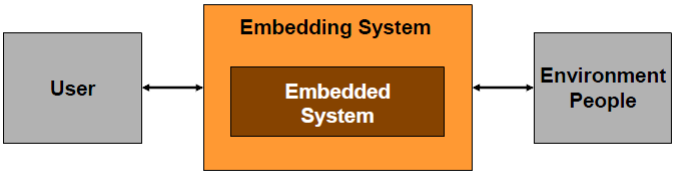
Introduction to embedded systems

What is an embedded system



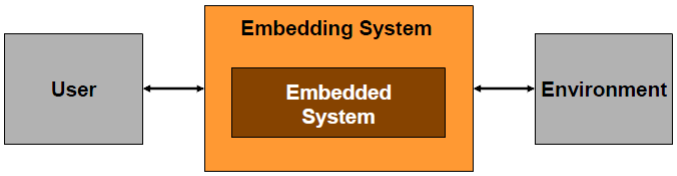
Embedded System =

A computer system which is integrated into another system, the embedding system. The requirements for the embedded system must be derived from the requirements for the embedding system.

Examples?

Examples of embedding systems

Two different main application areas



Embedding system = product
Examples:
•Automotive Electronics
• Avionics
• Health Care Systems

Embedding system = production system
Examples:
• Manufacturing Control
• Chemical Process Control
• Logistics

What are embedded systems doing?

Typical functionalities

- Measuring physical variables (sensing)
- Storing data
- Processing sensor signals and data
- Influencing physical variables (actuating)
- Monitoring
- Supervising
- Enable manual and automatic operation

In one word:

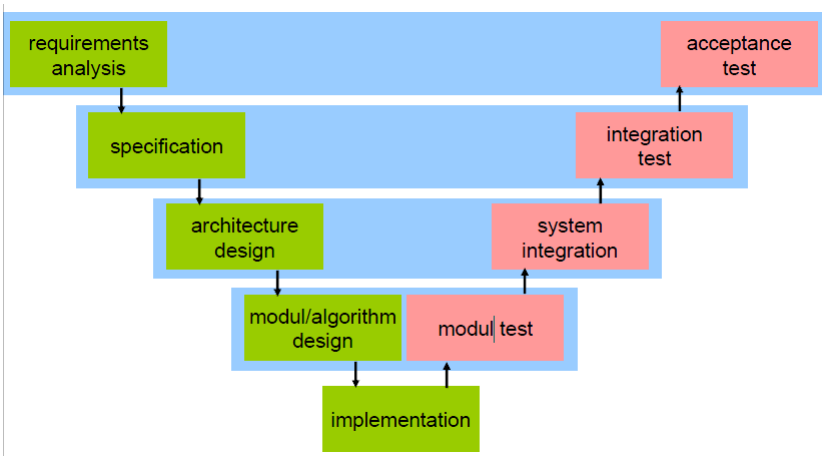
Control

Differences to “desktop“ computing?

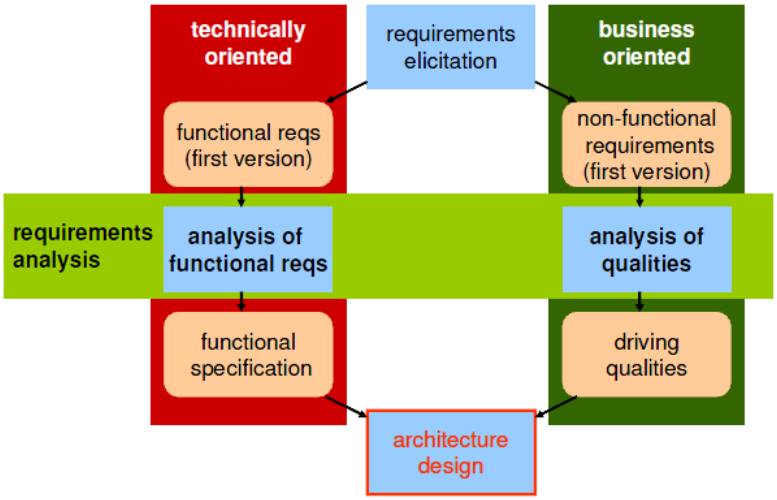
- Interaction with physical environment
- ***Closed loop***
- Malfunction may lead to damage
- No or very restricted human/computer interface
- No or very restricted maintenance possibilities
- Part of competitively priced products (high volumes)
- Tight resource constraints
- Often special hardware
- Part of engineering product
- High product generation frequency
- Often many variants

⇒ **Implications for SW engineering?**

V Model



Architecture design needs functional spec and driving qualities



What is architecture?

Bass, Clements, Kazman, 2003 (modified):

The architecture of a system is the structure or the structures of the system, which comprise elements, the externally visible properties of those elements, and the relationship among them.

- The architecture defines elements of the system.
  - Architecture design is the first phase in which the system is no longer a black box.
  - The designer begins to structure the system into parts.
  - Architecture manifests the earliest design decisions.
  - Architecture is the blueprint for system integration.

What is architecture? /2

Bass, Clements, Kazman, 2003 (modified):

The architecture of a system is the structure or the structures of the system, which comprise elements, the externally visible properties of those elements, and the relationship among them.

- The architecture is only one step further in refinement.
  - Now the elements are black boxes.
  - The architecture specifies what the elements do and how they interact **from an outside (system's) perspective** (often regarded as the element's **responsibilities)**
  - Central concept of architectures: **Interfaces.**

What is architecture? /3

Bass, Clements, Kazman, 2003 (modified):

The architecture of a system is the structure or the structures of the system, which comprise elements, the externally visible properties of those elements, and the relationship among them.

- A system can have and usually has more than one structure.
  - Examples:
    - Design time elements (files, components, modules)
    - Run-time elements (processes, tasks, threads)
    - Behavioral elements (states, messages, queues)
  - The designer must consider different **architectural views.**

Importance of architecture

Bass, Clements, Kazman, 2003 (again):

- Architecture represents earliest design decisions.
- They are the most difficult to get correct and the hardest to change later in the design process.
- They have the most far-reaching effects.

Why?

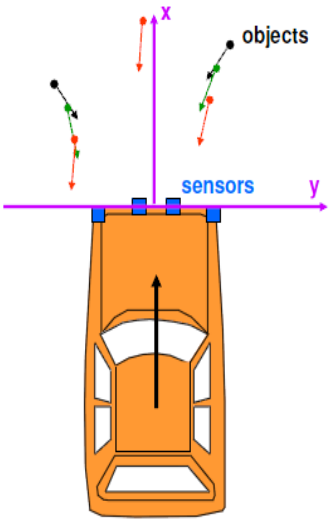
- Architecture defines constraints on implementation.
- Architecture dictates organizational structure.
- Architecture inhibits or enables a system‘s quality.
- It is possible to predict system qualities by studying the architecture.

Example: A pre-crash sensing system

The function of the pre-crash sensing system (PCSS) is preprocessing of sensor data to provide information about potentially colliding objects to pre-crash system (PCS), which will then decide what to do

Required sequence:

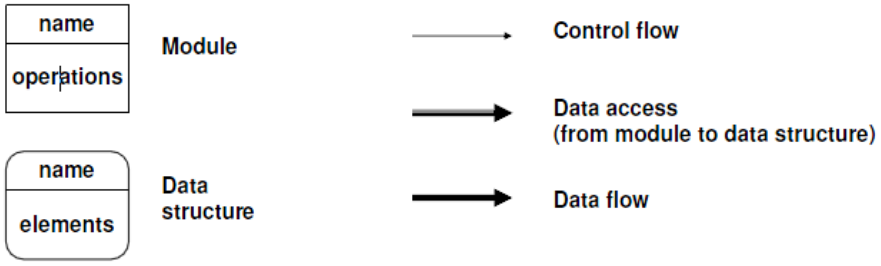
1. Read sensor data
2. Update list of objects (location, rel. speed, time of last measurement)
3. Identify dangerous objects
4. Send information to PCS

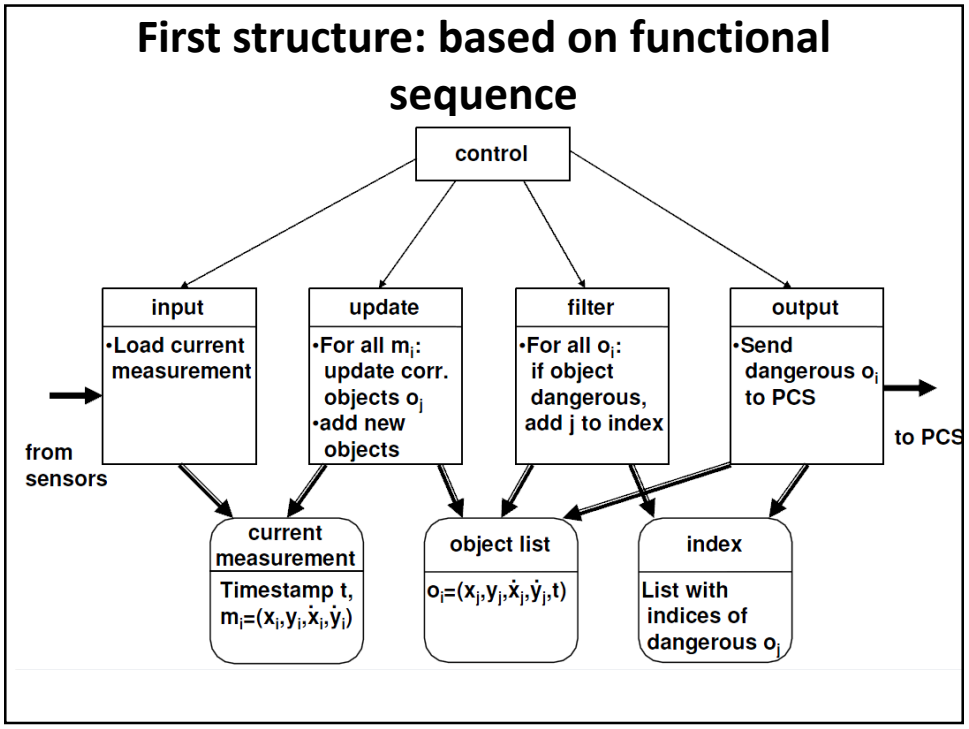


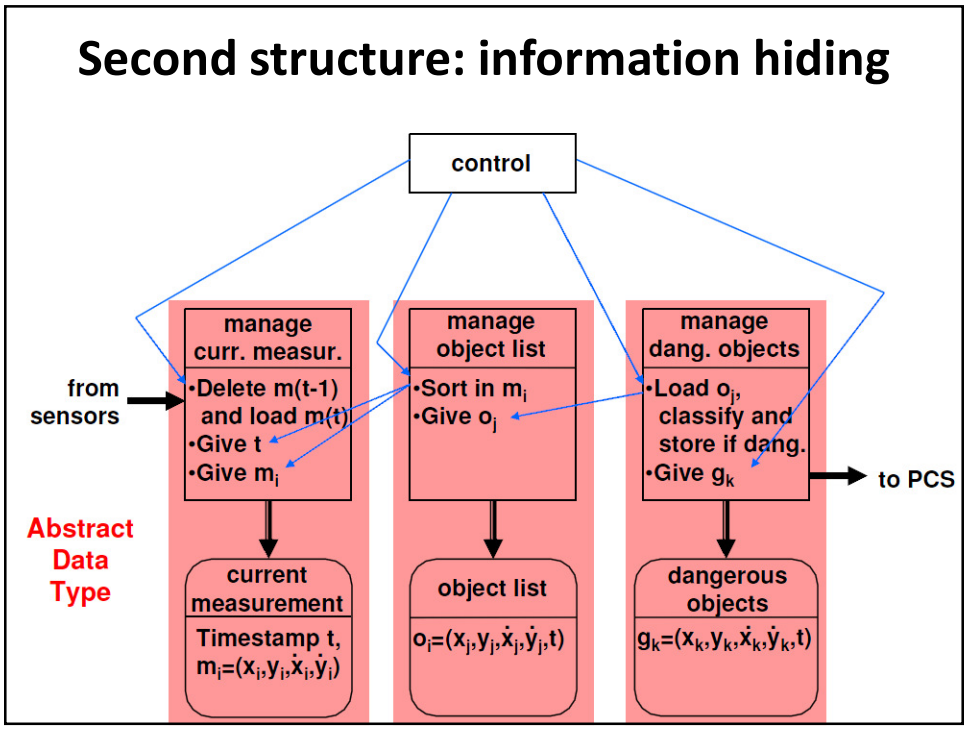
Two alternative structures

- In the following two alternative structures will be presented.
- The first one is based on the functional sequence, the second one on information hiding

Representation elements:




First structure: based on functional sequence
control
input
•Load current measurement
update
•For all m_i: update corr. objects o_j
•add new objects
filter
•For all o_i: if object dangerous, add j to index
output
•Send dangerous o_i to PCS
from sensors
to PCS
current measurement
Timestamp t, $m_i=(x_i,y_i,\dot{x}_i,\dot{y}_i)$
object list
$o_i=(x_j,y_j,\dot{x}_j,\dot{y}_j,t)$
index
List with indices of dangerous o_j


Second structure: information hiding
control
manage curr. measur.
•Delete m(t-1) and load m(t)
•Give t
•Give m_i
manage object list
•Sort in m_i
•Give o_j
manage dang. objects
•Load o_j, classify and store if dang.
•Give g_k
from sensors
to PCS
Abstract Data Type
current measurement
Timestamp t, $m_i=(x_i,y_i,\dot{x}_i,\dot{y}_i)$
object list
$o_i=(x_j,y_j,\dot{x}_j,\dot{y}_j,t)$
dangerous objects
$g_k=(x_k,y_k,\dot{x}_k,\dot{y}_k,t)$

Which is the better structure?

- Depends on criteria.
- Parnas, 1972:
  - Best criterion for modularization is maintainability/modifiability, i.e. the support of changes.
  - Changes mostly affect data structures
- Example:
  - Objects shall be stored in polar coordinates instead of Cartesian coordinates.
  - Changes in first structure: 3 Modules
  - Changes in second structure: 1 Module

Microcontrollers

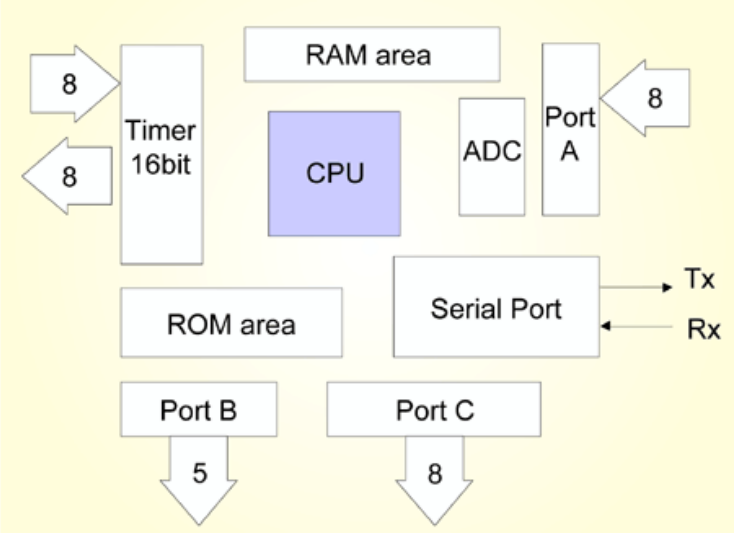
overview

- Microcontroller architecture. Address modes and instruction sets. Subroutines and interrupts. Handling software tools including IDE, editor, assembler, simulator and C compiler. Interface techniques including parallel, serial, timer, and analogue peripheral interface.

History

- In the year 1969, first microprocessor was born.
- In 1971 Intel obtained the right to sell this integrated circuit.
  - Before that Intel bought the license from the Japanese company BUSICOM company which first came with the idea.
- During that year, a microprocessor called the 4004 appeared on the market.
  - The first 4-bit microprocessor with the speed of 6000 operations per second.
- Not long after that, American company CTC requested from Intel and Texas Instruments to manufacture 8-bit microprocessor.
  - In April 1972 the first 8-bit microprocessor called the 8008 appeared on the market.
  - It was able to address 16Kb of memory, had 45 instructions and the speed of 300 000 operations per second. That microprocessor was the predecessor of all today's microprocessors.
- In April 1974, Intel launched 8-bit processor called the 8080.
  - Address 64Kb of memory, had 75 instructions and initial price was $360.
- In another company called Motorola launched 8-bit microprocessor 6800.
- At the WESCON exhibition in the USA in 1975, MOS Technology announced that it was selling processors 6501 and
- 6502 at $25 each.
  - In response to the competitor, both Motorola and Intel cut the prices of their microprocessors to $69.95.
- Due to low price, 6502 became very popular so it was installed into computers such as KIM-1, Apple I, Apple
- II, Atari, Commodore, Acorn, Oric, Galeb, Orao, Ultra and many others.
- Soon appeared several companies manufacturing the 6502 (Rockwell, Sznertek, GTE, NCR, Ricoh, Commodore took
- over MOS Technology).
- Other companies such as Zilog Inc have their own microprocessor.
  - In 1976 Zilog announced the Z80. When designing this microprocessor Faggin
  - The new processor was compatible with the 8080, i.e. it was able to perform all the programs written for the 8080.
  - Apart from that, many other features was included so that the Z80 was the most powerful microprocessor at that time.
  - It was able to directly address 64Kb of memory, had 176 instructions, a large number of registers, built in option for refreshing dynamic RAM memory, single power supply, greater operating speed etc.
  - The Z80 was a great success and everybody replaced the 8080 by the Z80.
- In 1976 Intel came up with an upgraded version of 8-bit microprocessor called the 8085. However, the Z80 was so much better that Intel lost the battle.
- Even though a few more microprocessors appeared later on the market (6809, 2650, SC/MP etc.), everything was actually decided. There were no such great improvements which could make manufacturers to change their mind, so the 6502 and Z80 along with the 6800 remained chief representatives of the 8-bit microprocessors of that time.

uProcessors < uControllers < SoCs



Microprocessors (uP) differ from microcontrollers (uC)

uC: suited for controlling I/O devices that requires a minimum component count
uP: suited for processing information in computer systems
Instruction sets:
- uP: processing intensive
  - powerful addressing modes
  - instructions to perform complex operations & manipulate large volumes of data
  - processing capability of MCs never approaches those of MPs
  - large instructions -- e.g., 80X86 7-byte long instructions
- uC: cater to control of inputs and outputs
  - instructions to set/clear bits
  - boolean operations (AND, OR, XOR, NOT, jump if a bit is set/cleared), etc.
  - Extremely compact instructions, many implemented in one byte
    - (Control program must often fit in the small, on-chip ROM)

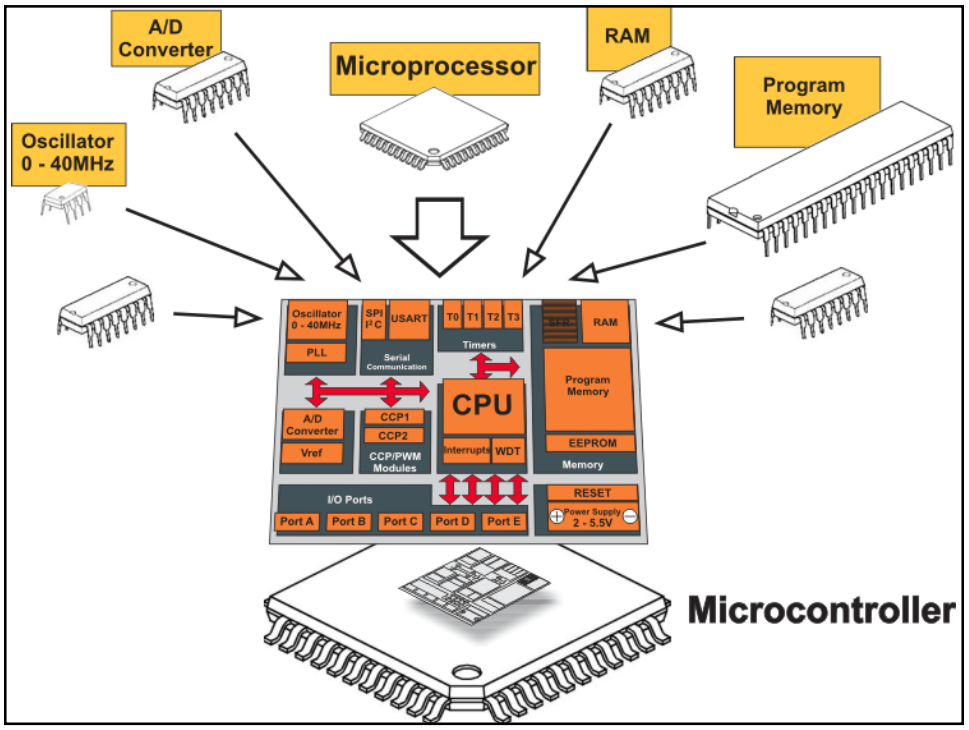
Hardware & Instructionset support:
- uC: built-in I/O operations, event timing, enabling & setting up priority levels for interrupts caused by external stimuli
- uP: usually require external circuitry to do similar things (e.g, 8255 PPI, 8259 etc)

Microprocessors (uP) and microcontrollers (uC)

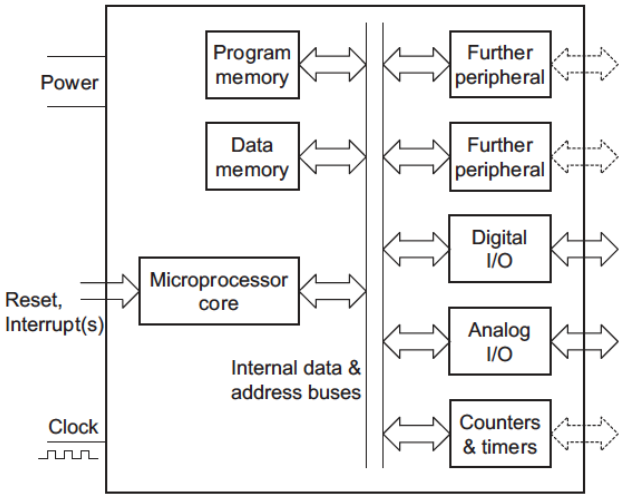
Bus widths:
- uP: very wide
  - large memory address spaces (>4 Gbytes)
  - lots of data (Data bus: 32, 64, 128 bits wide)
- uC: narrow
  - relatively small memory address spaces (typically kBytes)
  - less data (Data bus typically 4, 8, 16 bits wide)

Clock rates:
- uP very fast (> 1 GHz)
- uC: Relatively slow (typically 10-20 MHz)
  - since most I/O devices being controlled are relatively slow

