Stream: ECE

Paper Name: Advanced Microcontroller and Embedded system

Paper Code: EC 604B **Contacts: 3L** **Credits: 3** **Total Contact: 36**

Semester: 6th

Prepared by: Mr.Sayan Roy Chaudhuri.

Pre requisite:

(1) Concepts in 8085 ,8086 Microprocessor

(2) concept of MCS51 series of Microcontroller.

Course Objectives:

- To familiarize the students with concepts related to the fundamental principles embedded systems design, explain the process and apply it .
- To understand knowledge of the advanced microcontroller technology both for hardware and software.
- Student will able to understand Hardware/Software design techniques for microcontroller-based embedded systems and apply techniques in design problems.
- Student will able to develop microcontrollers programming in C and assembly language using Integrated Development Environments and using debugging technique.

Module I INTRODUCTION TO PIC MICROCONTROLLER : PIC 18F4550 Microcontroller – Hardware Architecture & GPIOs ((Pin Diagram, Memory Organization, SFRs description, Program Counter, Accumulator (or Working Register), Reset, Clock Cycle, Machine Cycle, Instruction Cycle, Interrupts, SFRs & GPRs, Stack, Stack Pointer, Stack Operation, Timers and serial communication in PIC 16F877A). Microcontroller PIC Assembly Language, Programming in Embedded C, Introduction to programming software, Examples programs for PIC.

Module II: INTERFACING PIC 16F877A WITH INPUT OUTPUT DEVICES : LED Display,7-Segment, DIP Switch, Intelligent LCD Display, Matrix Keyboard, Stepper Motors and Types of Stepper Motors, Serial Communication Concepts, Practices on interfacing circuits, serial and parallel communication devices, wireless communication devices, timer and counting devices, watchdog timer, real time clock, serial bus communication protocols, USB, Bluetooth, Practices of ICP, ADC, EEPROM, Opto-Isolators, Relay, I2C, SPI Protocol, Serial Memory, On chip Peripherals PWM.

Module III: ARM ARCHITECTURE AND PROGRAMMING: Introduction of ARM Processors, Evolution of ARM, 32 - bit Programming.ARM7 Architecture, Instruction Set Architecture, LPC21xx Description, Memories & Peripherals. ARM Processor Programming in C, Using ARM Programming Tools.

Module IV: INTRODUCTION TO EMBEDDED SYSTEM: Basics of Embedded computer Systems, Microprocessor and Microcontroller difference, Hardware architecture and software components of embedded system List of various applications [Mobile phones, RFID, WISENET, Robotics, Biomedical Applications, Brain machine interface etc.], Difference between embedded computer systems and generalpurpose computer Systems. Characteristics of embedded systems, Classifications of embedded system.

Module V: HARDWARE SOFTWARE CO- DESIGN: Co-Design Types: Microprocessors/Microcontrollers/DSP based Design, FPGA / ASIC /pSOC based Design, Hybrid Design. Methodology: i) System specifications ii)) co-specifications of hardware and software) iii)) System Design Languages (capturing the specification in a single Description) iv) System modeling /simulation v) Partitioning (optimizing hardware/software partition) vi) Co-verification (simulation interaction between custom hardware and processor) f) Co-implementation vii) Embedded Systems Design development cycle.
Programming concepts and embedded programming in C.

MODULE VI: - REAL TIME OPERATING SYSTEM (RTOS): - Introduction, Types, Process Management, Memory Management, Interrupt in RTOS, Task scheduling, Basic design using RTOS; Basic idea of Hardware and Software testing in Embedded Systems

---

Text Books:

1. Steve Furber, 'ARM system on chip architecture', Addision Wesley
2. Microchip's PIC microcontroller is rapidly becoming the microcontroller of choice throughout the world, Myke Predco
3. Embedded system Design: Peter Marwedel, Springer
4. Embedded Systems - Raj Kamal
5. PIC Microcontroller – Mazidi and Mazidi

Reference books:

1. Andrew N. Sloss, Dominic Symes, Chris Wright, John Rayfield 'ARM System Developer's Guide Designing and Optimizing System Software', Elsevier 2007.
3. ARM Architecture Reference Manual

Course outcome

EC604B.1. Analyze the performance of PIC microcontroller.

EC604B.2. Design and develop the systems based on ARM controllers.

EC604B.3. an ability to use the techniques, skills, and modern engineering tools in embedded system.

Program outcome

1.Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

2.Problem Analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics,natural sciences and Engineering sciences.

3.Design/Development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4.Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5.Modern tool usage : Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

6.The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7.Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8.Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9.Individual and team work: Function effectively as an individual and as a member or leader in diverse teams, and in multidisciplinary settings.

10.Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

Mapping of POs with COs:

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	P10	P11	P12
EC604B.1	3	3	2	-	**1**	**-**	**-**	**-**	**1**	**1**	**-**	**1**
EC604B.2	3	2	2	-	**-**	**-**	**-**	**1**	**2**	**1**	**-**	1
EC604B.3	3	2	1	2	**1**	**1**	**-**	**-**	**2**	**1**	**-**	-
EC603avg	3	2	2	1	**1**	**1**	**0**	**1**	**2**	**1**	**0**	1

OCW

PIC Microcontroller

PIC Microcontrollers------Introduction

PIC stands for Peripheral Interface Controller coined by Microchip Technology to identify its single- chip microcontrollers. These devices have been phenomenally successful in 8-bit microcontroller mar- ket. The main reason is that Microchip Technology has constantly upgraded the device architecture and added needed peripherals to the microcontroller to 'suit customers' requirements. The develop- ment tools such as assembler and simulator are freely available on the internet at www.microchip.com

Low-end Architectures

Microchip PIC microcontrollers are available in various types. When *PI C − M icro* MCU first became available from General Instruments in early 1980's, the microcontroller consisted of a very simple processor executing 12-bit wide instructions with basic I/O functions. These devices are known as low-end architectures.
Some of the low-end device past numbers are 12C5XX, 16C5X, and16C505

Mid-range Architectures

Mid-range Architectures are built by upgrading low-end architecture with more number of peripherals, more numbers of register and more data memory. Some of the mid-range devices are
16C6X
16C7X, 16F87X
↑Program memory type
C = EPROM
F= Flash
RC = Mask ROM
Popularity of PIC microcontrollers is due to the following factors- 1. Speed: Harvard Architecture, RISC Architecture 1 instruction Cycle = 4 clock cycles.

For 20 MHz clock, most of the instructions are executed in 0.2μs or five instructions per mi-

crosecond.

2. Instruction Set Simplicity:

The instruction set consists of just 35 instructions (as opposed to 111 instructions for 8051)

3. Power on reset

   Power-out reset

   Watch-dog timer

   Oscillator Options

   - low-power Crystal
   - Mid-range Crystal
   - High-range Crystal
   - RC Oscillator

4. Programmable timer options on chip AD

5. Up to 12 independent interrupt sources

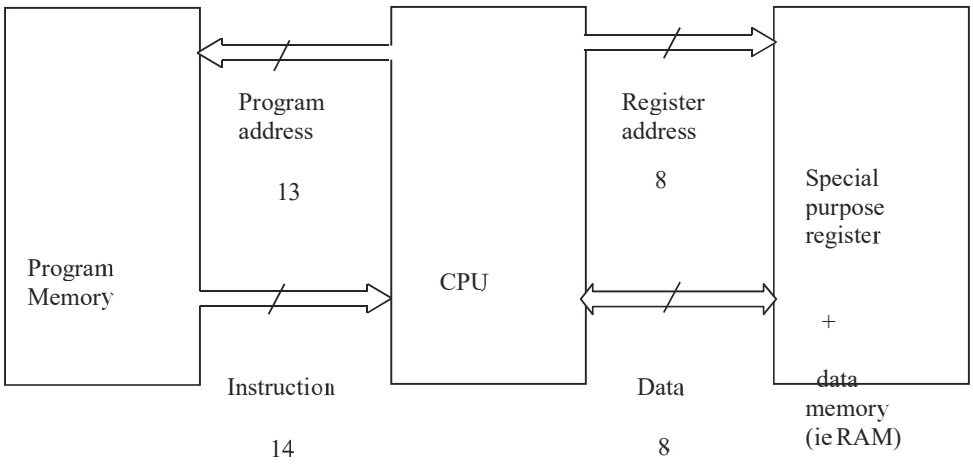
6. Powerful output pin control

   25mA (max.) current sourcing capability.

