Introduction to Programming Embedded Systems

Sebastian Fischmeister
sfischme@seas.upenn.edu
Department of Computer and Information Science
University of Pennsylvania

Goals

- Rough understanding of the underlying hardware.
- Understand how to develop software for the lab platform.

What is An Embedded System?

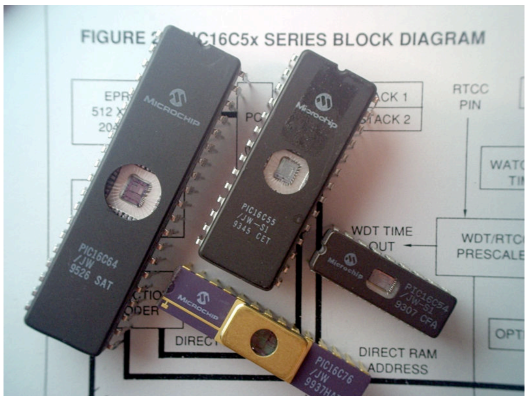
A general-purpose definition of embedded systems is that they are devices used to control, monitor or assist the operation of equipment, machinery or plant. "Embedded" reflects the fact that they are an integral part of the system. In many cases, their "embeddedness" may be such that their presence is far from obvious to the casual observer.

Institute of Electrical Engineers (IEE)

For Us

- PIC18F2680
  - 3,328 B RAM
  - 64kB ROM
  - 1.024 B EEPROM
  - 5 MIPS @ 20MHz

  - A/D converters
  - 1x UART
  - 1x 8bit Timer
  - 3x 16bit Timer

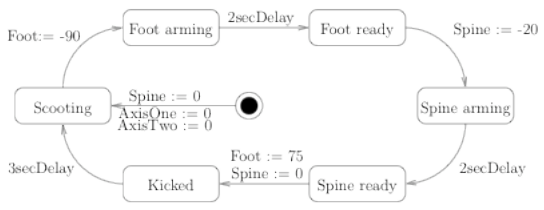


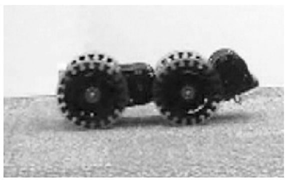
Will use in the PICDEM2 board to

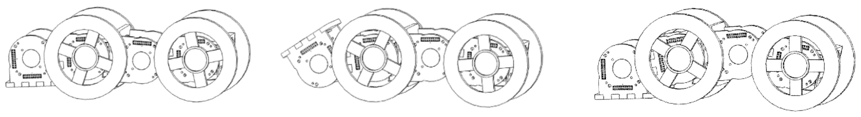
- Blink LEDs
- Control an LCD display
- Communicate via the serial line with a PC
- Communicate via the CAN protocol with other microchips
- Drive a stepper motor

Use it further to

- Control a modular robot:







The Hardware

A Microprocessor

- Introduced as a programmable replacement for logic-based circuits in the 1970s.
- Advantages compared to logic-based circuits:
  - Provide functional upgrades (e.g., add new feature to machine tool after deployment)
  - Provide easy maintenance upgrades (e.g., fix a bug in the cell phone via an SMS firmware upgrade)
  - Less fragile (e.g., instead of hundreds discrete logic chips and wiring only one microprocessor)
  - Protection of intellectual property (it is more difficult to copy software burnt in the on-chip memory than to check the part numbers and the wiring)

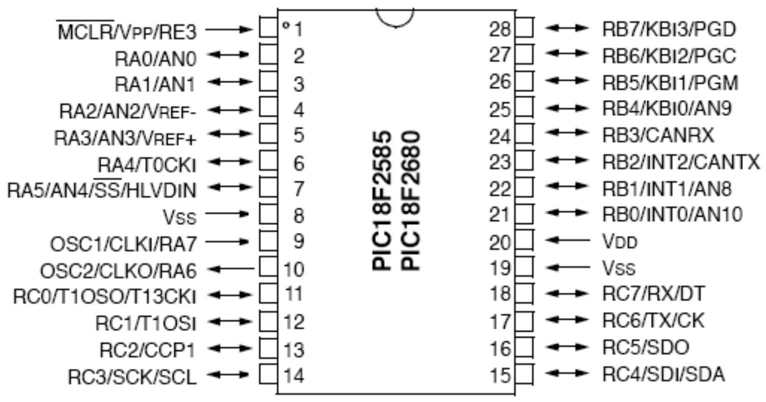
What makes a Microprocessor?

- Processor
  - An arithmetic logic unit (ALU) for processing.
- Memory
  - Permanent memory for keeping the program (= ROM)
  - Volatile memory for computation (= RAM)
  - Rewritable permanent memory for logging, tuning, storing intermediate data (= EEPROM)
- Connectivity to peripherals
  - Binary outputs via single chip pins
  - Integrated asynchronous and synchronous serial interfaces such as UART, I2C, RS232, CAN

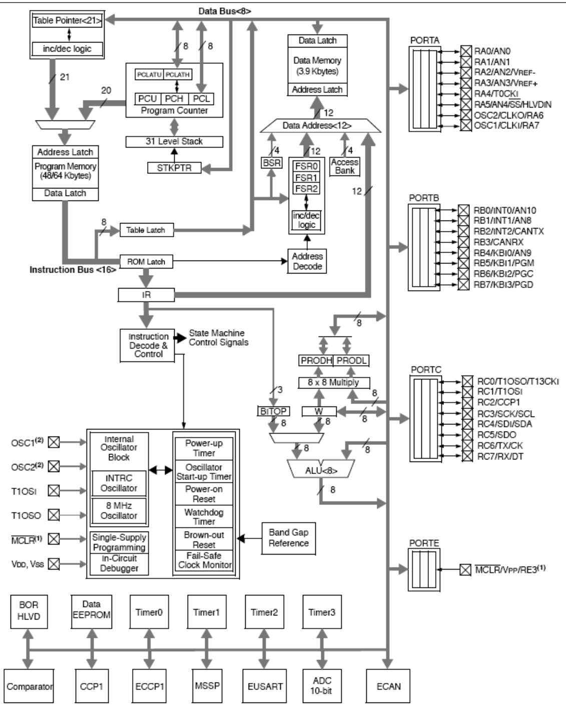
What makes a Microprocessor?

- Timers
  - Event counting, input capture, real-time interrupt, watchdog timer
  - Pulse-width modulation (PWM)
- Support for the analogue world
  - Analog-to-digital converter (ADC)
  - Digital-to-analog converter (DAC)
- Software debug support hardware
  - JTAG

Meet the PIC18F2680



Inside



Harvard Architecture

- Assign data and program instructions to different memory spaces.
- Each memory space has a separate bus.