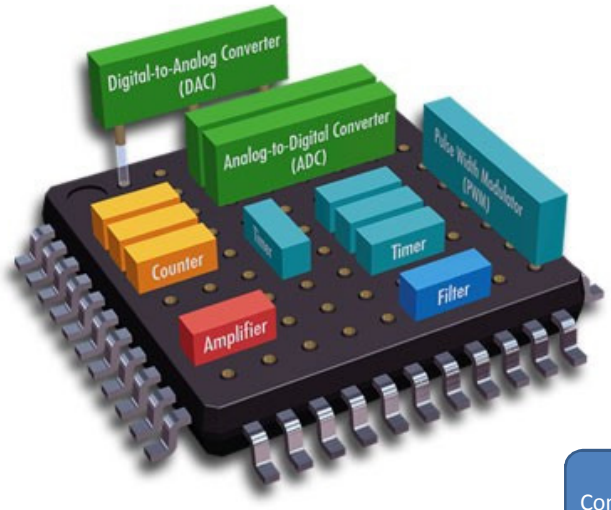
Cost:
- uP's expensive (often > $100)
- uCs cheap (often $1 - $10)
  - 4-bit: < $1.00
  - 8-bit: $1.00 - $8.00
  - 16-32-bit: $6.00 - $20.00

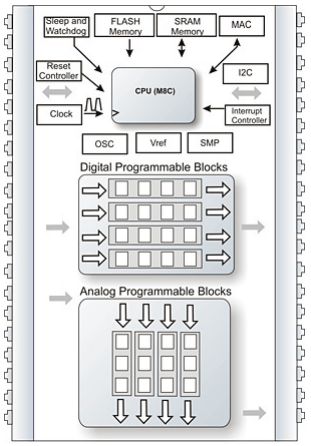


A generic microcontroller



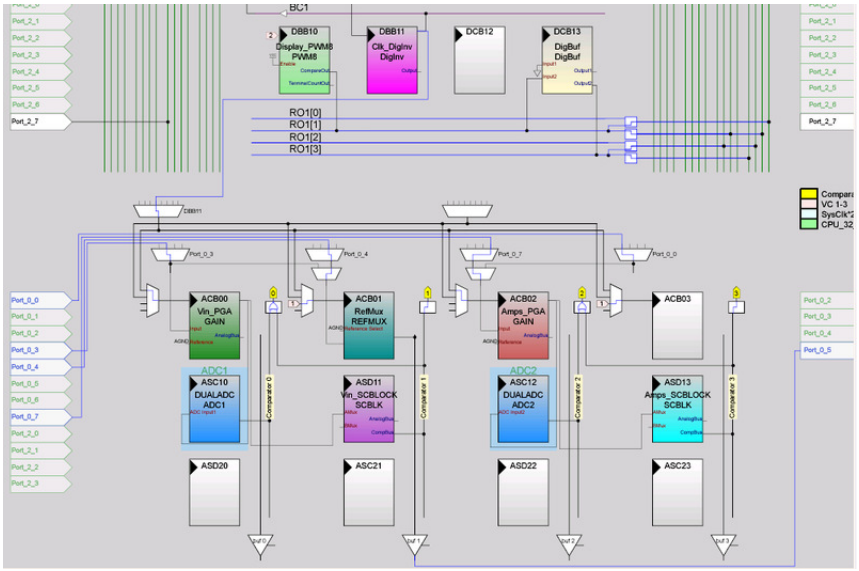
System-on-a-chip





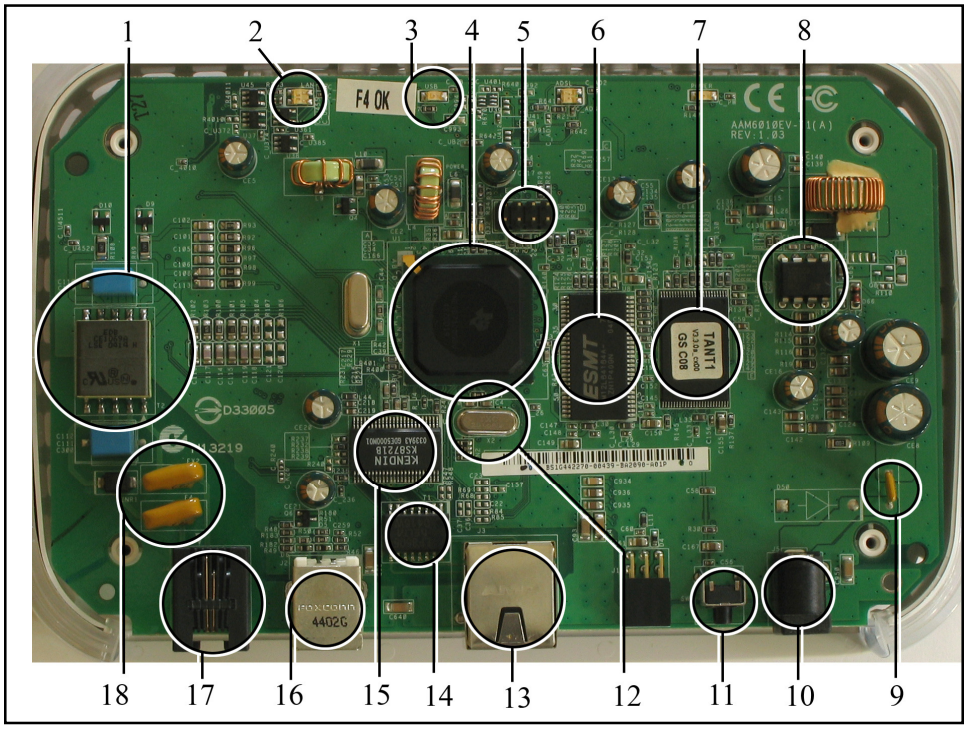
Compare to lab-on-a-chip ideas

Example: PSoC (Cypress)



Example of circuit

- Notice the processor/controller
- What does it do?



Memory types

- Volatile. This is memory that only works as long as it is powered. It loses its stored value when power is removed, but can be used as memory for temporary data storage.
  - commonly been called RAM (Random Access Memory)
- Non-volatile. This is memory that retains its stored value even when power is removed.
  - hard disk
  - In an embedded system it is achieved using non-volatile semiconductor memory, commonly been called ROM (Read-Only Memory)
- An ideal memory reads and writes in negligible time, retains its stored value indefinitely, occupies negligible space and consumes negligible power.
  - In practice no memory technology meets all these happy ideals

Memory – cont.

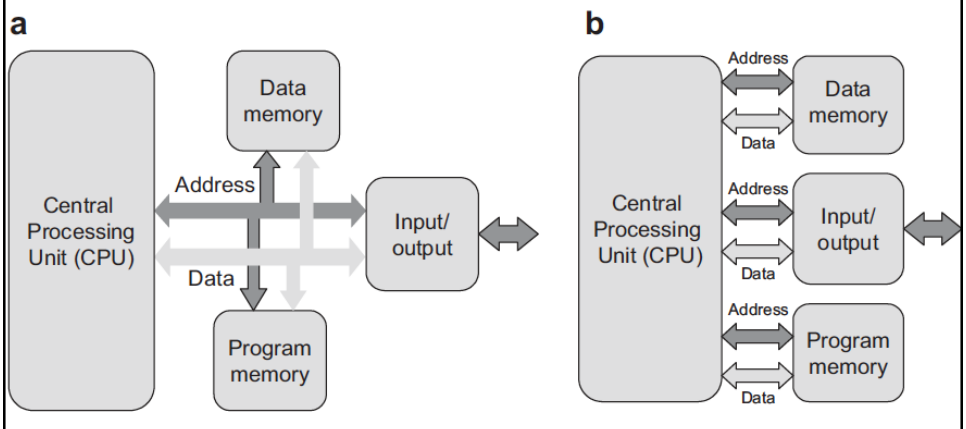
- Static RAM (SRAM)
  - Each memory cell is designed as a simple flip-flop.
  - Data is held only as long as power is supplied (volatile).
  - Consume very little power, and can retain its data down to a low voltage (around 2 V).
  - Each cell taking six transistors, SRAM is not a high-density technology.
- EPROM (Erasable Programmable Read-Only Memory)
  - erased by exposing it to intense ultraviolet light.
  - each memory cell is made of a single MOS transistor (very high density and robust)
  - Within the transistor there is embedded a 'floating gate'. Using a technique known as hot electron injection (HEI), the floating gate can be charged. When it is not charged, the transistor behaves normally and the cell output takes one logic state when activated. When it is charged, the transistor no longer works properly and it no longer responds when it is activated. The charge placed on the floating gate is totally trapped by the surrounding insulator. Hence EPROM technology is non-volatile.
  - Requires quartz window and ceramic packaging.
  - As a technology, EPROM has now almost completely given way to Flash

Memory – cont.

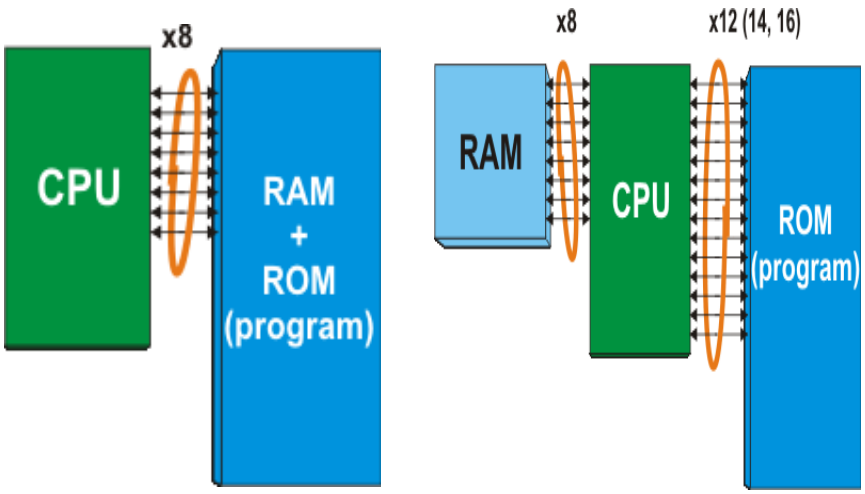
- EEPROM (Electrically Erasable Programmable Read-Only Memory)
  - Uses floating gate technology.
    - This is known as Nordheim–Fowler tunnelling (NFT).
    - With NFT, it is possible to electrically erase the memory cell as well as write to it.
    - To allow this to happen, a number of switching transistors need to be included around the memory element itself, so the high density of EPROM is lost.
    - EEPROM is non-volatile
      - Because the charge on the floating gate is totally trapped by the surrounding insulator
    - Write and erase byte by byte.
- Flash
  - A further evolution of floating-gate technology.
  - can only erase in blocks.
  - Non-volatile

Organizing memory

- Two important buses
  - Address
  - Data
- Two architectures
  - Von Neumann architecture
    - One bus for data and one for address
    - Serve for memory and others (program and I/O)
    - Disadvantages:
      - uses the same data bus for all sizes (areas) of memory
      - Shared between many things, can be used for one thing at a time
  - The Harvard architecture
    - Every memory area gets its own address bus and its own data bus
    - Disadvantages:
      - Complex
      - Data and program are separated (tables inside program memory can't be treated as data)



(a) The Von Neumann way. **(b) The Harvard way**



(a) The Von Neumann way.

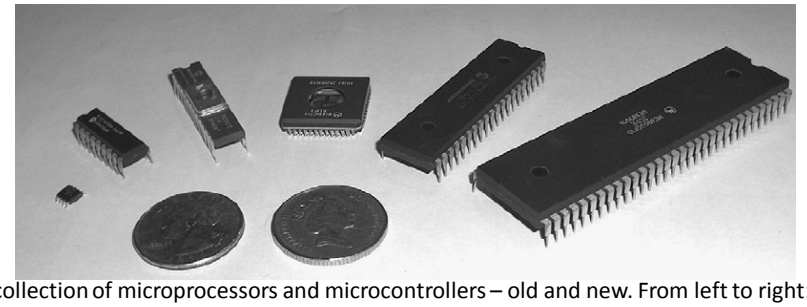
(b) The Harvard way

uC

- Many families
  - Each family is generally the same processor core with different peripherals combination and different memory size.
  - One core might be 8-bit with limited power, another 16-bit and another a sophisticated 32-bit machine.
  - Because the core is fixed for all members of one family, the instruction set is fixed and users have little difficulty in moving from one family member to another.

Microcontroller packaging and appearance

- Usually plastic or ceramic are used as the packaging material.
- Interconnection with the outside world is provided by the pins on the package
- Dual-in-line package (DIP), with its pins arranged in two rows along the longer sides of the IC, the pin spacing being 0.1 inches
- Other incudes: Pin grid array (PGA), leadless chip carrier (LCC) packages, Small-outline integrated circuit (SOIC), Plastic leaded chip carrier (PLCC) packages, plastic quad flat pack(PQFP), and thin small-outline packages (TSOP)
  - Used when number of pins is very large compared to IC size



A collection of microprocessors and microcontrollers – old and new. From left to right: PIC 12F508, PIC 16F84A, PIC 16C72, Motorola 68HC705B16, PIC 16F877, Motorola 68000

Microchip and the PIC microcontroller

- Has a wide range of different families
  - 8-bit, 16-bit, 32-bit
- All 8-bit PIC microcontrollers are lowcost, self-contained, pipelined, RISC, use the Harvard structure, have a single accumulator (the Working, or W, register), with a fixed reset vector.
- Microchip offer 8-bit microcontrollers with four different prefixes, 10-, 12-, 16-, and 18-, for example 10F200, or 18F242. We call each a 'Series', for example '12 Series', '16 Series', '18 Series'.
  - Each Series is identified by the first two digits of the device code.
- Letters are used as follows:
  - The 'C' insert implies CMOS technology
  - The 'F' insert indicates incorporation of Flash memory technology (still using CMOS as the core technology).
  - An 'A' after the number indicates a technological upgrade on the first issue device.
  - An 'X' indicates that a certain digit can take a number of values.
  - For example, the 16C84, the 16F84, and the 16F84A.
- In some cases microcontrollers of one Series can fall into more than one family.

RISC vs. CISC

- Reduced Instruction Set Computer (RISC)
  - Used in: SPARC, ALPHA, Atmel AVR, etc.
  - Few instructions (usually < 50)
  - Only a few addressing modes
  - Executes 1 instruction in 1 internal clock cycle (Tcyc)
- Complex Instruction Set Computer (CISC)
  - Used in: 80X86, 8051, 68HC11, etc.
  - Many instructions (usually > 100)
  - Several addressing modes
  - Usually takes more than 1 internal clock cycle (Tcyc) to execute

Family Core Architecture Differences

- The PIC Family: Cores
  - 12bit cores with 33 instructions: 12C50x, 16C5x
  - 14bit cores with 35 instructions: 12C67x,16Cxxx
  - 16bit cores with 58 instructions: 17C4x,17C7xx
  - 'Enhanced' 16bit cores with 77 instructions: 18Cxxx

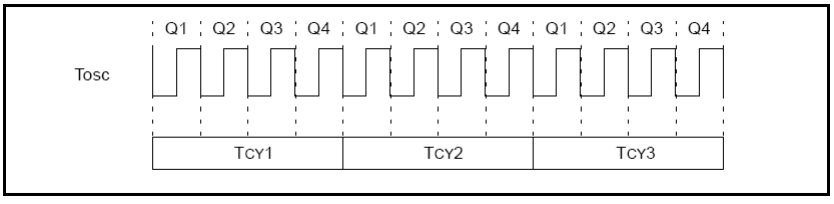
The PIC Family: Speed

- Can use crystals, clock oscillators, or even an RC circuit.
- Some PICs have a built in 4MHz RC clock, Not very accurate, but requires no external components!
- Instruction speed = 1/4 clock speed (Tcyc = 4 * Tclk)
- All PICs can be run from DC to their maximum specified speed:

12C50x	4MHz
12C67x	10MHz
16Cxxx	20MHz
17C4x / 17C7xxx	33MHz
18Cxxx	40MHz

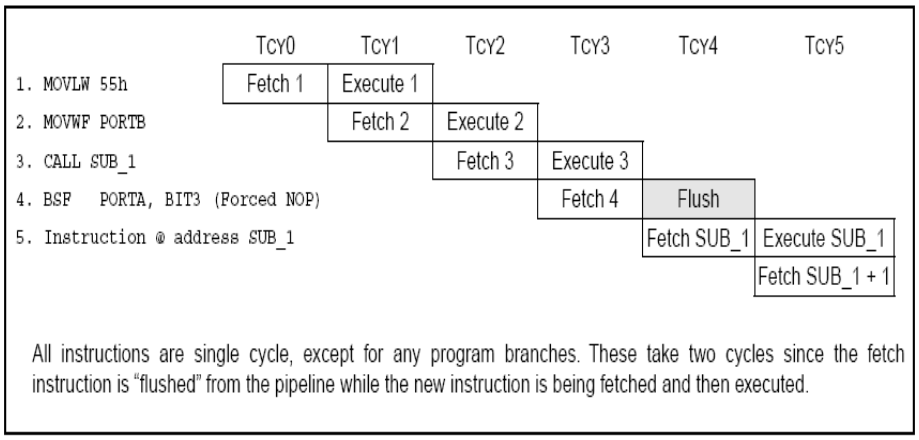
Clock and Instruction Cycles

- Instruction Clock
  - Clock from the oscillator enters a microcontroller via OSC1 pin where internal circuit of a microcontroller divides the clock into four even clocks Q1, Q2, Q3, and Q4 which do not overlap.
  - These four clocks make up one **instruction cycle** (also called machine cycle) during which one instruction is executed.
  - Execution of instruction starts by calling an instruction that is next in string.
  - Instruction is called from program memory on every Q1 and is written in instruction register on Q4.
  - Decoding and execution of instruction are done between the next Q1 and Q4 cycles. On the following diagram we can see the relationship between instruction cycle and clock of the oscillator (OSC1) as well as that of internal clocks Q1-Q4.
  - Program counter (PC) holds information about the address of the next instruction.



Pipelining in PIC

- Instruction Pipeline Flow



All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is "flushed" from the pipeline while the new instruction is being fetched and then executed.

The PIC Family: Program Memory

- Technology: EPROM, FLASH, or ROM
- It varies in size from one chip to another.
  - examples:

12C508	**512**	**12bit instructions**
16C711	**1024 (1k)**	**14bit instructions**
16F877	**8192 (8k)**	**14bit instructions**
17C766	**16384 (16k)**	**16bit instructions**

The PIC Family: Data Memory

- PICs use general purpose "File registers" for RAM (each register is 8bits for all PICs)
  - examples:

12C508	25B RAM
16C71C	36B RAM
16F877	368B RAM + 256B of nonvolatile EEPROM
17C766	902B RAM

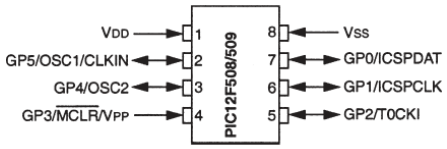
Comparison of 8-bit PIC families

Family	Example devices	Instruction word size	Stack size (words)	Number of instructions	Interrupt vectors
Baseline	10F200, 12F508, 16F57	12 bit	2	33	None
Mid range	12F609, 16F84A, 16F631, 16F873A	14 bit	8	35	1
High Performance	18F242, 18F2420	16 bit	32	75, including hardware multiply	2 (prioritised)

- Every member of any one family shares the same core architecture and instruction set.
- The processing power is defined to some extent by the parameters quoted, for example the instruction word size, and the number of instructions.

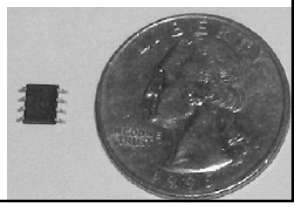
An introduction to PIC microcontrollers using the Baseline Series

- We will look at the PIC 12F508/509.
- The only difference between the 508 and 509 is that the latter has slightly larger program and data memories.

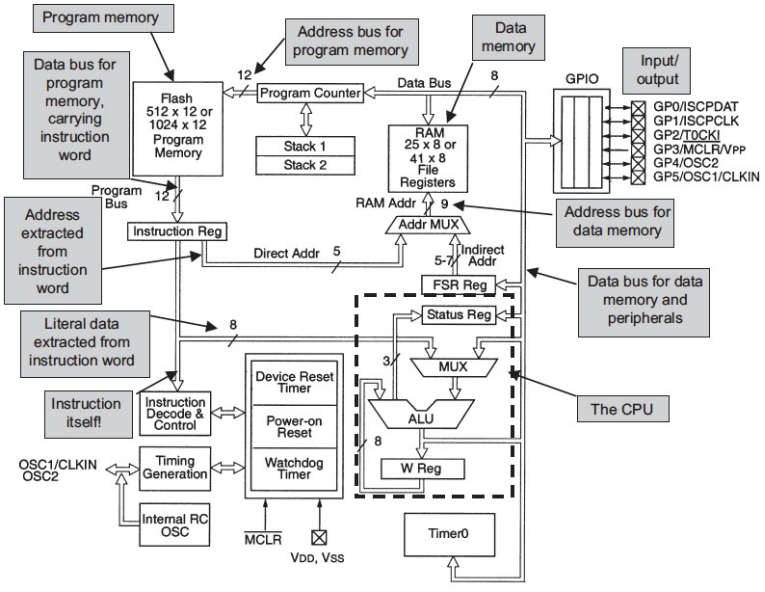


Key

V_{DD}:	Power supply	V_{SS}:	Ground
V_{PP}:	Programming voltage input	MCLR:	Master clear
OSC1, OSC2:	Oscillator pins	CLKIN:	External clock input
GP0 to GP5:	General Purpose input/output pins (bidirectional except GP3)		
CSPDAT:	In Circuit Serial Programming™ data pin.		
CSPCLK:	In Circuit Serial Programming™ clock pin.		



The architecture of the 12F508



Key *(See also Key to Figure 1.11)*

FSR:	File Select Register	GPIO:	General Purpose Input/Output
MUX:	Multiplexer	RC:	Resistor capacitor
W reg:	Working register		

The architecture of the 12F508 – cont.

- As this microcontroller is a RISC computer, each instruction word must carry not only the instruction code itself, but also any address or data information needed.
- Depending on the instruction itself, five bits of the instruction word may carry address information and hence be sent down the 'Direct Addr' bus to the address multiplexer ('Addr MUX'). Eight bits of the instruction word may carry a data byte that is to be used as literal data for the execution of that instruction. This goes to the multiplexer ('MUX'), which feeds into the ALU. Finally, there is the instruction data itself, which feeds into the 'Instruction Decode and Control' unit.
- A 'Power-on Reset' function detects when power is applied and holds the microcontroller in a Reset condition while the power supply stabilizes.
- The MCLR input can be used to place the CPU in a Reset condition and to force the program to start again.
  - An internal clock oscillator ('Internal RC OSC') is provided so that no external pins whatsoever need be committed to this function.
  - External oscillator connections can, however, be made, using input/output pins GP4 and GP5. The oscillator signal is conditioned for use through the microcontroller in the 'Timing Generation' unit.
- The 'Watchdog Timer' is a safety feature, used to force a reset in the processor if it crashes.

The PIC 16F877

- We will concentrate on just one device, the PIC 16F877
  - A good range of features and allows most of the essential techniques to be explained.
  - It has a set of serial ports built in, which are used to transfer data to and from other devices, as well as analogue inputs, which allow measurement of inputs such as temperature.
  - All standard types of microcontrollers work in a similar way, so analysis of one will make it possible to understand all the others.

MCU

- The microcontroller contains the same main elements as any computer system:
  - Processor
  - Memory
  - Input/Output
- In a PC, these are provided as separate chips, linked together via bus connections on a printed circuit board, but under the control of the microprocessor (CPU).
- A bus is a set of lines which carry data in parallel form which are shared by the peripheral devices.
- The system can be designed to suit a particular application, with the type of CPU, size of memory and selection of input/output (I/O) devices tailored to the system requirements.
- In the microcontroller, all these elements are on one chip. This means that the MCU (microcontroller) for a particular application must be chosen from the available range to suit the requirements.

