Programming PIC Microcontrollers

Module: EE2A2 Embedded Microprocessor Systems

Lecturer: James Grimbleby
URL: http://www.personal.rdg.ac.uk/~stsgrimb/
email: j.b.grimbleby@reading.ac.uk

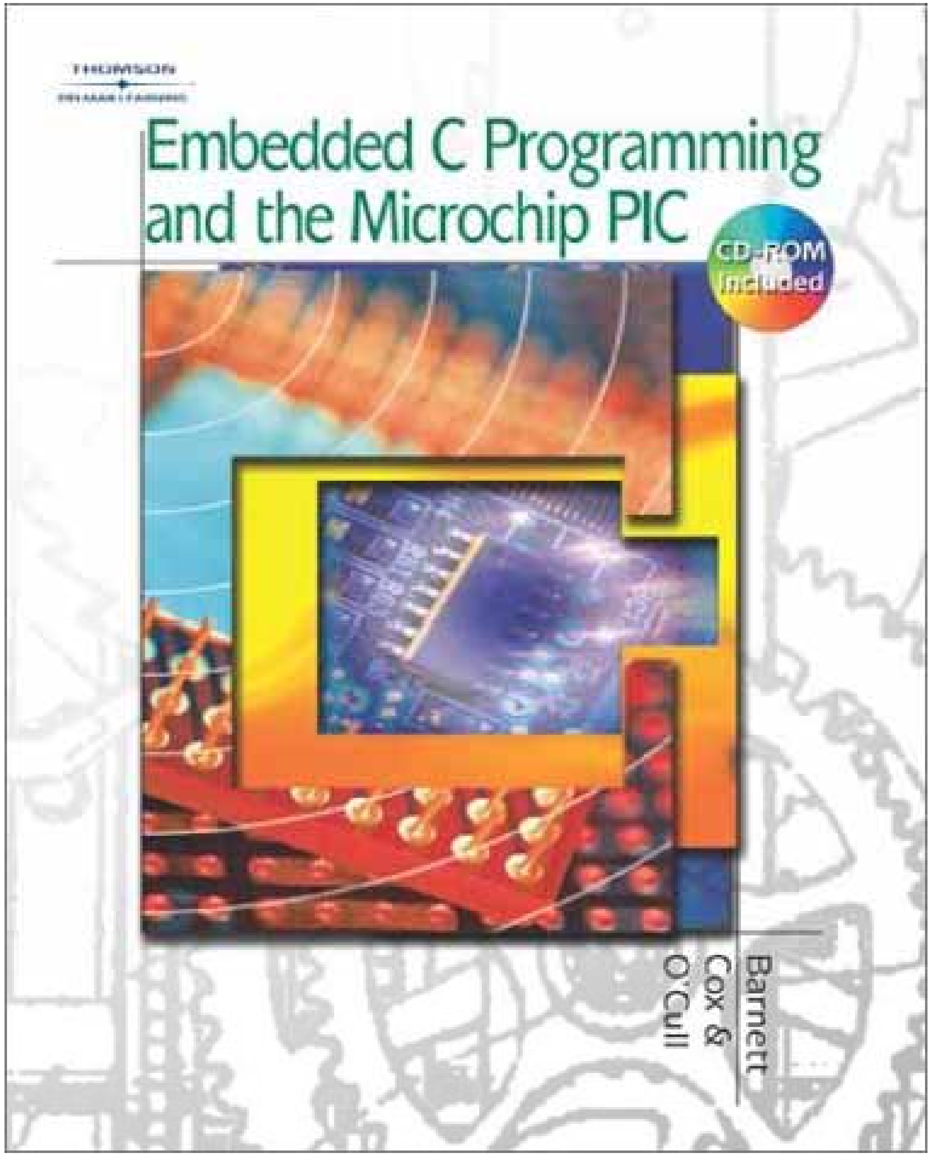
Number of Lectures: 5

Recommended text book:
R. Barnett, L O'Cull and S. Fox
Embedded C Programming and the Microchip PIC
Thomson (2004)
ISBN 1401837484

Programming PIC Microcontrollers

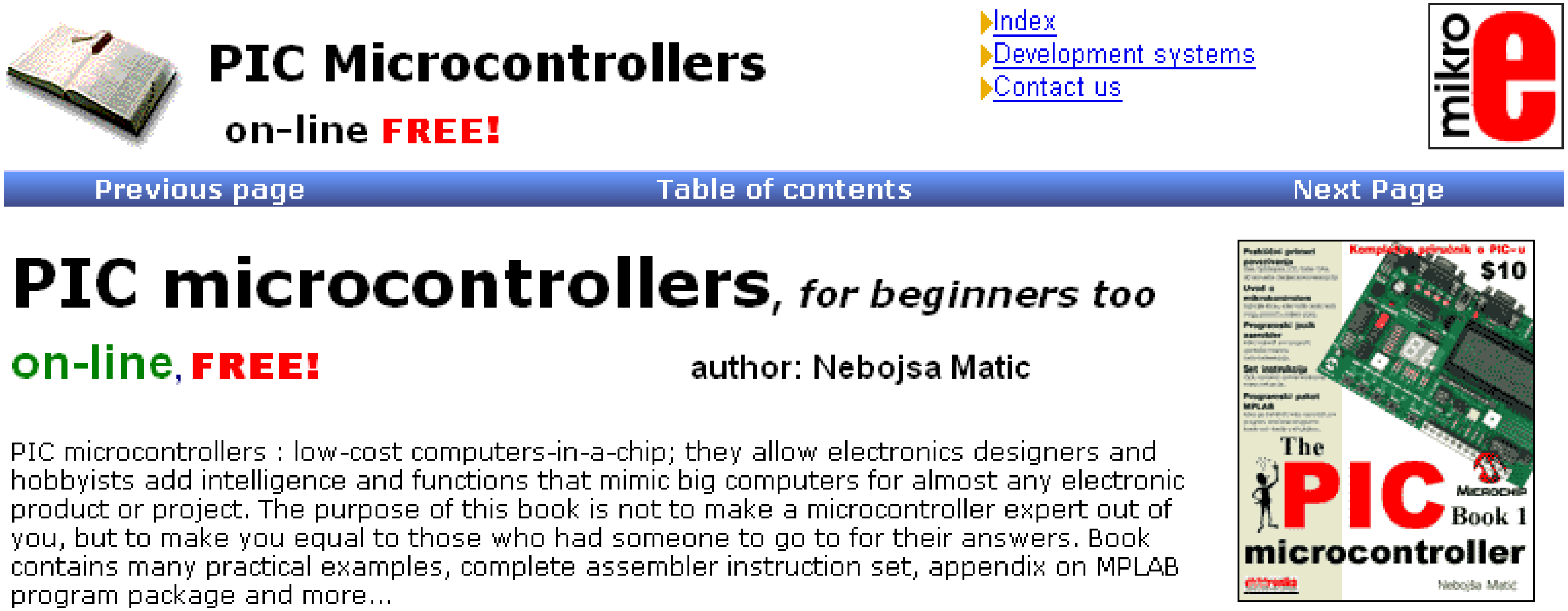
Recommended Text Book:

R. Barnett, L O’Cull and S. Fox
Embedded C Programming
and the Microchip PIC
Thomson (2004)
ISBN 1401837484
Price (Amazon) £47



Programming PIC Microcontrollers

On-line book describing PIC microcontrollers:



PIC Microcontrollers

on-line **FREE!**

- Index
- Development systems
- Contact us

Previous page   Table of contents   Next Page

PIC microcontrollers, ***for beginners too***

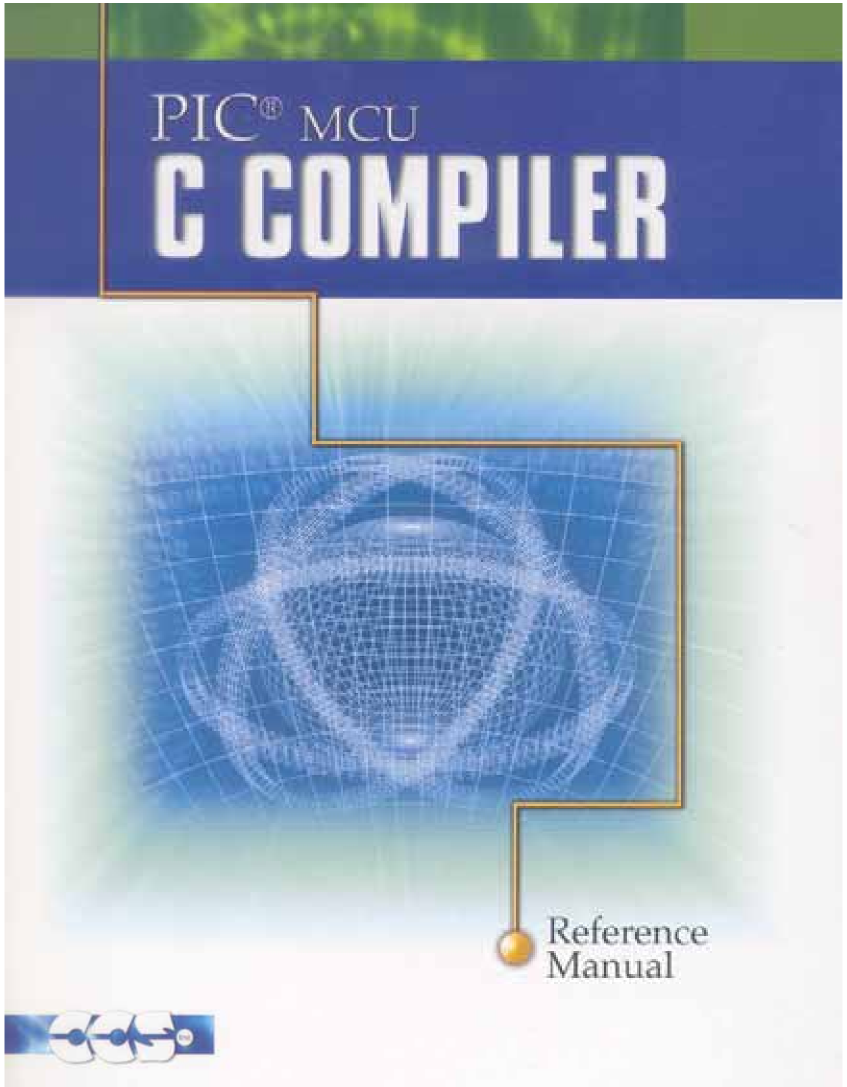
on-line, **FREE!**   author: Nebojsa Matic

PIC microcontrollers : low-cost computers-in-a-chip; they allow electronics designers and hobbyists add intelligence and functions that mimic big computers for almost any electronic product or project. The purpose of this book is not to make a microcontroller expert out of you, but to make you equal to those who had someone to go to for their answers. Book contains many practical examples, complete assembler instruction set, appendix on MPLAB program package and more...

http://www.mikroelektronika.co.yu/english/product/books/PICbook/0_Uvod.htm

Programming PIC Microcontrollers

Manual for CCS PIC C compiler:



http://www.ccsinfo.com/downloads/ccs_c_manual.pdf

Programming PIC Microcontrollers

University of Reading

This course is about programming PIC microcontrollers in C using the CCS PIC-C compiler

Topics covered include:

PIC architecture
PIC-specific limitations and extensions to C
Programming PIC hardware such as ports, ADC, timers, etc
Using software libraries

You should already be familiar with the C and C++ programming languages

Assessment

This unit will be assessed by a multiple-choice test

The multiple-choice test will last for 30 minutes, during which 20 questions must be answered

You will be permitted to bring your notebooks and the course notes into the test

The test will be held at the end of the Autumn term

The marks from this test will contribute to the overall mark for the module EE2A2

Multi-Choice Test Example

This question relates to the use of the CCS PIC C compiler.