- This allows:
  - Different timing, size, and structure for program instructions and data.
  - Concurrent access to data and instructions.
  - Clear partitioning of data and instructions (=> security)

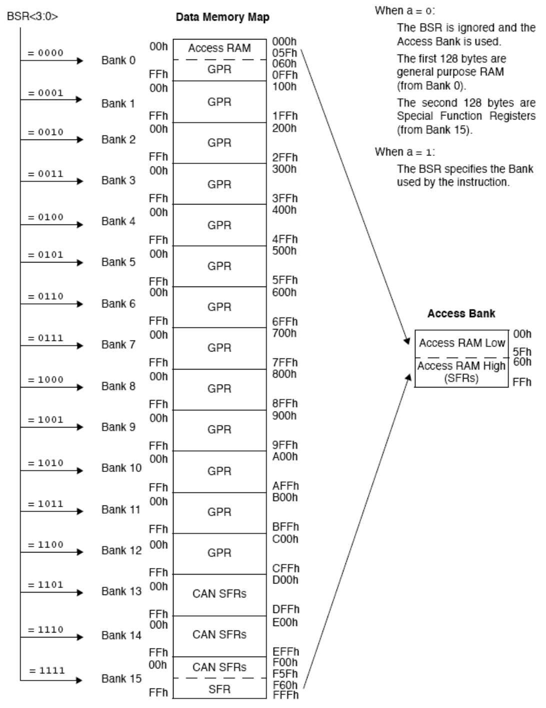
- This makes it harder to program, because static data can be in the program space or in the data space.
- If the program space and the data space are incompatible, copying data is no longer a *(<start>,len)* dump.

Data Memory

- Memory layout
  - Instructions in the PIC18 are limited to 16 bits.
  - To address the whole area you would need 12 bit => too many.
  - Memory is split into 256B banks. Only one is active.

- Register types
  - General-purpose registers (GPR)
  - Special function registers (SFR)

- SFR control the MCU and the peripherals.



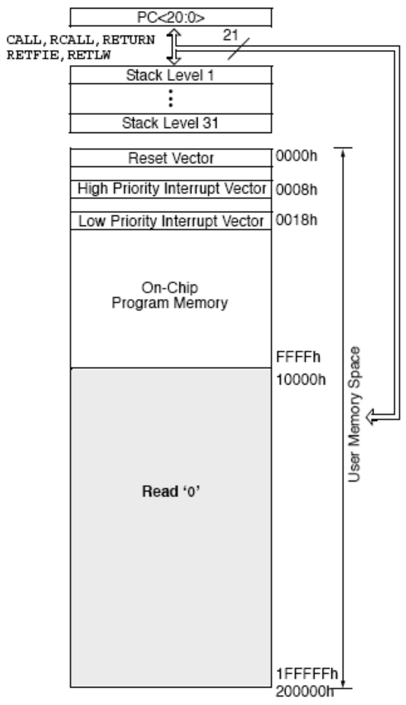
BSR<3:0>
Data Memory Map
= 0000 Bank 0 00h Access RAM GPR FFh 000h 05Fh 060h 0FFh
= 0001 Bank 1 00h GPR FFh 100h 1FFh
= 0010 Bank 2 00h GPR FFh 200h 2FFh
= 0011 Bank 3 00h GPR FFh 300h 3FFh
= 0100 Bank 4 00h GPR FFh 400h 4FFh
= 0101 Bank 5 00h GPR FFh 500h 5FFh
= 0110 Bank 6 00h GPR FFh 600h 6FFh
= 0111 Bank 7 00h GPR FFh 700h 7FFh
= 1000 Bank 8 00h GPR FFh 800h 8FFh
= 1001 Bank 9 00h GPR FFh 900h 9FFh
= 1010 Bank 10 00h GPR FFh A00h AFFh
= 1011 Bank 11 00h GPR FFh B00h BFFh
= 1100 Bank 12 00h GPR FFh C00h CFFh
= 1101 Bank 13 00h CAN SFRs FFh D00h DFFh
= 1110 Bank 14 00h CAN SFRs FFh E00h EFFh
= 1111 Bank 15 00h CAN SFRs SFR FFh F00h F5Fh F60h FFFh
Access Bank
Access RAM Low 00h 5Fh
Access RAM High (SFRs) 60h FFh

When a = 0:
The BSR is ignored and the Access Bank is used.
The first 128 bytes are general purpose RAM (from Bank 0).
The second 128 bytes are Special Function Registers (from Bank 15).

When a = 1:
The BSR specifies the Bank used by the instruction.

Program Memory

- Return address stack (31-entries) for subroutine calls and interrupt processing.
- Reset vector (0000h) is the program-starting address after power-on or manual reset.
- High priority int. vec (0008h) is the starting address of this ISR with at most 16B.
- Low priority int. vec (0018h) ditto but without a restriction.
- The user program follows the low priority int. vector program.



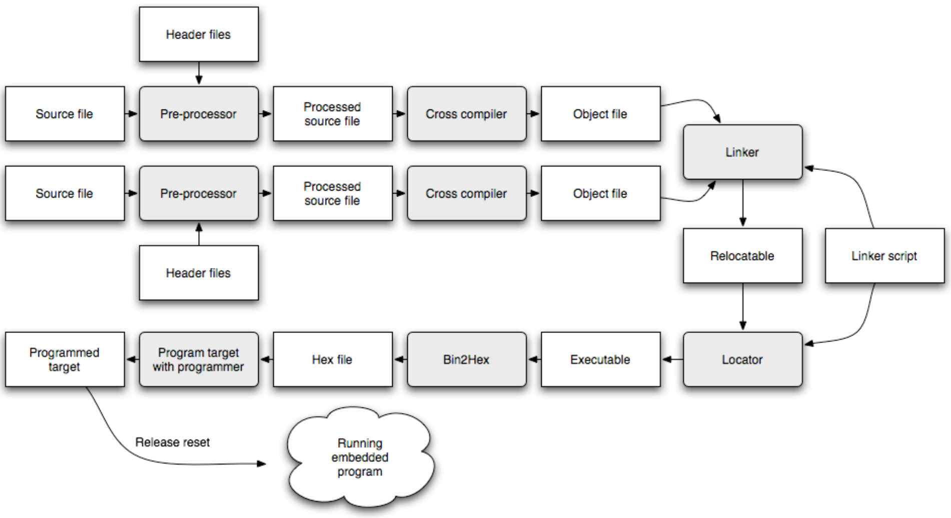
Further Processor Information

- It has a long list of CPU registers (see specification).
  - Not important when programming C, not irrelevant either.
  - For example STKPTR, INTCON*, STATUS
- PIC18 supports instruction pipelining with a depth of two steps
  - Instruction fetch
  - Instruction execute

The Programming Process

17

Overview of the Programming Process



Source file

```
#include <p18cxxx.h>
#define SHIFT_ME 3
#define LOOP_FOREVER() while(1);

void delay(unsigned int x) {
  while(x--);
}

void main (void) {
  unsigned int xx = 100%2 << SHIFT_ME;
  delay(xx);
  LOOP_FOREVER();
}
```