PIC 16F877 Architecture

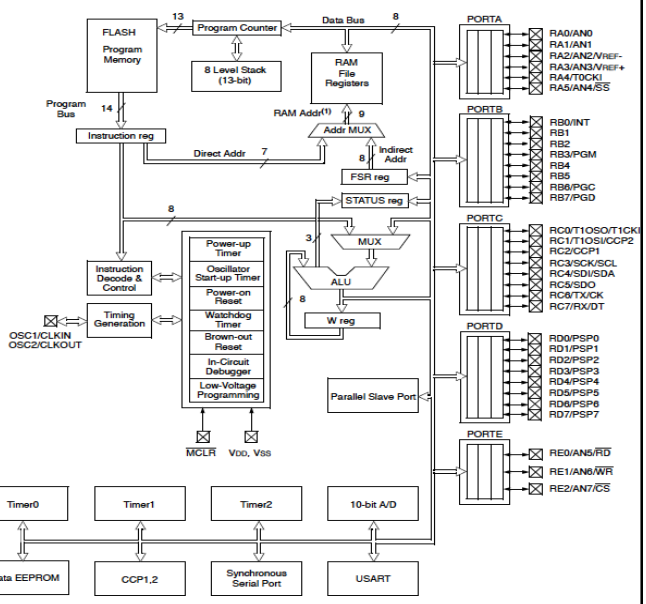
- Microcontrollers contain all the components required for a processor system in one chip: a CPU, memory and I/O.
- A complete system can therefore be built using one MCU chip and a few I/O devices such as a keypad, display and other interfacing circuits.

Datasheet

http://bit.ly/XgtauM

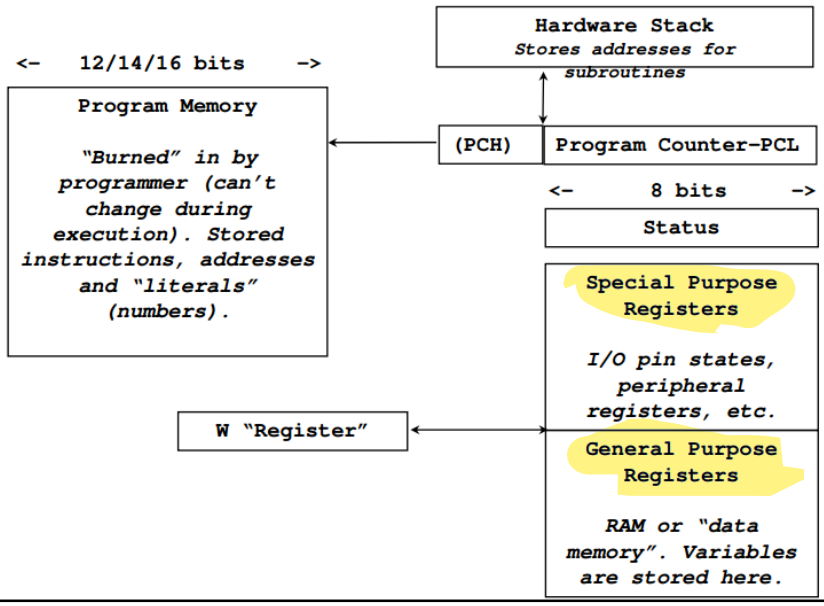
An architecture overview of the PIC16F877

Device	Program FLASH	Data Memory	Data EEPROM
PIC16F874	4K	192 Bytes	128 Bytes
PIC16F877	8K	368 Bytes	256 Bytes



Note 1: Higher order bits are from the STATUS register.

Software: Programmers Model



PIC Programming Procedure

- For example: in programming an embedded PIC featuring electronically erasable programmable read-only memory (EEPROM). The essential steps are:
  - Step 1: On a PC, type the program, successfully compile it and then generate the HEX file.
  - Step 2: Using a PIC device programmer, upload the HEX file into the PIC. This step is often called **"burning"**.
  - Step 3: Insert your PIC into your circuit, power up and verify the program works as expected. This step is often called **"dropping"** the chip. If it isn't, you must go to Step 1 and **debug** your program and repeat burning and dropping.

PIC16F877A Features

High Performance RISC CPU:

- Only 35 single word instructions to learn
- All single cycle instructions except for program branches, which are two-cycle
- Operating speed: DC - 20 MHz clock input DC - 200 ns instruction cycle

PIC 16F877 Pin IN/Out

- The chip can be obtained in different packages, such as conventional 40-pin DIP (Dual In-Line Package), square surface mount or socket format.
- Most of the pins are for input and output, and arranged as 5 ports: port A (5 pins), port B(8), C(8), D(8) and E(3), giving a total of 32 I/O pins.
  - These can all operate as simple digital I/O pins, but most have more than one function.
  - The mode of operation of each is selected by initializing various control registers within the chip.
  - Note, in particular, that Ports A and E become ANALOGUE INPUTS by default (on power up or reset), so they have to set up for digital I/O if required.

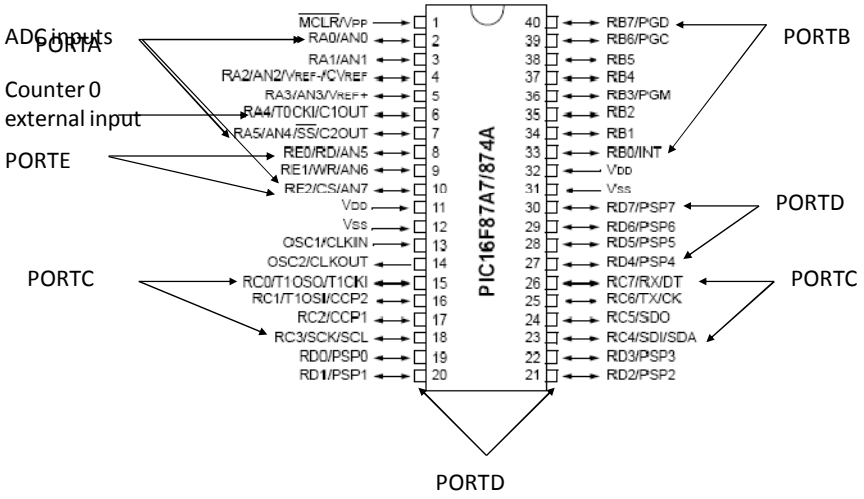
PIC 16F877 Pin IN/Out – cont.

- Port B is used for downloading the program to the chip flash ROM (RB6 and RB7), and RB0 and RB4–RB7 can generate an interrupt.
- Port C gives access to timers and serial ports.
- Port D can be used as a slave port, with Port E providing the control pins for this function.

PIC 16F877 Pin IN/Out – cont.

- The chip has two pairs of power pins ($\mathbf{V}_{DD}$=5 V nominal and $\mathbf{V}_{ss}$ =0 V) (11,12 or 31, 32), and either pair can be used.
- The chip can work down to about 2 V supply, for battery and power-saving operation.
- A low-frequency clock circuit using only a capacitor and resistor to set the frequency can be connected to CLKIN, or a crystal oscillator circuit can be connected across CLKIN and CLKOUT.
- MCLR is the reset input; when cleared to 0, the MCU stops, and restarts when MCLR=1. This input must be tied high allowing the chip to run if an external reset circuit is not connected.
  - It is usually a good idea to incorporate a manual reset button in all but the most trivial applications.

PIC16F877A Pin Layout



PIC 16F877 Pin IN/Out – cont.

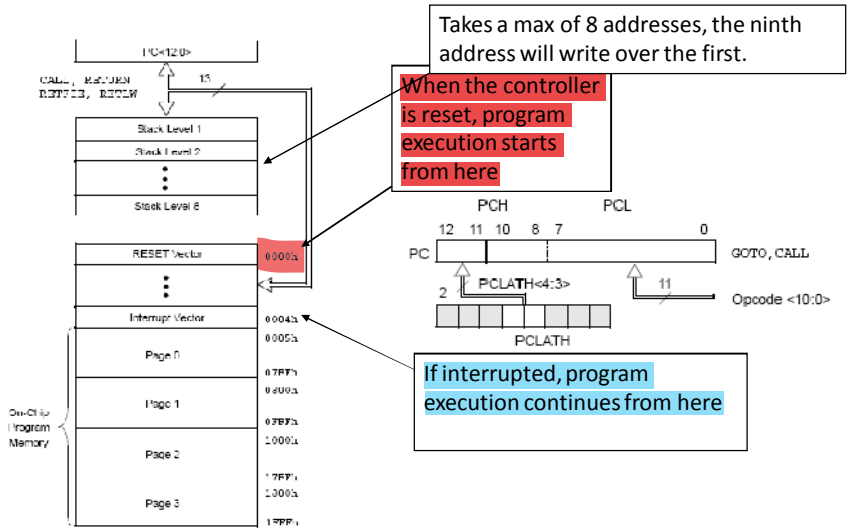
Reset = 0, Run = 1	**MCLR**	1	40	**RB7**	Port B, Bit 7 (Prog. Data, Interrupt)
Port A, Bit 0 (Analogue AN0)	**RA0**	2	39	**RB6**	Port B, Bit 6 (Prog. Clock, Interrupt))
Port A, Bit 1 (Analogue AN1)	**RA1**	3	38	**RB5**	Port B, Bit 5 (Interrupt)
Port A, Bit 2 (Analogue AN2)	**RA2**	4	37	**RB4**	Port B, Bit 4 (Interrupt)
Port A, Bit 3 (Analogue AN3)	**RA3**	5	36	**RB3**	Port B, Bit 3 (LV Program)
Port A, Bit 4 (Timer 0)	**RA4**	6	35	**RB2**	Port B, Bit 2
Port A, Bit 5 (Analogue AN4)	**RA5**	7	34	**RB1**	Port B, Bit 1
Port E, Bit 0 (AN5, Slave control)	**RE0**	8	33	**RB0**	Port B, Bit 0 (Interrupt)
Port E, Bit 1 (AN6, Slave control)	**RE1**	9	32	**V_{DD}**	+5V Power Supply
Port E, Bit 2 (AN7, Slave control)	**RE2**	10	31	**Vss**	0V Power Supply
+5V Power Supply	**V_{DD}**	11	30	**RD7**	Port D, Bit 7 (Slave Port)
0V Power Supply	**Vss**	12	29	**RD6**	Port D, Bit 6 (Slave Port)
(CR clock) XTAL circuit	**CLKIN**	13	28	**RD5**	Port D, Bit 5 (Slave Port)
XTAL circuit	**CLKOUT**	14	27	**RD4**	Port D, Bit 4 (Slave Port)
Port C, Bit 0 (Timer 1)	**RC0**	15	26	**RC7**	Port C, Bit 7 (Serial Ports)
Port C, Bit 1 (Timer 1)	**RC1**	16	25	**RC6**	Port C, Bit 6 (Serial Ports)
Port C, Bit 2 (Timer 1)	**RC2**	17	24	**RC5**	Port C, Bit 5 (Serial Ports)
Port C, Bit 3 (Serial Clocks)	**RC3**	18	23	**RC4**	Port C, Bit 4 (Serial Ports)
Port D, Bit 0 (Slave Port)	**RD0**	19	22	**RD3**	Port D, Bit 3 (Slave Port)
Port D, Bit 1 (Slave Port)	**RD1**	20	21	**RD2**	Port D, Bit 2 (Slave Port)

PIC Memory

- The PIC16F877A has an 8192 (8k) 14bit instruction program memory
- 368 Bytes Registers as Data Memory :
  - Special Function Registers: used to control peripherals and PIC behaviors
  - General Purpose Registers: used to a normal temporary storage space (RAM)
- 256 Bytes of nonvolatile EEPROM

PIC Program Memory

- The PIC16F877 8192 (8k) 14bit instructions



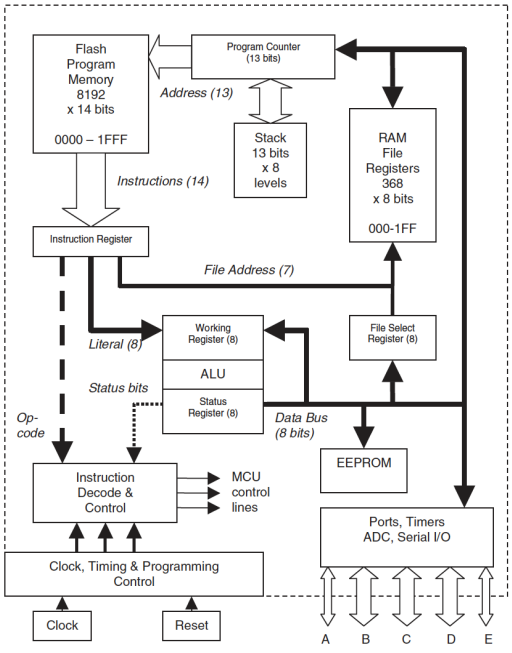
PIC16F877 block diagram – cont.

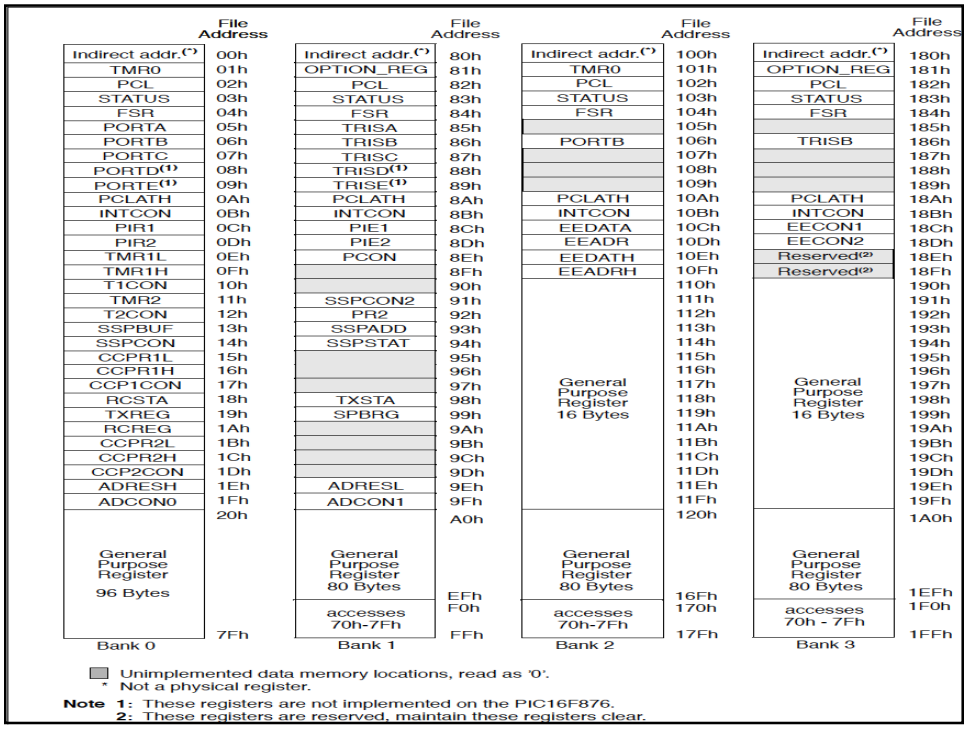
- The main program memory is flash ROM, which stores a list of 14-bits instructions.
- Instructions are fed to the execution unit, and used to modify the RAM file registers.
- The file register include special control registers, the port registers and a set of general purpose registers which can be used to store data temporarily.
- A separate working register (W) is used with the ALU (Arithmetic Logic Unit) to process data. Various special peripheral modules provide a range of I/O options

PIC16F877 block diagram – cont.

- There are 512 RAM File Register addresses (0–1FFh), which are organized in 4 banks (0–3), each bank containing 128 addresses.
- The default (selected on power up) is Bank 0 which is numbered from 0 to 7Fh, Bank 1 from 80h to FFh and so on.
- These contain both Special Function Registers (SFRs), which have a dedicated purpose, and the General Purpose Registers (GPRs).
- The file registers are mapped as seen in the next slid.
- The SFRs may be shown in the block diagram as separate from the GPRs, but they are in fact in the same logical block, and addressed in the same way.
- Deducting the SFRs from the total number of RAM locations, and allowing for some registers which are repeated in more than one bank, leaves 368 bytes of GPR (data) registers.

PIC 16F877 Block Diagram



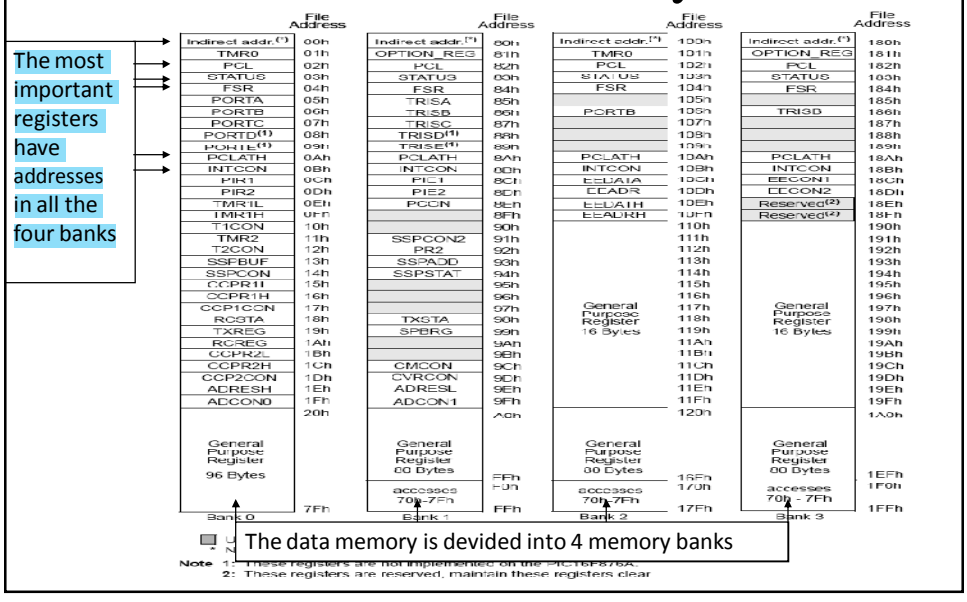


File Address	Bank 0	File Address	Bank 1	File Address	Bank 2	File Address	Bank 3
00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	180h	Indirect addr.(*)
01h	TMR0	81h	OPTION_REG	101h	TMR0	181h	OPTION_REG
02h	PCL	82h	PCL	102h	PCL	182h	PCL
03h	STATUS	83h	STATUS	103h	STATUS	183h	STATUS
04h	FSR	84h	FSR	104h	FSR	184h	FSR
05h	PORTA	85h	TRISA	105h		185h	
06h	PORTB	86h	TRISB	106h	PORTB	186h	TRISB
07h	PORTC	87h	TRISC	107h		187h	
08h	PORTD(1)	88h	TRISD(1)	108h		188h	
09h	PORTE(1)	89h	TRISE(1)	109h		189h	
0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah	PCLATH
0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh	INTCON
0Ch	PIR1	8Ch	PIE1	10Ch	EEDATA	18Ch	EECON1
0Dh	PIR2	8Dh	PIE2	10Dh	EEADR	18Dh	EECON2
0Eh	TMR1L	8Eh	PCON	10Eh	EEDATH	18Eh	Reserved(2)
0Fh	TMR1H	8Fh		10Fh	EEADRH	18Fh	Reserved(2)
10h	T1CON	90h		110h		190h	
11h	TMR2	91h	SSPCON2	111h		191h	
12h	T2CON	92h	PR2	112h		192h	
13h	SSPBUF	93h	SSPADD	113h		193h	
14h	SSPCON	94h	SSPSTAT	114h		194h	
15h	CCPR1L	95h		115h		195h	
16h	CCPR1H	96h		116h		196h	
17h	CCP1CON	97h		117h	General Purpose Register 16 Bytes	197h	General Purpose Register 16 Bytes
18h	RCSTA	98h	TXSTA	118h		198h	
19h	TXREG	99h	SPBRG	119h		199h	
1Ah	RCREG	9Ah		11Ah		19Ah	
1Bh	CCPR2L	9Bh		11Bh		19Bh	
1Ch	CCPR2H	9Ch		11Ch		19Ch	
1Dh	CCP2CON	9Dh		11Dh		19Dh	
1Eh	ADRESH	9Eh	ADRESL	11Eh		19Eh	
1Fh	ADCON0	9Fh	ADCON1	11Fh		19Fh	
20h		A0h		120h		1A0h	
	General Purpose Register 96 Bytes		General Purpose Register 80 Bytes		General Purpose Register 80 Bytes		General Purpose Register 80 Bytes
7Fh		EFh	accesses 70h-7Fh	16Fh	accesses 70h-7Fh	1EFh	accesses 70h - 7Fh
		F0h		170h		1F0h	
	Bank 0	FFh	Bank 1	17Fh	Bank 2	1FFh	Bank 3

Unimplemented data memory locations, read as '0'.
* Not a physical register.

Note 1: These registers are not implemented on the PIC16F876.
2: These registers are reserved, maintain these registers clear.

PIC Data Memory

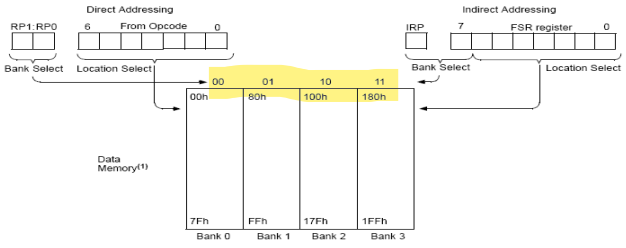


Register Addressing Modes

- There are 3 types of addressing modes in PIC
  - Immediate Addressing
    - Movlw H'0F'
  - Direct Addressing
  - Indirect Addressing

Register Addressing Modes

Immediate Addressing:
Movlw H'0F'



Direct Addressing:

Uses 7 bits of 14 bit instruction to identify a register file address 8th and 9th bit comes from RP0 and RP1 bits of STATUS register.

i.e. Z equ D'2' ; Z=2
btfss STATUS, Z ; test if the 3rd bit of the STATUS register is set

Indirect Addressing:

- Full 8 bit register address is written the special function register FSR
- INDF is used to get the content of the address pointed by FSR
- Exp : A sample program to clear RAM locations H'20' – H'2F:

```
        MOVLW 0x20 ;initialize pointer
        MOVWF FSR ;to RAM
NEXT    CLRF INDF ;clear INDF register
        INCF FSR,F  ;inc pointer
        BTFSS FSR,4 ;all done?
        GOTO NEXT ;no clear next
CONTINUE
: ;
```

0010 0000 -> 0011 0000

b:4

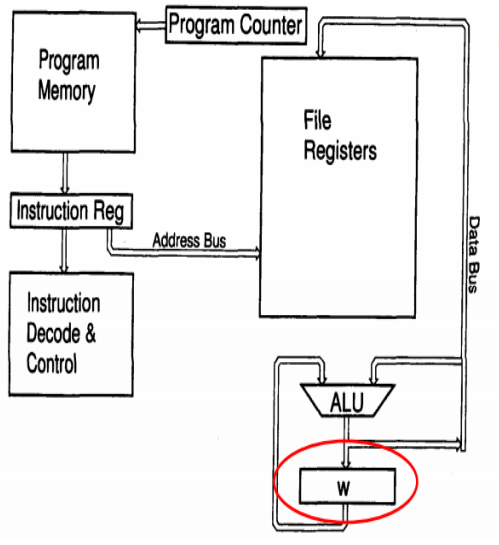
Indirect Addressing: /2

- for instance,
  - one general purpose register (GPR) at address 0Fh contains a value of 20
  - By writing a value of 0Fh in FSR register we will get a register indicator at address 0Fh,
  - and by reading from INDF register, we will get a value of 20, which means that we have read from the first register its value without accessing it directly (but via FSR and INDF).
- It appears that this type of addressing does not have any advantages over direct addressing, but certain needs do exist during programming which can be solved smoothly only through indirect addressing.
- Indirect addressing is very convenient for manipulating data arrays located in GPR registers.
  - In this case, it is necessary to initialize FSR register with a starting address of the array, and the rest of the data can be accessed by incrementing the FSR register.