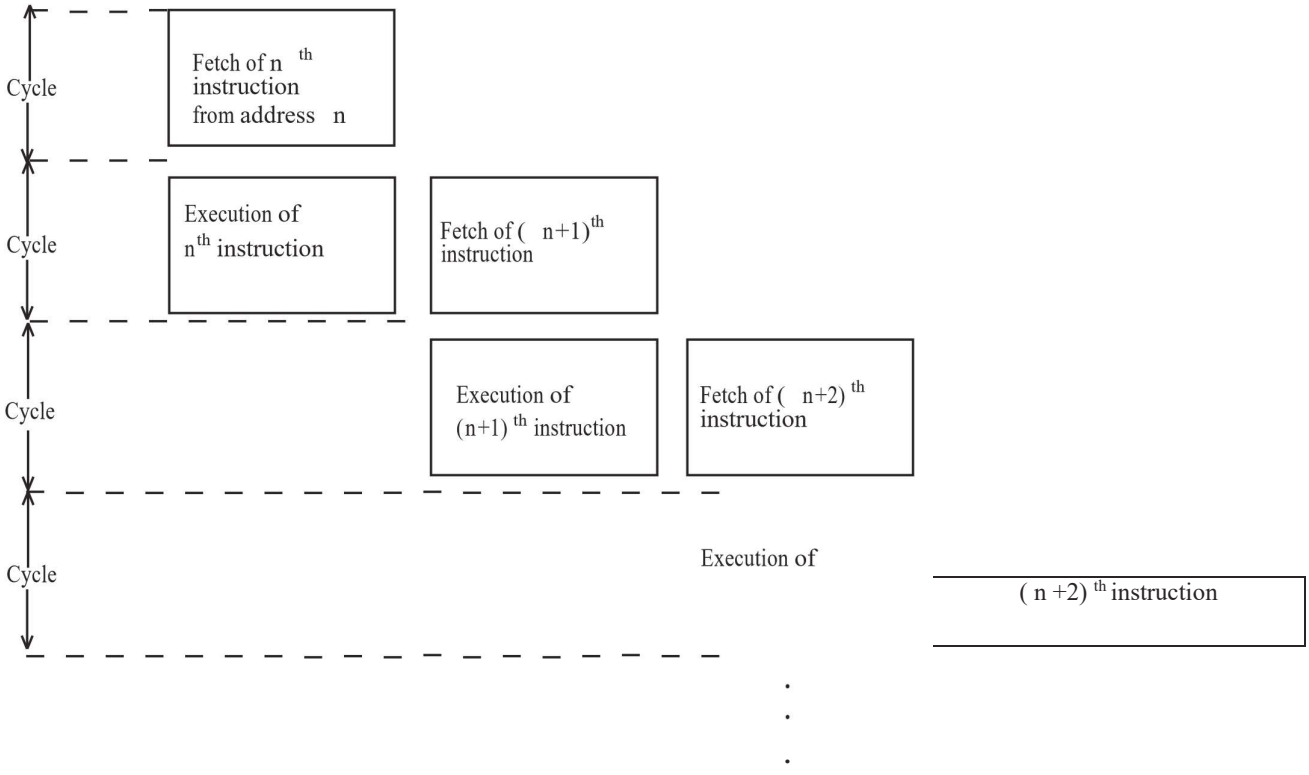
7. EPROM/OTP/ROM/Flash memory options.

8. Free assembler and simulator support from microchip at http://www.microchip.com

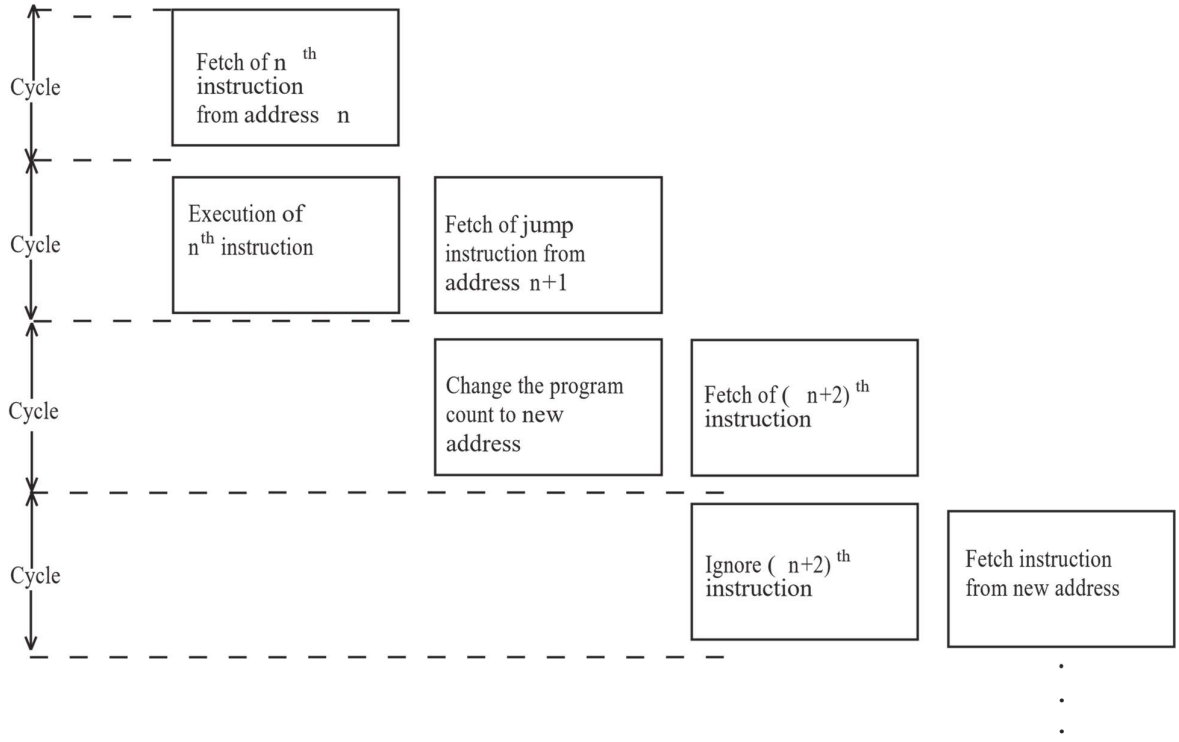
CPU Architecture and Instruction Set



Pipelining of instruction fetch successive addressing



Introduction of extra cycle for a jump/goto instruction

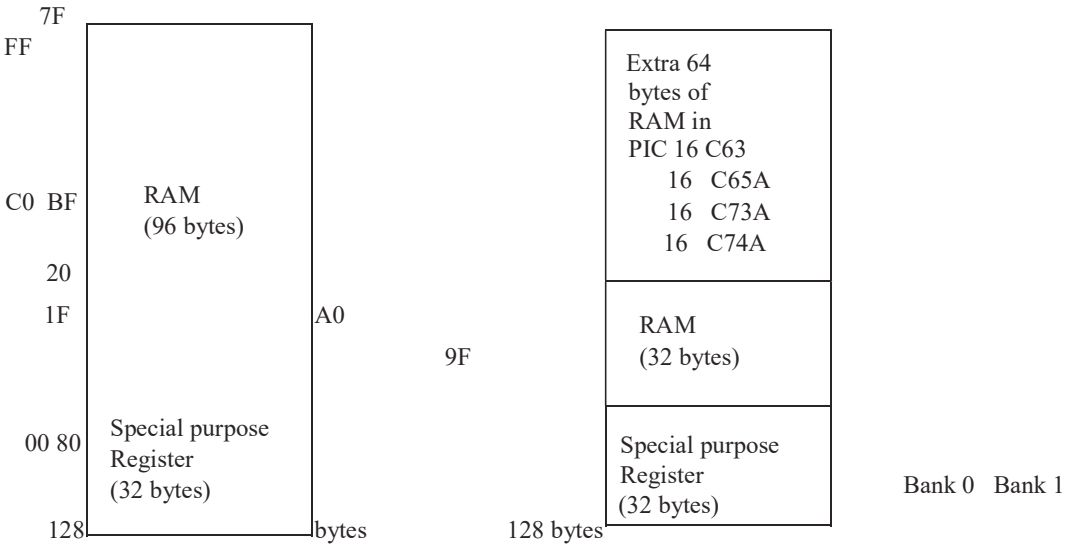


Register File Structure and Addressing Modes

Register file → locations that an instruction can access via an address. Register file consists of two components.

1. General purpose register file (same as RAM)

2. Special purpose register file



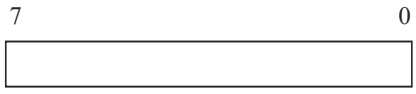
RPO bit in the Status register detects the bank. 7 bit of direct address TRPO determines the absolute address of the register.

