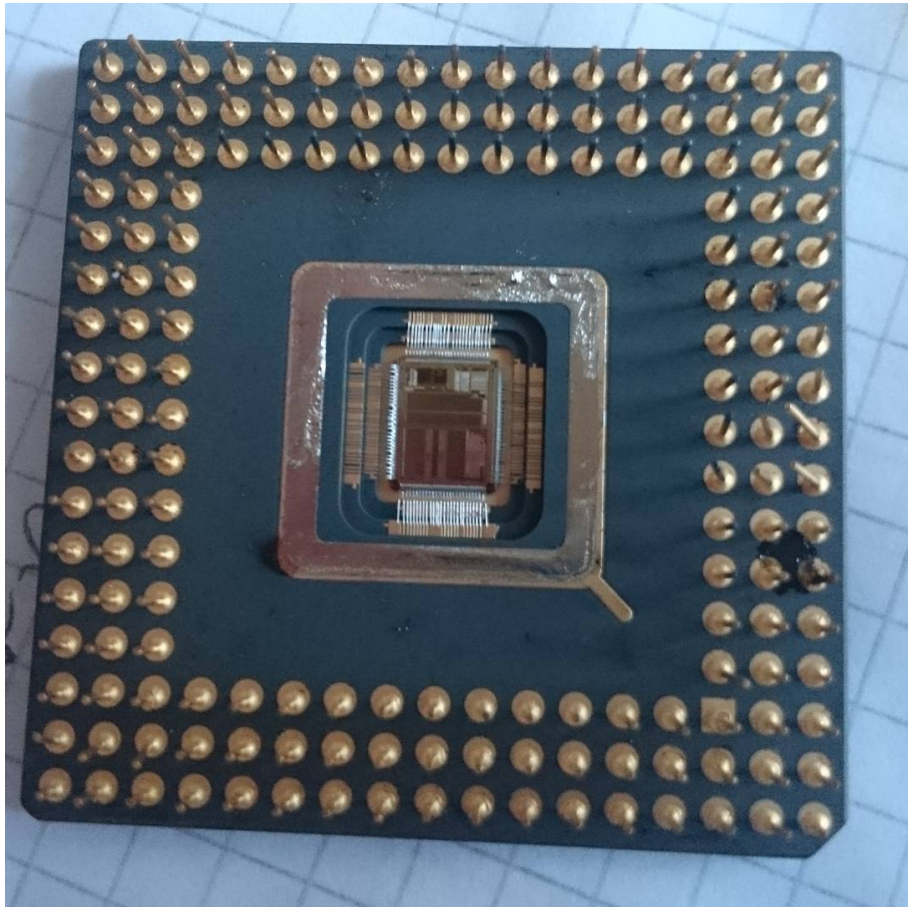
Hardware devices outside the CPU and memory are sometimes referred to as peripheral devices.

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.

RCC Base Address

Table 1. STM32F401xB/C and STM32F401xD/E register boundary addresses

Boundary address	Peripheral	Bus	Register map
0x5000 0000 - 0x5003 FFFF	USB OTG FS	AHB2	*Section 22.16.6: OTG_FS register map on page 755*
0x4002 6400 - 0x4002 67FF	DMA2	AHB1	*Section 9.5.11: DMA register map on page 198*
0x4002 6000 - 0x4002 63FF	DMA1		
0x4002 3C00 - 0x4002 3FFF	Flash interface register		*Section 3.8: Flash interface registers on page 60*
0x4002 3800 - 0x4002 3BFF	RCC		*Section 6.3.22: RCC register map on page 137*
0x4002 3000 - 0x4002 33FF	CRC		*Section 4.4.4: CRC register map on page 70*
0x4002 1C00 - 0x4002 1FFF	GPIOH		*Section 8.4.11: GPIO register map on page 164*
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		