PIC Family Control Registers

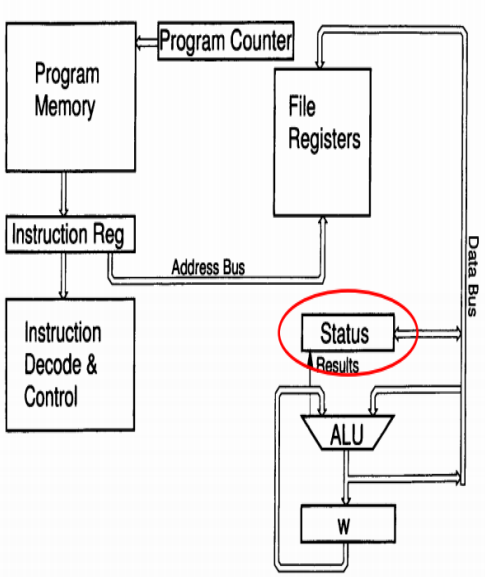
- Uses a series of "Special Function Registers" for controlling peripherals and PIC behaviors.
  - **STATUS** ➔ Bank select bits, ALU bits (zero, borrow, carry)
  - **INTCON** ➔ Interrupt control: interrupt enables, flags, etc.
  - **OPTION_REG** ➔ contains various control bits to configure the TMR0 prescaler/WDT postscaler ,the External INT Interrupt, TMR0 and the weak pull-ups on PORTB

the accumulator



- to add two numbers together
  - first move the contents of one file register into the w register
  - then add the contents of the second file register to w
  - the result can be written to w or to the second file register

the status register



- the STATUS register stores 'results' of the operation
- three of the bits of the STATUS register are set based on the result of an arithmetic or bitwise operation

the STATUS register

- three of the bits of the STATUS register are set based on the result of an arithmetic or bitwise operation
  - zero flag ; this bit is set whenever the result of an operation is zero
  - carry flag ; this bit is set whenever the result of an operation is greater than 255 (0xFF) ; can be used to indicate that higher order bytes need to be updated
  - digit carry flag ; this bit is set whenever the least significant four bits of the result of an operation is greater than 15 (0x0F)

Special Function Register "STATUS Register"

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 7 **IRP:** Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h - 1FFh)
0 = Bank 0, 1 (00h - FFh)

bit 6-5 **RP1:RP0**: Register Bank Select bits (used for direct addressing)
1 1 = Bank 3 (180h - 1FFh)
1 0 = Bank 2 (100h - 17Fh)
0 1 = Bank 1 (80h - FFh)
0 0 = Bank 0 (00h - 7Fh)
Each bank is 128 bytes

bit 4 **$\overline{TO}$**: Time-out bit
1 = After power-up, CLRWDT instruction, or SLEEP instruction
0 = A WDT time-out occurred

bit 3 **$\overline{PD}$**: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit carry/$\overline{borrow}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions)
(for $\overline{borrow}$, the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{borrow}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions)
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

Note: For $\overline{borrow}$, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high- or low-order bit of the source register.

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Special Function Register "INTCON Register"

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **PEIE**: Peripheral Interrupt Enable bit
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **TMR0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **TMR0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state; a mismatch condition will continue to set the bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared (must be cleared in software).
0 = None of the RB7:RB4 pins have changed state

We should write this

MOVF PORTB, F

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

X14 Instruction set

- 35 instructions
  - –Byte Oriented Operations
  - –Bit Oriented Operations
  - –Literal and control Operations

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 module.

3: If Program Counter (PC) is modified, or a conditional test is true, the instruction requires two cycles. The second cycle is

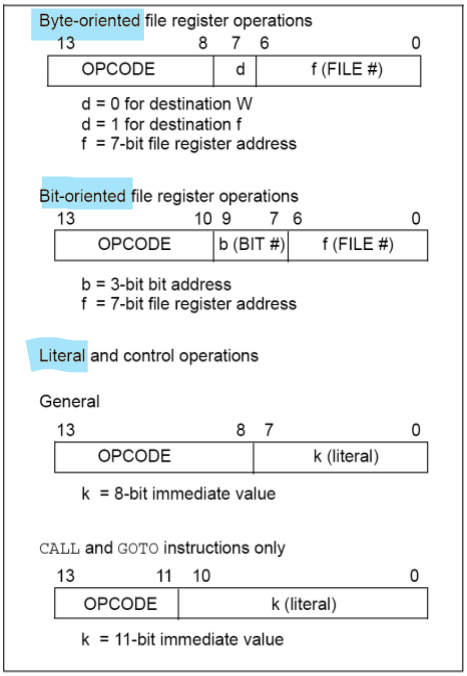
- Data Movement
  - movf,movlw,movwf
- Arithmetic
  - addlw,addwf,sublw,subwf,incf,decf
- Logical
  - andlw,andwf,iorlw,iorwf,xorlw,xorwf,rrf,rlf,clrf,clrw,swapf,comf
- Bit Operators
  - bsf,bcf
- Branching
  - goto,btfss,btfsc,decfsz,incfsz
- Subroutine
  - call,return,retlw,retfie
- Misc.
  - sleep,clrwdt,nop

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff	Z	1,2
CLRF	f	Clear f	1	00 0001 lfff ffff	Z	2
CLRW	-	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff	Z	1,2
DECF	f, d	Decrement f	1	00 0011 dfff ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff		1,2,3
INCF	f, d	Increment f	1	00 1010 dfff ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00 1111 dfff ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	1,2
MOVF	f, d	Move f	1	00 1000 dfff ffff	Z	1,2
MOVWF	f	Move W to f	1	00 0000 lfff ffff		
NOP	-	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	1,2

For **byte-oriented** instructions, 'f' represents a file register designator and 'd' represents a destination designator. The file register designator specifies which file register is to be used by the instruction.
The destination designator specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the W register. If 'd' is one, the result is placed in the file register specified in the instruction

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		1,2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	k	Call subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	-	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	-	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	-	Go into standby mode	1	00 0000 0110 0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	

For **bit-oriented** instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the address of the file in which the bit is located.
For **literal and control** operations, 'k' represents an eight or eleven bit constant or literal value.



1. Copy value from/to file register or literal to/from w

Mnemonic	Description	Status	Function
movf fr, d	Move file register	Z	fr => d
movwf fr	Move W to file register		w => fr
movlw k	Move literal to W		k => W

Move Commands:

movlw 0xF2 : stores the number 0xF2 into the W register

movwf 0x0C : stores the W register contents into file H'0C'

movf 0x0C,w : loads the contents of file H'0C' into W register

movf 0x0C,f : loads the contents of file H'0C' into file H'0C'

2. Logic / arithmetic instructions with a file register and w

Mnemonic	Description	Status	Function
addwf fr,d	addition	Z, DC,C	fr + W => d
subwf fr,d	subtraction	Z, DC,C	fr - W => d
andwf fr,d	Logical and	Z	fr AND W => d
iorwf fr,d	Logical or	Z	fr OR W => d
xorwf fr,d	xor	Z	fr XOR W => d

addwf instruction

General form:

addwf *floc*, *d* *d*← [*floc*] + w

floc is a memory location in the file registers (data memory)

w is the working register

d is the destination, can either be the literal 'f' or 'w'

[floc] means "the contents of memory location *floc*"

addwf 0x70,w w ← [0x70] + w

addwf 0x70,f [0x70] ← [0x70] + w

3. Logic / arithmetic instructions with literal and w

Mnemonic	Description	Status	Function
addlw k	addition	Z, DC,C	W + k => W
sublw k	subtraction	Z, DC,C	W- k => W
andlw k	Logical and	Z	W AND k => W
iorlw k	Logical or	Z	W OR k => W
xorlw k	xor	Z	W XOR k => W

4.One operand logic / arithmetic instructions

Mnemonic	Description	Status	Function
clrw	Clear accumulator W	Z	0 => W
clrf fr	Clear file register fr	Z	0 => fr
decf fr,d	Decrement file register fr	Z	fr - 1 => d
incf fr, d	Increment file register fr	Z	fr + 1 => d
comf fr,d	1's complement file register fr	Z	not fr => d
rlf fr, d	Rotate file register fr left thru C	C	C <= fr(7), fr(i) <= fr(i-1), fr(0) <= C
rrf fr, d	Rotate file register fr right thru C	C	C => fr(7), fr(i) => fr(i-1), fr(0) => C
bcf fr, b	Bit clear on file register fr		0 => fr(b)
bsf fr, b	Bit set on file register fr		1 => fr(b)
swapf fr,d	swap halves of fr		(fr(0:3) <=> fr(4:7)) => d
nop	No operation		

Bit Set/Clear Commands

```
bcf    0x0C,0      : clear the 0th bit of file H'0C'
bsf    0x0D,3      : set the 3rd bit of file H'0D'
btfsc  0x42,0      : test the 0th bit of the file H'42', if it
                     is 0, then skip the next line of code.
btfss  0x43,1      : test the 1st bit of the file H'43', if it
                     is 1, then skip the next line of code.
```

5. Branch, Skip and Call instructions

Mnemonic	Description	Status	Function
goto addr	branch to addr		addr => PC(0:10)
call addr	call routine at addr		PC => TOS addr => PC(0:10)
decfsz fr,d	Decrement fr, skip if zero		fr - 1 => d, skip if 0
incfsz fr,d	Increment fr, skip next instr if zero		fr + 1 => d, skip next instr if 0
btfsc fr,b	Bit test fr, skip if clear		skip next instr if fr(b) =0
btfss fr,b	Bit test fr, skip if set		skip next instr if fr(b)=1
return	return from subroutine		TOS => PC
retlw k	return with literal in w		k =>w, TOS => PC
retfie	return from interrupt		TOS => PC, 1 => GIE

TEST, SKIP & JUMP

- Conditional jumps are initiated using a bit test and conditional skip, followed by a GOTO or CALL.
- The bit test can be made on any file register bit.
  - This could be a port bit, to check if an input has changed, or a status bit in a control register.
- BTFSC (Bit Test and Skip if Clear) and BTFSS (Bit Test and Skip if Set) are used to test the bit and skip the next instruction, or not, according to the state of the bit tested.
- DECFSZ and INCFSZ embody a commonly used test – decrement or increment a register and jump depending on the effect of the result on the zero flag (Z is set if result 0).
- The bit test and skip may be followed by a single instruction to be carried out conditionally, but GOTO and CALL allow a block of conditional code.
- Using GOTO label simply transfers the program execution point to some other point in the program indicated by a label in the first column of the source code line.
- A CALL label means that the program returns to the instruction following the CALL when RETURN is encountered at the end of the subroutine.
  - Another option is RETLW (Return with Literal in W). See the KEYPAD.
  - RETFIE (Return From Interrupt) will be explained later.

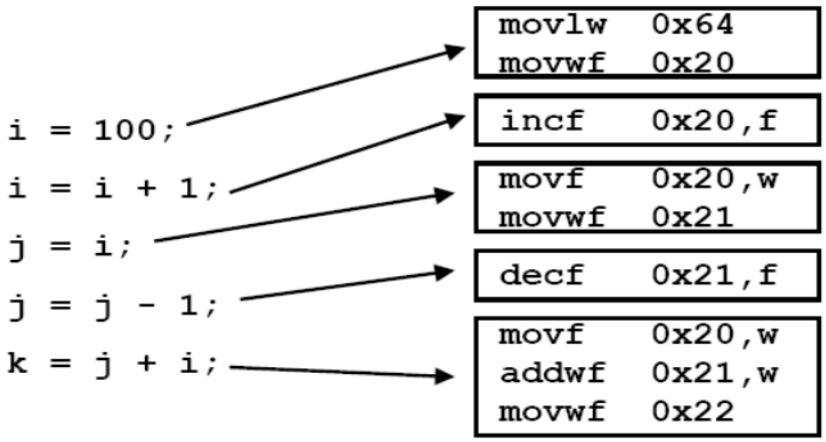
CONTROL

- NOP simply does nothing for one instruction cycle (four clock cycles).
  - very useful for putting short delays in the program
- SLEEP stops the program, such that it can be restarted with an external interrupt.
- The unused locations contain the code 3FFF (all 1 s), which is a valid instruction (ADDLW FF).
- CLRWDT means clear the watchdog timer. If the program gets stuck in a loop or stops for any other reason, it will be restarted automatically by the watchdog timer.
- To stop this from happening, the watchdog timer must be reset at regular intervals of less than, say, 10 ms, within the program loop, using CLRWDT.

OPTIONAL INSTRUCTIONS

- TRIS was an instruction originally provided to make port initialization simpler.
  - It selects register bank 1 so that the TRIS data direction registers (TRISA, TRISB, etc.) can be loaded with a data direction code (e.g. 0→output).
- The manufacturer no longer recommends use of this instruction, although it is still supported
- The assembler directive BANKSEL can be used
  - It gives more flexible access to the registers in banks 1, 2, 3.
- The other option is to change the bank select bits in the STATUS register directly, using BSF and BCF.
- OPTION, providing special access to the OPTION register, is the other instruction, which is no longer recommended.
  - It can be replaced by BANKSEL to select bank 1 which contains the OPTION register, which can then be accessed directly.

C to PIC Assembly



mptest.asm

```
INCLUDE "p16f877.inc"

;  Register Usage
CBLOCK 0x020   ;
 i, j,k    ; reserve space
ENDC

myid equ  D'100'    ; define myid label

  org    0
  movlw  myid    ; w <- 100
  movwf  i       ; i <- w;

  incf   i,f     ; i <- i + 1

  movf   i,w      ; w <- i
  movwf  j       ; j <- w

  decf   j,f     ; j <- j - 1


  movf   i,w     ; w <- I
  addwf  j,w     ; w <- w + j
  movwf  k     ; k <- w

here
  goto   here      ; loop forever
  end
```

This file can be assembled by MPLAB into PIC machine code and simulated.

Labels used for memory locations 0x20 (i), 0x21(j), 0x22(k) to increase code clarity

Chip Configuration Word

- The assembler directive __CONFIG is included at the top of the program, which sets up aspects of the chip operation which cannot be subsequently changed without reprogramming.
- The configuration word is a special area of program memory located outside the normal range (address 2007h) and stores chip configurations such as the clock type.
  - Done by loading the configuration bits with a suitable binary code (see next slide).

Chip Configuration Word

Bit	Label	Function	Default	Enabled	Typical
15	–	None	0	x	0
14	–	None	0	x	0
13	CP1	Code protection	1	0	1
12	CP0	(4 levels)	1	0	1
11	DEBUG	In-circuit debugging (ICD)	1	0	0
10	–	None	1	x	1
9	WRT	Program memory write enable	1	1	1
8	CPD	EEPROM data memory write protect	1	0	1
7	LVP	Low-voltage programming enable	1	1	0
6	BODEN	Brown-out reset (BoR) enable	1	1	0
5	CP1	Code protection (CP)	1	0	1
4	CP0	(repeats)	1	0	1
3	PWRTE	Power-up timer (PuT) enable	1	0	0
2	WDTE	Watchdog timer (WdT) enable	1	1	0
1	FOSC1	Oscillator type select	1	x	0
0	FOSC0	RC = 11, HS = 10, XT = 01, LP = 00	1	x	1

Default = 3FFF (RC clock, PuT disabled, WdT enabled).
Typical RC clock = 3FF3 (RC clock, ICD disabled, PuT enabled, WdT disabled).
Typical XT clock = 3731 (XT clock, ICD enabled, PuT enabled, WdT disabled).

CODE PROTECTION

- Normally, the program machine code can be read back to the programming host computer, be disassembled and the original source program recovered.
- This can be prevented if commercial or security considerations require it. The code protection bits (CP1:CP0) disable reads from selected program areas.
- Program memory may also be written from within the program itself, disabled via the WRT bit.
- Data EEPROM may also be protected from external reads in the same way via the CPD bit, while internal read and write operations are still allowed, regardless of the state-of-the code protection bits.
- bit 13-12, bit 5-4
- CP1:CP0: FLASH Program Memory Code Protection bit, All of the CP1:CP0 pairs have to be given the same value to enable the code protection scheme listed.
- 11 = Code protection off
- 10 = 1F00h to 1FFFh code protected
- 01 = 1000h to 1FFFh code protected
- 00 = 0000h to 1FFFh code protected

IN-CIRCUIT DEBUGGING

- In-circuit debugging (ICD) allows the program to be downloaded after the chip has been fitted in the application circuit, and allows it to be tested with the real hardware.
- The normal debugging techniques of single stepping, breakpoints and tracing can be applied in ICD mode.

bit 11 **DEBUG:** In-Circuit Debugger Mode
1 = In-Circuit Debugger disabled, RB6 and RB7 are general purpose I/O pins
0 = In-Circuit Debugger enabled, RB6 and RB7 are dedicated to the debugger.

LOW VOLTAGE PROGRAMMING

- Normally, when the chip is programmed, a high voltage (12–14 V) is applied to the PGM pin (RB3).
- To avoid the need to supply this voltage during in-circuit programming (e.g. during remote reprogramming), a low-voltage programming mode is available.
- Using this option means that RB3 is not then available for general I/O functions during normal operation.

bit 7 **LVP**: Low Voltage In-Circuit Serial Programming Enable bit
1 = RB3/PGM pin has PGM function, low voltage programming enabled
0 = RB3 is digital I/O, HV on $\overline{\text{MCLR}}$ must be used for programming

POWER-UP TIMER

- When the supply power is applied to the programmed MCU, the start of program execution should be delayed until the power supply and clock are stable, otherwise the program may not run correctly.
- The power-up timer may therefore be enabled (PWRTE 0) as a matter of routine.
- It avoids the need to reset the MCU manually at start up, or connect an external reset circuit, as is necessary with some microprocessors.
- At a clock frequency of 4 MHz, this works out to 256µs.

bit 3 **$\overline{\text{PWRTE}}$**: Power-up Timer Enable bit(3)
1 = PWRT disabled
0 = PWRT enabled

3: Enabling Brown-out Reset automatically enables Power-up Timer (PWRT), regardless of the value of bit $\overline{\text{PWRTE}}$. Ensure the Power-up Timer is enabled any time Brown-out Reset is enabled.

BROWN-OUT RESET

- Brown out refers to a short dip in the power-supply (PSU) voltage, caused by mains supply fluctuation, or some other supply fault, which might disrupt the program execution.
- If the Brown-Out Detect Enable bit (BODEN) is set, a PSU glitch of longer than about 100 µs will cause the device to be held in reset until the supply recovers, and then wait for the power-up timer to time out, before restarting. The program must be designed to recover automatically.

bit 6 **BODEN**: Brown-out Reset Enable bit
1 = BOR enabled
0 = BOR disabled

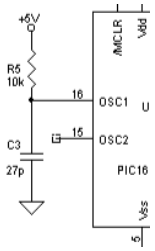
WATCHDOG TIMER

- The watchdog timer is designed to automatically reset the MCU if the program malfunctions, by stopping or getting stuck in loop.
- This could be caused by an undetected bug in the program, an unplanned sequence of inputs or supply fault.
- A separate internal oscillator and counter automatically generates a reset about every 18 ms, unless this is disabled in the configuration word.
- If the watchdog timer is enabled, it should be regularly reset by an instruction in the program loop (CLRWDT) to prevent the reset.
- If the program hangs, and the watchdog timer reset instruction not executed, the MCU will restart, and (possibly) continue correctly, depending on the nature of the fault.

bit 2 **WDTE**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

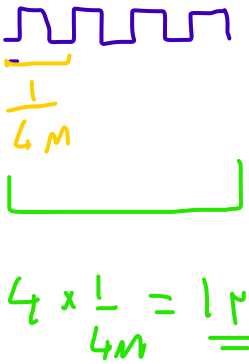
RC OSCILLATOR

- The MCU clock drives the program along, providing the timing signals for program execution.
- The RC (resistor–capacitor) clock is cheap and useful. It allows operating with the internal clock driver circuit, to generate the clock.
- The time constant (product R X C) determines the clock period.
- A variable resistor can be used to give a manually adjustable frequency, although is not very stable or accurate.



CRYSTAL (XTAL) OSCILLATOR

- Used for greater precision
  - uses the hardware timers to make accurate measurements
  - generate precise output signals
- Normally, it is connected across the clock pins with a pair of small capacitors (15 pF) to stabilize the frequency.
- The crystal acts as a self-contained resonant circuit, where the quartz or ceramic crystal vibrates at a precise frequency when subject to electrical stimulation.
- A convenient value (used in our examples later) is 4 MHz; this gives an instruction cycle time of 1 µs
  - This is the maximum frequency allowed for the XT configuration setting.
- Operating at higher frequency requires the selection of the HS configuration option.
- Each instruction takes four clock cycles



CRYSTAL (XTAL) OSCILLATOR/2

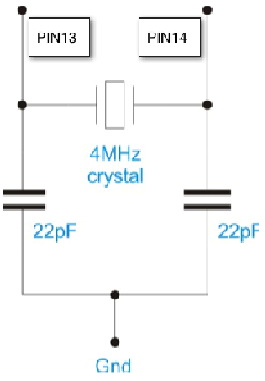


TABLE 14-2: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR

Osc Type	Crystal Freq.	Cap. Range C1	Cap. Range C2
LP	32 kHz	33 pF	33 pF
	200 kHz	15 pF	15 pF
XT	200 kHz	47-68 pF	47-68 pF
	1 MHz	15 pF	15 pF
	4 MHz	15 pF	15 pF
HS	4 MHz	15 pF	15 pF
	8 MHz	15-33 pF	15-33 pF
	20 MHz	15-33 pF	15-33 pF

CONFIGURATION SETTINGS

- The default setting for the configuration bits is 3FFF, which means
  - The code protection is off
  - In-circuit debugging disabled
  - Program write enabled
  - Low-voltage programming enabled
  - Brown-out reset enabled
  - Power-up timer disabled
  - Watchdog timer enabled
  - RC oscillator selected.
- A typical setting for basic development work would enable in-circuit debugging, enable the power-up timer
  - This would minimize the possibility of a faulty start-up.
- For reliable starting, disable the watchdog timer and use the XT oscillator type.
  - By default, the watchdog timer is enabled.