Indirect addressing mode

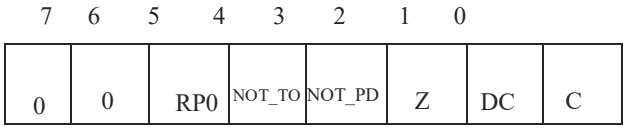
FSR contains the 8-bit address of the data/register.

CPU Registers

W, the working register, is used by many instructions as the source of an operand. It may also serve as the destination for the result of the instructions execution. It works as the accumulator.



W working register



STATUS
(address 03H,83H)

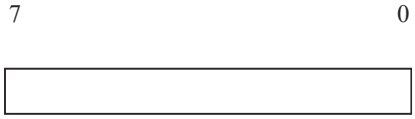
C = Carry bit

DC = Digit Carry (same as AC, Auxiliary Carry)

Z = Zero bit

NOT-TO, NOT-PD→ Used in conjunction with PIC's sleep mode

RPO → register bank select bit used in conjunction with the direct addressing mode.



FSR

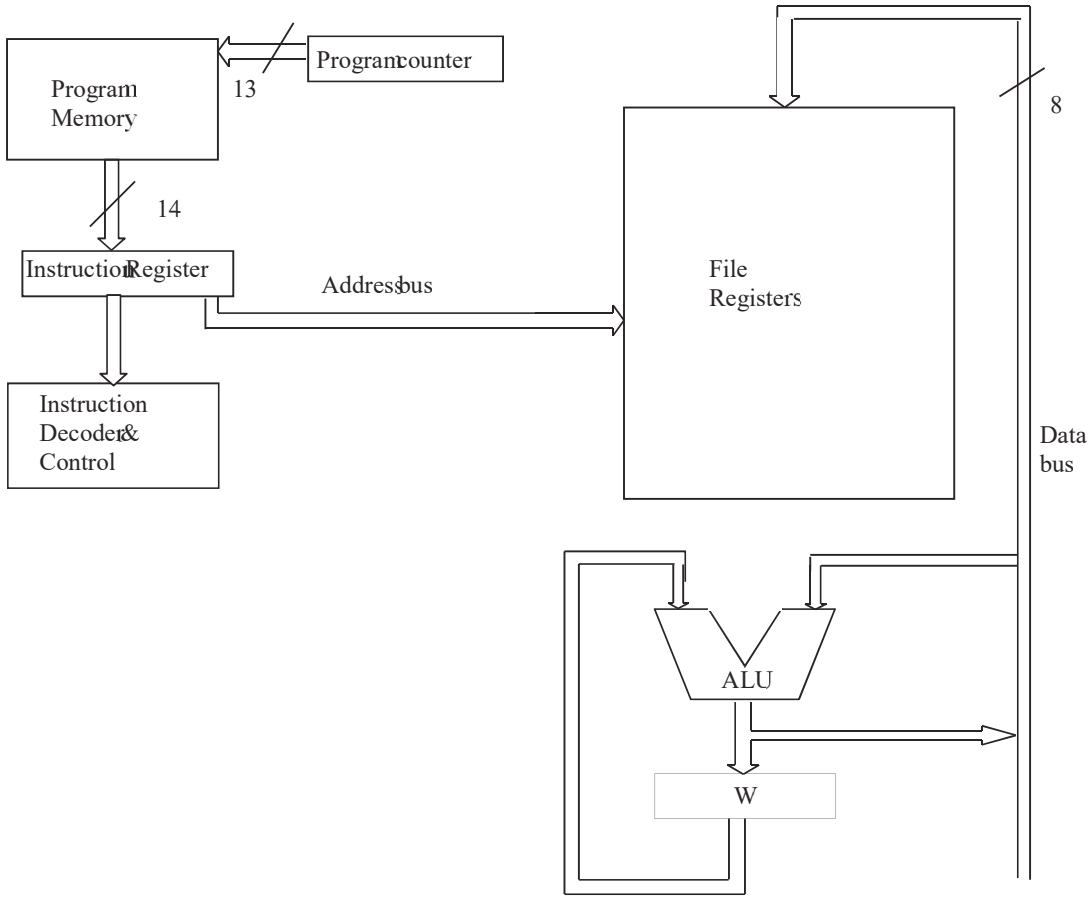
(address 04H,84H)

Indirect data memory address points.

FSR is the pointer used for indirect addressing.

The program is supported by an eight-level stack. When an interrupt occurs, the program counter is automatically pushed on to the stack. Since PIC microcontrollers programs are normally designed for handling one interrupt at a time, further

Basic Architecture of PIC Microcontroller



W → Temporary holding register, often called as an accumulator, cannot be accessed directly. Instead, contents must be moved to other registers that can be accessed directly.

Bank Addressing

Bank 0 Bank 1

00	INDF	INDF
01	TMRO	OPTION
02	PCL	PCL
03	STATUS	STATUS
04	FSR	FSR
05	PORTA	TRISA
06	PORTB	TRISB
07	PORTC	TRISC
08	PORTD	TRISD
	PORTE	TRISE
0A 8A	PCLATH	PCLATH
0B 8B	INTCON	INTCON
0C	. . .	
1F 9F		
07F 0FF		

TRIS bit set → Post bit in I mode Reset →
Post bit in 0 mode.

EX: To set PORT B bit 0 as an output and loaded with a 1, PIC micro MCU code would exe- cute as:
Port B. Bit = 1
STATUS. RPO = 1
TRIS B. Bit 0 = 0
STATUS. RPO = 0

PIC 16C74A

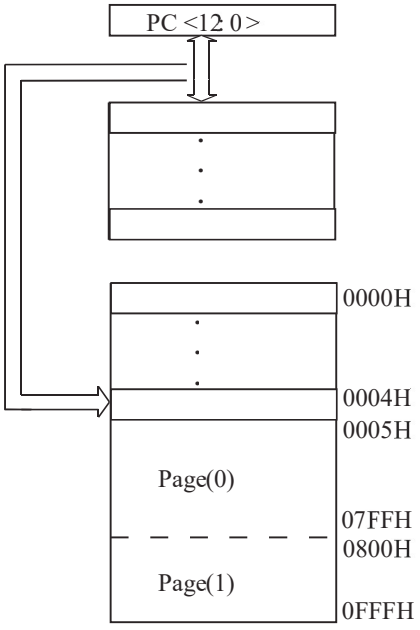
Program Memory (EPROM)×14	4k
Data Memory (Bytes)×8	192
I/O Pins	33
Parallel slave port	Yes
A/D channel Serial Comm	8 SPI/I^2C, USART
Interrupt sources	12

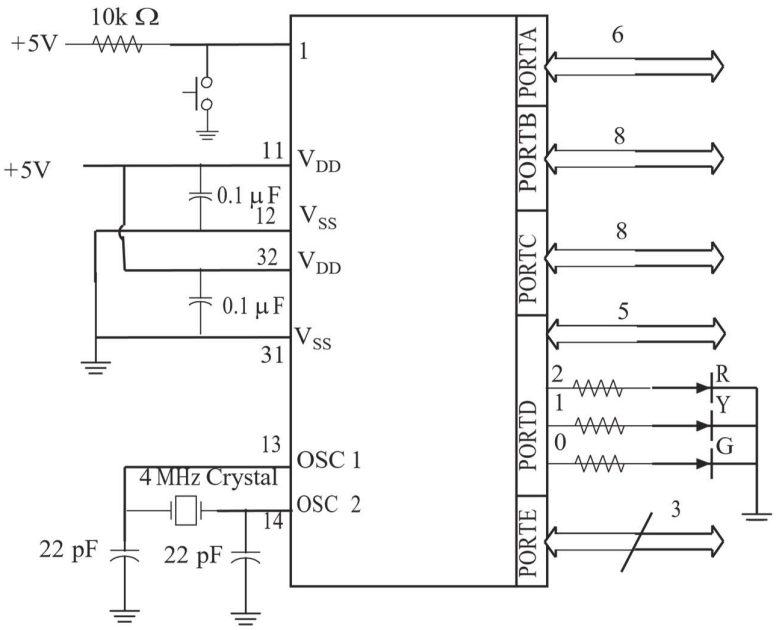
Memory Organization

The PIC 16C7X family has a 13-bit program counter capable of addressing 8k×14 program memory. PIC16C74A has 4k×14 program memory. For those devices with less than 8k program memory, accessing a location above the physically implemented address will cause a wraparound.