A variable q is declared:

```
long int q;
```

q can take on any value in the range:

(a) -128 to +127

(b) 0 to 255

(c) -32768 to +32767

(d) 0 to 65535

(e) -2147483648 to + 2147483647

Answer: □

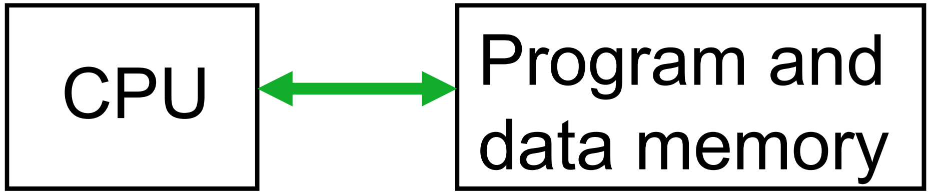
Lecture 1

PIC Architecture

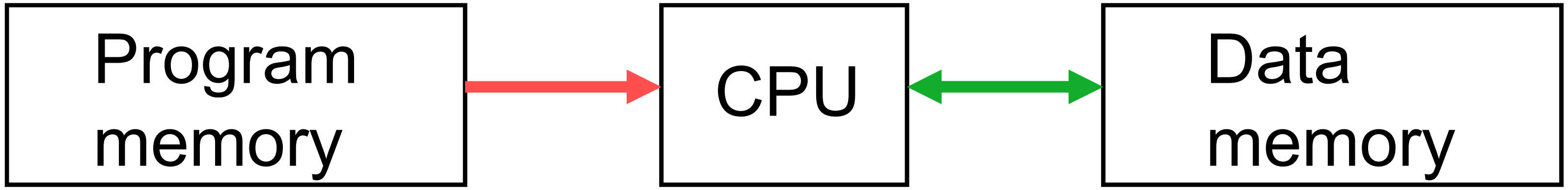
PIC Microcontroller Architecture

PICs use Harvard architecture and a RISC instruction set

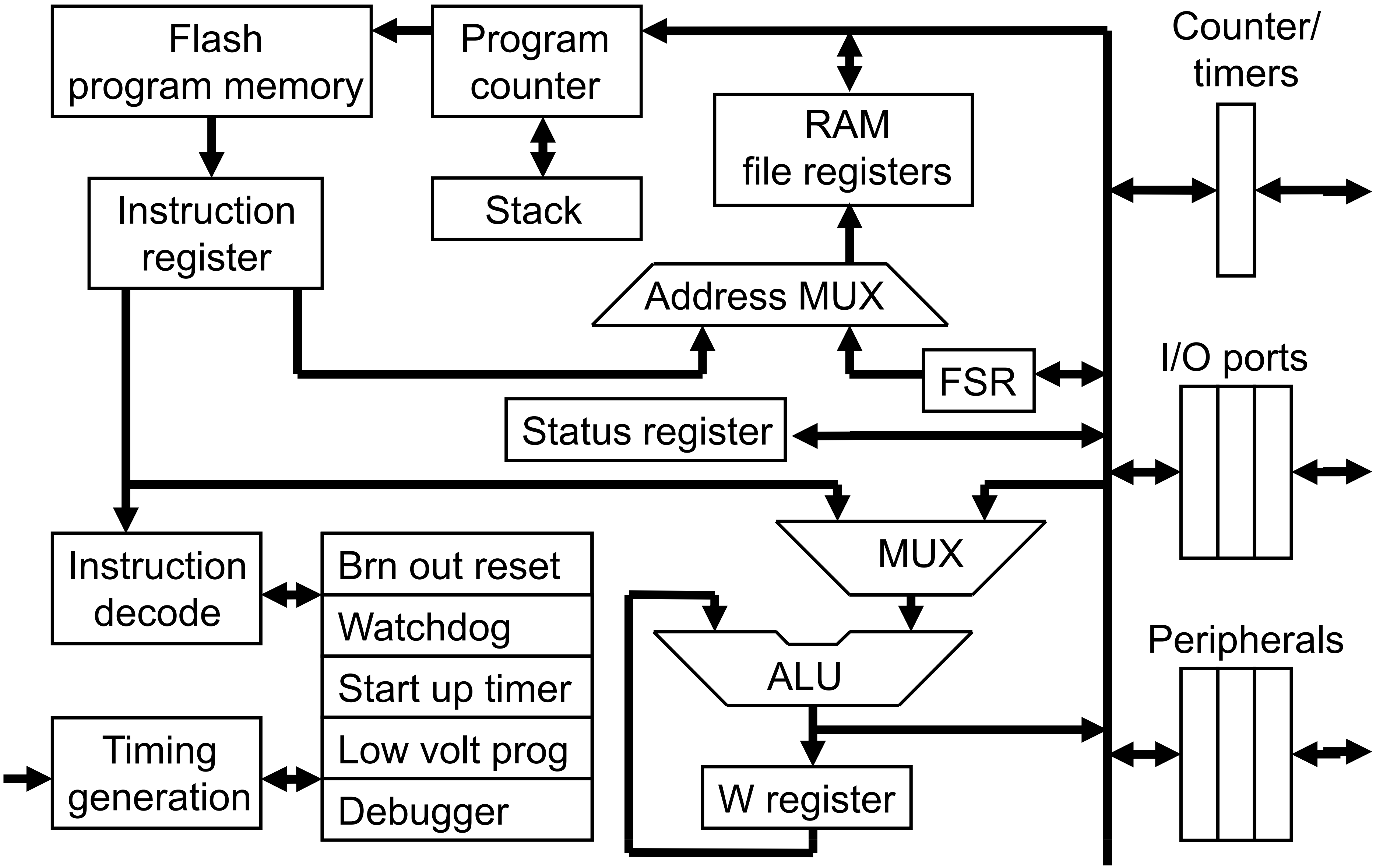
von Neuman Architecture:



Harvard Architecture:



PIC Microcontroller Architecture



PIC Microcontroller Peripherals

The 18F452 PIC has the following peripherals:

Data ports A (6-bit), B (8-bit), C (8-bit), D (8-bit), E (3-bit)

Timer/counter modules 0 (8-bit), 1 (16-bit), 2 (8-bit), 3 (16-bit)

CCP/PWM modules (2)

I^2C/SPI serial port

USART (RS-232, RS-485)

Analogue-to-digital converter (10-bit) with 10 way input multiplexer

EEPROM (256 byte)

Clock Generator

PICs use a fully static design so that any clock frequency up to the specified maximum can be used

There are 4 possible clock configurations:

- external clock (eg crystal oscillator module)
- self-oscillating with external crystal or ceramic resonator
- external or self-oscillating with phase-locked loop
- self-oscillating with external RC

In practice the choice will normally be a compromise between cost and clock speed or clock stability

Reset

A reset puts the PIC in a well-defined initial state so that the processor starts executing code from the first instruction

Resets can result from:

- external reset by MCLR pulled low
- reset on power-up
- reset by watchdog timer overflow
- reset on power supply brown-out

Reset can be used as a last resort for recovering from some catastrophic software event but all current data will be lost

Central Processing Unit

The CPU fetches instructions from memory, decodes them, and passes them to the ALU for execution

The arithmetic logic unit (ALU) is responsible for adding, subtracting, shifting and performing logical operations

The ALU operates in conjunction with:

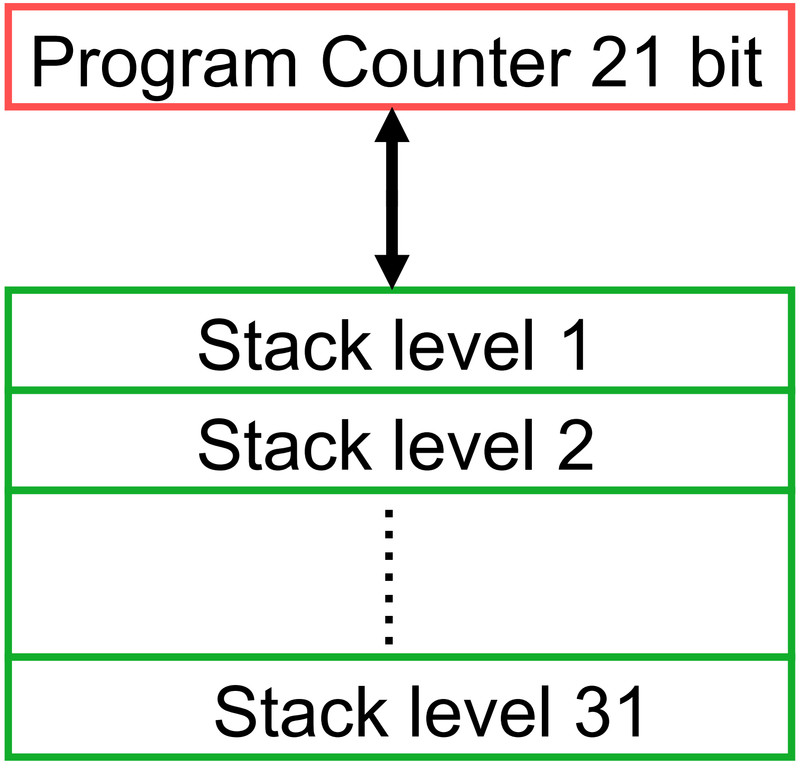
- a general-purpose register called the W register

- an f register that can be any location in data memory

- literals embedded in the instruction code

Memory Organisation - Stack

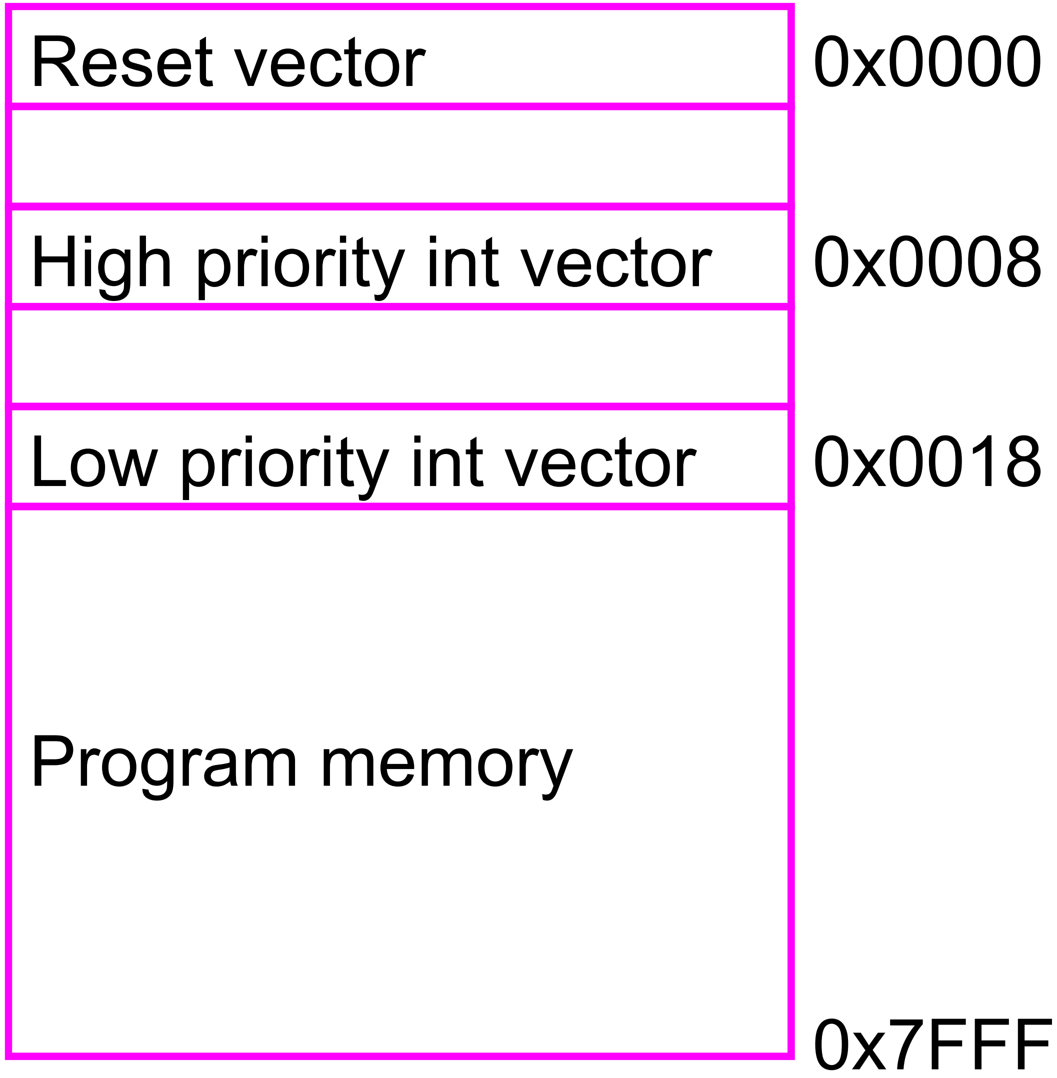
A 31-level stack stores the return address during interrupts and subroutine calls