Program Execution

- The program counter keeps track of program execution; it clears to zero on power up or reset.
- With 8k of program memory, a count from 0000 to 1FFF (8191) → requires (13 bits).
- The PCL (Program Counter Low) register (SFR 02) contains the low byte, and this can be read or written like any other file register.
- The high byte is only indirectly accessible via PCLATH (Program Counter Latch High, SFR 0Ah).

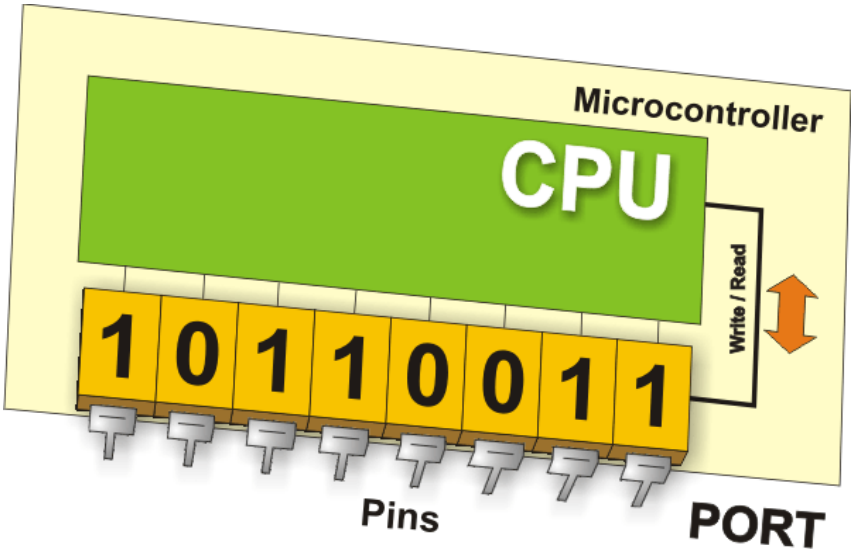
SUBROUTINES

- A label is used at the start of the subroutine
- When a subroutine is called (Using the CALL instruction),
  - the destination address is copied into the program counter
  - the return address (the one following the CALL) is pushed onto the stack
- In the PIC, there are 8 stack address storage levels, which are used in turn.
- The subroutine is terminated with a RETURN instruction
  - causes the program to go back to the original position and continue.
  - achieved by popping the address from the top of the stack and replacing it in the program counter.
- CALL and RETURN must always be used in sequence to avoid a stack error, and a possible program crash.
- In the PIC, the stack is not directly accessible

PAGE BOUNDARIES

- Jump instructions (CALL or GOTO) provide only an 11-bit destination address, so the program memory is effectively divided into four 2k blocks, or pages.
- A jump across the program memory page boundary requires the page selection bits to be modified by the user program.
- Sections 2.3 and 2.4 in the 16F877 data sheet contain detail how to handle these problems.

Input/output ports



Input/output ports

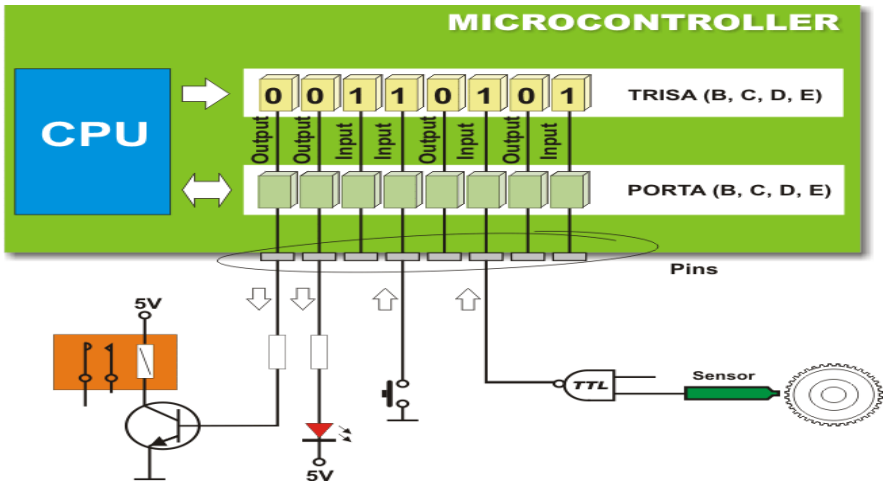
- There are five parallel ports in the PIC 16F877, labelled A–E.
- All pins can be used as bit- or byte-oriented digital input or output with
  - Some having alternate functions depending on the initialization of the relevant control registers.
- The TRIS (data direction) register bits in bank 1, default to 1, setting the ports B, C and D as inputs.
- Ports A and E are set to ANALOGUE INPUT by default, because the analogue control register ADCON1 in bank 1 defaults to 0 - - - 0000.
- To set up these ports for digital I/O, this register must be loaded with the code x - - - 011x (x don't care),e.g. 06h.
- ADCON1 can be initialized with bit codes that give a mixture of analogue and digital I/O on Ports A and E.
- ADCON1 is in bank 1 so BANKSEL is needed to access it.

if we put value in portA register 8-bit more than 5bit it will remove the upper bits values and just save the lower 5 bits

ADC
0-4: A
5-7: E

As you are only using the port in digital IO mode, set ADCON1 to a value of either 0x06 or 0x07 which makes all the pins digital. The LSB is ignored when 0x06 or 0x07 are used so both values do exactly the same thing.

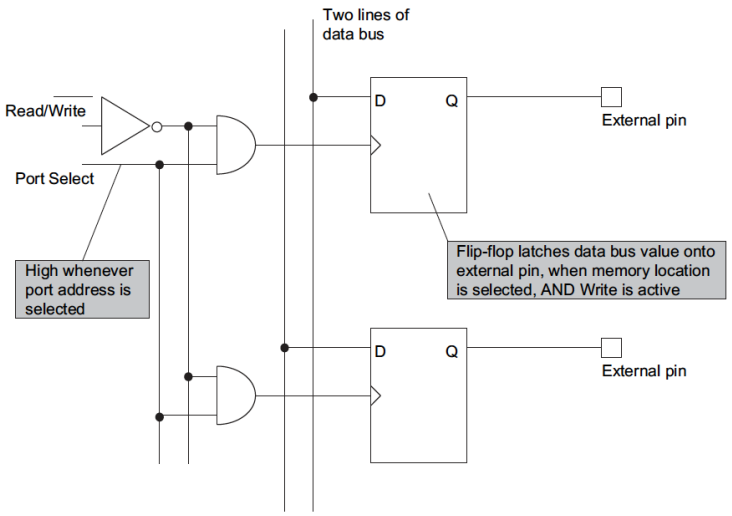
Input/output ports



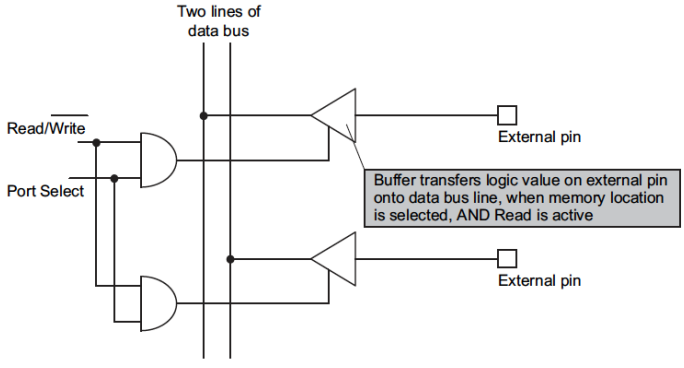
Port functions

	Bits	Pins	Alternate function/s	Bit	Default
Port A	6	RA0–RA5	Analogue inputs Timer0 clock input Serial port slave select input	0,1,2,3,5 4 5	Analogue Input
Port B	8	RB0–RB7	External interrupt Low-voltage programming input Serial programming In-circuit debugging	0 3 6,7 6,7	Digital I/O
Port C	8	RC0–RC7	Timer1 clock input/output Capture/Compare/PWM SPI, I^2C synchronous clock/data USART asynchronous clock/data	0,1 1,2 3,4,5 6,7	Digital I/O
Port D	8	RD0–RD7	Parallel slave port data I/O	0–7	Digital I/O
Port E	3	RE0–RE2	Analogue inputs Parallel slave port control bits	0,1,2 0,1,2	Analogue Input

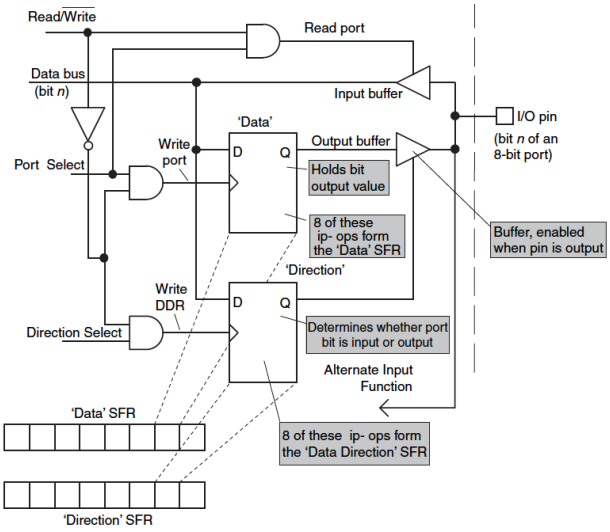
Two bits of a possible digital output port



Two bits of a possible digital input port

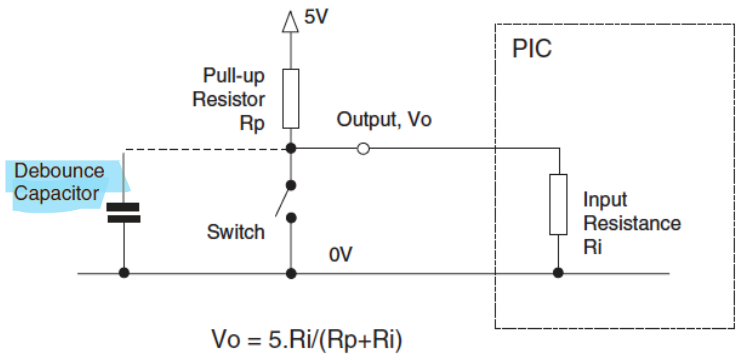


Combine the two circuits to create a programmable bidirectional input/output pin



Input & Output (Interfacing)

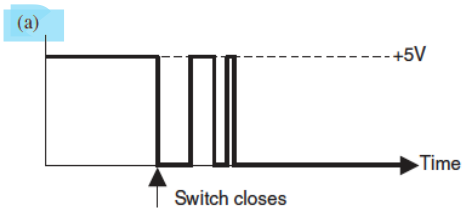
- Switch Input
  - input loading and debouncing.

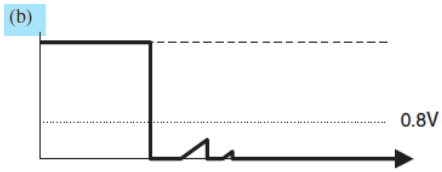


$Vo = 5.Ri/(Rp+Ri)$

Switch Debouncing

- Delay
- Schmit trigger
- capacitor



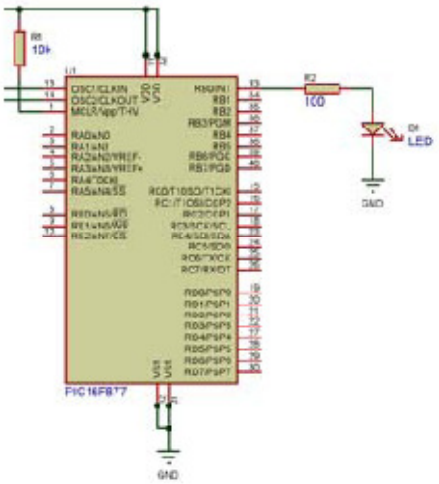


Switch hardware debounce (a) without debounce capacitor; (b) with debounce capacitor

PIC Applications

- LED Flasher

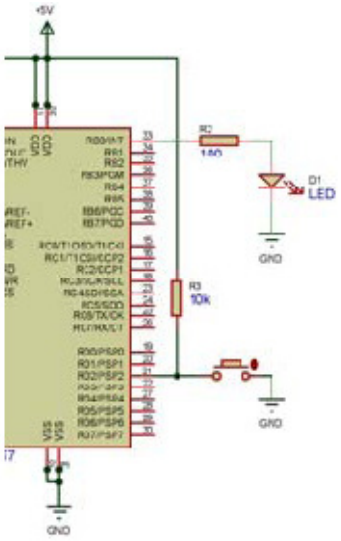
```
Loop:
        bsf     PORTB, 0
        call    Delay_500ms
        bcf     PORTB, 0
        call    Delay_500ms
        goto    Loop
```



PIC Applications

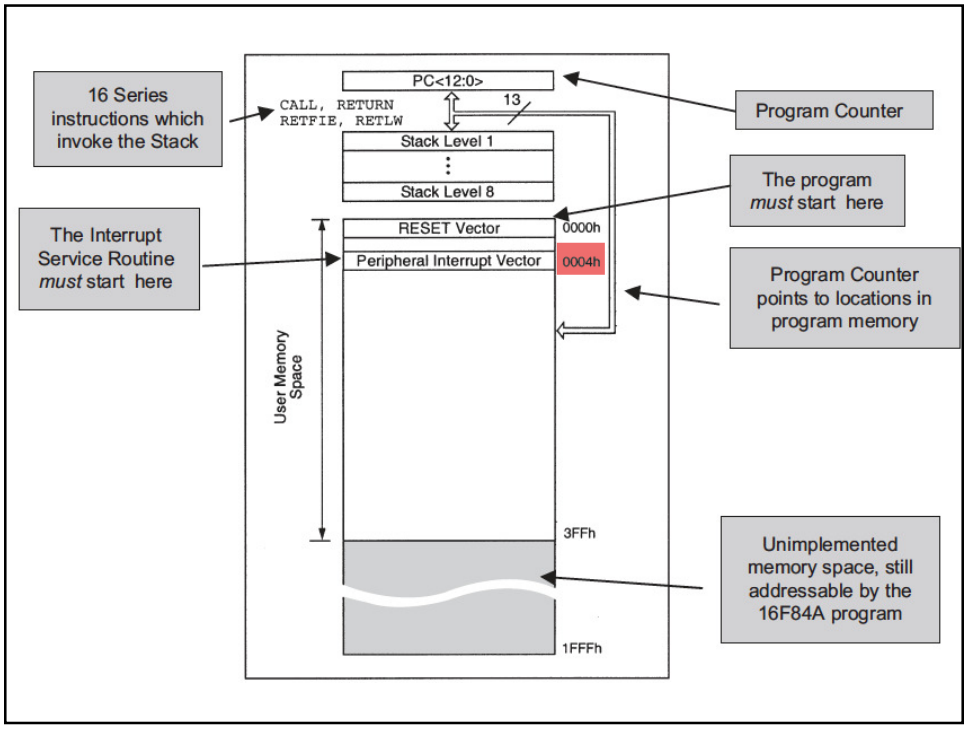
- Button Read

```
        Movlw   0
        movwf   TRISD, f
        bsf     TRISD, 2
Loop:
        btfsc   PORTD, 2
        goto    light
        goto    No_light
Light:
        bsf     PORTB,0
        goto    Loop
No_light:
        bcf     PORTB,0
        goto    Loop
```

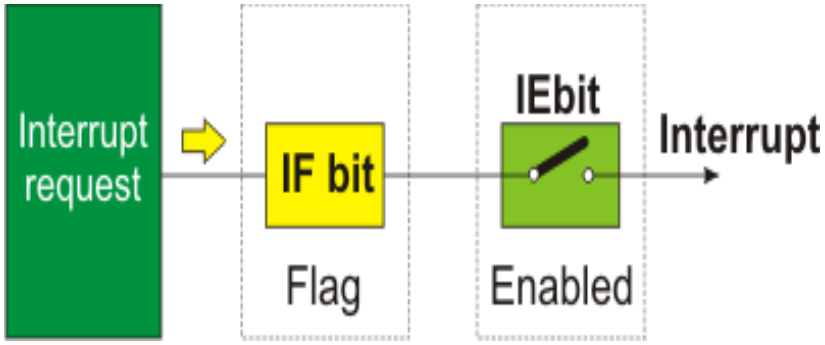


INTERRUPTS

- The stack is used when an interrupt is processed.
- An interrupt is effectively a call and return which is initiated by an external hardware signal
- Forces the processor to jump to a dedicated instruction sequence, an Interrupt Service Routine (ISR).
  - For example, the MCU can be set up so that when a hardware timer times out (finishes its count), the process required at that time is called via a timer interrupt.
- When an interrupt signal is received,
  - the current instruction is completed and
  - the address of the next instruction (the return address) is pushed into the first available stack location.
  - The ISR is called
  - The ISR is terminated with the instruction RETFIE (return from interrupt), which causes the return address to be pulled from the stack.
  - Program execution then restarts at the original location.
- If necessary, the registers must be saved at the beginning of the ISR, and restored at the end, in spare set of file registers.



Interrupt Control Registers

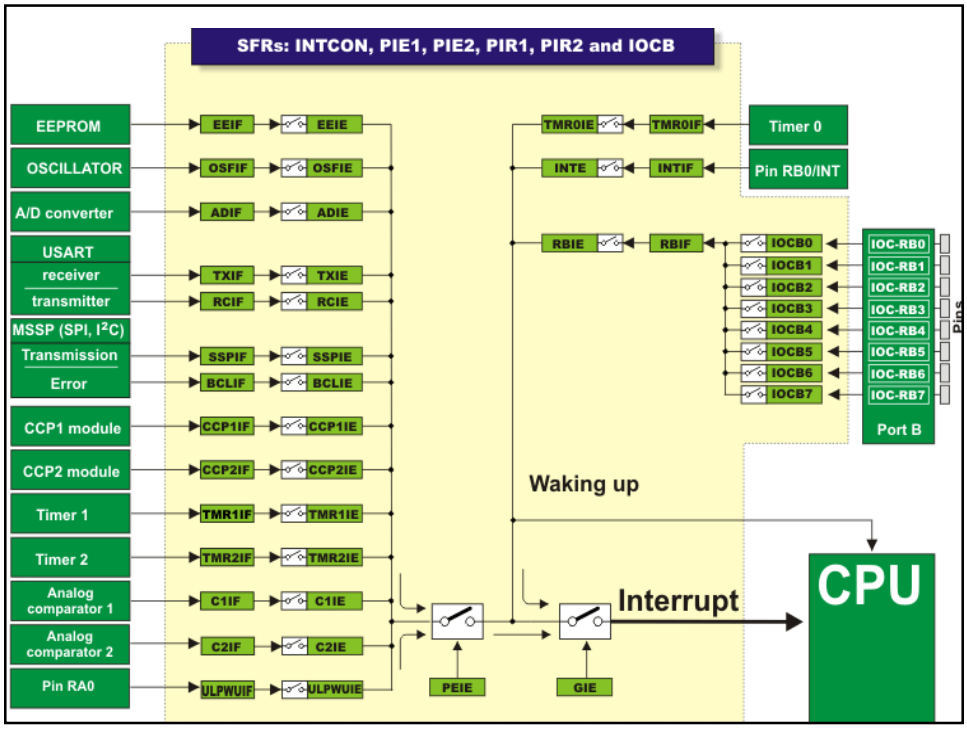


Interrupt Control Registers

- The registers involved in interrupt handling are INTCON, PIR1, PIR2, PIE1, PIE2 and PCON.
- Interrupts are external hardware signals which force the MCU to suspend its current process, and carry out an Interrupt Service Routine (ISR).
- In PIC when an interrupt occurs the program execution jumps to address 004.
- By default, interrupts are disabled.
- If interrupts are to be used
  - the main program start address needs to be 0005, or higher, and a 'GOTO start' (or similar label) placed at address 0000.
  - A 'GOTO ISR' instruction can then be placed at 004, using the ORG directive, which sets the address at which the instruction will be placed by the assembler.
  - The Global Interrupt Enable bit (INTCON, GIE) must be set to enable the interrupt system.
  - The individual interrupt source is then enabled.
  - For example, the bit INTCON, **TOIE** is set to enable the Timer0 overflow to trigger the interrupt sequence.
  - When the timer overflows, INTCON, **TOIF** (Timer0 Interrupt Flag) is set to indicate the interrupt source, and the ISR called.
- The flags can be checked by the ISR to establish the source of the interrupt, if more than one is enabled.

Interrupt sources and control bits

Source	Enable Bit Set	Flag Bit Set	Interrupt Trigger Event
TMR0	INTCON,5	INTCON,2	Timer0 count overflowed
RB0	INTCON,4	INTCON,1	RB0 input changed (also uses INTEDG)
RB4-7	INTCON,3	INTCON,0	Port B high nibble input changed
Peripherals	INTCON,6		
TMR1	PIE1,0	PIR1,0	Timer1 count overflowed
TMR2	PIE1,1	PIR1,1	Timer2 count matched period register PR2
CCP1	PIE1,2	PIR1,2	Timer1 count captured in or matched CCPR1
SSP	PIE1,3	PIR1,3	Data transmitted or received in Synchronous Serial Port
TX	PIE1,4	PIR1,4	Transmit buffer empty in Asynchronous Serial Port
RC	PIE1,5	PIR1,5	Receive buffer full in Asynchronous Serial Port
AD	PIE1,6	PIR1,6	Analogue to Digital Conversion completed
PSP	PIE1,7	PIR1,7	A read or write has occurred in the Parallel Slave Port
CCP2	PIE2,0	PIR2,0 CCPR2	Timer2 count captured in or matched
BCL	PIE2,3	PIR2,3	Bus collision detected in SSP (I^2C mode)
EE	PIE2,4	PIR2,4	Write to EEPROM memory completed



Macros, Special Instructions, Assembler Directives
Another structured

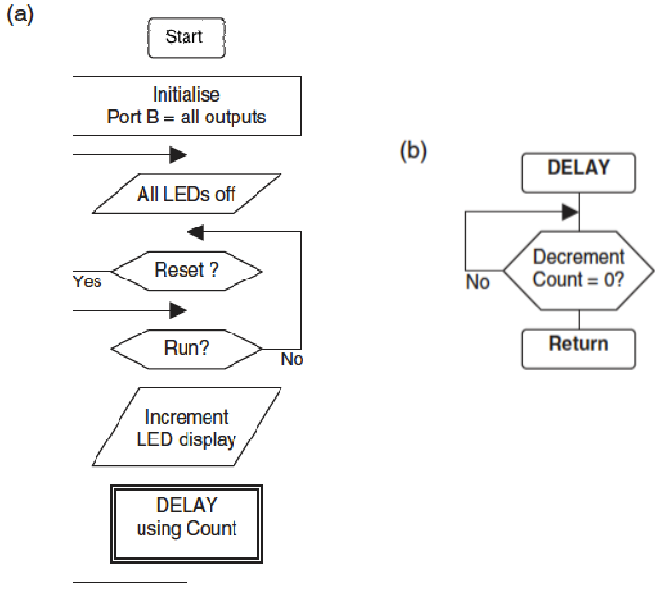
- Supplementary instructions

S.	Instruction		Meaning	Assembler Code	
BZ		addlab	Branch to destination (address label) if result of previous operation zero	BTFSC GOTO	STATUS,Z addlab
BNZ		addlab	Branch to destination (address label) if result of previous operation not zero	BTFSS GOTO	STATUS,Z addlab
BC		addlab	Branch to destination (address label) if carry set	BTFSC GOTO	STATUS,C addlab
BNC		addlab	Branch to destination (address label) if carry not set	BTFSS GOTO	STATUS,C addlab
NEG		num1	Negate (2s complement) a file register (labelled num1)	COMF INCF	num1 num1
TSTF		num1	Test a file register (labelled num1) to modify status bits	MOVF	num1

Program Design and flowcharts

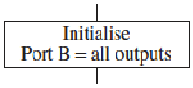
- There are two main forms of flowchart.
  - Data flowcharts: used to represent complex data processing systems
  - Program flow charts: used to represent overall program structure and sequence, but not the details.