Program memory map and stack

16C74A has 4k program memory. The address range is 0000H - 0FFFH. The reset vector is 0000H and the interrupt vector is 0004H.



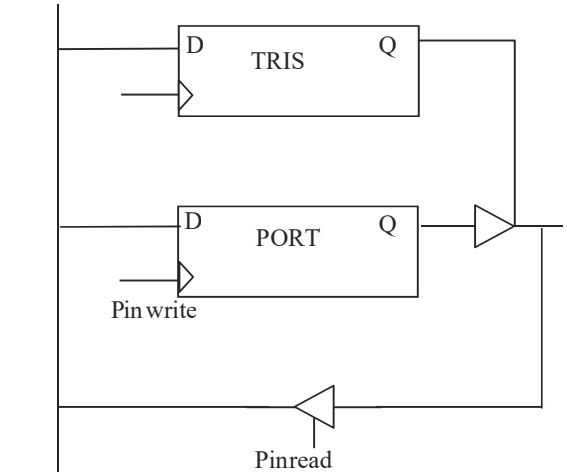


PIC 16C74A

PIC 16C74A has five ports. Each port is a bidirectional I/O port. In addition, they have the following alternative functions.

Port	Alternative uses of I/O pins	I/O pins	
		64A 65A 74A	62A 63A 73A
PORTA	A/D Converter inputs (PIC 16C7X parts)	6	6
PORTB	External interrupt inouts	8	8
PORTC	Serial port, Timer I/O	8	8
PORTD	Parallel slave port	8	0
PORTE	A/D Convertor inputs (PIC 16C 7X)	3	0
	Total I/O pins	33	22
	Total pins	40/44	28

Port D alternative function is parallel slave port which enables one PIC microcontroller to be connected to the data bus of another microprocessor. Since three LED's are connected to three pins of Port D to be used as normal I/O operation, the special alternative function is ruled out.



TRIS register controls the direction of data flow. TRIS = 1
Sets the pin in the input mode.
TRIS = 0 Sets the pin in the output mode.

```
; Toggle the green LED every half second.
    List        P = PIC16C74A,        F = INHX8M,        C = 160,        N = 80
            ST  =  FF,  MM  =  OFF,  R  =  DEC  include
    "C:\MPLAB\P16C74A.INC"
    - config (_CP OFF & _PWRTE ON &_XT-OSC & _WDT OFF &  BODEN OFF) error level -302
; Equates
     Bank0 RAM     equ       20H
      MaxCount equ      50
     Green equ      0000000HB
    TenMsH    equ    13
    TenMsL    equ    250
; Variables
       cblock Bank0RAM        ;      Variables are declared
    BLNKCNT
    COUNTH
    COUNTL endc
; Vectors
    org       000H
     goto       Mainline
    org       004H
Stop: goto
stop
Mainline: call
        Initial
        ;
Initialize
Main loop:
    call    Blink        ; Blink LED
```

```
      call     TenMs          ; Inset ten millisecond delay
 indent goto Mainloop
 ;Initial Subroutine Initial:
          movlw      MaxCount        0.5 second
          movwf      BLNKCNT           BLKCNT ← N
          movlw     Green
          movwf      PORTD           PORTD ← W
          bsf STATUS,RPO Set register access to bank 1 clrf TRISD
          Set PORTD as O/P port bcf STATUS,RPO Set register
          access to bank 0 return
 ; Blink Subroutine. This subroutine blinks a green LED in evey 0.5 sec Blink:
          decfsz          BLNKCNT,F     ;          decrement loop counter and return if not zero
          goto BlinkEnd
          movlw MaxCount ; Reinitialize BLNKCNT movwf BLNKCNT
          movlw           GREEN           w ← Green Toggle green LED Xorwf
          PORTD,F         w ← Green Toggle green LED
      Blink End; return
 ; Ten Ms subroutine (delay of 10ms) Ten Ms:
      nop movlw          TenMsH          movwf
         COUNTH          ; COUNTH ←w movlw
         TenMsL movwf           COUNTL
 TEN_1:
          decfsz  CountL.f  goto
          Ten       1       decfsz
          COUNTH,f goto Ten 1
          return
```

Ten Ms subroutine introduces a delay of 10ms by counting 10,000 instruction cycles. This is achieved by nested loops. The sequence of instructions executed from calling Ten Ms is listed and correspond- ing instruction cycles are mentioned against the instructions.

Instructions Call ten Ms		Instruction Cycles 2
nop		1
movlw 13 (TenMsH)		1
movwf COUNTH		1
movlw 250 (Ten MsL)		1
movwf COUNTL		1
decfsz COUNTL,F Ten_1	COUNTL:250 →249→...→1 goto	3 × 249 = 747
decfsz COUNTL,F	COUNTL: 1 → 0	2
decfsz COUNTH,F	COUNTH: 13 → 12	1
goto Ten_1		2
decfsz COUNTL,F		

COUNTL: 0→255→254...→1 goto Ten 1

255 × 3 = 765

decfsz COUNTL,F COUNTL:1 →0 2 decfsz COUNTH,F COUNTH: 12 →11 1 goto Ten_1 2

770 ×11=8470

Repeat this block 11 times as

COUNTH:12→11→...→2→1

decfsz COUNTL,F goto Ten_1	COUNTL:0→255→...2→1	3×255=765
decfsz COUNTL,F	COUNTL:1→0	2
decfsz COUNTH,F	COUNTH:1→0	2
return		2

Total = 10,000

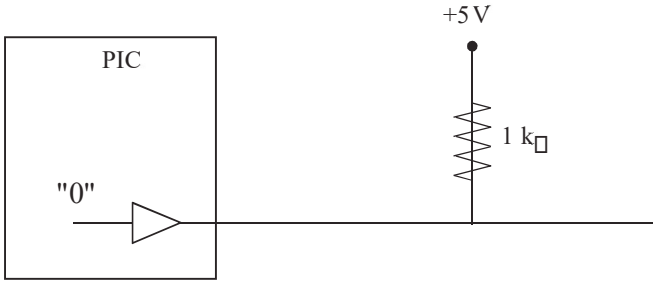
I2C Bus for Peripheral Chip Access

Requires two open-drain I/O pins.

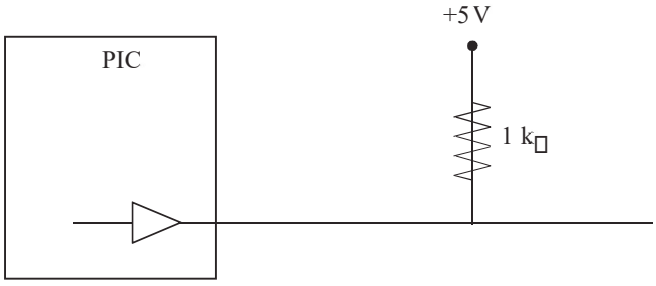
Port-C of PIC IC can be used for I^2C communication.

SCL (Serial Clock) RC3/SCK/SCL

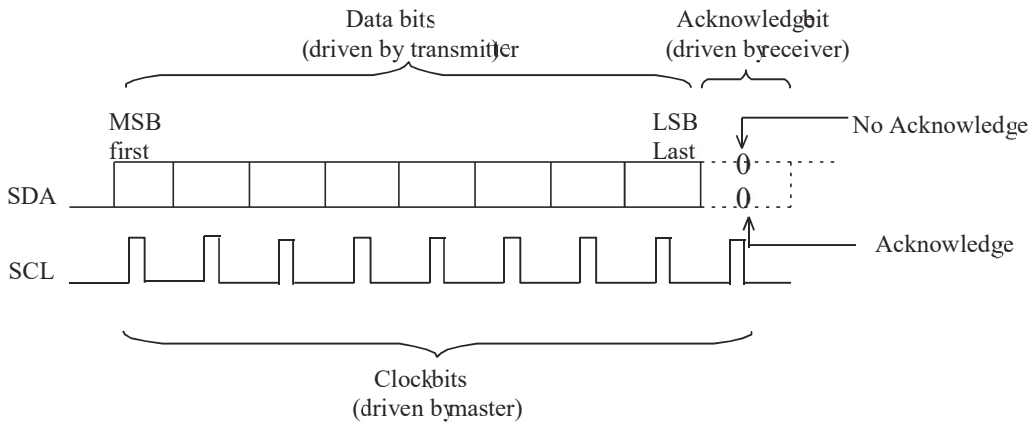
SDA (Serial Data) RC4/SDI/SDA



Low output on SCL or SDA I/O pin set to be an output with ”0” written to it.