Pre-processor

- The pre-processor processes the source code before it continues with the compilation stage.
- The pre-processor
  - Resolves #define statements (constants, variable types, macros)
  - Concatenates #include files and source file into one large file
  - Processes #ifdef - #endif statements
  - Processes #if - #endif statements
- Specifically for embedded systems the pre-processor also processes vendor-specific directives (non-ANSI)
  - #pragma

Source File After Pre-Processing

… the p18cxx.h file …

```
void delay(unsigned int x) {
  while(x--);
}

void main (void) {
  unsigned int xx = 100%2 << 3;
  delay(xx);
  while(1);
}
```

Compiler

- The compiler turns source code into machine code packaged in object files.
- Common file format are object file format (COFF) or the extended linker format (ELF).
- A cross-compiler produces object files that will then be linked for the target instead of the computer running the compiler (compare Linux, embedded Linux, PIC18)
- Details about the compilation process and how the compiler works look at Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 2006.
- Practical approach in embedded systems:
  - **TURN OFF ALL OPTIMIZATION !!**
  - In MPLAB: -Ou- -Ot- -Ob- -Op- -Or- -Od- -Opa- -On-

Linker

- The linker performs the following
  - It combines object files by merging object file sections.
    - .text section for code
    - .data section for initialized global variables
    - .bss section for uninitialized global variables
  - It resolves all unresolved symbols.
    - External variables
    - Function calls
  - Reports errors about unresolved symbols.
  - Appends the start-up code (see next slide)
  - Provide symbolic debug information
- The linker produces a relocatable file. For standard operating systems with a dynamic loader, the processes is now finished - not so for embedded systems which need absolutely located binaries.

Startup Code

- Startup is a small fragment of assembly code that prepares the machine for executing a program written in a high-level language.
  - For C in Unix it is called crt1.o or crt0.S (assembly)
  - For PIC it is typically also an object file specified in the linker script.
- Tasks of the startup code
  - Disable all interrupts
  - Initialize stack pointers for software stack
  - Initialize *idata* sections
  - Zero all uninitialized data areas in data memory (ANSI standard)
  - Call loop: main(); goto loop;

Relocator

- The relocator converts a relocatable binary into an absolutely located binary.
- The relocator is guided by a linker script that specifies:
  - Program and data memory for the target part
  - Stack size and location
  - Logical sections used in the source code to place code and ata
- The relocator then produces an ‘executable’, that is ready for deployment on the target or for the simulator.

Linker File for MPLINK

- The linker directives fall into four basic categories. Since MPLINK combines the linker and relocator in one, there is no clean separation.
- Command line directives
  - LIBPATH: Search path for library and object files.
  - LKRPATH: Search path for linker command files.
  - FILES: Additional files to be linked.
  - INCLUDE: Additional linker command files to be included.
- Stack definition directive.
  - STACK SIZE=allocSize [RAM=memName]
    - Specifies the size and location of the software stack.

Linker File for MPLINK

- Memory region definition directives.
  - CODEPAGE NAME=memName START=addr END=addr [PROTECTED] [FILL=fillvalue]
    - Specifies a ROM directive that is used for program code, initialized data values, constant data values, and external memory
    - PROTECTED specifies that it can only by explicit request
    - Useful for reflashing on the fly.
  - DATABANK NAME=memName START=addr END=addr [PROTECTED]
    - Specifies a RAM directive that is used for volatile memory.
    - Useful for correlated data tables.

- Logical sections definition directives.
  - SECTION NAME=secName [ROM=memName | RAM=memName]
    - The code or data specified using the #pragma directive will then be located in the specified memory area.

Sample Linker File

```
LIBPATH .
FILES c018i.o
FILES clib.lib
FILES p18f8720.lib

CODEPAGE   NAME=vectors    START=0x0          END=0x29         PROTECTED
CODEPAGE   NAME=page       START=0x2A         END=0x1FFFF
CODEPAGE   NAME=eeprom     START=0x20000      END=0x1FFFFF     PROTECTED
CODEPAGE   NAME=idlocs     START=0x200000     END=0x200007     PROTECTED
CODEPAGE   NAME=config     START=0x300000     END=0x30000D     PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE     END=0x3FFFFF     PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000     END=0xF003FF     PROTECTED

ACCESSBANK NAME=accessram  START=0x0          END=0x5F
DATABANK   NAME=gpr0       START=0x60         END=0xFF
DATABANK   NAME=gpr1       START=0x100        END=0x1FF
…
DATABANK   NAME=gpr13      START=0xD00        END=0xDFF
DATABANK   NAME=gpr14      START=0xE00        END=0xEFF
ACCESSBANK NAME=accesssfr  START=0xF60        END=0xFFF        PROTECTED

SECTION    NAME=CONFIG     ROM=config
SECTION    NAME=STARTUP    ROM=vectors    // Reset and interrupt vectors
SECTION    NAME=PROG       ROM=page       // main application code space
SECTION    NAME=INTHAND    ROM=eeprom     // Interrupt handlers
SECTION    NAME=DATTBL     ROM=eeprom     // Data tables
STACK SIZE=0x100 RAM=gpr14
```

Map File after Linking and Relocating

- Open external file…

Bin2Hex

- The executable still has to be transferred to the target via a serial line (or even Ethernet with applicable boot loaders).
- For standard compliance, the binary is converted into an ASCII representation useful to PROM programmers and emulators.
- A number of standards exists, Intel HEX is widespread.
  - Each line in an Intel HEX file contains one HEX record.

Example Program after Bin2Hex

```
:020000040000FA
:0600000094EF00F0120075
:02002A000000D4
:04002C002A0EF66E34
:10003000000EF76E000EF86E00010900F550656FB6
:100040000900F550666F03E1656701D03DD00900F6
:10005000F550606F0900F550616F0900F550626F4F
:1000600009000900F550E96E0900F550EA6E090033
:1000700009000900F550636F0900F550646F09002D
:100080000900F6CF67F0F7CF68F0F8CF69F060C0ED
:10009000F6FF61C0F7FF62C0F8FF0001635302E1A1
:1000A000645307E00900F550EE6E6307F8E2640759
:1000B000F9D767C0F6FF68C0F7FF69C0F8FF000115
:0A00C0006507000E665BBFD7120053
:0600CA00D9CFE6FFE1CFF3
:1000D000D9FFFD0EDBCF02F0DB06FE0EDBCF03F017
:1000E00001E2DB060250031001E0F3D7E552E7CF4F
:1000F000D9FF1200D9CFE6FFE1CFD9FF020EE126EA
:10010000DE6ADD6ADECFE6FFDDCFE6FFDEDFE55249
:10011000E552FFD7020EE15C02E2E16AE552E16ED0
:08012000E552E7CFD9FF120000
:080128001BEE00F02BEE00F0CD
:10013000F86A019C16EC00F07AEC00F0FDD7120092
:00000001FF
```

MCC18 Compiler Extensions

Embedded Systems C Compilers

- Embedded systems developers need more control over the generated file than traditional C developers.
  - Access to assembly instructions for high-performance functions
  - Specify memory area for code and data
  - Extra functionality for saving memory
  - Define ISRs
  - Define chip configuration
- Every compiler provides different extensions.
- GCC is available for a small set of targets, but not for too many.