Memory Organisation - Program

Program memory contains the Reset and Interrupt vectors

The PIC18F452 has 32k (0x8000) locations of program memory



Reset vector 0x0000

High priority int vector 0x0008

Low priority int vector 0x0018

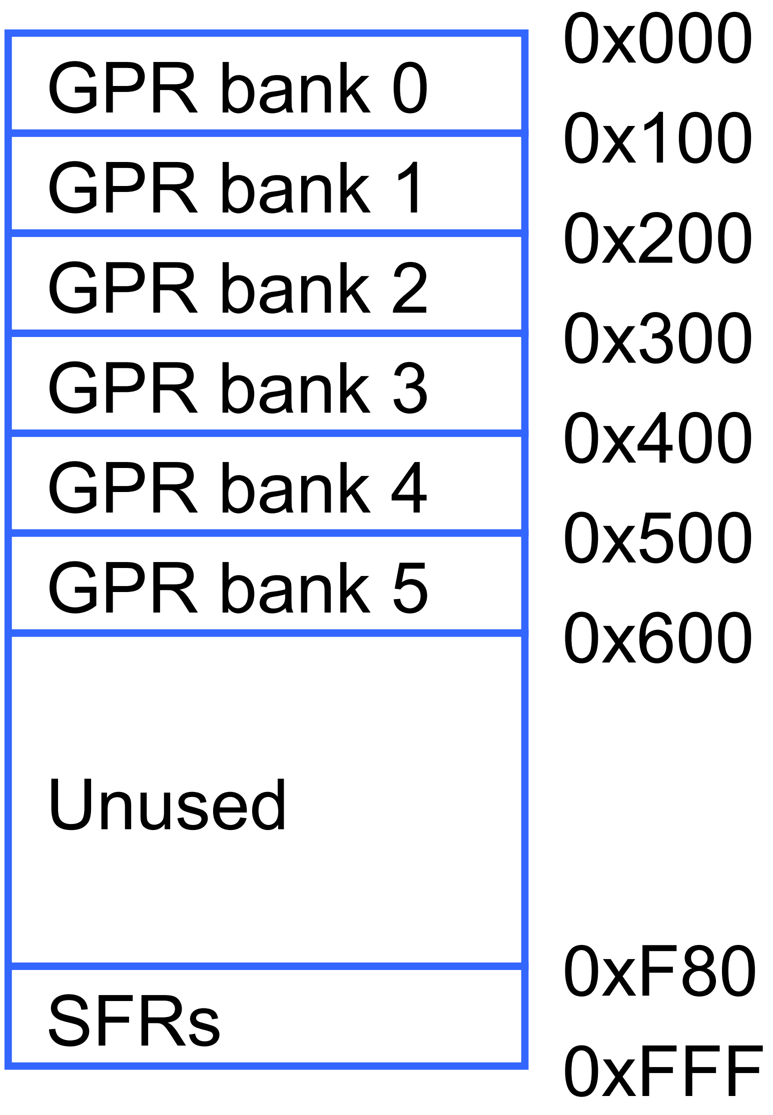
Program memory

0x7FFF

Memory Organisation - Data

Data memory contains general purpose registers (GPRs) and special function registers (SFRs)

The PIC18F452 has 1536 (0x600) locations of GPR data memory



GPR bank 0	0x000
GPR bank 1	0x100
GPR bank 2	0x200
GPR bank 3	0x300
GPR bank 4	0x400
GPR bank 5	0x500
Unused	0x600
SFRs	0xF80
	0xFFF

Memory Organisation – SFRs

University of Reading

The memory block 0xF80 to 0xFFF (128 locations) references special function registers (SFRs)

Some of the SFRs are shown here

Port A	0xF80	SPBRG	0xFAF
Port B	0xF81	⋮	
Port C	0xF82	Timer1L	0xFCE
Port D	0xF83	Timer1H	0xFCF
Port E	0xF84	⋮	
⋮		Timer0L	0xFD6
Tris A	0xF92	Timer0H	0xFD7
Tris B	0xF93	⋮	
Tris C	0xF94	Wreg	0xFE8
Tris D	0xF95	⋮	
Tris E	0xF96	StkPtr	0xFFC

PIC Instruction Set

The PIC instruction set has a small number of simple (RISC) instructions

PIC16 series: 35 instructions coded into 14 bits

PIC 18 series: 59 instructions coded into 16 bits

PIC 24 series: 71 instructions coded into 24 bits

Most instructions are executed in one instruction cycle which corresponds to 4 clock cycles

Thus a PIC operating at 40 MHz clock frequency will have an instruction rate of 10 MIPS.

PIC 18Fxxx Instruction Set

Most PIC 18Fxxx instructions occupy a single 16-bit program memory location

Each instruction consists of an opcode and one or more operands

The instruction set is highly orthogonal and can be partitioned:

- 31 byte-oriented file register operations
- 5 bit-oriented file register operations
- 23 control instructions
- 10 literal instructions
- 8 data memory – program memory operations

PIC 18Fxxx Instruction Set

Byte-oriented file register operations :

ADDWF	Add W and f: result in W or f
CLRF	Clear f
DECF	Decrement f
MOVF	Move contents of f to f or W

Bit-oriented file register operations:

BCF	Clear bit in f
BTFSC	Test bit in f; skip if clear

PIC 18Fxxx Instruction Set

Control instructions :

BRA	Branch unconditionally
CALL	Call subroutine (function)
RETURN	Return from subroutine (function)
BNZ	Branch if not zero

Literal instructions :

MOVLW	Move literal to W
ADDLW	Add literal to W

Data memory – program memory operations:

TBLRD*+	Table read with post-increment

Status Register

The 8-bit status register is set during arithmetic operations

-	-	-	N	OV	Z	DC	C

N Negative bit - result of arithmetic operation was negative
OV Overflow bit – overflow occurred for signed arithmetic
Z Zero bit - result of arithmetic operation was zero
DC Digit Carry bit – carry out from 4th low order bit of result
C Carry bit – carry out from most-significant bit of result

The bits of the status register can then be used in conditional branches, for example:

BNZ Branch if Not Zero
BOV Branch of OVerflow

Lecture 2

CCS Compiler

What is C ?

In 1970 a team at Bell Labs led by Brian Kernighan were developing the UNIX computer operating system

They required a high-level computer language for writing computer operating systems

Starting from an existing language called BCPL they developed C

C was used to write the next version of UNIX system software

UNIX eventually became the world's first portable operating system

What is C ?

C has now become a widely used professional language for various reasons:

It has high level constructs

It can handle low level activities

It produces efficient programs

It can be compiled on a wide variety of computers

The standard for C programs was originally the features set by Brian Kernighan

Later an international standard was developed: ANSI C (American National Standards Institute)

What is C++ ?

More recently another group at AT&T led by Bjarne Stroustrup developed C to reflect modern programming techniques

The new language was called C++

C++ has stronger type checking and supports object-oriented programming

C++ may be considered in several ways.:

An extension of C

A "data abstraction" improvement on C

A base for "object oriented" programming

Why Program PICs in C?

C is a portable language, requiring minimal modification when transferring programs from one processor to another

Programming in a high-level language rather than assembler allows programs to be developed much more rapidly

Typically a program which takes a few weeks in assembler can be written in C in a few days

Code efficiency of compiled C programs is typically 80% of well-written assembler programs

The related language C++ is too complex for use with the present generation of PICs

CCS PIC Compiler

A compiler converts a high-level language program to machine instructions for the target processor

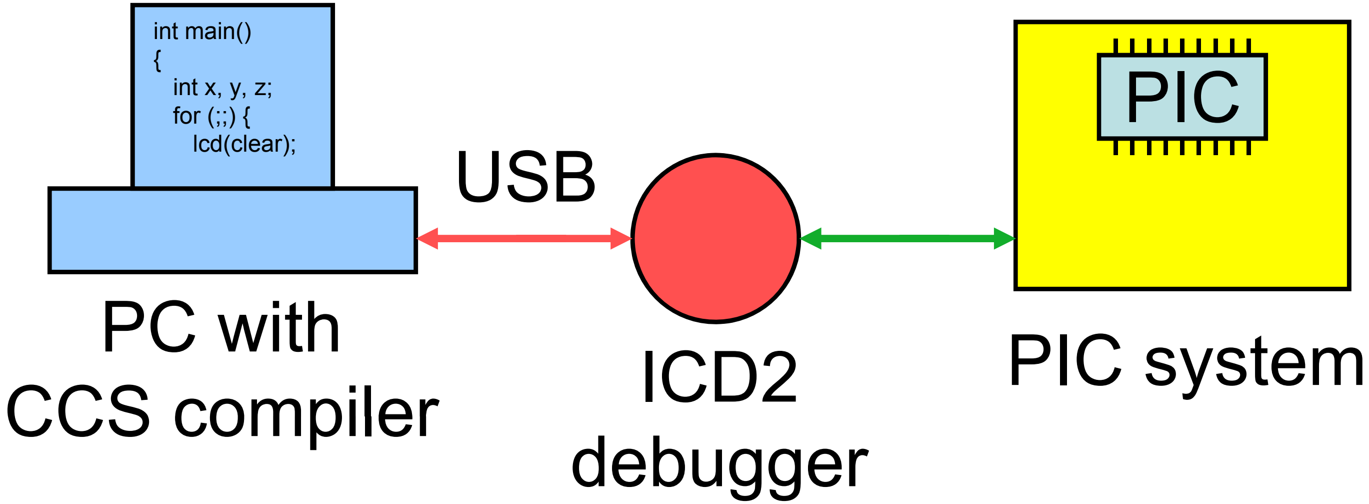
A cross-compiler is a compiler that runs on a processor (usually a PC) that is different from the target processor

Most embedded systems are now programmed using the C/C++ language

Several C compilers are available that target Microchip PICs, for example HiTech, Microchip and CCS

The PIC programming laboratory at Reading is equipped with the CCS cross-compiler

CCS PIC Compiler



```
int main()
{
  int x, y, z;
  for (;;) {
    lcd(clear);
```