High output on SCL or SDA I/O pin set to be an input. Transfers on the I^2C bus take place a bit at a time.



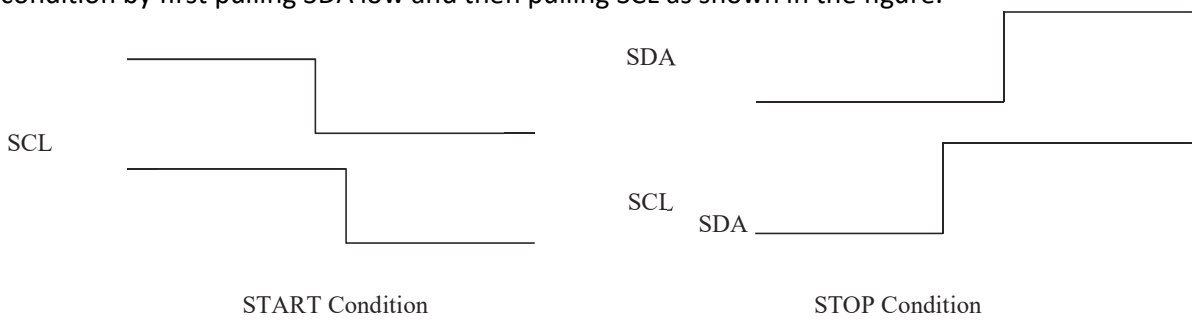
The clock line, SCL, is driven by the PIC chip, which server as *bus master*. The open drain feature of every chip's bus driver can be used by the receiver to hold the clock line low, there by signalling the transmitter to pause until the clock line is released by the receiver. The open drain feature is also needed if this PIC will ever become an I^2C slave to another PIC, in which it must relinquish control of the SCL line.

The previous figure illustrates that the first eight bits on the SDA line are sent by the transmitter whereas the ninth bit is the acknowledgment bit which is sent by the receiver in response to the byte sent by the transmitter. For instance, when the PIC sends out a chip address, it is the transmitter, while every other chip on the I^2C bus is a receiver. During the acknowledgment bit time, the addressed chip is the only one that drives the SDA line, pulling it low in response to the masters pulse on SCL, acknowledging the reception of its chip address.

When the data transfer direction is reversed that is form a peripheral chip to the PIC, which is the master , the peripheral chip drives the eight daa bits in response to the clock pulse from PIC. In this case, the acknowledge bit is driven in a special way by the PIC, which is serving as receive but also as bus master. If the peripheral chip is one that can send the contents of successive internal address back to the PIC, then PIC completes the reception of each byte and signals a request for the next byte by pulling *SDA line low* in acknowledgment. After any number of bytes have been received by the master from the peripheral, the PIC can signal the peripheral to stop any further transfers by not pulling the SDA line low in acknowledgment.

SDA line should be *stable during high period of the clock (SCL)*. When the slave peripheral is driving SDA line , either as transmiter or acknowledge, it initiates the new bit in response to the falling edge of SCL, after a specified time. It maintains that bit on SDA line until the next falling edge of SCL, again afte r a specified hold time.

I^2C bus transfers consist of a number of byte transfers framed between a START condition and either another START condition or a STOP condition. Both SDA and SCL lines are released by all drives and *float high* when bus transfers are not taking place. The PIC (I^2C bus controller) initiates a transfer with a START condition by first pulling SDA low and then pulling SCL as shown in the figure.



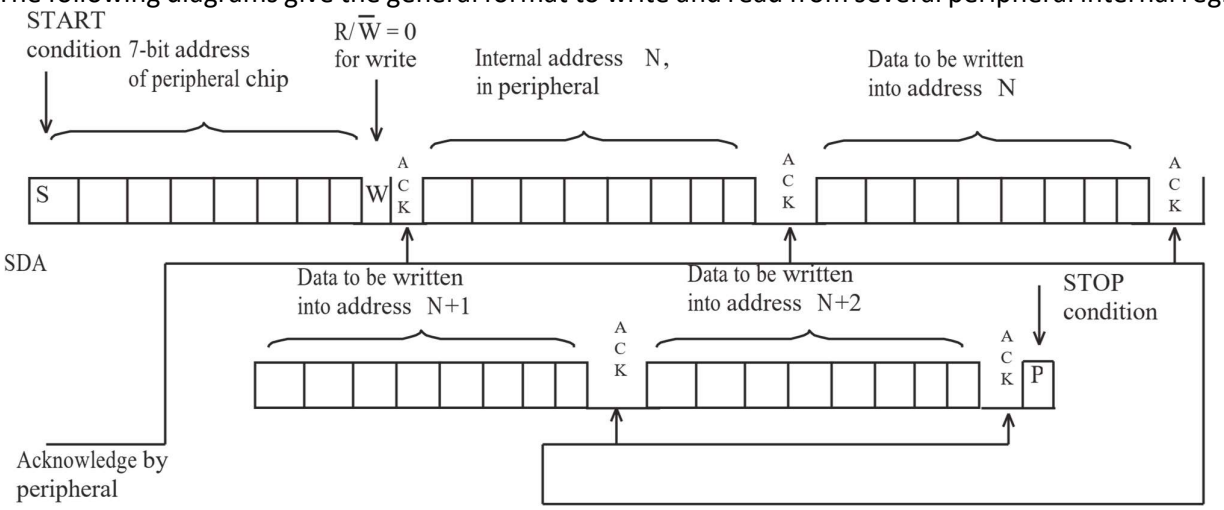
Similarly, the PIC terminates a multiple byte transfer with the STOP condition. With both SDA and SCL initially low, it first releases SCL and then SDA. Both then occurrences are easily recognized by I^2C hardware in each peripheral chip since they both consist of a chage in SDA line which SCL is high, a condition that never happens in the middle of a byte transfer.

Data Communication protocol

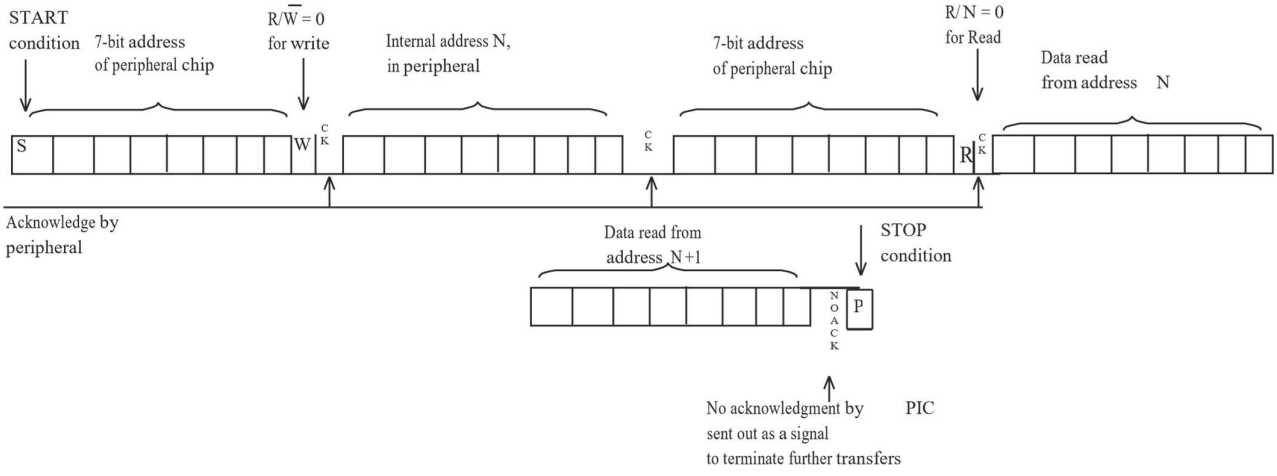
In I^2C communication standard, there is one bus master and several slaves. It can be assumed here that the PIC microcontroller is the bus master and several peripheral devices connected to SDA and SCL bus are slaves.

Following a start condition, the master sends a 7-bit address of the slave on SDA line. The MSB is sent first. After sending 7 bit address of the slave peripheral a R/~~W~~ bit (8^{th} bit) is sent by the master. If R/~~W~~ bit is 0 the following byte (after the acknowledgment) is written by the master to the addressed slave peripheral. If R/~~W~~ bit is 1, the following byte after the acknowledgment bit has to be read from the slave by the master. After sending the 7-bit address of the slave, the master sends the address of the internal register of the salve where from the data has to be used or written to. The subsegment access is automatically directed to the next address of the internal register.

The following diagrams give the general format to write and read from several peripheral internal registers.