Data Types and Limits

Type	Size	Minimum	Maximum
`char`[1,2]	8 bits	-128	127
`signed char`	8 bits	-128	127
`unsigned char`	8 bits	0	255
`int`	16 bits	-32,768	32,767
`unsigned int`	16 bits	0	65,535
`short`	16 bits	-32,768	32,767
`unsigned short`	16 bits	0	65,535
`short long`	24 bits	-8,388,608	8,388,607
`unsigned short long`	24 bits	0	16,777,215
`long`	32 bits	-2,147,483,648	2,147,483,647
`unsigned long`	32 bits	0	4,294,967,295

Note 1: A plain *char* is signed by default.
2: A plain *char* may be unsigned by default via the `-k` command-line option.

Storage Classes

- Auto
  - An *auto* variable is stored in the software stack.
  - Enables basic reentrancy for functions.
  - The default for variables and parameters.
  - But, can be changed in the MPLAB settings to a different setting.
- Static
  - A *static* variable is allocated globally.
  - Slowly but surely eat up your memory.
- Register
  - Can be ignored, because PIC18 only has WREG.

Storage Classes

- Extern
  - Declares a variable that is defined somewhere else.
  - Useful when splitting software in multiple files.
  - Watch out for the type and storage qualifiers!
- Overlay
  - Allows more than one variable to occupy the same physical memory location.
  - Used to reduce stack and global memory requirements.
  - Only available for variables.
  - The compiler decides which variables share the same memory location by analyzing the code.

Overlay Example

Two simple functions:

```
int f (void) {
  overlay int x = 1;
  return x;
}

int g (void) {
  overlay int y = 2;
  return y;
}
```

Overlay works, *x* and *y* will share the same memory region.

Again the two functions:

```
int f (void) {
  overlay int x = 1;
  return x;
}

int g (void) {
  overlay int y = 2;
  y = f ();
  return y;
}
```

Overlay will not work, because of the dependency between f() and g().

Storage Qualifiers

- Const
  - Defines a constant value, i.e., the value cannot be changed at runtime.
  - The value is stored in the program memory.
- [Default]
  - Defines a variable whose value can change at runtime.
  - The value is stored in the data memory.
- Volatile
  - Defines a variable whose value can change at runtime - anytime.
  - The value is stored in the data memory.
  - Turns off certain compiler optimizations.

Volatile Example

```
void do_i_exit (void) {
  volatile int dummy;

  do {
    dummy = -1;
  } while ( dummy == -1 );

}
```

Near/Far Qualifiers

- Near/far data memory objects
  - *Far* specifies that the object is anywhere in the data memory => requires a bank switching instruction.
  - *Near* specifies that it is in the access bank.
- Near/far program memory objects
  - *Far* specifies that the object is anywhere in the program memory.
  - Near specifies that it is within 64KB.
- The default storage qualifier for program and data memory objects is *far*.

Ram/Rom Qualifiers

- In the Harvard architecture, program and memory space are separated => require means to specify where to find look.
- Ram qualifier
  - *Ram* specifies that the object is located in the data memory.
  - It is the default for variables.
- Rom qualifier
  - *Rom* specifies that the object is located in the program memory.
  - Useful for constant data such as lookup tables.

MCC18 ANSI/ISO Divergences

- MPLAB C18 implements some optimizations that are not specified or differ from the ANSI/ISO C standard.
- Integer promotions
  - C18 will perform arithmetic at the size of the largest operand, even if both operands are smaller than an integer.

```
unsigned char j, k;
unsigned i;
j = 0x79;
k = 0x87;
i = j+k;
```

```
#define X 0x20
#define Y 0x5
#define Z (X)*(Y)

unsgined i;
i = Z;
```

i will be 0x0 instead of 0x100.

MCC18 ANSI/ISO Divergences

- Numeric constants
  - C18 allows specifying binary values using the *0b* prefix.
  - *0b0111001* = 0x39 = 57
- String constants
  - Strings are typically stored in the program memory.
  - Usual qualifiers are: *const rom char []*
  - Two ways to declare string arrays

```
rom const char table[][20] = { "string 1", "string 2",
                               "string 3", "string 4" };      80B
rom const char *rom table2[] = { "string 1", "string 2",
                               "string 3", "string 4" };      44B
```

Copying Data between ROM & RAM

- Pointers to data memory and program memory are incompatible!
- A data memory pointer cannot be passed as a program memory pointer and vice versa.
- Copying between data and program memory looks like this:

```
void str2ram(static char *dest, static char rom *src)
{
  while ((*dest++ = *src++) != '\0');
}
```

Inline Assembly

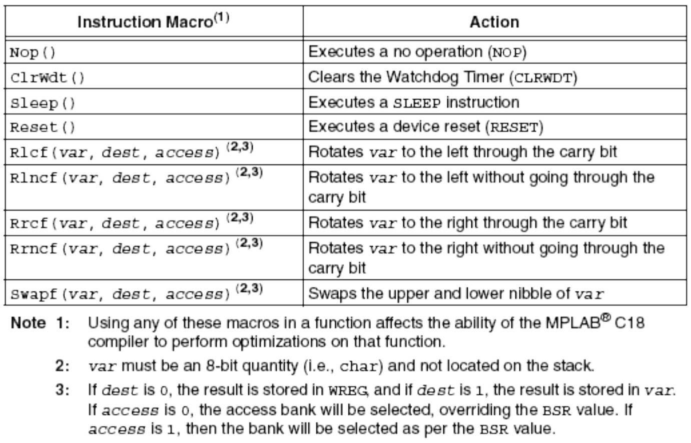
- An assembly section starts with _*asm* and ends with _*endasm*.

```
_asm
  nop
_endasm
```

- Useful for optimization and implanting explicit code in the program (e.g., for traces or benchmarks).
- Should be kept to a minimum, because it turns off compiler optimization.

Access to Assembly Instructions

- Assembly provides instructions that are not typically accessible from the high-level language (e.g., swap upper and lower nibble, nop)



Instruction Macro[1]	Action
`Nop()`	Executes a no operation (`NOP`)
`ClrWdt()`	Clears the Watchdog Timer (`CLRWDT`)
`Sleep()`	Executes a `SLEEP` instruction
`Reset()`	Executes a device reset (`RESET`)
`Rlcf(var, dest, access)` [2,3]	Rotates `var` to the left through the carry bit
`Rlncf(var, dest, access)` [2,3]	Rotates `var` to the left without going through the carry bit
`Rrcf(var, dest, access)` [2,3]	Rotates `var` to the right through the carry bit
`Rrncf(var, dest, access)` [2,3]	Rotates `var` to the right without going through the carry bit
`Swapf(var, dest, access)` [2,3]	Swaps the upper and lower nibble of `var`

Note 1: Using any of these macros in a function affects the ability of the MPLAB® C18 compiler to perform optimizations on that function.