USB

PIC

PC with
CCS compiler

ICD2
debugger

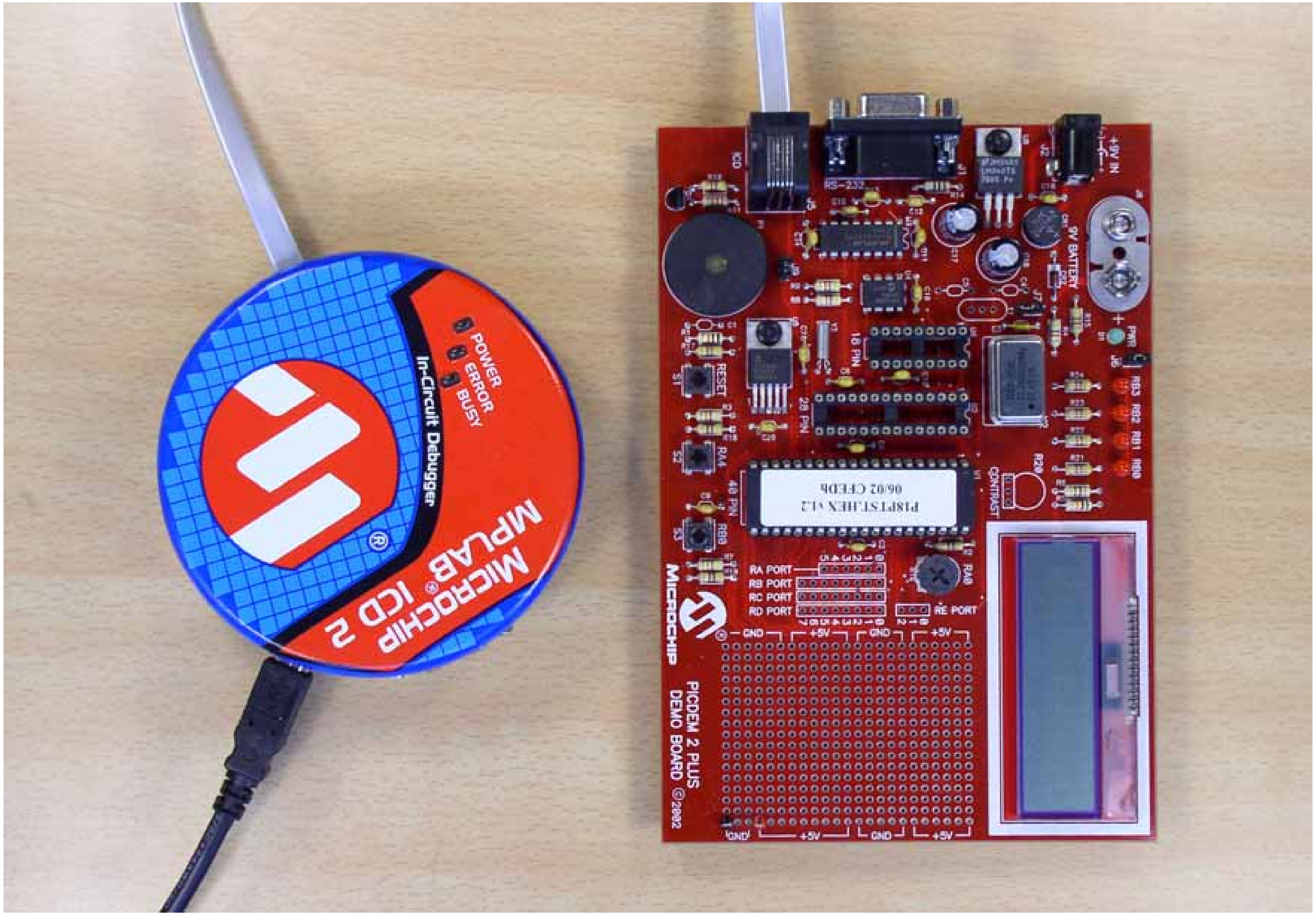
PIC system

Programs are edited and compiled to PIC machine instructions on a PC

PIC machine instructions are uploaded from PC to PIC system via the ICD2 debugger

Code is executed on the PIC system and can be debugged (break points, inspect variables, single step etc.) using PC

CCS PIC Compiler



CCS PIC Compiler

The CCS compiler comes with an integral syntax-aware editor

CCS C is standard C plus limited support for reference parameters in functions

PIC-specific pre-processor directives are provided in addition to the standard directives (#include, #define etc):

`#inline`	implement the following function inline
`#priority`	set priority of interrupts

Additional functions supporting PIC hardware are provided:

`output_low()`	set an I/O port bit low
`delay_us()`	delay by a specified number of μs

CCS PIC Compiler Data Types

PICs are optimised for processing single bits or 8-bit words, and this is reflected the CCS compiler word sizes:

short int (or int1)	1 bit	0 or 1
int (or int8)	8 bit	0 to 255
long int (or int16)	16 bit	0 to 65535
int32	32 bit	0 to 4294967295
char	8 bit	0 to 255
float	32 bit	$\pm 3\times10^{-38}$ to $\pm 3\times10^{+38}$

Contrary to the C standard, CCS C integers are by default unsigned

CCS PIC Compiler Data Types

In CCS C it is necessary to use the signed qualifier if signed integer are required:

short int	1 bit	0 or 1
signed int	8 bit	-128 to +127
signed long int	16 bit	-32768 to +32767
signed int32	32 bit	-2147M to +2147M
char	8 bit	0 to 255
float	32 bit	$\pm 3\times10^{-38}$ to $\pm 3\times10^{+38}$

It is not appropriate to use the signed qualifier with char or short int, and floats are signed by default

Constants

Constants can be specified in either decimal, octal, hexadecimal or binary, or as a special character:

123	Decimal
0123	Octal
0x123	Hex
0b010010	Binary
'x'	Character
'\010'	Octal character
'\0xA5’	Hex character

'\n'	Line Feed
'\r'	Return Feed
'\t'	TAB
'\b'	Backspace
'\f'	Form Feed
'\a'	Bell
'\v'	Vertical Space
'\?'	Question Mark
'\''	Single Quote
'\"'	Double Quote
'\\'	A Single Backslash

CCS PIC Compiler Data Types

In CCS C a short int is effectively a boolean variable

To make programs more readable it is a helpful to make use of the definitions (already in the device definition files):

```
#define boolean short int
#define false 0
#define true 1
```

Now it is possible to declare boolean variables:

```
boolean finished = true;
```

The standard boolean operators (||, &&, ! etc) can be used with these variables

Multi-Precision Operations

It is often necessary to process data words that are larger than can be operated on by a single instruction

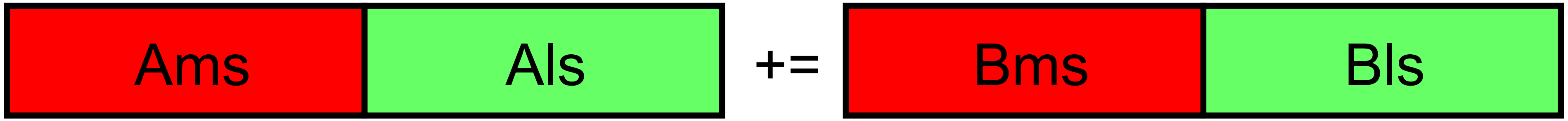
PIC instructions only operate on 8-bit words

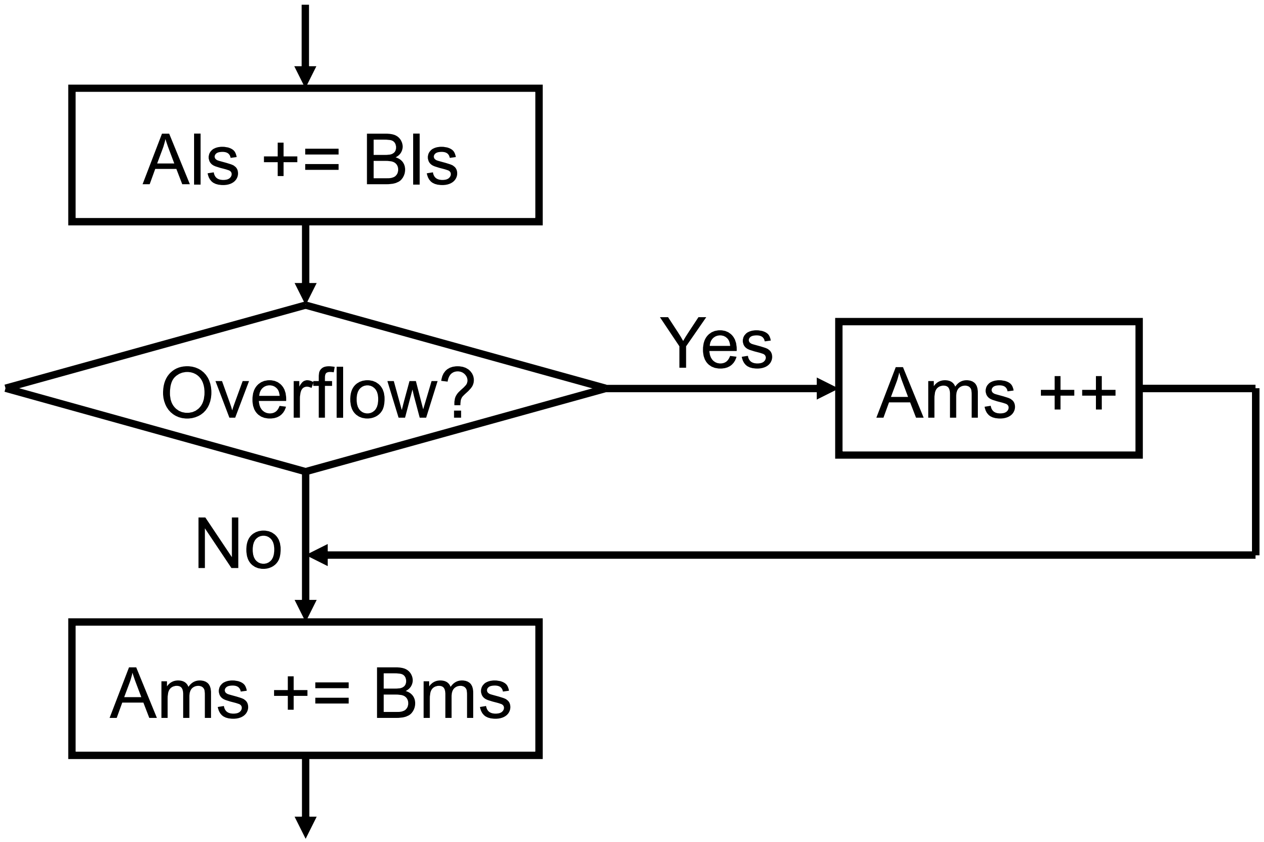
Multi-precision arithmetic uses a sequence of basic instructions on existing data types

In CCS C the long int (16 bit) and int32 (32 bit) types are processed using multi-precision arithmetic

This is much more expensive in time and code size than single instructions

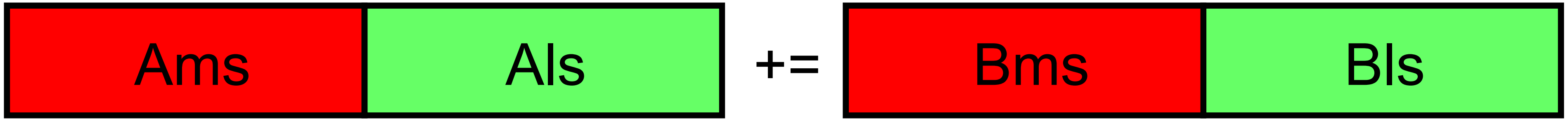
Multi-Precision Operations





Multi-Precision Operations

16-bit addition using 8-bit operations:

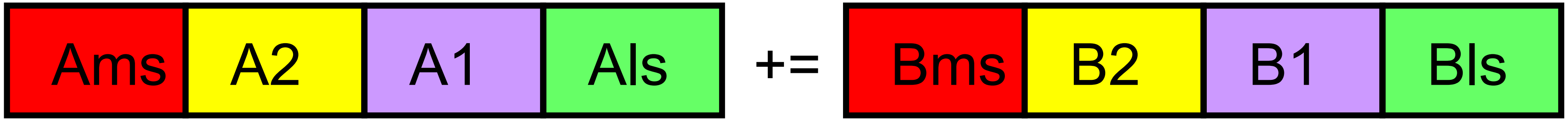


Ams Als += Bms Bls

```
MOVF Bls, W
ADDWF Als
MOVF Bms, W
ADDWFC Ams
```

Multi-Precision Operations

32-bit addition using 8-bit operations:



Ams A2 A1 Als += Bms B2 B1 Bls

```
MOVF  Bls, W
ADDWF Als
MOVF B1, W
ADDWFC A1
MOVF B2, W
ADDWFC A2
MOVF Bms, W
ADDWFC Ams
```

Reference Parameters

CCS C provides C++ like reference parameters to functions:

Traditional C:

```
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int j = 5, k = 8;
swap(&j, &k);
```

CCS C (C++):

```
void swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

int j = 5, k = 8;
swap(j, k);
```

Built-In Functions

University of Reading

RS-232 I/O:

getc()
putc()
fgetc()
gets()
puts()
fgets()
fputc()
fputs()
printf()
kbhit()
fprintf()
set_uart_speed()
perror()
assert()
getchar()
putchar()
setup_uart()