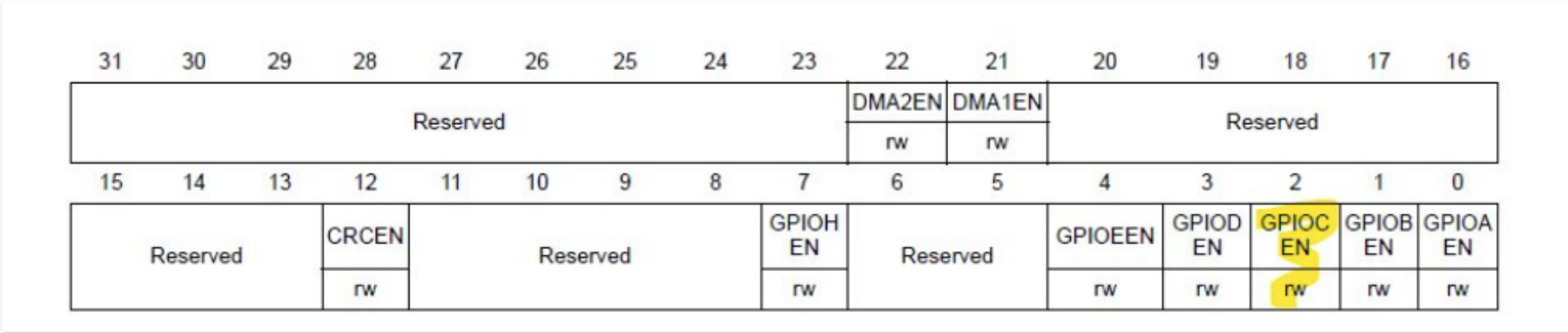
RCC_AHB1ENR

What's RCC_AHB1ENR? AHB is "Advanced High-Speed Bus"

This is a bus that controls some peripherals including GPIO.

This register controls whether the input has a clock signal.

Offset is `0x30`.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									DMA2EN	DMA1EN	Reserved				
									rw	rw					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved				GPIOH EN	Reserved		GPIOEEN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw					rw			rw	rw	rw	rw	rw

GPIO?!

What is GPIO then? General-Purpose Input/Output

The GPIO system is controlling the voltages on the pins.

What’s it good for?

GPIO Ideas

Slow digital input – signal detection, not data transfer.

Slow digital output – turning things on/off, not data transfer.

Analog input (ADC)

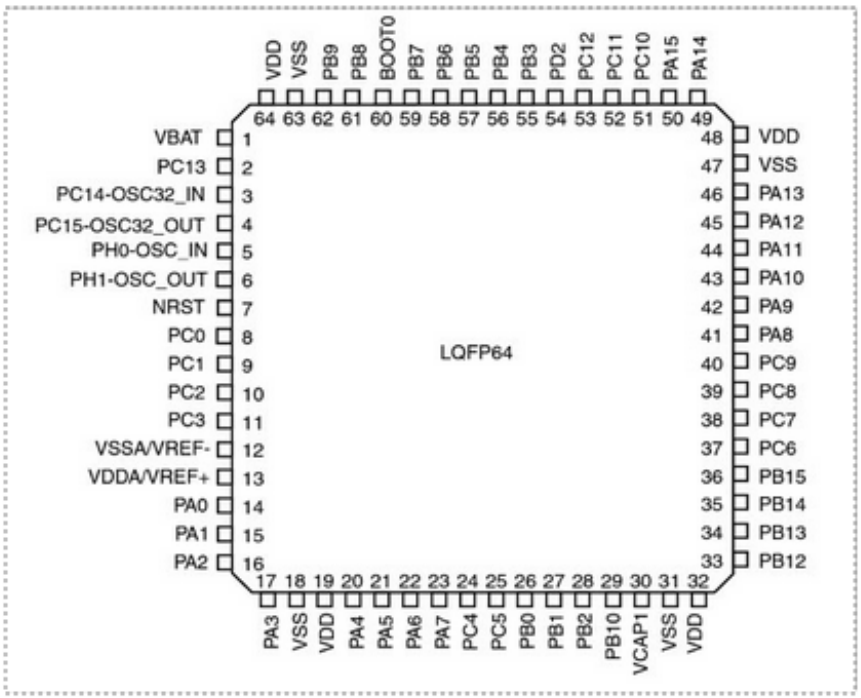
Other fun stuff if needed: “Alternate Functions”.

GPIO Organization

Pins are organized into ports of up to 16 pins.

Ports are PA∗ through PC∗ on our chip.