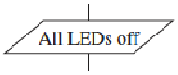
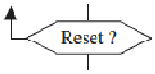
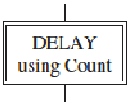
Previous Example flow chart



Flowchart implementation

Operation	Symbol	Implementation
Start End		Source code file/project name in start box. End not needed if program loops endlessly

Process
Sequence
		`BANKSEL TRISB`
		`MOVLW B'00000000'`
		`MOVWF PORTB`

Input or Output		`CLRF PORTB`
Branch Selection		`BTFSS PORTD,Inres` `GOTO reset`
Subroutine Procedure or Function		`MOVLW OFF` `CALL delay`

PIC Peripherals

- Each peripheral has a set of SFRs to control its operation.
- Different PICs have different on-board peripherals

Timers

- The PIC 16F877 has three hardware timers.
  - Used to carry out timing operations simultaneously with the program.
  - Ex.: Generating a pulse every second at an output.
- Timer0 uses an 8-bit register
  - TMR0, file register address 01.
  - The register counts from 0 to 255, and then rolls over to 00 again.
  - When the register goes from FF to 00, an overflow flag, T0IF, bit 2 in the Interrupt Control Register INTCON, address 0B, is set.
- The timer register is incremented via a clock input from either the MCU oscillator (f_{OSC}) or an external pulse train at RA4.
- If the internal clock is used, the register acts as a timer.
- The timers are driven from the instruction clock ($f_{OSC}/4$).
- If the chip is driven from a crystal of 4 MHz, the instruction clock will be 1 MHz, and the timer will update every 1us.

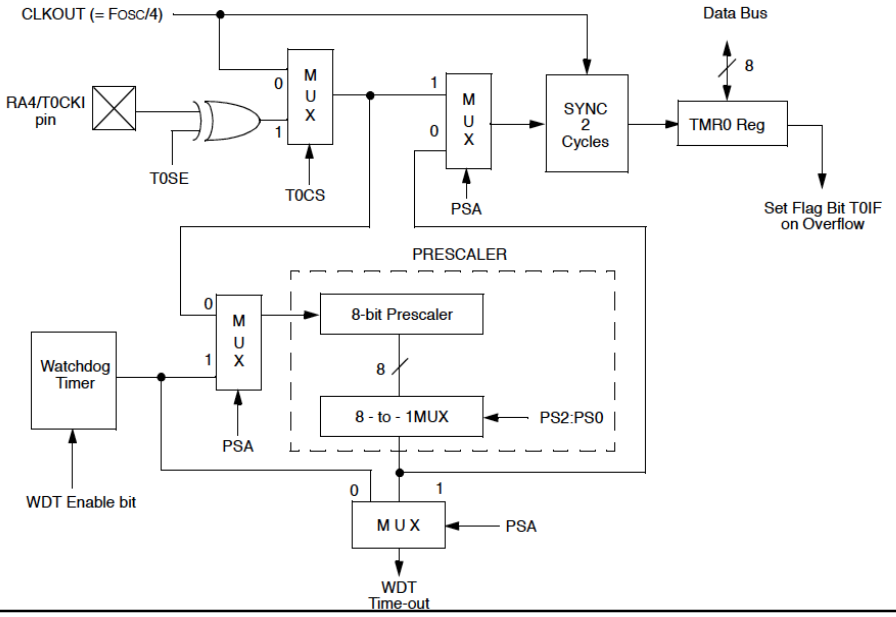
Timers – cont.

- A timer can work as a counter.
  - Counts external pulses
  - timers can also be used as counters.
- Timer0, can be controlled by a pre-scaler, see next slide.
- The pre-scaler is a divide by N register, where N 2, 4, 8, 16, 32, 64, 128 or 256, meaning that the output count rate is reduced by this factor.
- This extends the count period or total count by the same ratio, giving a greater range to the measurement.
- The watchdog timer interval can also be extended, if this is selected as the clock source.
- The pre-scale select bits, and other control bits for Timer0 are found in OPTION_REG.

N=2 -> MAX_TIME = MAX_TIME * 2

4MHz -> I MHz clk inst.
1u time
N = 2
Then 1u -> 2u

BLOCK DIAGRAM OF THE TIMER0/WDT PRESCALER



OPTION_REG REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$**

bit 6 **INTEDG**

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

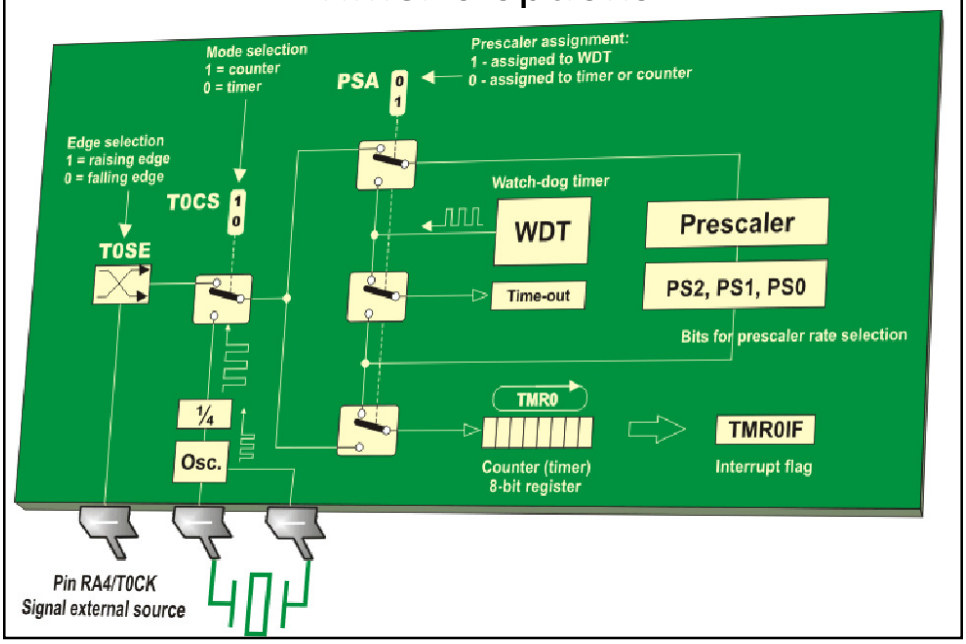
Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Each of the PORTB pins has a weak internal pull-up. A single control bit can turn on all the pull-ups. This is performed by clearing bit **RBPU** (OPTION_REG<7>). The weak pull-up is automatically turned off when the port pin is configured as an output. The pull-ups are disabled on a Power-on Reset.

RB0/INT is an external interrupt input pin and is configured using the **INTEDG** bit (OPTION_REG<6>).

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Timer 0 options



TIMER0 registers

TABLE 5-1: REGISTERS ASSOCIATED WITH TIMER0

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
01h,101h	TMR0	Timer0 Module Register								xxxx xxxx	uuuu uuuu
0Bh,8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
81h,181h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'. Shaded cells are not used by Timer0.

Typical configurations for Timer0

OPTION_REG	Configuration	Effect	Applications
11010000 *Active bits in bold*	Internal clock ($f_{OSC}/4$) No pre-scale	Timer mode using instruction clock	1. Preload Timer0 with initial value, and count up to 256 2. Clear Timer0 initially and read count later to measure time elapsed
11010011	Internal clock ($f_{OSC}/4$) Pre-scale = 16	Timer mode using instruction clock with pre-scale	Extend the count period × 16 for applications 1 and 2
11110111	External clock T0CKI pin	Counter mode Pre-scale = 256	Count one pulse in 256 at RA4
11111110	Watchdog timer selected pre-scale = 64	Extend watchdog reset period to 18 × 64 = 1152 ms	Watchdog timer checks program every second

- show (LED1H design and code) which illustrates the use of a hardware timer and the use of interrupt.

Timers – cont.

high low

- Timer1 is a 16-bit counter, consisting of TMR1H and TMR1L (addresses 0E AND 0F).
  - When the low byte rolls over from FF to 00, the high byte is incremented.
  - The maximum count is therefore 65535, which allows a higher count without sacrificing accuracy.
- Timer2 is an 8-bit counter (TMR2) with a 4-bit pre-scaler, 4-bit post-scaler and a comparator.
  - It can be used to generate Pulse Width Modulated (PWM) output which is useful for driving DC motors and servos, among other things.
- These timers can also be used in capture and compare modes, which allow external signals to be more easily measured.

Assuming a clock frequency of 4 MHz and a prescaler value of 256, which is a common configuration for Timer0 in the PIC16F877A, each Timer0 count would take approximately 64 microseconds to complete. This can be calculated as follows:

The clock period at 4 MHz is 0.25 microseconds (1/4 MHz).
The prescaler value of 256 divides the clock frequency by 256, resulting in a Timer0 clock frequency of 15.625 kHz (4 MHz / 256).
Each Timer0 count therefore takes 1 / 15.625 kHz, or approximately 64 microseconds (1/15625 seconds).
To count from 0 to 255, Timer0 would need to complete 256 counts, so the total time it would take for Timer0 to count from 0 to 255 would be 256 x 64 microseconds, or approximately 16.384 milliseconds.

T1CON: TIMER1 CONTROL REGISTER (ADDRESS 10h)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

bit 7-6 **Unimplemented:** Read as '0'

bit 5-4 **T1CKPS1:T1CKPS0:** Timer1 Input Clock Prescale Select bits
11 = 1:8 prescale value
10 = 1:4 prescale value
01 = 1:2 prescale value
00 = 1:1 prescale value

bit 3 **T1OSCEN**: Timer1 Oscillator Enable Control bit
1 = Oscillator is enabled
0 = Oscillator is shut-off (the oscillator inverter is turned off to eliminate power drain)

bit 2 **$\overline{\text{T1SYNC}}$**: Timer1 External Clock Input Synchronization Control bit
When TMR1CS = 1:
1 = Do not synchronize external clock input
0 = Synchronize external clock input
When TMR1CS = 0:
This bit is ignored. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS**: Timer1 Clock Source Select bit
1 = External clock from pin RC0/T1OSO/T1CKI (on the rising edge)
0 = Internal clock (FOSC/4)

bit 0 **TMR1ON**: Timer1 On bit
1 = Enables Timer1
0 = Stops Timer1

TIMER1 summary

TABLE 6-2: REGISTERS ASSOCIATED WITH TIMER1 AS A TIMER/COUNTER

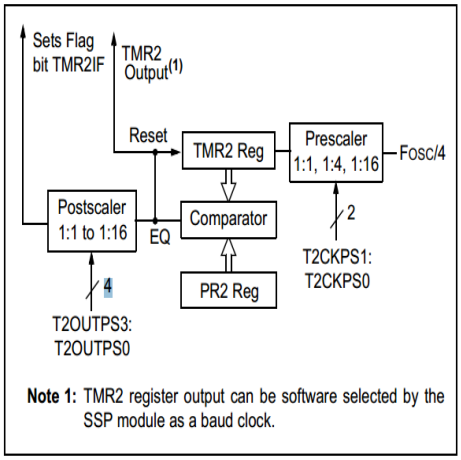
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	--00 0000	--uu uuuu

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer1 module.

Note 1: Bits PSPIE and PSPIF are reserved on the 28-pin devices; always maintain these bits clear.

TIMER2

FIGURE 7-1: TIMER2 BLOCK DIAGRAM



Note 1: TMR2 register output can be software selected by the SSP module as a baud clock.

- Timer2 is an 8-bit timer with a prescaler and a postscaler. It can be used as the PWM time base for the PWM mode of the CCP module(s). The TMR2 register is readable and writable and is cleared on any device Reset.

TIMER2 summary

TABLE 7-1: REGISTERS ASSOCIATED WITH TIMER2 AS A TIMER/COUNTER

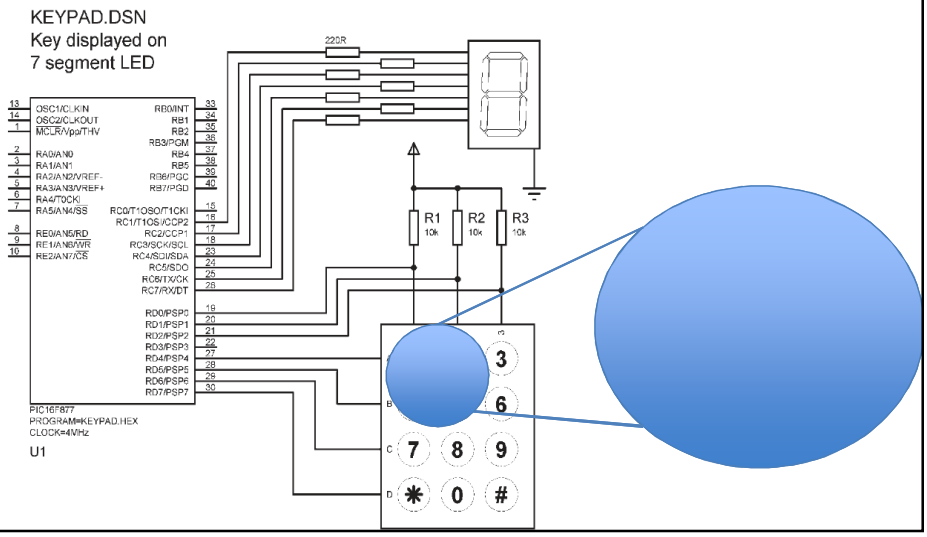
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
11h	TMR2	Timer2 Module's Register								0000 0000	0000 0000
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
92h	PR2	Timer2 Period Register								1111 1111	1111 1111

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer2 module.

Note 1: Bits PSPIE and PSPIF are reserved on 28-pin devices; always maintain these bits clear.

Keypad Input

- A keypad is simply an array of push buttons connected in rows and columns, so that each can be tested for closure with the minimum number of connections.
- There are 12 keys on a phone type pad (0–9, #, ∗), arranged in a 3X4 matrix. The columns are labeled 1, 2, 3 and the rows A, B, C, D.



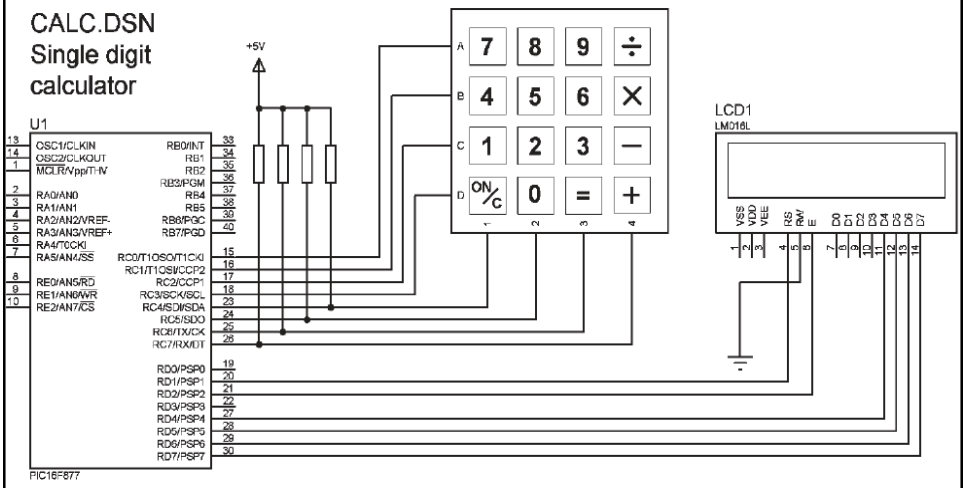
Keypad Input – cont.

- If we assume that all the rows and columns are initially high, a keystroke can be detected by setting each row low in turn and checking each column for a zero.
- In the KEYPAD circuit, the 7 keypad pins are connected to Port D.
  - Bits 4–7 are initialized as outputs, and bits 0–2 used as inputs.
- The input pins are pulled high to logic 1. The output rows are also initially set to 1.

Keypad operation

- If a 0 is now output on row A, there is no effect on the inputs unless a button in row A is pressed. If these are checked in turn for a 0, a button in this row which is pressed can be identified as a specific combination of output and input bits.
- A simple way to achieve this result is to increment a count of keys tested when each is checked, so that when a button is detected, the scan of the keyboard is terminated with current key number in the counter.
- This works because the (non-zero) numbers on the keypad arranged in order:
  - Row A 1, 2, 3
  - Row B 4, 5, 6
  - Row C 7, 8, 9
  - Row D *, 0, #
- Following this system, the star symbol is represented by a count of 10 (0Ah), zero by 11(0Bh) and hash by 12 (0C).
- Show Keypad design and code.

Calculator



- The calculator operates as follows:
  - To perform a calculation, press a number key, then an operation key, then another number and then equals.
  - The calculation and result are displayed. For the divide operation, the result is displayed as result and remainder.

Pseudo code for the calculator

- CALC
  - Single digit calculator produces two digit results.
  - Hardware: x12 keypad, 2x16 LCD, P16F887 MCU
- MAIN
- Initialise
  - PortC = keypad
    - RC0 – RC3 = output rows
    - RC4 – RC7 = input columns
  - PortD = LCD
    - RD1, RD2 = control bits
    - RD4– RD7 = data bits
  - CALL Initialise display
- Scan Keypad
  - REPEAT
    - CALL Keypad input, Delay 50ms for debounce
    - CALL Keypad input, Check key released
    - IF first key, load Num1, Display character and restart loop
    - IF second key, load sign, Display character and restart loop
    - IF third key, load Num2 Display character and restart loop
    - IFfourth key, CALL Calculate result
    - IF fifth key, Clear display
  - ALWAYS

Subroutines

- Included LCD driver routines
  - Initialise display
  - Display character
- Keypad Input
  - Check row A, IF key pressed, load ASCII code
  - Check row B, IF key pressed, load ASCII code
  - Check row C, IF key pressed, load ASCII code
  - Check row D, IF key pressed, load ASCII code
  - ELSE load zero code
- Calculate result
  - IF key = '+', Add
  - IF key = '-', Subtract
  - IF key = 'x', Multiply
  - IF key = '/', Divide
  - Add Add Num1 + Num2
    - Load result, CALL Two digits
  - Subtract Subtract Num1 – Num2
    - IF result negative, load minus sign, CALL Display character
    - Load result, CALL Display character

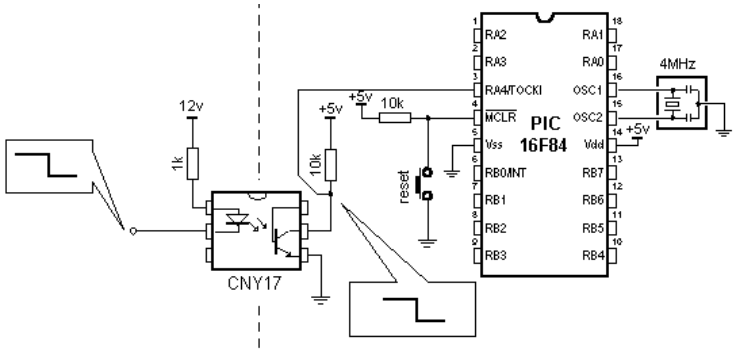
Subroutines – cont.

  - Multiply
    - REPEAT
    - Add Num1 to Result
    - Decrement Num2
    - UNTIL Num2= 0
    - Load result, CALL Two digits
  - Divide
    - REPEAT
    - Subtract Num2 from Num1
    - Increment Result
    - UNTIL Num1 negative
    - Add Num2 back onto Num1 for Remainder
    - Load Result, CALL Display character
    - Load Remainder, CALL Display character
- Two digits
  - Divide result by 10, load MSD, CALL Display character
  - Load LSD, CALL Display character

- Show calc code and design

Optocouplers

- The way it works is simple: when a signal arrives, the LED within the optocoupler is turned on, and it illuminates the base of a photo-transistor within the same case. When the transistor is activated, the voltage between collector and emitter falls to 0.7V or less and the microcontroller sees this as a logic zero on its RA4 pin.



156

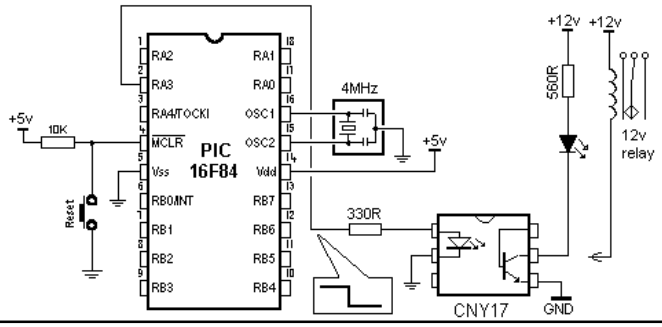
Makro : OPTOIN.ASM

```
;***** Declaring and configuring a microcontroller *****

      PROCESSOR 16f84
      #include "p16f84.inc"

      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Structure of program memory *****

      ORG    0x00          ;Reset vector
      goto   Main

      ORG    0x04          ;Interrupt vector
      goto   Main          ;no interrupt routine


Main                       ;Main program
      banksel TRISA
      movlw  0xef          ;Initialization of port A
      movwf  TRISA         ;TRISA <- 0xff
      movlw  0x00          ;Initialization of port B
      movwf  TRISB         ;TRISB <- 0x00
      movlw  b'00110000'   ;RA4 -> TMR0, PS=1:2
      banksel OPTION
      movwf  OPTION_REG    ;Increment TMR0 upon falling edge
      banksel PORTB

      clrf   PORTB         ;PORTB <- 0
      clrf   TMR0          ;TMR0 <- 0

Loop  movf   TMR0,w        ;Send value of the counter
      movwf  PORTB         ;to PORTB
      goto   Loop          ;Remain at this line

      End                  ;End of program
```

157

Optocoupler - output

- An Optocoupler can be also used to separate the output signals. If optocoupler LED is connected to microcontroller pin, logical zero on pin will activate optocoupler LED, thus activating the transistor. This will consequently switch on LED in the part of device working on 12V. Layout of this connection is shown below.



158

Program

- Write a program to command the relay after each interrupt RB0

159

Program

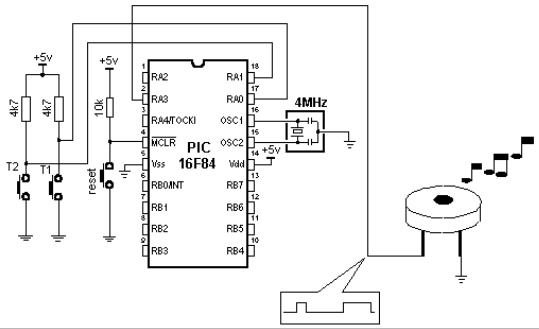
- Write a program to command the relay after each interrupt RB0

160

Sounds

- **Generating sound**

In microcontroller systems, beeper is used for indicating certain occurrences, such as push of a button or an error. To have the beeper started, it needs to be delivered a string in binary code - in this way, you can create sounds according to your needs. Connecting the beeper is fairly simple: one pin is connected to the mass, and the other to the microcontroller pin through a capacitor, as shown on the following image.