2: `var` must be an 8-bit quantity (i.e., `char`) and not located on the stack.

3: If *dest* is `0`, the result is stored in `WREG`, and if *dest* is `1`, the result is stored in `var`. If *access* is `0`, the access bank will be selected, overriding the `BSR` value. If *access* is `1`, then the bank will be selected as per the `BSR` value.

#pragma

- The *#pragma* statement is used to manage
  - Program memory with *#pragma code*, *#pragma romdata*
  - Data memory with *#pragma udata, #pragma idata*
  - Interrupt functions with *#pragma interrupt*
  - Configuration settings with *#pragma config*

- Program memory
  - *#pragma code [overlay] [section-name [=address]]*
    - Allows placing code at a specific location in the program memory.
    - `#pragma code uart_int_service = 0x08`
    - Overlay tells the compiler to try and overlay as many sections of the specified functions as possible.

#pragma

- Program memory
  - *#pragma romdata [sectionname [=address]]*
    - Allows to place the data following the #pragma in the program memory.
    - Useful for correlated lookup tables that can then be absolutely address from the program code (*table2_data=table1_data[i]+offset*).

- Data memory
  - *#pragma udata [attribute-list] [sectionname [=address]]*
    - Specifies a location for the following statically allocated uninitialized data (udata).
    - Per default, all global variables without initial value are placed in udata.
  - *#pragma idata [attribute-list] [sectionname [=address]]*
    - Similar to udata, but for statically allocated initialized data, only.
    - Useful for 256B bank restriction.
  - Attribute *access* and *overlay*
    - Allows placing a specific section into the access region of the data memory (=>ACCESSBANK)
    - Must be declared with a near keyword.

#pragma

- Interrupt service routines
  - Interrupt service routines preempt the current execution. After finishing the ISR complete, the execution resumes. (=> context switch)
  - The ISR saves a minimal context of WREG, BSR, STATUS, etc.
  - Interrupt functions have a separate temporary sections in memory that are not overlaid with other sections (see the .map file).

  - *#pragma interrupt <fname> [tmpname] [save=<savelist>] [nosave=<nosavelist]*
  - *#pragma interruptlow <fname> [tmpname] [save=<savelist>] [nosave=<nosavelist]*

#pragma

- Define variable locations
  - *#pragma varlocate bank <variable-name>*
  - *#pragma varlocate "section-name" <variable-name>*
  - Useful to location variables in specific banks for performance reasons.

```
// ** place c1 into bank 1
#pragma varlocate 1 c1
extern signed char c1;

// ** place c2 into bank 1
#pragma varlocate 1 c2
extern signed char c2;

void main (void) {
  c1 += 5;
  /* No MOVLB instruction needs to be generated here. */
  c2 += 5;
}
```

#pragma

- Chip configuration
  - *#pragma config <settings-list>*
  - Allows specifying processor-specific configuration settings.
  - E.g.,
    - #pragma config WDT = ON, WDTPS = 128
    - #pragma config OSC = HS

Debugging and Emulation

Introduction Debugging

- Restrict the introduction of untested and flawed software (real programmer use *gcc -x c - << EOF)*.
- Identify and isolate bugs.

- The standard mechanism for debugging are:
  - printf
    - Board must be connected via the serial line.
    - Output must be redirected to the UART.
  - LED blinking
    - It shows that a certain code line has been executed.
    - It shows an specific error status (have fun with Morse).
    - In case of critical errors, flash all LEDs.
  - Breakpoints
    - Somewhat like LEDs but allow inspecting the chip state.

Simulation

- High-level language simulation
  - Test parts of the software without I/O or with simulated I/O.
  - Compile the binary not for the target but for the workstation.
  - Bind the target with specific libraries to allow scripted I/O.
  - Problem: what if the libraries do not behave as the real world?

- Low-level simulation
  - Compile the binary for the target.
  - Load the executable in a CPU simulator and run.
  - CPU simulators are widespread, some with I/O support.
    - MPLAB supports virtually all PIC chips.
    - Check GCC and it's supported targets via GDB.
  - Level of complexity differs:
    - Solely CPU and memory simulation or also
    - Cache performance, memory usage, cycle count.

Onboard Debugging

- Instead of simulating on the workstation, run the software on the target.
- Special software support for debugging linked with the compiled object files.
- One way to do it (depends on the chip and the programmer):
  - Tweak the interrupt handler to provide your debugging features.
  - For a breakpoint flash the program memory and either insert a specific breakpoint instruction or an illegal instructions (traps).
  - Run the program and wait for the trap signals.
- JTAG is the standard, defined in 1985 as IEEE-Standard 1149.1.

Emulation

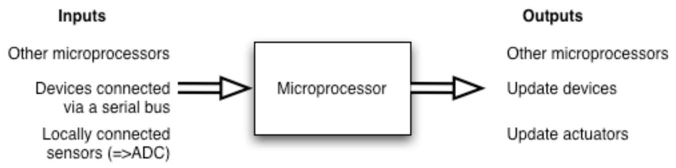
- A hardware device simulates the chip and interfaces with the target board.
- A very costly solution, but the way to go for efficiency and effectiveness of the developer (compare 2,500.- for the ICE vs. 160.- for the ICD).
- For example the PIC ICE allows:
  - Debug your application on your own hardware in real time.
  - Debug with both hardware and software breakpoints.
  - Measure timing between events using the stopwatch or complex trigger.
  - Set breakpoints based on internal and/or external signals.
  - Monitor internal file registers.
  - Select the oscillator source in software.
  - Trace data bus activity and time stamp events.
  - Set complex triggers based on program and data bus events, and external inputs

Interrupt-based Input/Output Programming

57

Input/Output Programming

- I/O programming is the most important task in an embedded system:

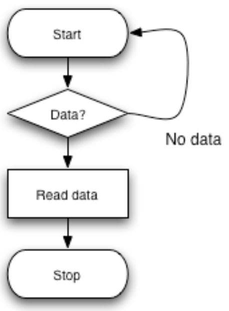


- Inputs can be
  - Random: I.e., have an unpredictable timing
  - Periodic: I.e., have a known timing
  - Low rate
  - High rate

Polled I/O Programming

- Polled programming uses waiting loops:

```
char serial_input(void) {
  while ( (inportb(SERIAL_STATUS_PORT)
            & RX_READY) == 0 ) {
    // do nothing, just idle
  }
  return inportb(SERIAL_DATA_PORT);
}
```