General format to write to several peripheral internal registers or addresses.

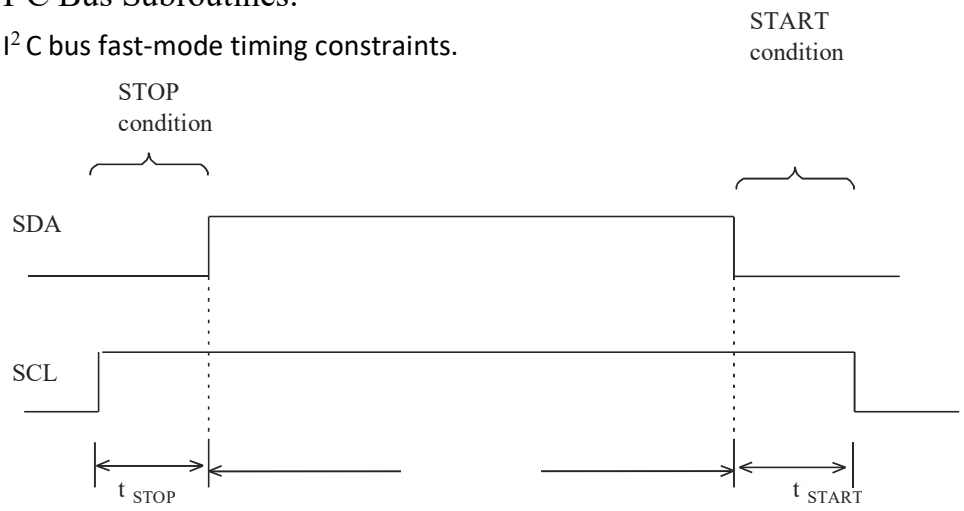


General format to from several peripheral internal registers or addresses.

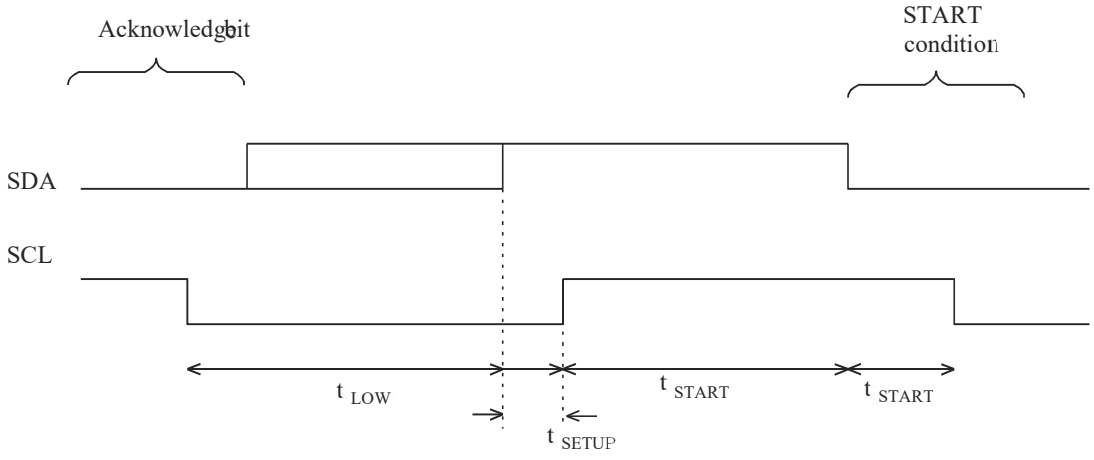
The 1995 I^2C bus specification includes the timing constraints for older chips designed for a maximum bit rate of 100kbits/s. It also includes constraints for newer fast-mode 400kbits/s parts.

I^2C Bus Subroutines:

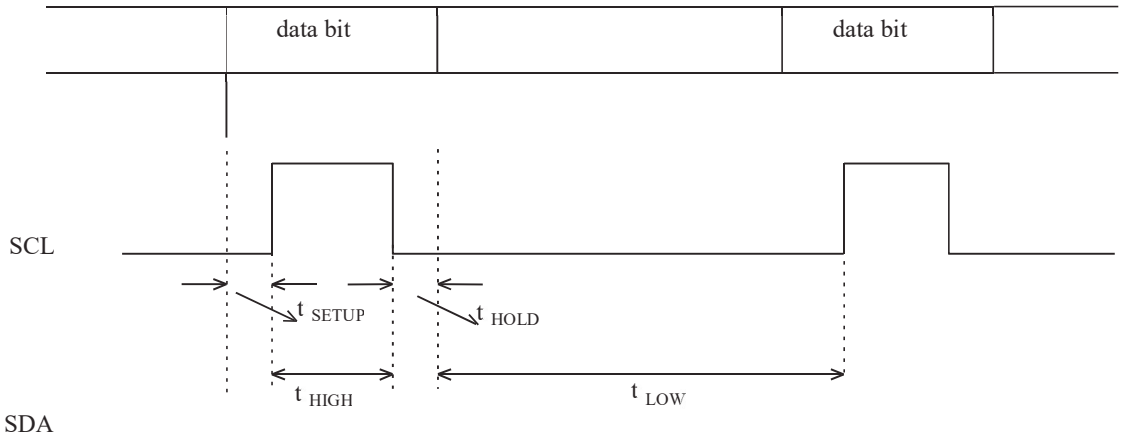
I^2C bus fast-mode timing constraints.



STOP- to - START Constraints



Acknowledge bit to START (restart condition)



Data bit to data bit

Parameter	Constraint	Cycles required to meet constraint		
		ocs = 4MHz Period = 1μs	osc = 10MHz Period = 0.4μs	osc = 20MHz Period = 0.2μs
t_{START}	> 0.6μ s	1 1	2 1	3 1
t_{SETUP} t_{HIGH}	> 0.1μ s	1	2	3
t_{HOLD} t_{LOW}	> 0.6μ s >	1 2	1 4	1 7
t_{STOP}	0μ s	1	2	3
$t_{STOP-START}$	> 1.3μ s	2	4	7
	> 0.6μ s			
	> 1.3μ s			

Because the SCL pin must have an open pin output which the SDA pin must be either an input or have an open drain output, the I^2C subroutines will repeatedly access TRISC, the data direction register for PORTC, However, TRISC is located at the bank 1 address 87H, which cannot be accessed by direct addressing without changing RPO bit to 1.

bsf STATUS, RPO

Then required bit of TRISC can be changed followed by clearing RPO and reveting back to Bank 0. bcf STATUS, RPO

Instead of doing this, the indirect pointer FSR can be loaded with the address of TRISC and the bit setting and bit clearing of TRISC can be done indirectly.

For example, with the following definitions

SCL equ 3

SDA equ 4 bsf INDF, SDA will release the SDA line, letting the external pull up register pull it high or some I^2C chp pull it low. When FSR is used for indirect addressing, care should be taken to restore FSR value when a subroutine is completed and the program returns to the mainline program.

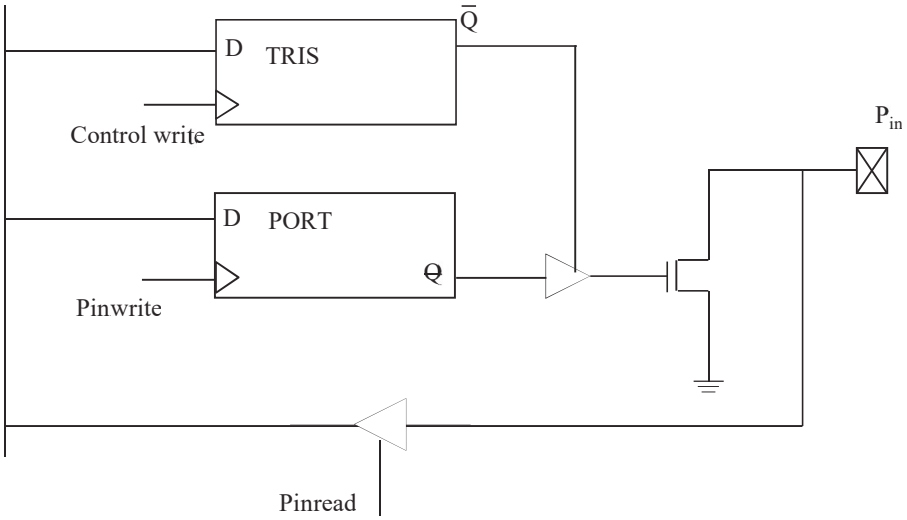
I2C Subroutines

```
Freq equ 4 SDA equ 4
SCL              equ 3


cblock

..

 DEVADD                  ;The I2Cout subroutine transfers out three bytes:
INTADD
DATAOUT
DATAIN
TXBUFF RXBUFF
.
.
endc
;DEVADD, INTADD, and DATAOUT
I2C out :
        call start
        movf DEVADD, W ; Send peripheral address with R/W~ =0 (write) Call Tx
        movf INTADD, W Call Tx
        movf DATAOUT, W
        Call Tx
        Call Stop       ; Generate Stop condition return
; The I2C in subroutine transfers out DEVADD (with R/W~ =0) ; and
INTADD, restarts, transfers out DEVADD (with R/W=1) ; and read one
byte back into DATAIN.

I2C in:

        Call Start                  ; Generate start condition
         movf DEVADD, W          ; Send peripheral address          R/W~ =0 (write)
        Call Tx
        movf INTADD, W Call Tx      ; Send peripheral's internal address

        Call  ReStart  movf         ; Re START
        DEVADD ,W                   ; Send peripheral's address.
        iorlw  0000000.1  B  Call   ; with R/W~ =1 (read)
        Tx
        bsf TXBUFF, 7               ; NOACK the following reading of one byte
         Call Rx                    ; Read  byte
        movwf DATAIN ; inte  DATAIN Call stop  ; Generate stop
        condition return
```