SPI two wire I/O:

read_bank()
setup_spi()
spi_read()
spi_write()
spi_data_is_in()

Discrete I/O:

output_low()
output_high()
output_float()
output_bit()
input()
output_X()
output_toggle()
input_state()
input_X()
port_b_pullups()
set_tris_X()

Built-In Functions

Parallel Slave I/O:

```
setup_psp()
psp_input_full()
psp_output_full()
psp_overflow()
```

I^{2}C I/O

```
i2c_start()
i2c_stop()
i2C_read
i2c_write()
i2c_poll()
```

Processor control:

```
sleep()
reset_cpu()
restart_cause()
disable_interrupts()
enable_interrupts()
ext_int_edge()
read_bank()
write_bank()
label_address()
goto_address()
getenv()
clear_interrupts
setup_oscillator()
```

Built-In Functions

Bit/Byte Manipulation:

```
shift_right()
shift_left()
rotate_right()
rotate_left()
bit_clear()
bit_set()
bit_test()
swap()
make8()
make16()
make32()
```

Standard C Math:

```
abs()
acos()
asin()
atan()
ceil()
cos()
exp()
floor()
labs()
sinh()
log()
log10()
pow()
sin()
cosh()
tanh()
```

```
fabs()
fmod()
atan2()
frexp()
ldexp()
modf()
sqrt()
tan()
div()
ldiv()
```

Built-In Functions

Standard C Char:

```
atoi()
atoi32()
atol()
atof()
tolower()
toupper()
isalnum()
isalpha()
isamoung()
isdigit()
islower()
isspace()
isupper()
isxdigit()
strlen()
strcpy()
strncpy()
strcmp()
stricmp()
strncmp()
strcat()
strstr()
strchr()
strrchr()
isgraph()
iscntrl()
strtok()
strspn()
strcspn()
strpbrk()
strlwr()
sprintf()
isprint()
strtod()
strtol()
strtoul()
strncat()
strcoll()
strxfrm()
```

Built-In Functions

A/D Conversion:

```
setup_vref()
setup_adc_ports()
setup_adc()
set_adc_channel()
read_adc()
```

Analog Compare:

```
setup_comparator()
```

Timers:

```
setup_timer_X()
set_timer_X()
get_timer_X()
setup_counters()
setup_wdt()
restart_wdt()
```

Standard C memory:

```
memset()
memcpy()
offsetof()
offsetofbit()
malloc()
calloc()
free()
realloc()
memmove()
memcmp()
memchr()
```

Built-In Functions

Capture/Compare/PWM:

```
setup_ccpX()
set_pwmX_duty()
setup_power_pwm()
setup_power_pwm_pins()
set_power_pwmx_duty()
set_power_pwm_override()
```

Delays:

```
delay_us()
delay_ms()
delay_cycles()
```

Internal EEPROM:

```
read_eeprom()
write_eeprom()
read_program_eeprom()
write_program_eeprom()
read_calibration()
write_program_memory()
read_program_memory()
write_external_memory()
erase_program_memory()
setup_external_memory()
```

Standard C Special:

```
rand()
srand()
```

Device Definition File

A CCS C program will start with a number of pre-processor directives similar to:

```
#include <18F452.H>
#fuses HS,NOWDT,NOBROWNOUT,NOPROTECT,PUT
#use delay(clock=20000000)
#include "lcd.c"
```

The first directive instructs the compiler to include the system header file 18F452.H

This is a device-specific file that contains information about the location of SFRs and the values to be written to them

Device Definition File

PIC 18F452 Definition File (18F452.H):

```
#define PIN_A0  31744
#define PIN_A1  31745
. . . . . . . .
#define PIN_B0  31752
#define PIN_B1  31753
. . . . . . . .
#define T1_DISABLED         0
#define T1_INTERNAL         0x85
#define T1_EXTERNAL         0x87
#define T1_EXTERNAL_SYNC    0x83
. . . . . . . .
#define CCP_OFF             0
#define CCP_CAPTURE_FE      4
#define CCP_CAPTURE_RE      5
#define CCP_CAPTURE_DIV_4   6
. . . . . . . .
```

Fuses

CCS C provides a fuse directive:

```
#fuses HS,NOWDT,NOBROWNOUT,NOPROTECT,PUT
```

which specifies the states of the configuration fuses that should be programmed onto the PIC

In this example:

HS	Clock is a high-speed crystal or resonator
NOWDT	Watchdog timer is disabled
NOBROWNOUT	Brown-out detector is disabled
NOPROTECT	Code protect off
PUT	Power-on timer is enabled

Delays

CCS C provides functions for generating delays:

```
delay_us()
delay_ms()
```

These delay functions actually delay by a number of machine cycles

The compiler needs to know the clock frequency in order to calculate the required number of machine cycles

```
#use delay(clock=20000000)
```

This use-delay directive specifies that the clock frequency of the PIC is 20 MHz

Multiple Source Code Files

CCS C does not allow separate compilation and linking of source code files

It is convenient (and good programming practice) to put commonly-used library functions in separate files

```
#include "lcd.c"
```

This directive instructs the compiler to include the user library file lcd.c in the file currently being compiled

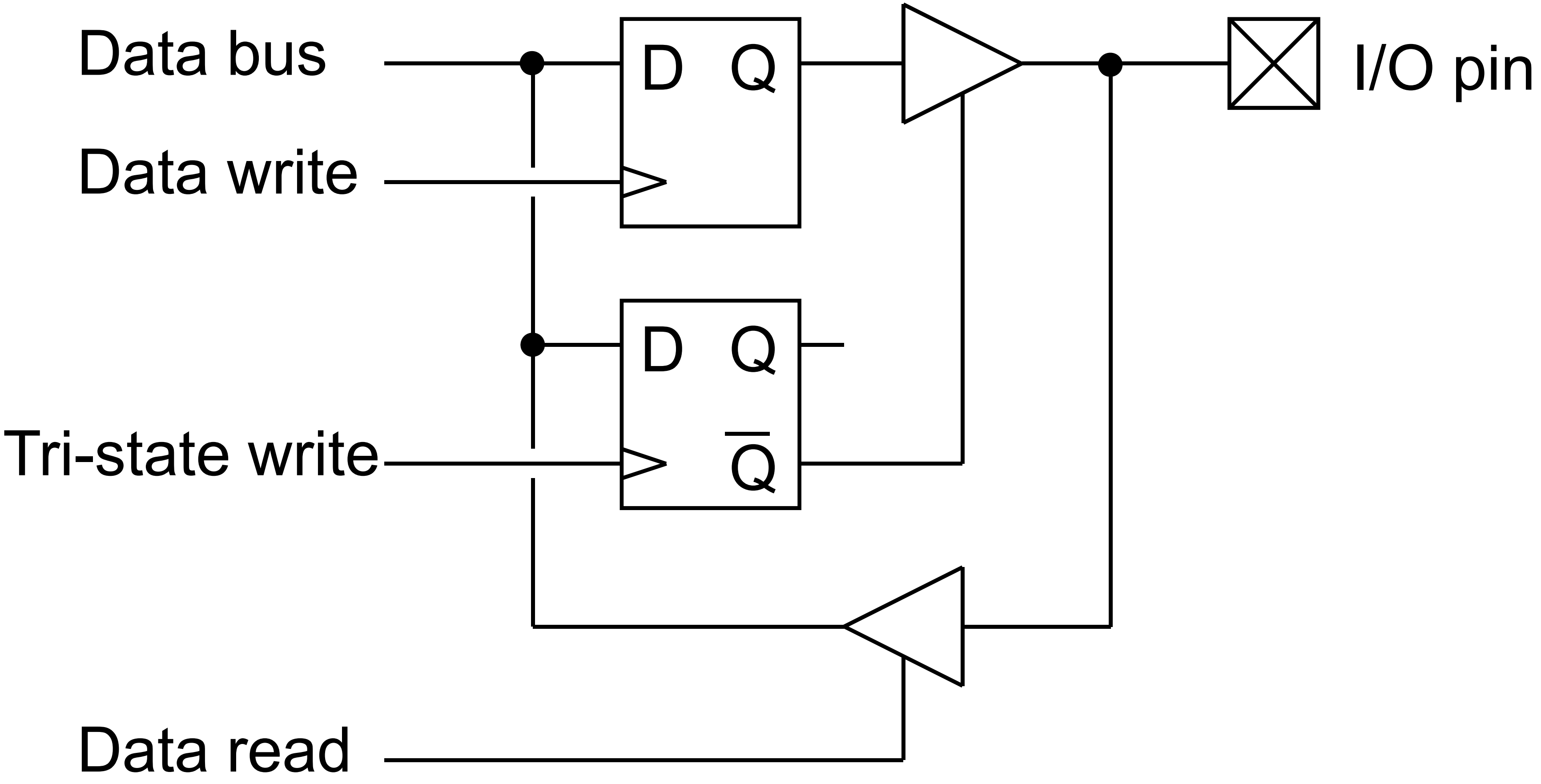
This is not particularly efficient (the library file is compiled every time) - however typical PIC programs compile in a few seconds

Lecture 3

Data Ports

Data Ports

Simplified diagram representing a single data I/O pin of one of the ports A-E:



Data Ports

Data I/O port functions:

Data write - this latches the data written to the pin which should be configured as an output

Tri-state write - this latches the data direction for the pin (0 = output, 1 = input)

Data read - this reads the current value of the pin which should be configured as an input

Each data port (A-E) consists of a number of pins, each of which can individually be configured as an input or output

Hardware Access in C

Memory-mapped hardware is traditionally accessed in C using pointers

If the hardware is byte (8-bit) organised then char or int (PIC) pointers are used

Example: an 8-bit input port memory-mapped to location 0xF81:

```
#define portb (int *) 0xF81
```

Thus *portb* is an int pointer whose value is the address of the bus device

Hardware Access in C

The port is accessed by the use of the indirection operator *:

```
int p;
p = *portb;
```

In this example the value of the data on the port mapped to memory location 0xF81 (port B) is assigned to variable *p*

Before the port can be read it is necessary to set the data direction register:

```
#define trisb (int *) 0xF93
*trisb = 0xFF;
```

Accessing the Data Ports

Complete program to toggle all pins on the B port:

```
#include <18F452.H>
#fuses HS,NOPROTECT,NOBROWNOUT,NOWDT,NOLVP,PUT
#use delay(clock=20000000)

#define trisb (int *) 0xF93
#define portb (int *) 0xF81

void main()
{
    *trisb = 0x00;
    for (;;) {
        *portb = ~*portb;
        delay_ms(100);
    }
}
```

Accessing the Data Ports

Or more elegantly using functions:

```
void set_portb_output()
{
    *trisb = 0x00;
}

void write_portb(int p)
{
    *portb = p;
}
```

```
void main()
{
    int q = 0x0F;
    set_portb_output();
    for (;;) {
        write_portb(q = ~q);
        delay_ms(100);
      }
}
```

Although this code is longer than the previous example it is better structured

Accessing the Data Pins

Data pins within a port can be set or read by using logical operators

To set pin 2 of data port B to logic 1:

```
*portb |= 0b00000100;
```

and to reset pin 2 of data port B to logic 0:

```
*portb &= 0b11111011;
```

To read the value of pin 7 of data port B:

```
if (*portb & 0b10000000) { ...
```

CCS C Support for Port I/O

Comprehensive support is provided in CCS C for accessing data ports and individual pins of the ports

Three different methods of I/O can be used, specified by the directives:

```
#use standard_io(port)

#use fast_io(port)

#use fixed_io(port_outputs=pin_x1,pin_x2, ...)
```

The differences between these I/O methods are to do with the way that the data direction registers are controlled

Standard I/O

#use standard_io(port) affects how the compiler will generate code for input and output instructions that follow

This directive takes effect until another #use xxx_io directive is encountered

The standard method of I/O will cause the compiler to generate code to set the direction register for each I/O operation

Standard_io is the default I/O method for all ports.