161

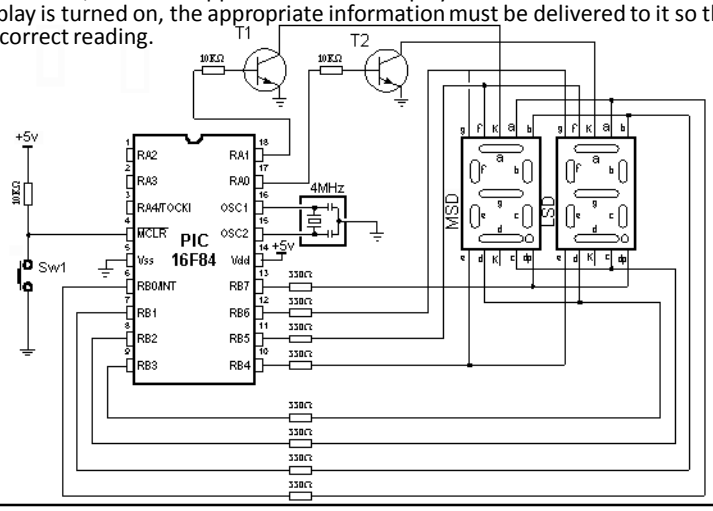
Program

- Write a program to make a sound of frequency 1Khz

162

7 segment-display

- To produce a 4, 5 or 6 digit display, all the 7-segment displays are connected in parallel. The common line (the common-cathode line) is taken out separately and this line is taken low for a short period of time to turn on the display. Each display is turned on at a rate above 100 times per second, and it will appear that all the displays are turned on at the same time. As each display is turned on, the appropriate information must be delivered to it so that it will give the correct reading.



163

Program

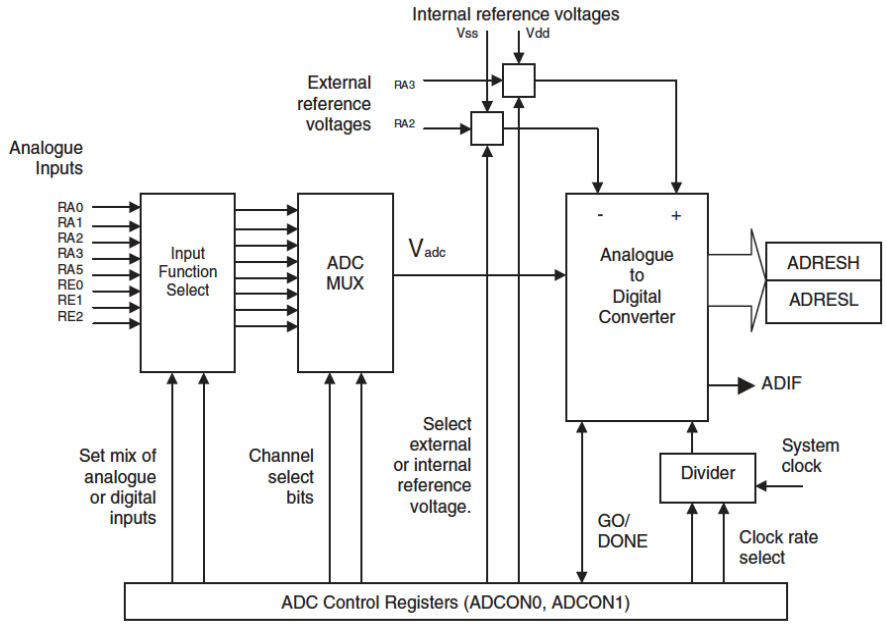
- Write a program to display 45 to the 7 segments displays

164

ADC

- Registers used
  - ADCON0
  - ADCON1
  - The output from the converter is stored in ADRESH (analogue to digital conversion result, high byte) and ADRESL (low byte).
- Show ADC code and design (VINTEST)

ADC block diagram



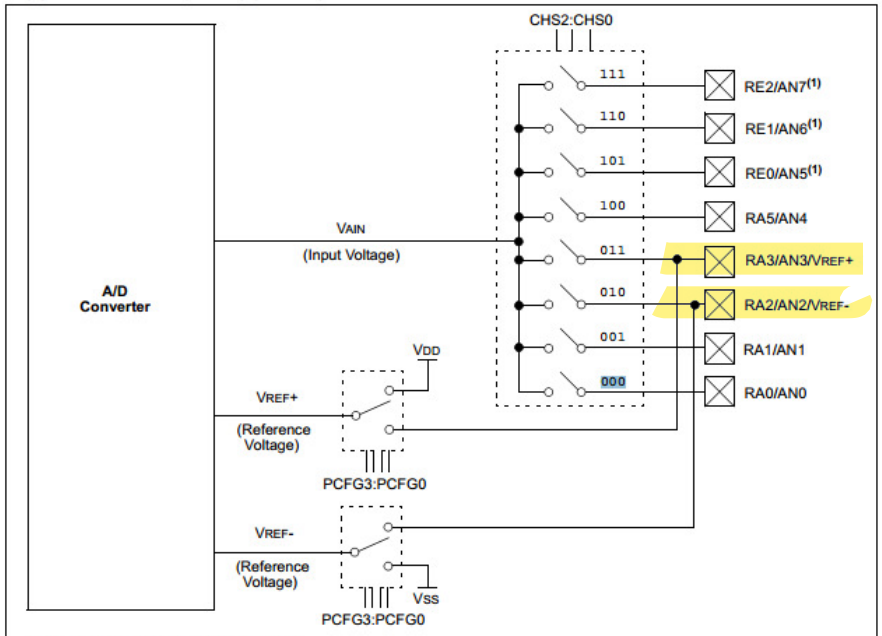
8-bit Conversion

- The 16F877 MCU has eight analogue inputs available, at RA0, RA1, RA2, RA3, RA5, RE0, RE1 and RE2.
- RA2 and RA3 may be used as reference voltage inputs, setting the minimum and maximum values for the measured voltage range.
- These inputs default to analogue operation, so the register ADCON1 has to be initialized explicitly to use these pins for digital input or output.
- The ADC converts an analogue input voltage (e.g. 0 – 2.56V) to 10-bit binary, but only the upper 8 bits of the result are used, giving a resolution of 10 mV per bit ((1/256) X 2.56 V).

ADC OPERATION

- The inputs are connected to a function selector block which sets up each pin for analogue or digital operation according to the 4-bit control code loaded into the A/D port configuration control bits, PCFG0–PCFG3 in ADCON1.
  - The code used, 0011, sets Port E as digital I/O, and Port A as analogue inputs with AN3 as the positive reference input.
- The analogue inputs are then fed to a multiplexer which allows one of the eight inputs to be selected at any one time.
  - This is controlled by the three analogue channel select bits, CHS0–CHS2 in ADCON0.
  - In the example, channel 0 is selected (000), RA0 input.
  - If more than one channel is to be sampled, these select bits need to be changed between ADC conversions.
- The conversion is triggered by setting the GO/DONE bit, which is later cleared automatically to indicate that the conversion is complete.

FIGURE 11-1: A/D BLOCK DIAGRAM



Note 1: Not available on 28-pin devices.

ADCON0 REGISTER (ADDRESS 1Fh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON
bit 7							bit 0

bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits
000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)
110 = Channel 6 (AN6)
111 = Channel 7 (AN7)

Note: The PIC16F873A/876A devices only implement A/D channels 0 through 4; the unimplemented selections are reserved. Do not select any unimplemented channels with these devices.

bit 2 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit
When ADON = 1:
1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)
0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit
1 = A/D converter module is powered up
0 = A/D converter module is shut-off and consumes no operating current

at least we need 1.6 us

from which bit we want to read the analog signal

1: when start convert then we will wait it to set to 0 when finish

ADCON1 REGISTER (ADDRESS 9Fh)

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **ADFM**: A/D Result Format Select bit
1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'.
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'.

bit 6 **ADCS2**: A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-4 **Unimplemented:** Read as '0'
bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references

A → Vref

will read the analog value as voltage in range

[Vref-, Vref+]

Vref+ = 5, Vref- = 0

then

(5 - 0) / 2 ^ 10 = 0.005

then if the result is 1000000000 = 512

then the voltage was 512 * 0.005 = 2.56 V

ADC control registers

Register	Setting	Flags	Function
ADRESH	XXXX XXXX		ADC result high byte
ADRESL	XXXX XXXX		ADC result low byte
ADCON0	0100 0X01	ADCS1,0 GO/DONE, ADON	Conversion frequency select ADC start, ADC enable
ADCON1	0000 0011	ADFM, PCFG3-0	Result justify, ADC input mode control
INTCON	1100 0000	GIE,PEIE	Peripheral interrupt enable
PIE1	0100 0000	ADIE	ADC interrupt enable
PIR1	0100 0000	ADIF	ADC interrupt flag

(c)

	ADRESH	ADRESL
ADFM = 1 Right justified	0000 00RR	RRRR RRRR
ADFM = 0 Left justified	RRRR RRRR	RR00 0000

R = Result bits

We can ignore ADRESL

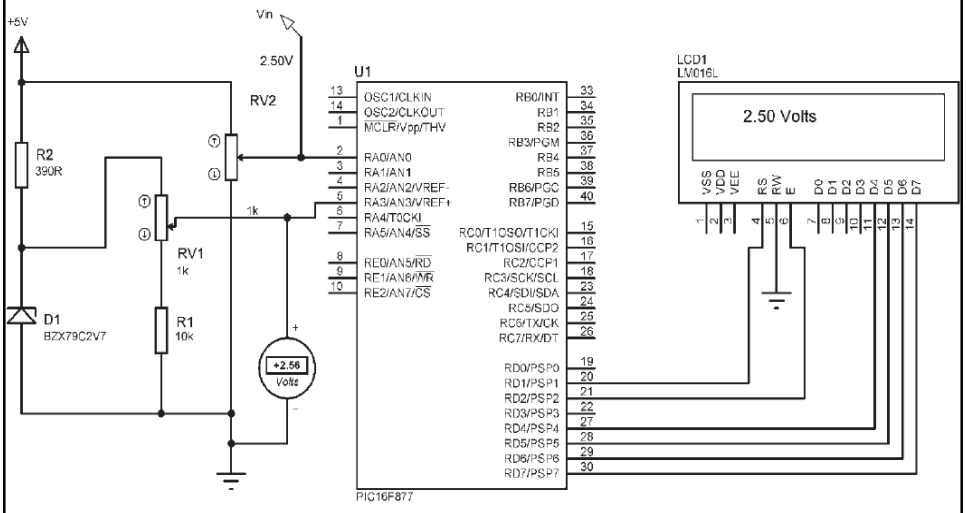
To do an A/D Conversion, follow these steps:

1. Configure the A/D module:
   - Configure analog pins/voltage reference and
   - digital I/O (ADCON1)
   - Select A/D input channel (ADCON0)
   - Select A/D conversion clock (ADCON0)
   - Turn on A/D module (ADCON0)

2. Configure A/D interrupt (if desired):
   - Clear ADIF bit
   - Set ADIE bit
   - Set PEIE bit
   - Set GIE bit
3. Wait the required acquisition time
4. Start conversion:
   - Set GO/DONE bit (ADCON0)
5. Wait for A/D conversion to complete by either:
   - Polling for the GO/DONE bit to be cleared (interrupts disabled); OR Waiting for the A/D interrupt

6. Read A/D Result register pair (ADRESH:ADRESL), clear bit ADIF if required.
7. For the next conversion, go to step 1 or step 2 as required. The A/D conversion time per bit is defined as TAD.

ADC Test Circuit



ADC clock

- The speed of the conversion is selected by bits ADSC1 and ADSC0.
- The ADC operates by successive approximation;
  - this means that the input voltage is fed to a comparator, and if the voltage is higher than 50% of the range, the MSB of the result is set high.
  - The voltage is then checked against the mid-point of the remaining range, and the next bit set high or low accordingly, and so on for 10 bits.
- This takes a significant amount of time: the minimum conversion time is 1.6 µs per bit, making 16 µs for a 10-bit conversion.
- The ADC clock speed must be selected such that this minimum time requirement is satisfied;
- The MCU clock is divided by 2, 8 or 32 as necessary.
- Our simulated test circuit is clocked at 4 MHz. This gives a clock period of 0.25 µs. We need a conversion time of at least 1.6 µs; if we select the divide by 8 option, the ADC clock period will then be 8 X 0.25 = 2 µs, which is just longer than the minimum required.
- The select bits are therefore set to 01.

Capture/compare/ PWM

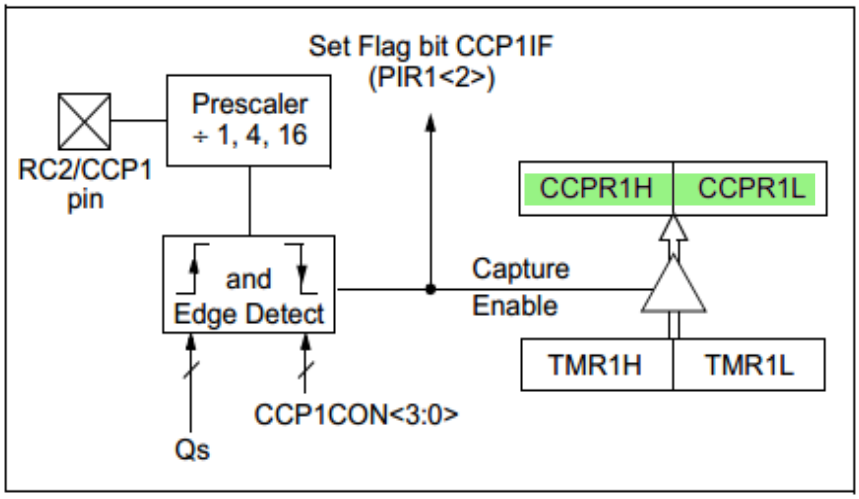
Each Capture/Compare/PWM (CCP) module contains a 16-bit register which can operate as:

- 16-bit Capture register
- 16-bit Compare register
- PWM Master/Slave Duty Cycle register

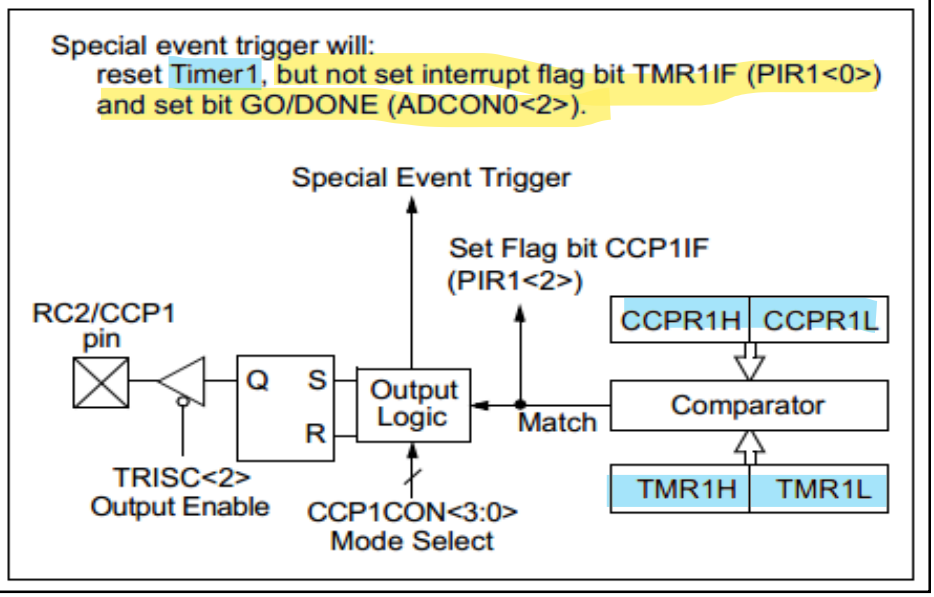
- Both the CCP1 and CCP2 modules are identical in operation

for find the period of the signal that enter on portc CCP1 or CCP2 from rising to rising edge or from falling to falling edge and will save the value in CCPR1H, CCPR1L

Capture mode



Compare mode



Will compare the value in timer1 if it become as the value in CCPR1H and CCPR1L
then will set/clear the portc CCP1 or CCP2

CCP1CON REGISTER/CCP2CON REGISTER (ADDRESS 17h/1Dh)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CCPxX	CCPxY	CCPxM3	CCPxM2	CCPxM1	CCPxM0
bit 7							bit 0

bit 7-6 **Unimplemented:** Read as '0'

bit 5-4 **CCPxX:CCPxY**: PWM Least Significant bits

Capture mode:
Unused.

Compare mode:
Unused.

PWM mode:
These bits are the two LSbs of the PWM duty cycle. The eight MSbs are found in CCPRxL.

bit 3-0 **CCPxM3:CCPxM0**: CCPx Mode Select bits

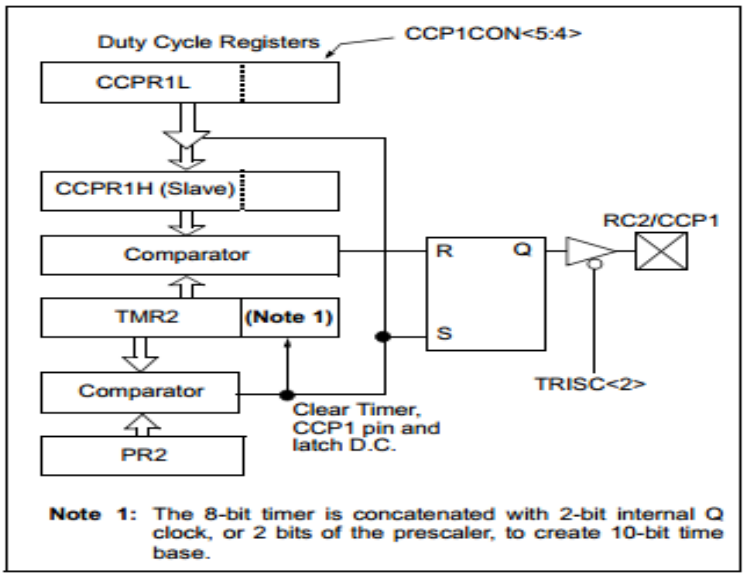
0000 = Capture/Compare/PWM disabled (resets CCPx module)
0100 = Capture mode, every falling edge
0101 = Capture mode, every rising edge
0110 = Capture mode, every 4th rising edge
0111 = Capture mode, every 16th rising edge
1000 = Compare mode, set output on match (CCPxIF bit is set)
1001 = Compare mode, clear output on match (CCPxIF bit is set)
1010 = Compare mode, generate software interrupt on match (CCPxIF bit is set, CCPx pin is unaffected)
1011 = Compare mode, trigger special event (CCPxIF bit is set, CCPx pin is unaffected); CCP1 resets TMR1; CCP2 resets TMR1 and starts an A/D conversion (if A/D module is enabled)
11xx = PWM mode

PWM value we save it as 10 bit in:
CCPR1L<7:0>:CCP1CON<5:4>

PWM Mode (PWM)

- In Pulse Width Modulation mode, the CCPx pin produces up to a 10-bit resolution PWM output. Since the CCP1 pin is multiplexed with the PORTC data latch, the TRISC<2> bit must be cleared to make the CCP1 pin an output

FIGURE 8-3: SIMPLIFIED PWM BLOCK DIAGRAM



Note 1: The 8-bit timer is concatenated with 2-bit internal Q clock, or 2 bits of the prescaler, to create 10-bit time base.

8-bit register

- PWM Period =

  [(PR2) + 1] • 4 • TOSC • (TMR2 Prescale Value)

- PWM Duty Cycle =

  (CCPR1L:CCP1CON<5:4>) •TOSC • (TMR2 $_{\text{Prescale}}$)

$$\text{Resolution} = \frac{\log\left(\frac{F_{OSC}}{F_{PWM}}\right)}{\log(2)} \text{ bits}$$

TABLE 8-5: REGISTERS ASSOCIATED WITH PWM AND TIMER2

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
0Dh	PIR2	—	—	—	—	—	—	—	CCP2IF	---- ---0	---- ---0
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
8Dh	PIE2	—	—	—	—	—	—	—	CCP2IE	---- ---0	---- ---0
87h	TRISC	PORTC Data Direction Register								1111 1111	1111 1111
11h	TMR2	Timer2 Module's Register								0000 0000	0000 0000
92h	PR2	Timer2 Module's Period Register								1111 1111	1111 1111
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	uuuu uuuu
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	uuuu uuuu
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	--00 0000
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	uuuu uuuu
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	uuuu uuuu
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	--00 0000

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PWM and Timer2.

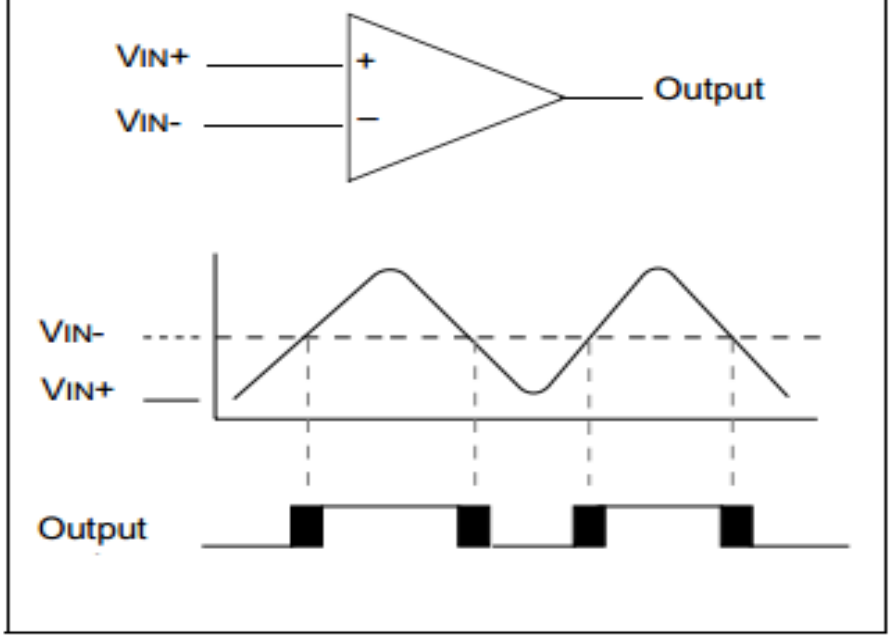
Note 1: Bits PSPIE and PSPIF are reserved on 28-pin devices; always maintain these bits clear.

SETUP FOR PWM OPERATION

1. Set the PWM period by writing to the PR2 register.
2. Set the PWM duty cycle by writing to the CCPR1L register and CCP1CON<5:4> bits.
3. Make the CCP1 pin an output by clearing the TRISC<2> bit.
4. Set the TMR2 prescale value and enable Timer2 by writing to T2CON.
5. Configure the CCP1 module for PWM operation

COMPARATOR MODULE

- The comparator module contains two analog comparators. The inputs to the comparators are multiplexed with I/O port pins RA0 through RA3, while the outputs are multiplexed to pins RA4 and RA5. The on-chip voltage reference can also be an input to the comparators.