; The start subroutine initializes the I2C bus and then

```
; generates  the  START condition on the I2C bus
; The ReStart entry point bypadd the initialization of the
; I2C  bus

Start:
        movlw  00111011
        movwf SSPCON          ; Enable I2C Master mode.
        bcf PORTC, SDA        ; DRIVE SDA low when it is an output
        bcf PORTC, SCL        ; DRIVE SCL low when it is an output
        movlw TRISC movwf     ; Set indirect pointer to TRISC
FSR ReStart:

        bsf INDF, SDA                 ; Make sure SDA is high - I/P mode
        bsf  INDF  ,SCL  delay        ; Make sure SCL is high - I/P mode
        0,1,2 not
        bcf  INDF  ,SDA  delay        ; Make SDA low
        0,1,2 nop
        bcf INDF, SCL return          ; Make SCL low


Stop:
          bcf INDF, SDA               ; Return SDA low
        bsf  INDF,  SCL  delay        ; Drive SCL high
        0,1,2
        bsf INDF, SDA return          ; and then drive SDA high


; The Tx subroutine sends out the byte passed to it in W.
; It returns with z = 1 if ACK occurs.
; It returns with z = 0 if NOACK occurs.
Tx:
        movwf TXBUFF
        bsf STATUS, C

Tx_1:
        rlf TXBUFF, F ; rotate  TXBUFF left, through carry movf TXBUFF, F ; Set  Z bit
        when all 8 bits have been transformed
          btfss STATUS, Z           ; until z = 1
        Call Bitout ; Send  carry  bit then clear carry bit btfss
        STATUS, Z goto TX 1 Call Bit In movlw 00000001 B
        End wf RXBUFF, W    ; z = 1  if ACK   z = 0 if NOACK return

; The Rx subroutine receives a byte from I2C bus into W,
; using  RXBUFF buffer
; Call Rx with bit 7 of TXBUFF clear for ACK
; Call Rx with bit 7 of TXBUFF set for NOACK

Rx:    movlw  00000001  B
movwf RXBUFF
```

```
Rx_1:
        rlf RXBUFF, F Call
        Bit In btfss
        STATUS, C goto Rx
        1 rlf TXBUFF, F
        Call BitOut movf
        RXBUFF, W return
; The BitOut subroutine transmits, hthen clears, the carry bit BitOut:
        bcf INDF, SDA          ; copy carrybit to SDA btfsc
        STATUS, c bcf INDF, SDA
          bsf INDF, SCL        ; pulse clockline
        delay 0,1,2 ; t: HIGH bcf INDF, SCL
        bcf STATUS, c return
; The bit In subroutine receives one bit into
; bit 0 of RXBUFF

BitIn:
        bsf INDF, SDA
        bsf INDF, SCL ; Drive clock line high bcf RXBUFF, 0 ; copy
        SDA to bit 0 of RXBUFF btfsc PORTC, SDA bsf RXBUFF, 0
        bcf INDF, SCL ; Drive clock low again return
```



Examples of I2C bus Interfacing

I. DAC Interfacing

Two digital-to-analog convste outputs are easily added to a PIC with MAX518 eight-pin DIP. Each output channel produces an output voltage that ranges from 0v to 256 ~~255~~ V_{DD} where V_{DD} is the power supply to the DAC chip. If V_{DD} = 5V , an output of 2.5V will appear on the OUT0 pin if the following three bytes are sent to the chip.

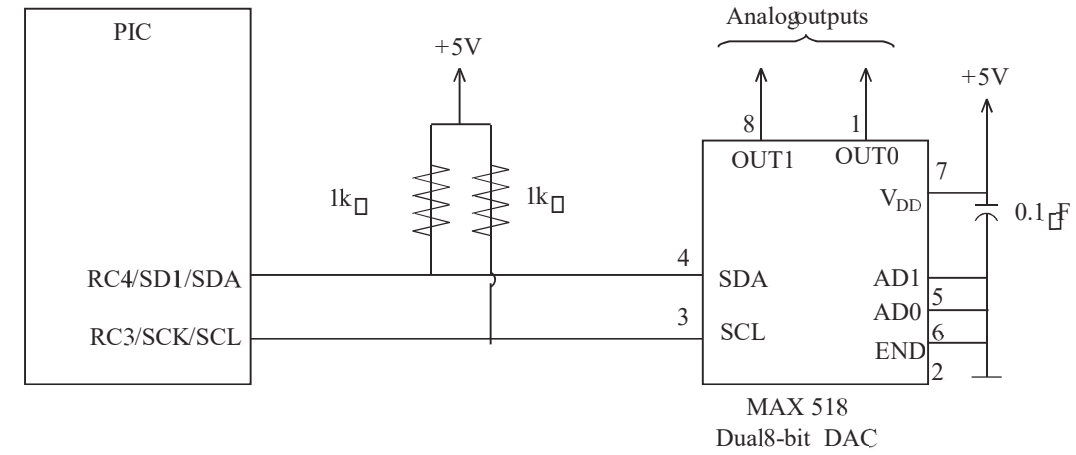
’01011000’, ’00000000’ ’10000000’

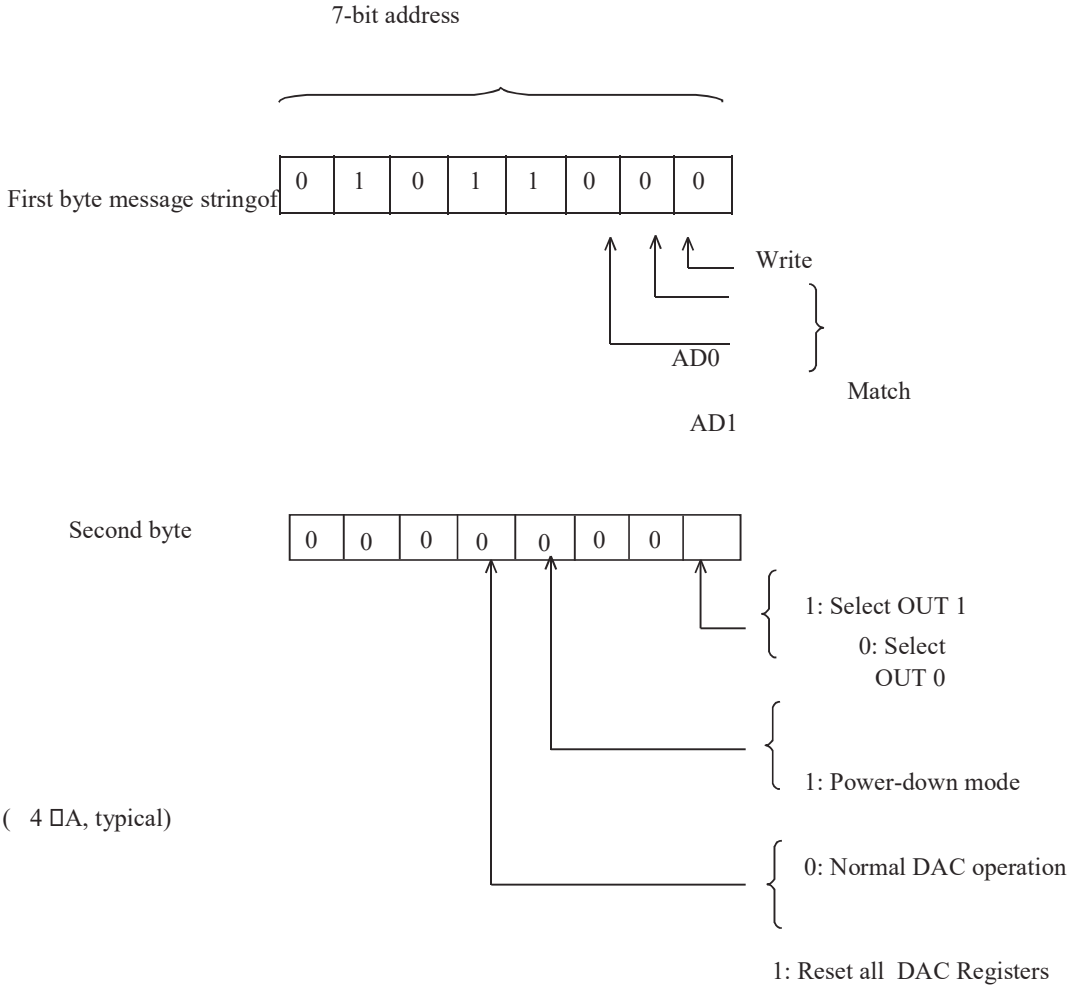
An output of 2.5 V will appear on the OUT1 pin by sending the following three bytes