Examples: `#use standard_io(A)`

Fast I/O

#use fast_io(port) affects how the compiler will generate code for input and output instructions that follow

This directive takes effect until another #use xxxx_io directive is encountered

The fast method of doing I/O will cause the compiler to perform I/O without programming of the direction register

The user must ensure the direction register is set correctly via `set_tris_X()` .

Example: `#use fast_io(A)`

Fixed I/O

#use fixed_io(port) affects how the compiler will generate code for input and output instructions that follow

This directive takes effect until another #use xxx_io directive is encountered

The fixed method of I/O will cause the compiler to generate code to set the direction register for each I/O operation

The pins are programmed according to the information in this directive (not the operations actually performed)

Examples: `#use fixed_io(a_outputs=PIN_A2,PIN_A3)`

CCS C Support for Port I/O

Functions are provided for reading from a complete port:

```
value = input_a()
value = input_b()
. . . . . . . . .
```

for writing to a complete port:

```
output_a(value)
output_b(value)
. . . . . . . . .
```

and for setting the data direction register:

```
set_tris_a(int)
set_tris_b(int)
. . . . . . . .
```

Standard I/O

```
#use standard_io(b)

void main()
{
    int q;
    for (q = 0b00000001;; q ^= 0b00000101) {
        output_b(q);
        delay_ms(100);
      }
}
```

```
output_b(q);
      CLRF 0xf93
      MOVFF 0x6, 0xf8a
```

Set DDR

Write port

Fast I/O

```
#use fast_io(b)

void main()
{
    int q;
    set_tris_b(0b11111010);
    for (q = 0b00000001;; q ^= 0b00000101) {
        output_b(q);
        delay_ms(100);
    }
}
```

```
output_b(q);
      MOVFF 0x6, 0xf8a
```

Write port

Fixed I/O

```
#use fixed_io(b_outputs=pin_b2,pin_b0)

void main()
{
    int q;
    for (q = 0b00000001;; q ^= 0b00000101) {
        output_b(q);
        delay_ms(100);
      }
}

output_b(q);
      MOVLW 0xfa
      MOVWF 0xf93
      MOVFF 0x6, 0xf8a
```

Set DDR

Write port

CCS C Support for Pin I/O

A function is provided for reading from a pin of a data port:

```
value = input(pin)
```

and for writing to a pin of a data port :

```
output_bit(pin,value)
output_low(pin)
output_high(pin)
output_toggle(pin)
```

Pin names are of the form:

```
pin_a1      pin_b1      pin_c1
pin_a2      pin_b2      pin_c2
. . . . . . . . . . . . . . . .
```

Standard I/O

```
#use standard_io(b)

void main()
{
    boolean q;
    for (q = false;; q = !q) {
        if (q)
            output_high(pin_b2);
        else
            output_low(pin_b2);
        delay_ms(100);
      }
}

output_high(pin_b2);
      BCF 0xf93, 0x2
      BSF 0xf8a, 0x2
```

Set DDR

Write pin

Fast I/O

```
#use fast_io(b)

void main()
{
    boolean q;
    set_tris_b(0b11111010);
    for (q = false;; q = !q) {
        if (q)
            output_high(pin_b2);
        else
            output_low(pin_b2);
        delay_ms(100);
      }
}

output_high(pin_b2);
      BSF 0xf8a, 0x2
```

Write pin

Fixed I/O

```
#use fixed_io(b_outputs=pin_b2,pin_b0)

void main()
{
    boolean q;
    for (q = false;; q = !q) {
        if (q)
            output_high(pin_b2);
        else
            output_low(pin_b2);
        delay_ms(100);
      }
}

output_high(pin_b2);
      MOVLW 0xfa
      MOVWF 0xf93
      BSF 0xf8a, 0x2
```

Set DDR

Write pin

More Efficient Program

```
#use fast_io(b)

void main()
{
    set_tris_b(0b11111010);
    for (;;) {
        output_toggle(pin_b2);
        delay_ms(100);
      }
}
```

```
output_toggle(pin_b2);
      BTG 0xf8a, 0x2
```

← Toggle pin

Pull-ups

Some data ports have optional internal weak pull-ups which pull the I/O lines high by default

A switch used as input can pull the line low (against the weak pull-ups) and no further hardware is required

These are only available on ports A and B

The commands to activate the pull-ups are:

```
port_a_pullups(value)
port_b_pullups(value)
```

where a value *true* will activate, and a value *false* de-activate, the internal pull-ups

Lecture 4

Timer/Counter/PWM

Timer/Counters

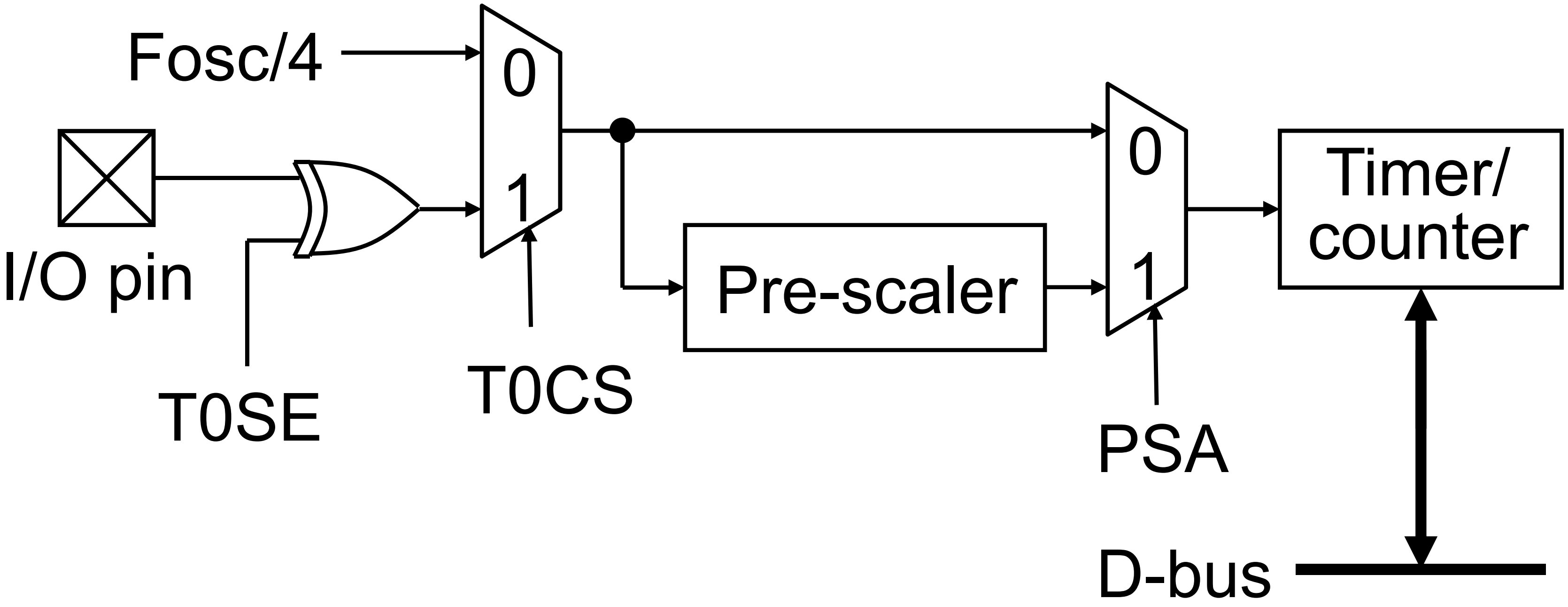
The PIC 18F452 has 4 timer/counters: Timer0, Timer1, Timer2, Timer3

Timer 0:	8 or 16-bit (selectable)
Timer 1:	16-bit
Timer 2:	8-bit
Timer 3:	16-bit

The timer/counters can be used to:

- generate timed interrupts
- count incoming logic transitions
- capture timer/counter on an input event
- generate variable PWM outputs

Timer/Counters



T0SE determines whether 0→1 or 1→0 transitions are active

T0CS determines the source (I/O pin or internal clock)

PSA determine whether the input is pre-scaled or not

Timer/Counter Control Register

The 8-bit timer control register T0CON controls the configuration for timer/counter 0:

TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

TMR0ON	Enable: off (0) or on(1)
T08BIT	Mode: 16-bit (0) or 8-bit (1)
T0CS	Time source: internal clock (0) or external (1)
T0SE	Edge select: 0→1 (0) or 1→0 (1)
PSA	Prescaler: on (0) or off (1)
T0PS0-2	Prescaler ratio: 1/2 (000) .. 1/256 (111)

For example for 8-bit mode, external source, 0→1 edge, no pre-scaler: T0CON = 0b11100000 = 0xE0

Counters

Program to count pulses on external input to timer/counter 0:

```
#define t0con (int *) 0xFD5
#define tmr0l (int *) 0xFD6

void main()
{
    *t0con = 0xE0;
    *tmr0l = 0;
    lcd_init();
    for (;;) {
        printf(lcd_putc, "\f%d", *tmr0l);
        delay_ms(200);
      }
}
```

Counters

Fortunately it is not necessary to manipulate the registers directly because special functions are provided in CCS C:

```
setup_timer_0(mode)
setup_timer_1(mode)
. . . . . . . . . .
```

where mode depends on the timer, but for timer 0 can be:

```
RTCC_INTERNAL, RTCC_EXT_L_TO_H or RTCC_EXT_H_TO_L

RTCC_DIV_2, RTCC_DIV_4, RTCC_DIV_8, RTCC_DIV_16,
RTCC_DIV_32, RTCC_DIV_64, RTCC_DIV_128 or RTCC_DIV_256

RTCC_OFF, RTCC_8_BIT
```

One constant may be used from each group ORed together with the | operator

Counters

To set the counter:

```
set_timer0(value)
set_timer1(value)
. . . . . . . . .
```

For timers/counters 0, 1 and 3 the value is a 16 bit int
For timer/counter 2 the value is an 8 bit int

To read the counter:

```
value = get_timer0()
value = get_timer1()
. . . . . .
```

Timer/counters 0, 1 and 3 return a 16 bit int
Timer/counter 2 returns an 8 bit int

Counters

Program to count pulses on external input to timer/counter 0:

```
void main()
{
    setup_timer_0(RTCC_EXT_L_TO_H | RTCC_8_BIT);
    set_timer0(0);
    lcd_init();
    for (;;) {
        printf(lcd_putc, "\f%d", (int) get_timer0());
        delay_ms(200);
      }
}
```

Timer Interrupts

Microprocessors normally execute code sequentially

Sometimes execution must be suspended temporarily to perform some other task

In PICs this happens as the result of interrupt requests

An interrupt is raised when a particular condition occurs:

- timer/counter overflow
- change in the state of an input line
- data received on the serial bus
- completion of an analogue-to-digital conversion
- power supply brown-out

Timer Interrupts

An interrupt can be generated each time a counter/timer overflows

This generates interrupts at a frequency determined by the clock speed and the timer/counter configuration

The clock, divided by 4 and pre-scaled, is applied to the counter which counts to 2^n-1 before overflowing back to 0

$$\textit{interrupt rate} = \frac{\textit{clock frequency}}{4 \times 65536 \times \textit{prescaler}}$$

(This assumes that a 16-bit timer/counter is being used)

Timer Interrupts

Suppose that the clock frequency is 20 Mz and a pre-scaler ratio of 16 is used:

$$\begin{aligned} \textit{interrupt}\ \text{rate} &= \frac{20000000}{4 \times 65536 \times 16} \\ &= \frac{20000000}{4194304} \\ &= 4.768\ \text{Hz} \end{aligned}$$

Note that only a limited number of discrete interrupt rates are possible with a given clock frequency

Timer Interrupts

CCS C provides the following functions to configure interrupts:

`disable_interrupts()`	disables the specified interrupt
`enable_interrupts()`	enables the specified interrupt
`clear_interrupt()`	clear specified interrupt flag

The are corresponding interrupt types and directives for each of the available interrupt sources:

`INT_TIMER0`	`#INT_TIMER0`	Counter/timer 0 oflo
`INT_AD`	`#INT_AD`	A/D conversion complete
`INT_RB`	`#INT_RB`	Change on B port
`INT_SSP`	`#INT_SSP`	I^2C Activity

. . . .