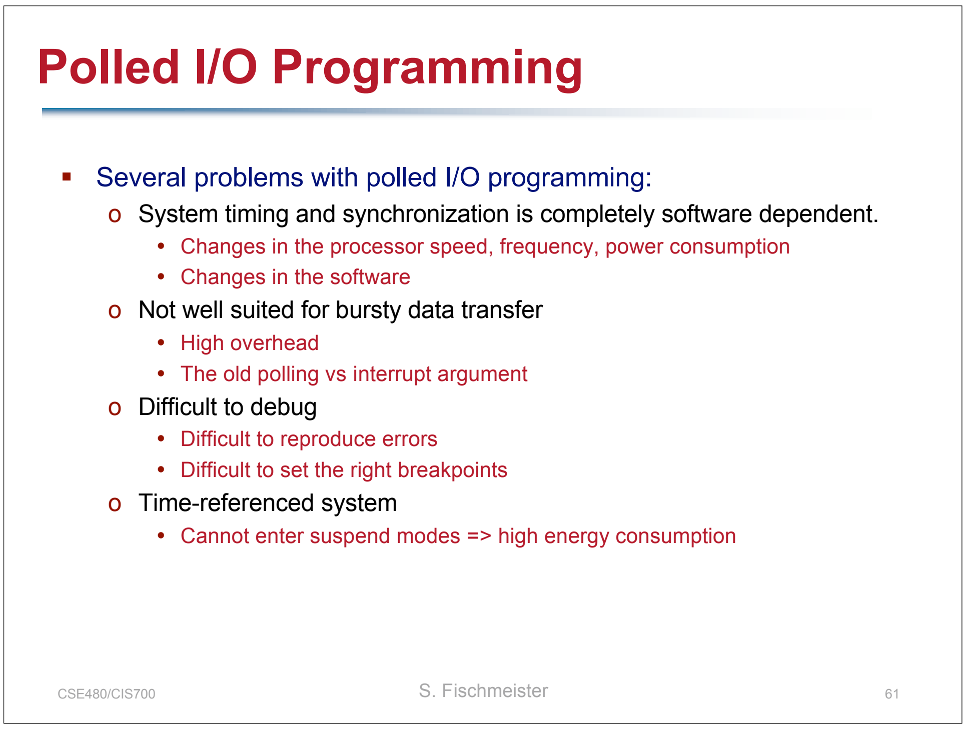


- The time required to execute one invocation determines the minimum time per transfer and thus the maximum data rate.
- Latency is unpredictable, because there's no guarantee on the function's execution.

Estimating Data-Transfer Rate

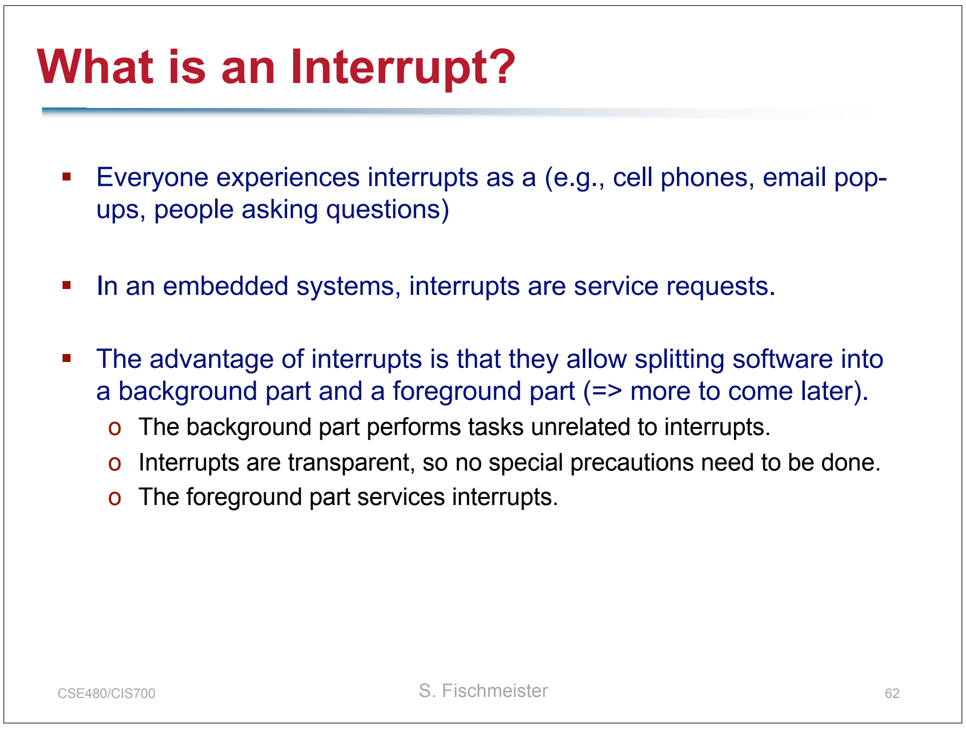
			Instruction Bytes Opcode	Instruction Bytes Immediates	Stack Bytes	I/O Transfer
	MOV	DX,02FDh	1	2		
SI1:	IN	AL,DX	1			1
	TEST	AL,00000001E	1	1		
	JZ	SI1	1	1		
	MOV	DX,02F8h	1	2		
	IN	AL,DX	1			1
	MOVZX	EAX,AL	1			
	RET		1		4	
			8	6	4	
		Sum	**18**			

- Assume that (1) I/O device is ready, (2) all opcodes use single byte, (3) no page faults, banking, etc, and (4) perfectly aligned.
- On an Intel 386 with 4byte memory bus, a 60ns memory cycle time, and an I/O card connected via a 33MHz bus.
- Memory: ceil(18/4)*60ns = 0.3us
- Transfer rate: 0.3 + 2*0.03 = 0.36us/byte => 2.78MB/Sec



Polled I/O Programming

- Several problems with polled I/O programming:
  - System timing and synchronization is completely software dependent.
    - Changes in the processor speed, frequency, power consumption
    - Changes in the software
  - Not well suited for bursty data transfer
    - High overhead
    - The old polling vs interrupt argument
  - Difficult to debug
    - Difficult to reproduce errors
    - Difficult to set the right breakpoints
  - Time-referenced system
    - Cannot enter suspend modes => high energy consumption



What is an Interrupt?