Each port is controlled via registers labelled `GPIOx...`



GPIO Base Addresses

Table 1. STM32F401xB/C and STM32F401xD/E register boundary addresses

Boundary address	Peripheral	Bus	Register map
0x5000 0000 - 0x5003 FFFF	USB OTG FS	AHB2	*Section 22.16.6: OTG_FS register map on page 755*
0x4002 6400 - 0x4002 67FF	DMA2	AHB1	*Section 9.5.11: DMA register map on page 198*
0x4002 6000 - 0x4002 63FF	DMA1		
0x4002 3C00 - 0x4002 3FFF	Flash interface register		*Section 3.8: Flash interface registers on page 60*
0x4002 3800 - 0x4002 3BFF	RCC		*Section 6.3.22: RCC register map on page 137*
0x4002 3000 - 0x4002 33FF	CRC		*Section 4.4.4: CRC register map on page 70*
0x4002 1C00 - 0x4002 1FFF	GPIOH		*Section 8.4.11: GPIO register map on page 164*
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		

GPIO Configuration Registers

`GPIOx_MODER`: I/O direction, analog or alternate function

`GPIOx_OTYPER`: Output type (push/pull, open-drain)

`GPIOx_OSPEEDR`: Speed for update/read data

`GPIOx_PUPDR`: Pullup/pulldown resistor enable, regardless of direction

What are the Modes?

The mode register is 32 bits with offset of `0x00`.

4 modes are available (2 bits per pin).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Alternate Functions

Alternate Functions just mean using one of the manufacturer-specific functions.

To use them, set the `GPIOx_AF` registers.

16 AFs per pin are allowed; 4 bits per pin.

Two AF registers per port: LOW and HIGH.

We won't worry about these until we must.

Other Registers

Other registers exist but we can usually just use them in their reset state.

If you need something, you can always look up what you need it to be.

GPIO Input/Output Registers

`GPIOx_ODR`: Output data register, must be OUTPUT mode.
This is the data! If bit `j` is 1, the pin `j` is HIGH.
Read-write.

`GPIOx_IDR`: Input data register, must be INPUT mode.
Stores data read by the pins, read-only.

GPIOx_ODR

Offset is `0x14`.

Only bits 0 to 15 are valid (pins 0 to 15); others are reserved (cannot use).

Used only in output mode – writing in other modes does nothing.

Write 1 or 0 to write HIGH or LOW on a given pin.
Can read back to know what was written.

GPIOx_IDR

Offset is `0x10`.

Only bits 0 to 15 are valid (pins 0 to 15); others are reserved (cannot use).

Used only in input mode – reads in other modes return garbage.

Read-only.

Basic GPIO Driver

Very basic GPIO driver: Read PC 13 (user button 1), write to PA5 (LED 2, green).

General plan:

- Define registers and offsets needed
- Create setup functions
- Create read functions for inputs
- Create write functions for outputs

Define Statements

```c
#include <stdint.h>
#include <stdbool.h>

#define _IDR_OFFSET 0x10
#define _ODR_OFFSET 0x14

#define _RCC_BASE 0x40023800
#define _RCC_GPIO_EN_OFFSET 0x30
#define _GPUI_CLOCK_EN *(uint32_t*) (_RCC_BASE + _RCC_GPIO_EN_OFFSET)

#define _GPIOC_BASE 0x40020800
#define _GPIOC_MODER *(uint32_t*) (_GPIOC_BASE)
#define _GPIOC_IDR *(uint16_t) (_GPIOC_BASE + _IDR_OFFSET)

#define _GPIOA_BASE 0x40020000
#define _GPIOC_MODER *(uint32_t*) (_GPIOA_BASE)
#define _GPIOC_ODR *(uint16_t) (_GPIOA_BASE + _ODR_OFFSET)
```

```
void setGPIOCInput() {
  // Enable clock
  _GPIO_CLOCK_EN |= 1<<2;
  _GPIOC_MODER &= ~(0x3 << 26);
}

void setGPIOAOutput() {
  // Clear it first
  _GPIOA_MODER &= ~(0x3 << 10);
  _GPIOA_MODER |= 1 << 10;
}
```

Access

```
int readGPIOC() {
  return (_GPIOC_IDR & 1<<13) >> 13;
}

void writeGPIOA(bool in) {
  if (in) {
    _GPIOA_ODR |= 1<<5;
  } else {
    _GPIOA_ODR &= ~(1<<5);
  }
}
```

```
while( 1 ) {
  writeGPIOA( !readGPIOC() );
}
```

It’s simple, but it does work!