CMCON REGISTER

R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-1
C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0
bit 7							bit 0

bit 7 **C2OUT**: Comparator 2 Output bit
When C2INV = 0:
1 = C2 VIN+ > C2 VIN-
0 = C2 VIN+ < C2 VIN-
When C2INV = 1:
1 = C2 VIN+ < C2 VIN-
0 = C2 VIN+ > C2 VIN-

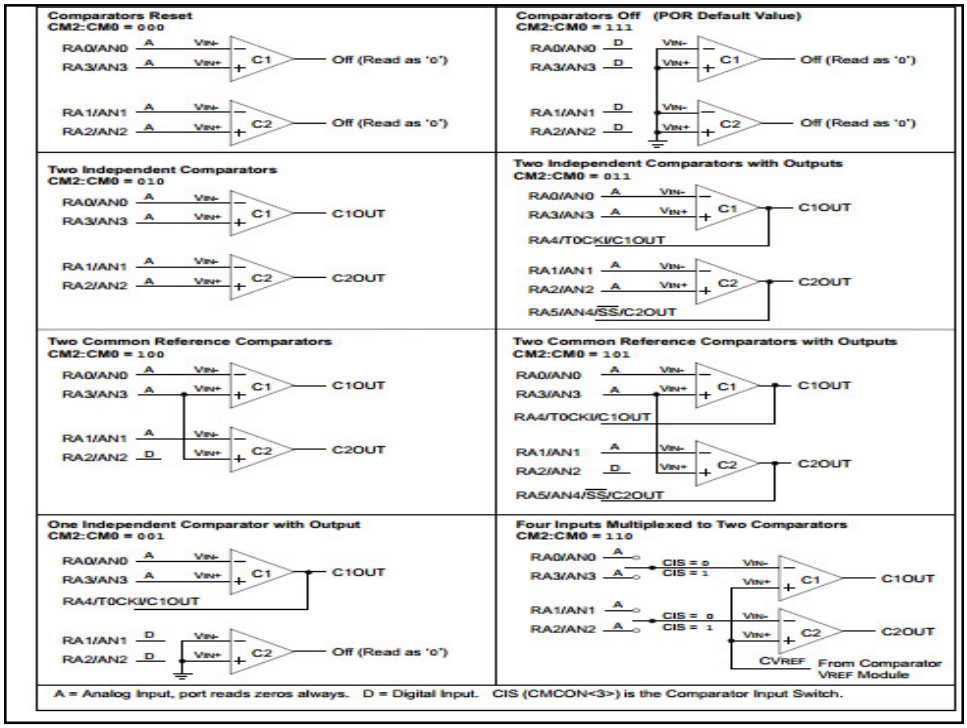
bit 6 **C1OUT**: Comparator 1 Output bit
When C1INV = 0:
1 = C1 VIN+ > C1 VIN-
0 = C1 VIN+ < C1 VIN-
When C1INV = 1:
1 = C1 VIN+ < C1 VIN-
0 = C1 VIN+ > C1 VIN-

bit 5 **C2INV**: Comparator 2 Output Inversion bit
1 = C2 output inverted
0 = C2 output not inverted

bit 4 **C1INV**: Comparator 1 Output Inversion bit
1 = C1 output inverted
0 = C1 output not inverted

bit 3 **CIS**: Comparator Input Switch bit
When CM2:CM0 = 110:
1 = C1 VIN- connects to RA3/AN3
C2 VIN- connects to RA2/AN2
0 = C1 VIN- connects to RA0/AN0
C2 VIN- connects to RA1/AN1

bit 2 **CM2:CM0**: Comparator Mode bits
Figure 12-1 shows the Comparator modes and CM2:CM0 bit settings.



PIC CCS 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

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

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	$\pm 3 \times 10^{-38}$ to $\pm 3 \times 10^{+38}$

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

Multi-Precision Operations

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

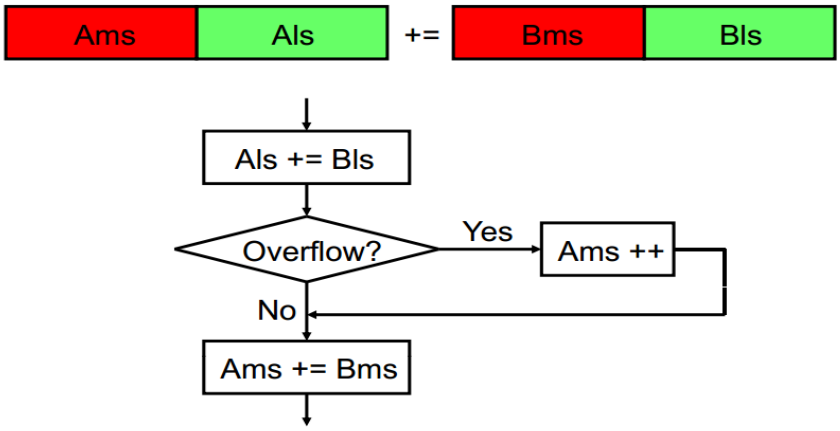
PIC instructions only operate on 8-bit words

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

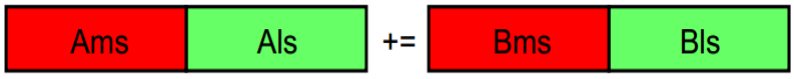
This is much more expensive in time and code size than single instructions

Multi-Precision Operations



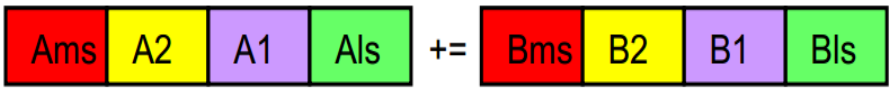
Multi-Precision Operations

16-bit addition using 8-bit operations:



Multi-Precision Operations

32-bit addition using 8-bit operations:



Built-in functions

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^2C 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

Delay

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 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

Access to IO 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);
    }
}
```

Support for 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

Support for 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)
. . . . . .
```

```
#use fast_io(b)

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

Support for 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
. . . . . . . . . . . . . . . .
```

Timers

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)
. . . . . . . . . . .
```

To set the counter:

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

To read the counter:

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

Counter example

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);
      }
}
```

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 interrupt

```
#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 (;;) {
    }
}
```

PWM

```
#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);
    }
}
```

LCD

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

LCD

```
#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

```
#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);
    }
}
```

ADC

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)

ADC

```
#device ADC=10
void main()
{
      long int q;
      float p;
      setup_adc(ADC_CLOCK_DIV_64);
      setup_adc_ports(AN0);
      set_adc_channel(0);

      for (;;) {
            q = read_adc();
            p = 5.0 * q / 1024.0;

            .....
            delay_ms(100);
      }
}
```

Evolution of Microcontroller Firmware Development

Overview

- Traditional microcontroller firmware
  - Polling
  - Interrupts
- Real-time Operating Systems
- Demonstration

Polling

- Polling is when a process continually evaluates the status of a register in an effort to synchronize program execution.
- Polling is not widely used because it is an inefficient use of microcontroller resources.

Polling Example

```
void main ( void )
{
   while(1)
   {
         while (!button1_pressed);
         turn_on_led;
         while (!button1_pressed);
         turn_off_led;
   }
}
```

Interrupts

- Interrupt
  - An *event* that causes the *Program Counter (PC)* to change. These events can be internal or external.
- Interrupt Service Routine (ISR)
  - Set of instructions that is executed when an interrupt occurs
- Interrupt Vector
  - Memory address of the first instruction in the ISR.

Interrupts

- Why should one use interrupts?
  - Provides more efficient use of microcontroller resources.
  - Provides a means to create firmware that can "multi-task".

Interrupts

- Interrupts are often prioritized within the system and are associated with a variety of on-chip and off-chip peripherals
  - Timers, A/D D/A converters, UART, GP I/O
- A majority of the µC interrupts can be enabled or disabled.
  - Globally
  - Individually
  - Those interrupts that cannot be disabled are called Non-Maskable Interrupts (NMI).
- Interrupts can be nested.
- Interrupts can occur at anyplace, at anytime.
  - ISRs should be kept *short* and *fast*.

Interrupts vs. Polling

- Allowed for more efficient use of the microcontroller.
  - Faster program execution
  - Multi-tasking
- Facilitated the development of complex firmware.
  - Supports a modular approach to firmware design.

Real-time Operating Systems

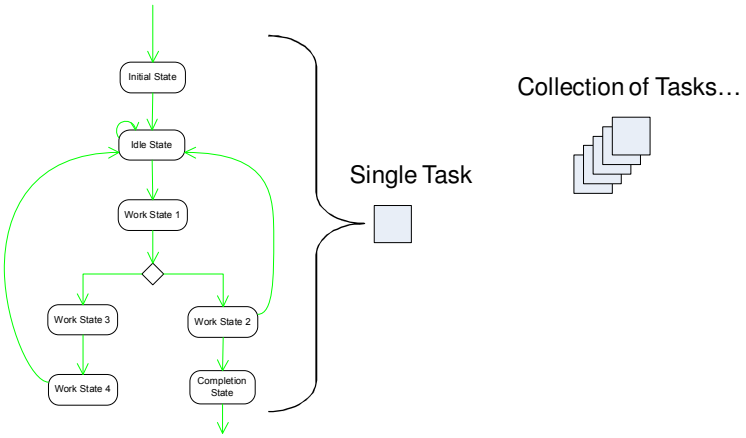
Real-time Operating System

- A real-time operating system (RTOS) is a multi-tasking operating system intended for real-time applications.
  - Mainly used in embedded applications.
  - Facilitates the creation of a real-time system.
  - Tool for the real-time software developer.
  - Provides a layer abstraction between the hardware and software.

Real-time Operating System

- State
  - A unique operating condition of the system.
- Task
  - A single thread of execution through a group of related states.
- Task Manager
  - Responsible for maintaining the current state of each task.
  - Responsible for providing each task with execution time.

Real-time Operating System



Real-time Operating System

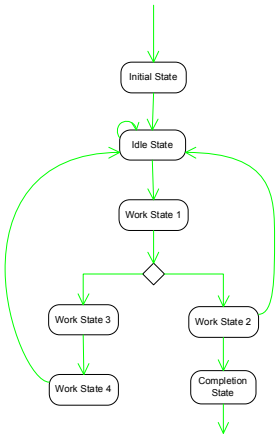
- A more detailed explanation state
  - A function

```
void idleState ( void );
```

  - Should be kept short and fast
  - Should represent a logical step in the task
    - i.e. Evaluating different parts of an incoming message.

Real-time Operating System

```
void idleState( void )
{
    if (rx_buffer_full)
    {
      read_rxBuffer;
      if (syncByte_received)
      {
         transition_to_workState1;
      }
      else
      {
        stay_in_idleState;
      }
    }
    else
    {
       stay_in_idleState;
    }
}
```

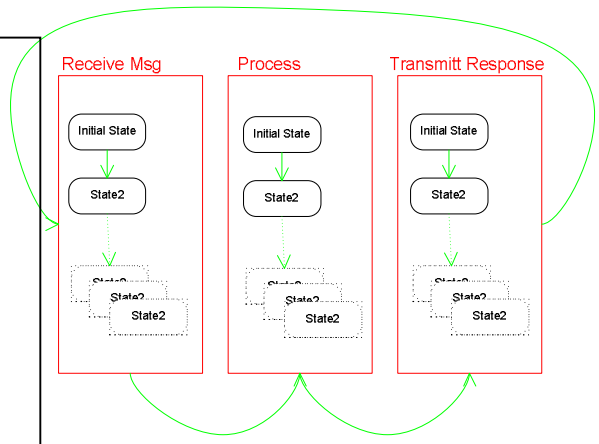


Real-time Operating System

- Event-driven
  - Tasks are granted execution time, based on an event (interrupt).
  - Tasks of higher priority are executed first (interrupt priorities).
- Time sharing
  - Each task is granted a given amount of time to execute.
  - Tasks may also be granted execution time based on events.
  - "Round-robin" approach
  - Creates a more deterministic multi-tasking system.

Real-time Operating Systems

```
void main ( void )
{
      initSystem();

        while (1)
        {
              work();
              sleep();
        }
}
void work (void)
{
      doTask(RecieveMsg);
      doTask(Process);
      doTask(TransmittResponse);
}
__interrupt void Timer_A (void)
{
      wakeUp();
}
```



Real-time Operating System

- Available commercially
  - TinyOS -an open source component-based operating system and platform targeting wireless sensor networks (WSNs).
  - Salvo - an RTOS developed by Pumpkin Inc.
  - FreeRTOS - free RTOS that runs on several architectures.
  - DrRTOS - works with ARM 7
- Implement a custom RTOS
  - Can be highly optimized to suit your application

Real-time Operating Systems

- A tool for real-time software developers.
- Allows increasingly complex systems to be developed in less time.
- Provides a level abstraction between software and hardware.
- Continues the evolution microcontroller system design.

SALVO & RTOS

Foreground/background (superloop) system

- The main application is a loop (background)
- Interrupts to handle time critical events (foreground)

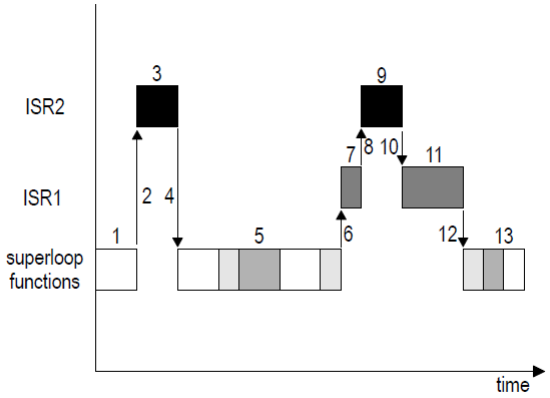


Figure 1: Foreground / Background Processing

- Simple
- Loop time is affected by any code change in ISR
- Poor response to input

multitasking

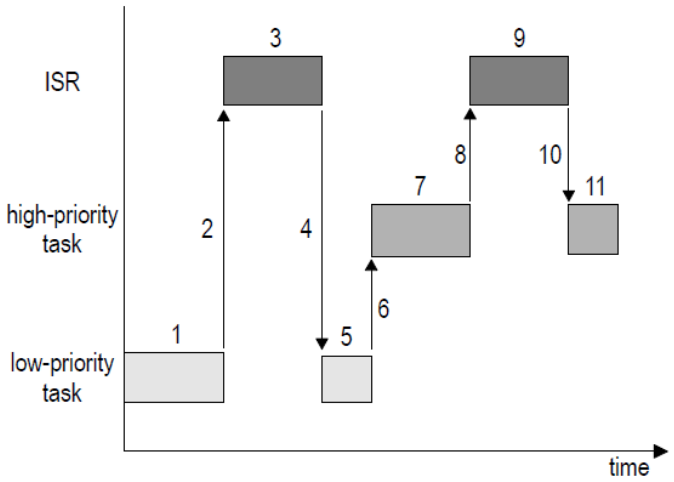


Figure 2: Interrupts Can Occur While Tasks Are Running

Preemptive vs cooperative scheduling

preemptive scheduler can cause the current task (i.e. the task that's currently running) to be preempted by another one. Preemption occurs when a task with higher priority than the current task becomes eligible to run. Because it can occur at any time,]

A *cooperative scheduler* is likely to be simpler than its preemptive counterpart. Since the tasks must all cooperate for context switching to occur, the scheduler is less dependent on interrupts and can be smaller and potentially faster. Also, the programmer knows exactly when context switches will occur, and can protect critical regions of code simply by keeping a context-switching call out of that part of the code.

Preemptive Scheduling

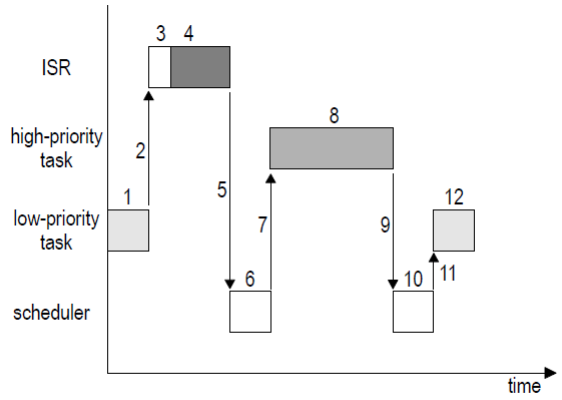


Figure 3: Preemptive Scheduling

- Stack intensive
- Complex

Cooperative Scheduling

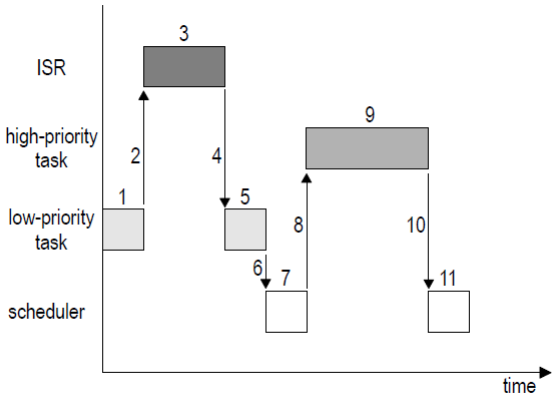


Figure 4: Cooperative Scheduling

- Simple
- Worse responsiveness

Task structure

```
Initialize();
for (;;) {
  ...
}
```

Listing 4: Task Structure for Preemptive Multitasking

```
Initialize();
for (;;) {
  ...
  TaskSwitch();
  ...
}
```

Listing 5: Task Structure for Cooperative Multitasking

Event driven multitasking

```
initialize binary semaphore #1 to 0;
initialize binary semaphore #2 to 1;

UpperTask()
{
  for (;;) {
    /* wait for LowerTask() */
    wait binary semaphore #1;
    do stuff;
    signal binary semaphore #2;
  }
}

LowerTask()
{
  for (;;) {
    do stuff;
    signal binary semaphore #1;
    /* wait for UpperTask() */
    wait binary semaphore #2;
  }
}
```

Listing 9: Task Synchronization with Binary Semaphores

RTOS Performance

The code to implement a multitasking RTOS may be larger than what's required in a superloop implementation. That's because each task requires a few extra instructions to be compatible with the scheduler. Even so, a multitasking application is likely to have much better performance and be more responsive than one with a superloop. That's because a well-written RTOS can take advantage of the fact that tasks that are not running often need not consume any processing power at all. This means that instead of spending instruction cycles testing flags, checking counters and polling for events, your multitasking application makes the most of the proc-

SALVO - basics

```
#include <salvo.h>

int main( void )

{
  OSInit();

  for (;;)
    OSSched();
}
```

Listing 23: A Minimal Salvo Application

Creating, Starting and Switching tasks

```
#include <salvo.h>

_OSLabel(TaskA1)
_OSLabel(TaskB1)


void TaskA( void )
{
  for (;;)
    OS_Yield(TaskA1);
}

void TaskB( void )
{
  for (;;)
    OS_Yield(TaskB1);
}

int main( void )
{
  OSInit();

  OSCreateTask(TaskA, OSTCBP(1), 10);
  OSCreateTask(TaskB, OSTCBP(2), 10);

  for (;;)
    OSSched();
}
```

Listing 24: A Multitasking Salvo Application with two Tasks

```
#include <salvo.h>

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)

unsigned int counter = 0;

void TaskCount( void )
{
  for (;;) {
    counter++;

    OS_Yield(TaskCount1);
  }
}

void TaskShow( void )
{
  TRISB = 0x00;
  PORTB = 0x00;

  for (;;) {
    PORTB &= ~0xFE;
    PORTB |= (counter >> 8) & 0xFE;

    OS_Yield(TaskShow1);
  }
}

int main( void )
{
  OSInit();

  OSCreateTask(TaskCount, OSTCBP(1), 10);
  OSCreateTask(TaskShow,  OSTCBP(2), 10);

  for (;;)
    OSSched();
}
```

Listing 25: Multitasking with two Non-trivial Tasks

Using events

```
#define TASK_COUNT_P       OSTCBP(1) /* task #1 */
#define TASK_SHOW_P        OSTCBP(2) /* task #2 */
#define PRIO_COUNT         10 /* task priorities*/
#define PRIO_SHOW          10 /*  ""             */
#define SEM_UPDATE_PORT_P OSECBP(1) /* sem #1  */

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)

unsigned int counter = 0;

void TaskCount( void )
{
  for (;;) {
    counter++;

    if ( !(counter & 0x01FF) )
      OSSignalSem(SEM_UPDATE_PORT_P);

    OS_Yield(TaskCount1);
  }
}

void TaskShow( void )
{
  TRISB = 0x00;
  PORTB = 0x00;
```

```
  for (;;) {
    OS_WaitSem(SEM_UPDATE_PORT_P, TaskShow1);

    PORTB &= ~0xFE;
    PORTB |= (counter >> 8) & 0xFE;
  }
}

int main( void )
{
  OSInit();

  OSCreateTask(TaskCount,
    TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow,
    TASK_SHOW_P,  PRIO_SHOW);

  OSCreateSem(SEM_UPDATE_PORT_P, 0);

  for (;;)
    OSSched();
}
```

Listing 26: Multitasking with an Event

```
#include <salvo.h>

#define TASK_COUNT_P       OSTCBP(1)   /* task #1 */
#define TASK_SHOW_P        OSTCBP(2)   /*   ""   #2 */
#define TASK_BLINK_P       OSTCBP(3)   /*   ""   #3 */
#define PRIO_COUNT         10   /* task priorities*/
#define PRIO_SHOW          10   /*   ""            */
#define PRIO_BLINK         2    /*   ""            */
#define SEM_UPDATE_PORT_P OSECBP(1)  /* sem #1  */
#define TMR2_RELOAD        250 /* for 100Hz ints */

unsigned int counter = 0;

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)
_OSLabel(TaskBlink1)

void TaskCount( void )
{
  for (;;) {
    counter++;

    if ( ! counter & 0x01FF )
      OSSignalSem(SEM_UPDATE_PORT_P);

    OS_Yield(TaskCount1);
  }
}

void TaskShow( void )
{
  for (;;) {
    OS_WaitSem(SEM_UPDATE_PORT_P, TaskShow1);

    PORTB &= ~0xFE;
    PORTB |= (counter >> 8) & 0xFE;
  }
}

void TaskBlink( void )
{
  TRISB = 0x00;
  PORTB = 0x00;

  for (;;) {
    PORTB ^= 0x01;

    OS_Delay(50, TaskBlink1);
```

```
  }
}

int main( void )
{
  PR2 = TMR2_RELOAD;
  T2CON = 0x4D;

  OSInit();

  TMR2IE = 1;
  PEIE = 1;
  OSEnableInts();

  OSCreateTask(TaskCount,
    TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow,
    TASK_SHOW_P,  PRIO_SHOW);
  OSCreateTask(TaskBlink,
    TASK_BLINK_P, PRIO_BLINK);

  OSCreateSem(SEM_UPDATE_PORT_P, 0);

  for (;;)
    OSSched();
}

void interrupt intVector( void )
{
  if ( TMR2IE && TMR2IF ) {
    TMR2IF = 0;

    OSTimer();
  }
}
```

Listing 27: Multitasking with a Delay