Timer Interrupts

```
#INT_TIMER0
void timer_irq()
{
    output_toggle(pin_b1);
}

void main()
{
    setup_timer_0(RTCC_INTERNAL | RTCC_DIV_16);
    enable_interrupts(INT_TIMER0);
    enable_interrupts(GLOBAL);
    for (;;) {
    }
}
```

Timer Interrupts

Starting with a 20 MHz clock there is no power-of-2 pre-scaler ratio that gives an interrupt rate close to 1 Hz

A pre-scaler ratio of 128 gives:

$$\begin{aligned} \textit{interrupt rate} &= \frac{20000000}{4 \times 65536 \times 128} \\ &= 0.59605 \end{aligned}$$

Interrupts occur when the counter overflows from 65535 to 0

If the counter is pre-loaded with a value *n* when an interrupt occurs then the counter only has to count from *n* to 65535

Timer Interrupts

Pre-loading with a value *n*:

$$interrupt\ rate = \frac{clock\ frequency}{4 \times (65536 - n) \times prescaler}$$

To generate a 1 Hz interrupt with a clock frequency of 20 MHz and a pre-scaler ratio of 128:

$$1 = \frac{20000000}{4 \times (65536 - n) \times 128}$$

$$n = 65536 - \frac{20000000}{4 \times 128}$$

$$= \quad 26474$$

1 Hz Timer Interrupts

```
#INT_TIMER0
void timer_irq()
{
    set_timer0(26474);
    output_toggle(pin_b1);
}

void main()
{
    setup_timer_0(RTCC_INTERNAL | RTCC_DIV_128);
    enable_interrupts(INT_TIMER0);
    enable_interrupts(GLOBAL);
    for (;;) {
    }
}
```

Pulse-Width Modulation

Pulse-width modulation (PWM) can be used to create an n-bit digital-to-analogue converter (DAC)

A rectangular wave with a given mark-space ratio (duty cycle) is generated and this is applied to a 1-bit DAC

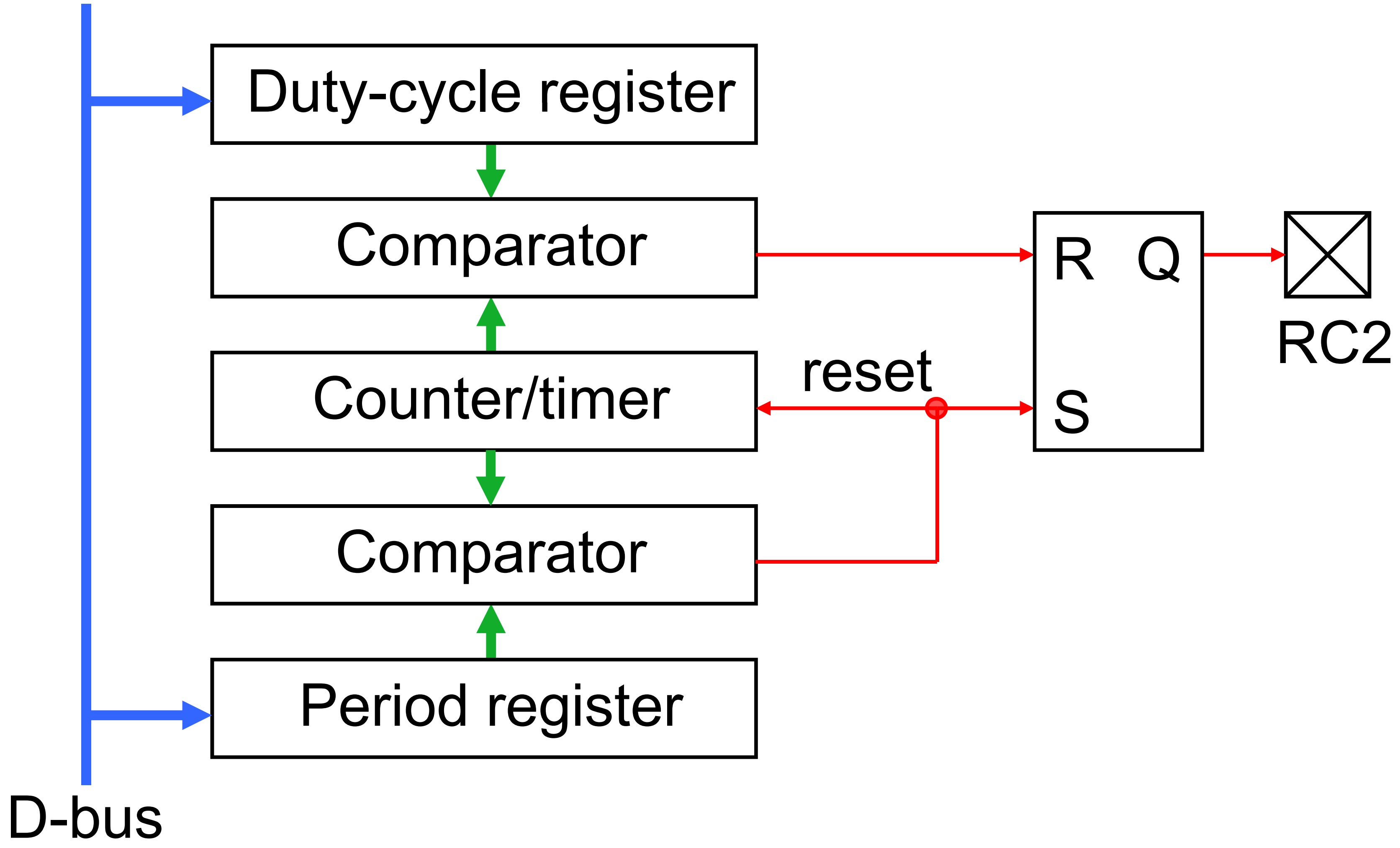
DACs of this type have only a limited bandwidth because of the need to filter out the rectangular wave.

Typical applications are in dc motor control, brightness control of lights and in dc-dc converters

The PIC18F452 has a PWM generator that make use of counter/timer 2

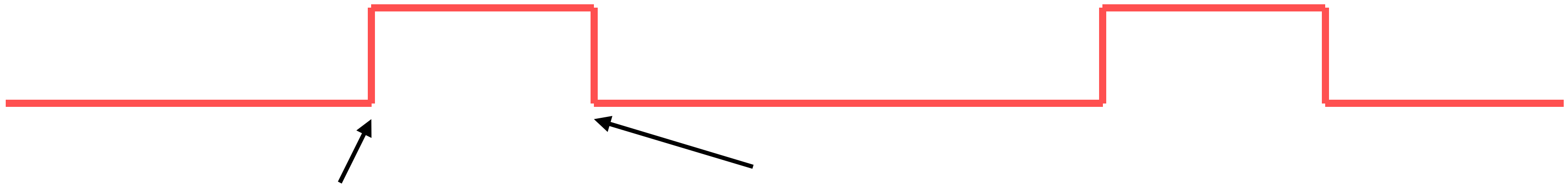
Pulse-Width Modulation

Simplified diagram of PWM generator:



Pulse-Width Modulation

PWM waveform:



$Count = Period\ Register$

$\rightarrow 0$

$Count = Duty\ Cycle\ Register$

PWM frequency:

$$PWM\ frequency = \frac{clock\ frequency}{4 \times prescaler \times (1 + period\ register)}$$

For example, 20 MHz clock, prescaler = 1, period register = 255:

$$PWM\ frequency = \frac{20000000}{4 \times 1 \times (1 + 255)} = 19.53\ \mathrm{kHz}$$

Pulse-Width Modulation

CCS C provides the following functions to control the PWM generator:

`setup_ccp1(CCP_PWM)`	sets PWM mode
`set_pwm1_duty(q)`	sets duty-cycle register to *q*

Note that *q* should not exceed the value of the period register

It is also necessary to configure counter/timer 2:

```
setup_timer_2(pre-scaler, period, 1);
```

pre-scaler is one of: `T2_DIV_BY_1, T2_DIV_BY_4, T2_DIV_BY_16`

period (the period register) is an int 0-255

Pulse-Width Modulation

```
#define period 100

void main()
{
    int q;
    setup_ccp1(CCP_PWM);
    setup_timer_2(T2_DIV_BY_4,period,1);
    for(;;) {
        if (++q >= period)
            q = 0;
        set_pwm1_duty(q);
        delay_ms(100);
    }
}
```

$$\textit{PWM frequency} = \frac{20000000}{4 \times 4 \times (1+100)} = 12.38\ \text{kHz}$$

Lecture 5

LCD, RS232, ADC, I^{2}C, EEPROM

Liquid Crystal Display

A convenient method for displaying information is the alpha-numeric liquid crystal display (LCD)

A 2-line 16-character LCD is provided on the PicDem2 board

The LCD has a 7-wire interface (4 data and 3 control) and the connections are hard-wired on the PicDem2 board:

```
#define en pin_a1        #define d4 pin_d0
#define rw pin_a2        #define d5 pin_d1
#define rs pin_a3        #define d6 pin_d2
                         #define d7 pin_d3
```

Routines to drive the LCD are available in a file lcd.c:

```
#include "lcd.c"
```

Liquid Crystal Display

Before writing to the LCD it is necessary to initialise it:

```
lcd_init();
```

This function sets up the PIC I/O pins used to communicate with the LCD and initializes the LCD registers

Then various routines can be used to control the display:

`lcd_clear()`	clear complete display
`lcd_home()`	goto 1st character on 1st line
`lcd_backspace()`	backspace by 1 character
`lcd_panleft()`	pan complete display left
`lcd_panright()`	pan complete display right
`lcd_gotoxy(int x, int y)`	goto x character on y line
`lcd_putc(char c)`	write character at current pos

Liquid Crystal Display

In most cases it is convenient to use the printf() (print formatted) function for all output to the LCD, for example:

```
printf(lcd_putc, "\fTime = %d s", t);
```

printf() can print characters, text, integers and floating-point numbers

The first parameter determines the output channel, in this case the LCD

The second parameter is the formatting string which determines how the following parameters ae displayed

Any further parameters are variables or constants to be printed

Liquid Crystal Display

The printf() format takes the generic form %nt where n is optional and may be:

1-9 to specify number of characters to be output
01-09 to indicate leading zeros
1.1 to 9.9 for floating point and %w output

t is the type and may be one of:

c	Character	s	String or character
u	Unsigned int	d	Signed int
Lu	Long unsigned int	Ld	Long signed int
x	Hex int (lower case)	X	Hex int (upper case)
Lx	Hex long int (lower case)	LX	Hex long int (upper case)
f	Float (truncated decimal)	g	Float (rounded decimal)
e	Float in exponential format	w	Int with decimal point

Liquid Crystal Display

```
#include "lcd.c"

void main()
{
    long int q;
    float p;

    lcd_init();

    for (;;) {
        q = read_adc();
        p = 5.0 * q / 1024.0;
        printf(lcd_putc, "\fADC = %4ld", q);
        printf(lcd_putc, "\nVoltage = %01.2fV", p);
        delay_ms(100);
    }
}
```

RS232

The PIC18F452 has a built-in Universal Synchronous Asynchronous Receiver Transmitter (USART)

This allows it to communicate using the RS232, RS422 and RS485 protocols

The 5 V logic-level receive and transmit signals of the PIC are converted to RS232 levels by a MAX232 device

Baud rates are generated by dividing down the system clock

The USART receive and transmit pins are c7 and c6 respectively

RS232

CCS C provides the following functions to control RS2323 communications:

`getc()`	returns character received on RS232
`kbhit()`	true when character received on RS232
`putc(char)`	transmits character over RS232
`printf(form,..)`	transmits formatted data over RS232

There is also a directive which sets up the USART for RS232 operation:

```
#USE RS232(options)
```

where options include: transmit pin, receive pin, baud rate, bits, and parity

RS232

```
#use rs232(baud=38400, xmit=PIN_C6, rcv=PIN_C7,
    parity=n, bits=8)

void main()
{
    float p;

    lcd_init();

    for (;;) {
        p = 5.0 * read_adc() / 1024.0;
        printf("\n\rVoltage = %01.2fV", p);
        if (kbhit())
            printf(lcd_putc, "%c", fgetc());
        delay_ms(100);
    }
}
```