We can take this template and use it to interact with other I/O devices.

But before we leave the topic, maybe you had a different idea of what memory-mapped I/O is…

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.

PROT_NONE

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

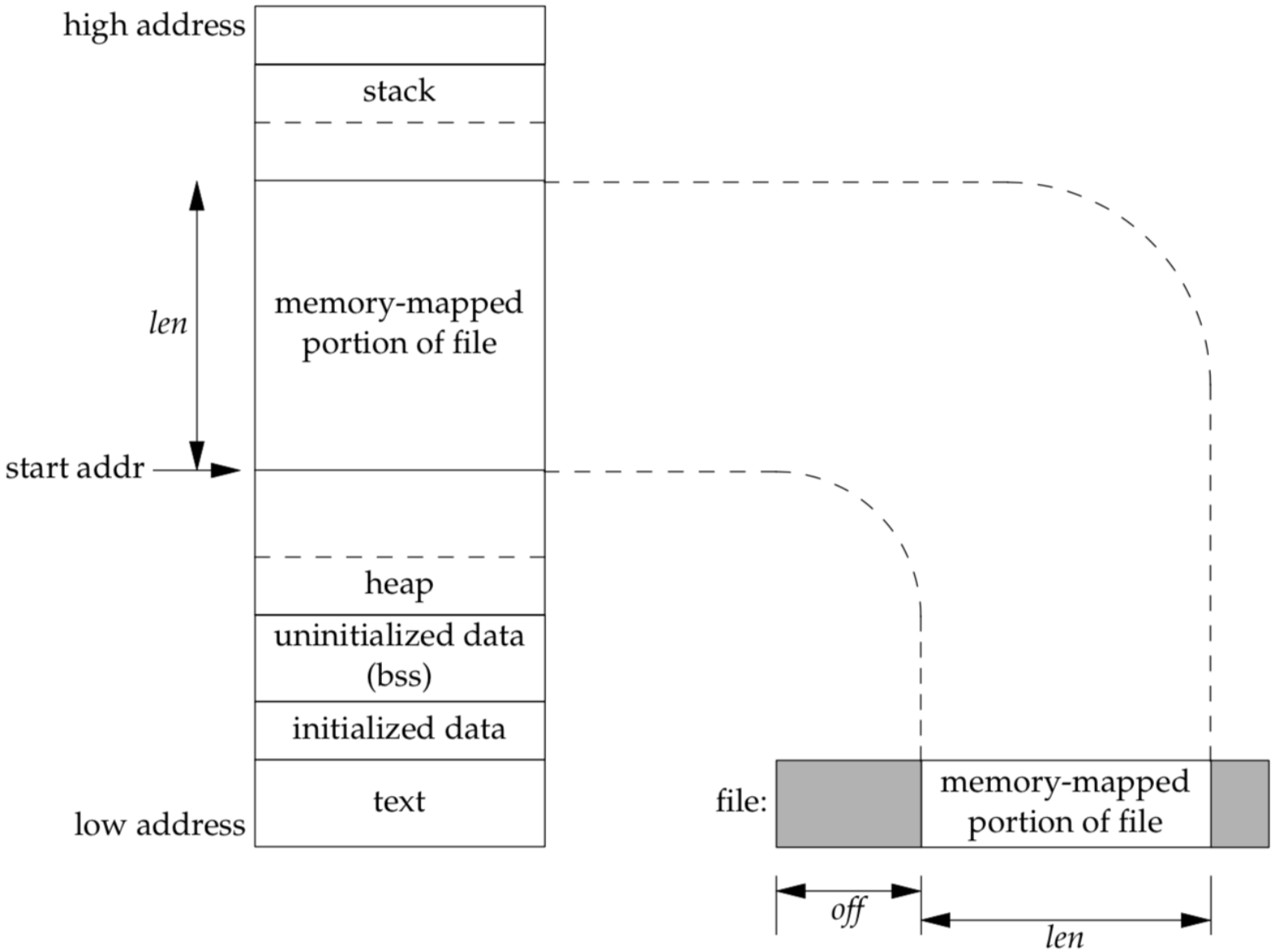
Flags

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



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

```
#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.