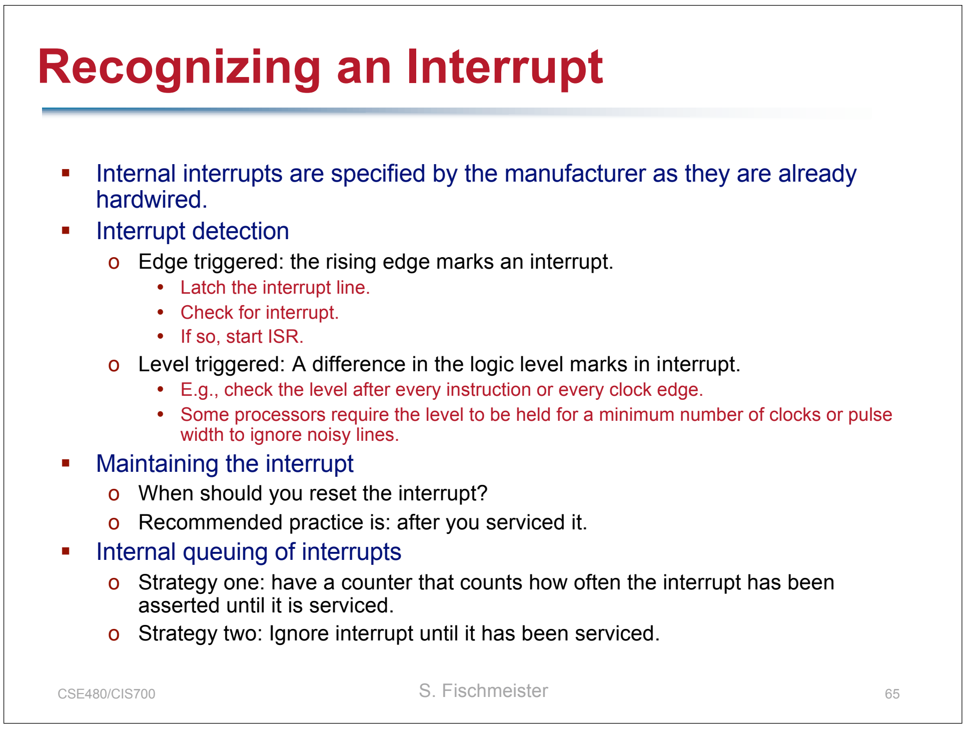
- Everyone experiences interrupts as a (e.g., cell phones, email pop-ups, people asking questions)
- In an embedded systems, interrupts are service requests.
- The advantage of interrupts is that they allow splitting software into a background part and a foreground part (=> more to come later).
  - The background part performs tasks unrelated to interrupts.
  - Interrupts are transparent, so no special precautions need to be done.
  - The foreground part services interrupts.

Interrupts

- Sources of interrupts are:
  - Internal interrupts generated by the on-chip peripherals such as serial or parallel ports, and timers.
  - External interrupts generated by peripherals connected to the processor.
  - Exceptions thrown by the processor.
  - Software interrupts
    - Useful to steer control flow in your application.
    - Are the source of a lot of evil, if not done right.
- Non-maskable interrupts (NMI)
  - Most interrupts can be turned off and on (=ignored).
  - Some cannot be turned off and on (=non-maskable interrupts):
    - Reset, watchdog timer, memory parity failure (=> restart machine).

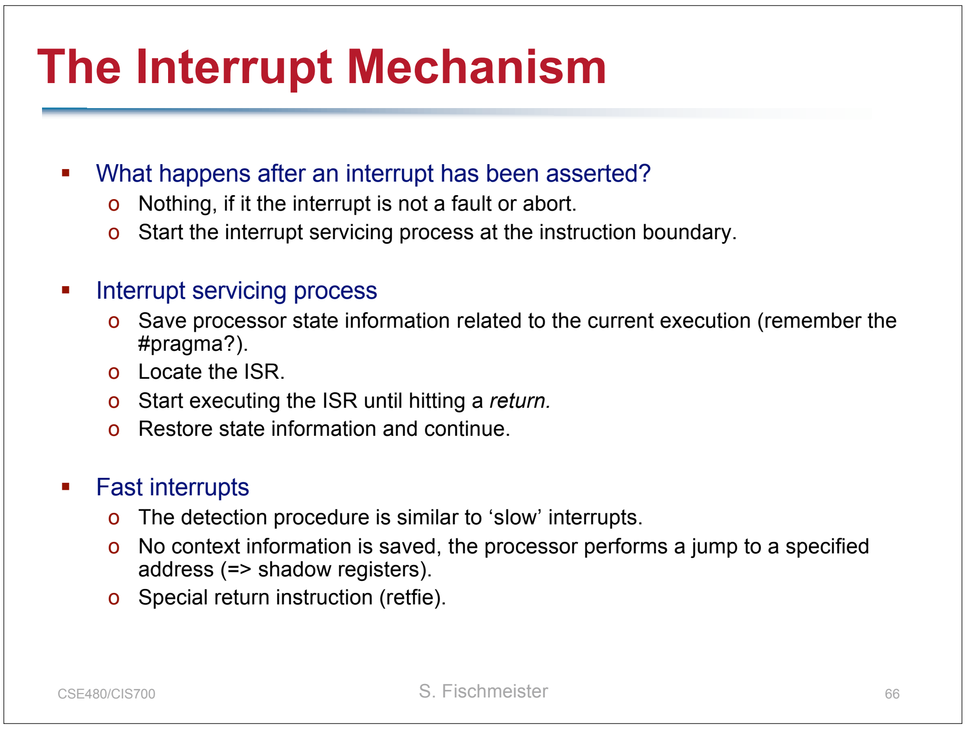
Exceptions

- Exceptions are broken down into traps, faults, and aborts.
- *Traps* are detected and serviced immediately **after** the execution of the instruction that caused the error condition (=> return address points to the next instruction)
- *Faults* are detected and serviced **before** execution of the instruction (=> return address points to the instruction causing the fault).
- *Aborts* are similar to faults, however, the machine state cannot be restored to the condition just prior to the exception.
- Exceptions detected by the Intel Processor are for example:
  - Faults: divide error, invalid opcode, no math coprocessor, segment not present
  - Traps: Debug, breakpoint, Overflow
  - Aborts: double fault, failure of internal cache



Recognizing an Interrupt

- Internal interrupts are specified by the manufacturer as they are already hardwired.
- Interrupt detection
  - Edge triggered: the rising edge marks an interrupt.
    - Latch the interrupt line.
    - Check for interrupt.
    - If so, start ISR.
  - Level triggered: A difference in the logic level marks in interrupt.
    - E.g., check the level after every instruction or every clock edge.
    - Some processors require the level to be held for a minimum number of clocks or pulse width to ignore noisy lines.
- Maintaining the interrupt
  - When should you reset the interrupt?
  - Recommended practice is: after you serviced it.
- Internal queuing of interrupts
  - Strategy one: have a counter that counts how often the interrupt has been asserted until it is serviced.
  - Strategy two: Ignore interrupt until it has been serviced.