Analogue-to-Digital Converter

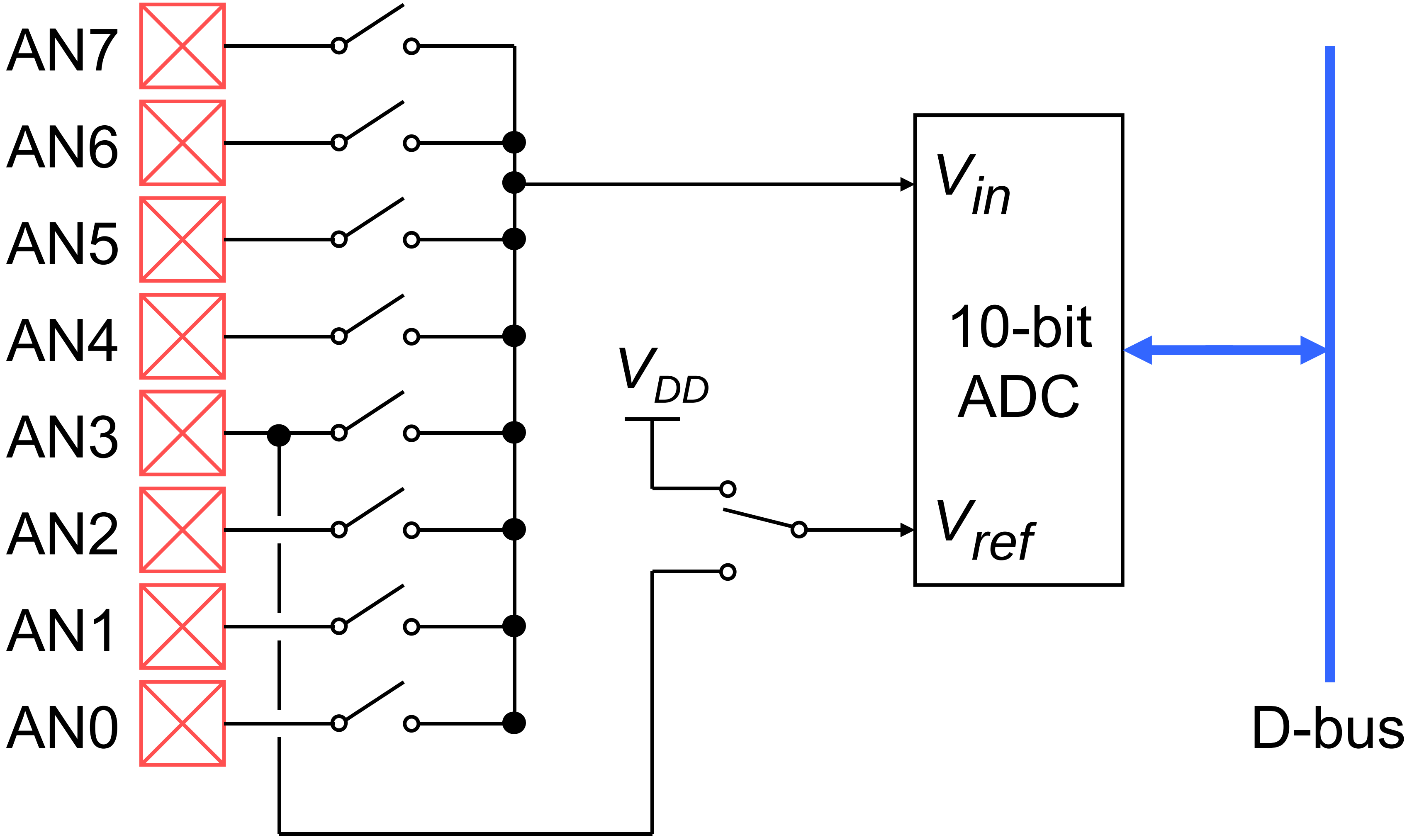
The PIC18F452 has a single 10-bit successive-approximation ADC with up to 8 multiplexed analogue inputs

The reference voltage V_{ref} is software selectable to be either the supply rail or the analogue input AN3

Analogue inputs should have a source resistance of less than 2.5 kΩ to allow for charging of the sample-hold capacitor

Conversion takes 11 cycles of the ADC clock which can be either a RC oscillator (2-6 µS) or the pre-scaled system clock

Analogue-to-Digital Converter



Analogue-to-Digital Converter

CCS C provides the following functions to control the ADC:

`setup_adc(mode)`	set the clock source
`setup_adc_ports(value)`	set which pins are analogue
`set_adc_channel(channel)`	set current input channel
`read_adc()`	perform conversion

There is also a directive which determines the return size for read_adc():

```
#DEVICE ADC=xx
```

where xx can be 8 or 10 (when set to 8 the ADC will return the most significant byte)

Analogue-to-Digital Converter

```
#device ADC=10

void main()
{
    long int q;
    float p;
    setup_adc(ADC_CLOCK_DIV_64);
    setup_adc_ports(AN0);
    set_adc_channel(0);
    lcd_init();
    for (;;) {
        q = read_adc();
        p = 5.0 * q / 1024.0;
        printf(lcd_putc, "\fADC = %4ld", q);
        printf(lcd_putc, "\nVoltage = %01.2fV", p);
        delay_ms(100);
    }
}
```

EEPROM

The PIC18F452 has 256 byte of internal data eeprom

EEPROM is not directly mapped to the data space but is accessed indirectly through the SFR: EEADR

This memory is non-volatile and can be used to store, for example, setup parameters

CCS C provides the following functions to read and write to the EEPROM:

`read_eeprom(address)`	read data from address
`write_eeprom(address, value)`	write data to address

Inter-Integrated Circuit (I^2C) Bus

The PIC18F452 has a Master Synchronous Serial Port (MSSP) which can operate in either SPI or I^2C mode

SPI is a synchronous serial protocol that uses 3 wires: SDO, SDI and SCK

I^2C is a synchronous serial protocol that uses 2 wires: SDA and SCL

The PicDem2 board used in the PIC laboratory has 2 devices connected to the I^2C bus:

- a TC74 digital thermometer with I^2C address 0x9A
- a 24LC256 EEPROM (32 kbytes) with I^2C address 0xA0

Inter-Integrated Circuit (I^{2}C) Bus

CCS C provides the following functions to control I^{2}C communications:

`i2c_start()`	Issues a start command on the I^{2}C
`i2c_write(data)`	Sends a single byte over the I^{2}C
`i2c_read()`	Reads a byte over the I^{2}C
`i2c_stop()`	Issues a stop command on the I^{2}C

There is also a pre-processor directive which configures the device as a Master or a Slave:

```
#use i2c
```

This directive also assigns the SDA and SCL pins used for the I^{2}C interface

24LC256 EEPROM

The Microchip Technology 24LC256/ is a 32K x 8 (256 Kbit) serial EEPROM

It has been developed for advanced, low-power applications such as personal communications or data acquisition

This device is capable of operation across a broad voltage range (1.8V to 5.5V)

Functional address lines allow up to eight devices on the same bus, for up to 2 Mbit address space

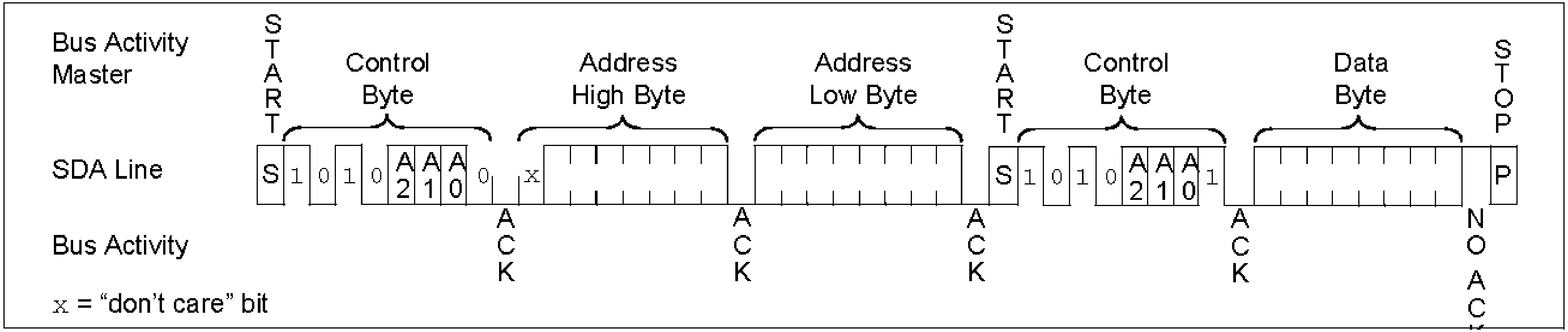
It is available in the standard 8-pin plastic DIP,SOIC, TSSOP, MSOP and DFN packages.

24LC256 EEPROM

To perform a read operation the master generates a Start with R/W=0 and sends the word address (MS byte first)

Then the master generates a Start with R/W=1 and reads the data

Finally the master generates a Stop



24LC256 EEPROM

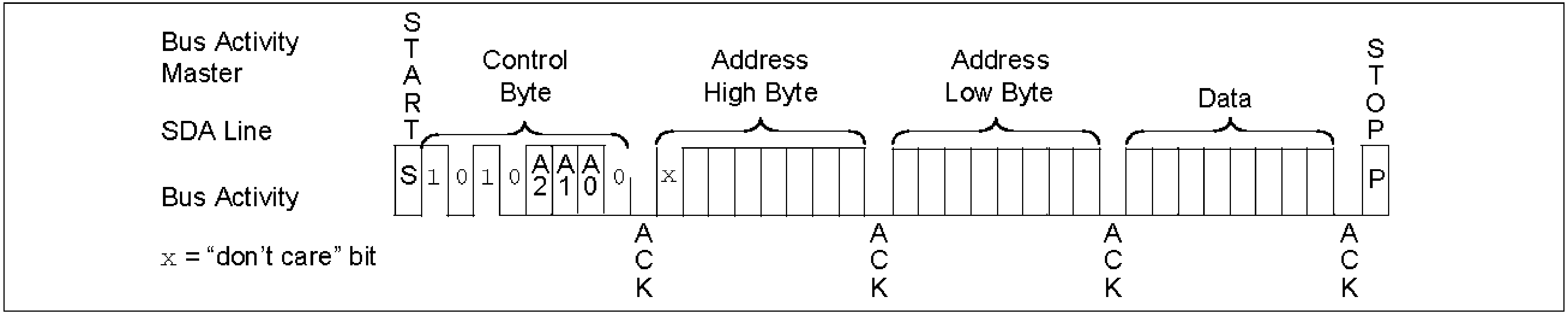
```
#use i2c(master, sda=pin_c4, scl=pin_c3)
#define eeprom_addr 0xa0

int read_ext_eeprom(long int i)
{
    int q;
    i2c_start();
    i2c_write(eeprom_addr & 0xfe);
    i2c_write(i >> 8);
    i2c_write(i & 0xff);
    i2c_start();
    i2c_write(eeprom_addr | 0x01);
    q = i2c_read(0);
    i2c_stop();
    return q;
}
```

24LC256 EEPROM

To perform a write operation the master generates a Start with R/W=0 and sends the word address (MS byte first) and data

Then the master generates a Stop

To prevent further writes while the device is busy the master should wait for acknowledge (ack=0) before proceeding



24LC256 EEPROM

```
boolean busy()
{
   boolean ack;
   i2c_start();
   ack = i2c_write(eeprom_addr & 0xfe);
   i2c_stop();
   return ack;
}

void write_ext_eeprom(long int i, int d)
{
    i2c_start();
    i2c_write(eeprom_addr & 0xfe);
    i2c_write(i >> 8);
    i2c_write(i & 0xff);
    i2c_write(d);
    i2c_stop();
    while (busy());
}
```

Programming PIC Microcontrollers