The Interrupt Mechanism

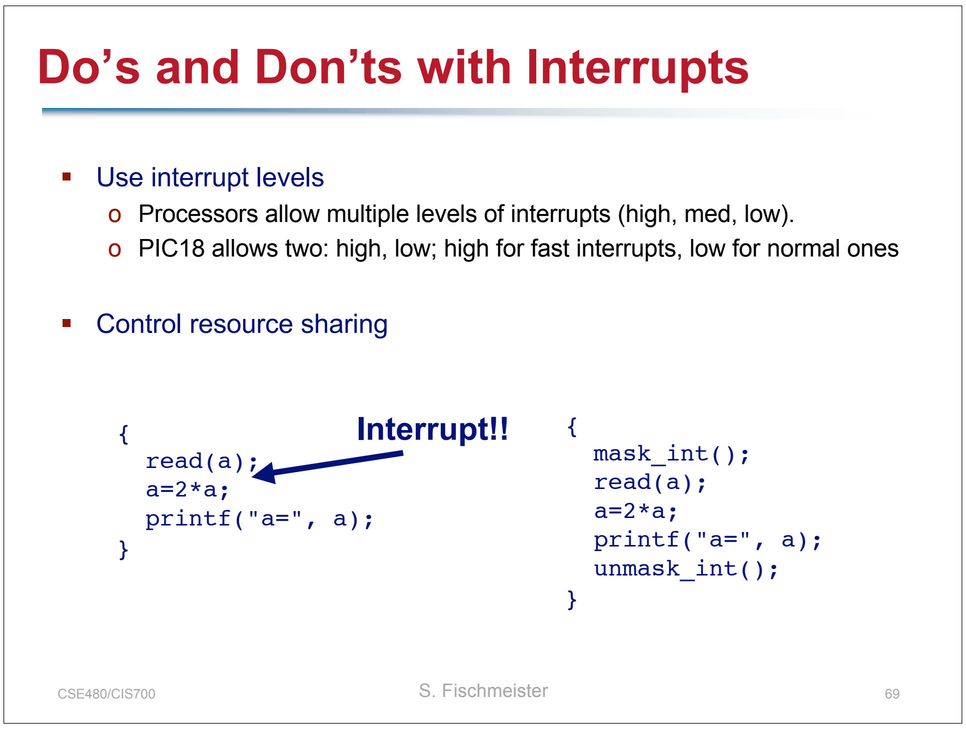
- What happens after an interrupt has been asserted?
  - Nothing, if it the interrupt is not a fault or abort.
  - Start the interrupt servicing process at the instruction boundary.
- Interrupt servicing process
  - Save processor state information related to the current execution (remember the #pragma?).
  - Locate the ISR.
  - Start executing the ISR until hitting a *return*.
  - Restore state information and continue.
- Fast interrupts
  - The detection procedure is similar to 'slow' interrupts.
  - No context information is saved, the processor performs a jump to a specified address (=> shadow registers).
  - Special return instruction (retfie).

Interrupt Latency

- Interrupt latency is the time it takes the processor from recognizing the interrupt until the start of the ISR execution.
- Elements that add to the interrupt latency:
  - Time taken to recognize the interrupt.
    - Reconsider multi sampling to lower faulty interrupt detection.
  - Time taken to complete the current instruction.
    - Low in RISC systems, potentially long in CISC systems.
    - With CISC some compilers restrict use to fast instructions to reduce interrupt latency (=> replace hardware instructions with software routines)
  - Time taken for the context switch.
  - Time taken to fetch the interrupt vector.
  - Time taken to start the ISR.
- For the microprocessor, computing the worst case interrupt latency is doable, but consider systems with caches, flexible interrupt vectors, large number of registers, deep pipelines, etc.

Do's and Don'ts with Interrupts

- Always expect the unexpected interrupt
  - Write a generic interrupt handler that saves the processor state for latter analysis (e.g., in the EEPROM).
- Interrupts are not negligible
  - Switching to the ISR costs time.
  - Too many interrupts will introduce a high switching overhead.
  - Too long ISRs will cause starvation for other computation tasks.
- Clear your interrupts
  - Leaving them set will have the processor ignore them.
- Beware false interrupts
  - Although hardware engineers give their best, they can occur.
  - Design the software accordingly.



Do’s and Don’ts with Interrupts

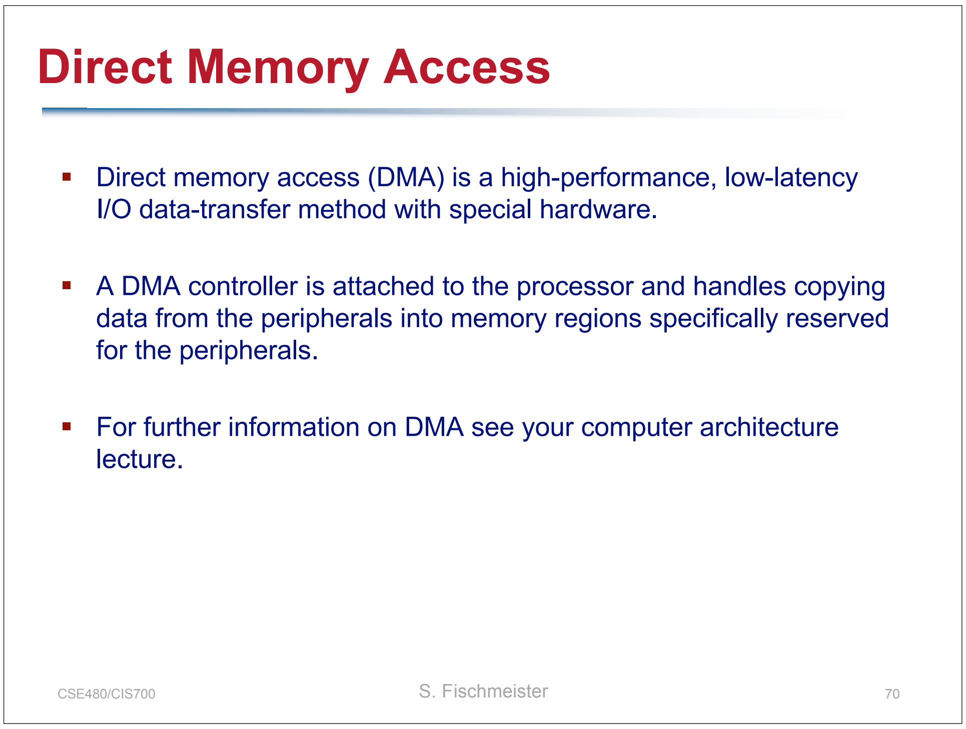
- Use interrupt levels
  - Processors allow multiple levels of interrupts (high, med, low).
  - PIC18 allows two: high, low; high for fast interrupts, low for normal ones
- Control resource sharing

```
{
  read(a);
  a=2*a;
  printf("a=", a);
}
```

Interrupt!!

```
{
  mask_int();
  read(a);
  a=2*a;
  printf("a=", a);
  unmask_int();
}
```

CSE480/CIS700 S. Fischmeister 69



Direct Memory Access

- Direct memory access (DMA) is a high-performance, low-latency I/O data-transfer method with special hardware.
- A DMA controller is attached to the processor and handles copying data from the peripherals into memory regions specifically reserved for the peripherals.
- For further information on DMA see your computer architecture lecture.

CSE480/CIS700 S. Fischmeister 70