’01011000’ ’00000001’ ’100000000’

The MAX518 chip includes a power-on reset circuit that drives the two outputs to 0V initially. The two address inputs, AD1 and Ad0, provide an adjustable part of the chip’s I^2C address. With 5 bits fixed at 01011 and two adjustable bits, it is possible to connect four MAC518 chips to a PIC.

DAC Interfacing on I^2C bus





Third byte B

Analog output voltage = V_{DD} ___

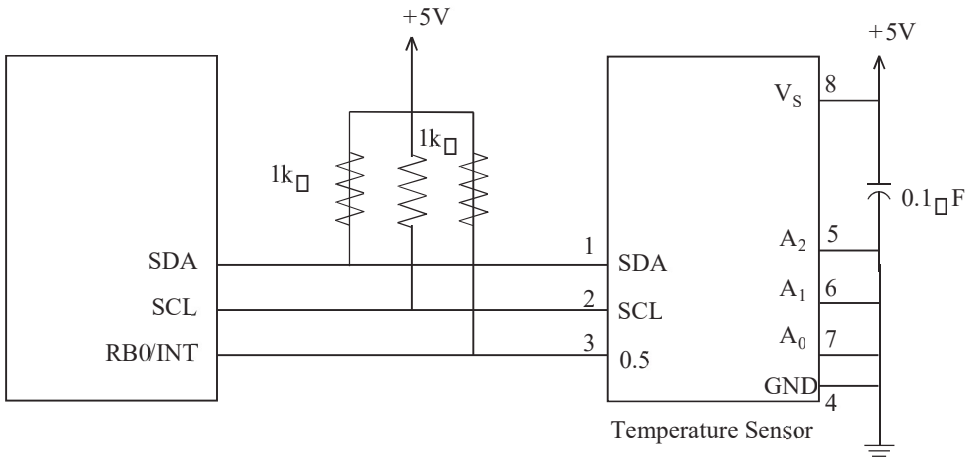
256

II. Interfacing a Temperature Sensor

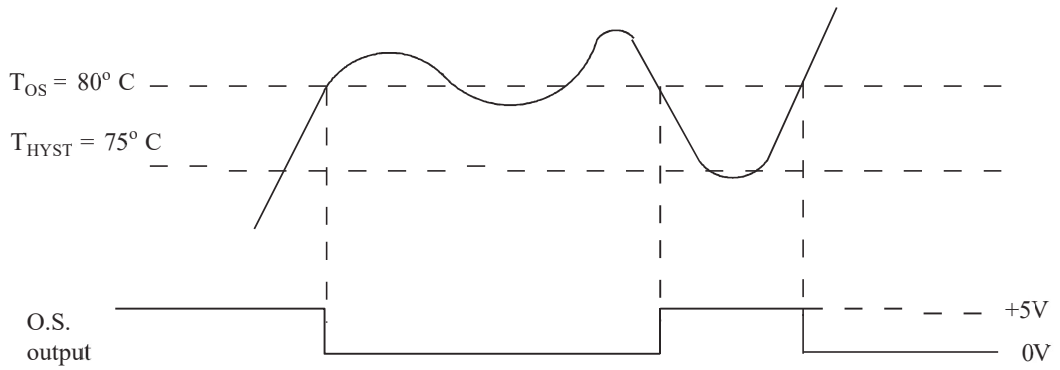
National Semiconductor's LM 75 chip combines an analog temperature transducer, an analog-to- digital convertor (9-bit), and an I^2C bus interface, all in a tiny S)-8 surface mount package. The temperature range covered is -25^o C to +100^o C with $\pm$2^o C accuracy. The two's complement form of the temperature is available from the 9-bit ADC. The resolution of the ADC is about 0.5^o C.

Temperature	Digital Output	
	Binary	Decimal
125^o C	01111 1010	250
25^o C	00011 0010	50
0.5^o C	000000001	1
0$_o$ C	00000 0000	0
-0.5^o C	11111 1111	
-25^o C	11100 1110	
-55^o C	11001 0010	

LM 75 chip also includes a thermal watch dog that can be setup to interrupt PIC on its RBO/INT edge-triggered interrupt input when the temperature rises above a programmable, T_{OS} . It also in- cludes programmable hysteresis so that the temperature must dip down below the setpoints T_{OS} threshold to a lower $T_{HY\ ST}$ threshold before rising againpast the T_{OS} setpoint to generate another output edge.



LM 75

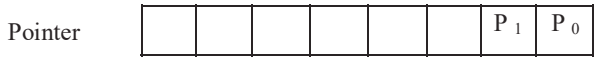


O.S. stands for over temperature shutdown

O.S. stands for over temperature shutdown.

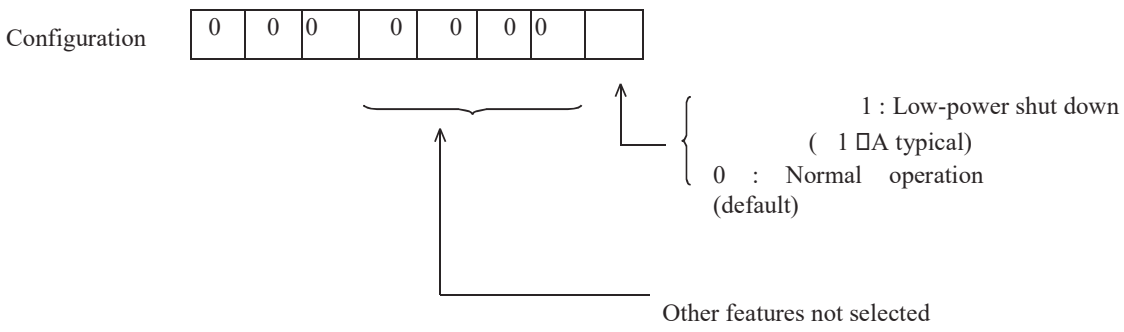
Register Structure

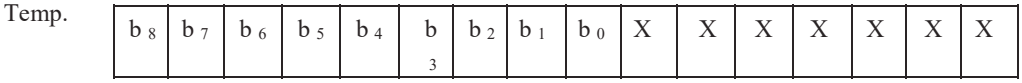
When a ”write” message string is sent, the first byte selects the chip for a write and the second byte loads the pointer register. The write message string can stop there or it can continue with a 2-byte write of T_{OS} (Over tem shutdown). Once the pointer has been set, any of their register can be read, reading two bytes for temperature, T_{OS}, or T_{HYST} or reading just 1 byte for the configuration register.

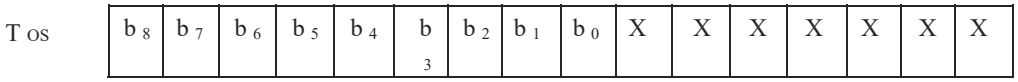


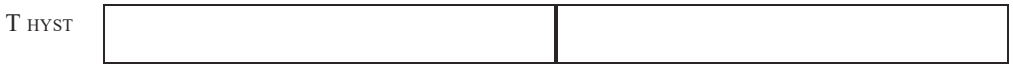
0	0	Temperature (read only) default
0	1	Configuration (read / write)
1	0	T_{HYST} (read / write)

1 1 T_{OS} (read / write)









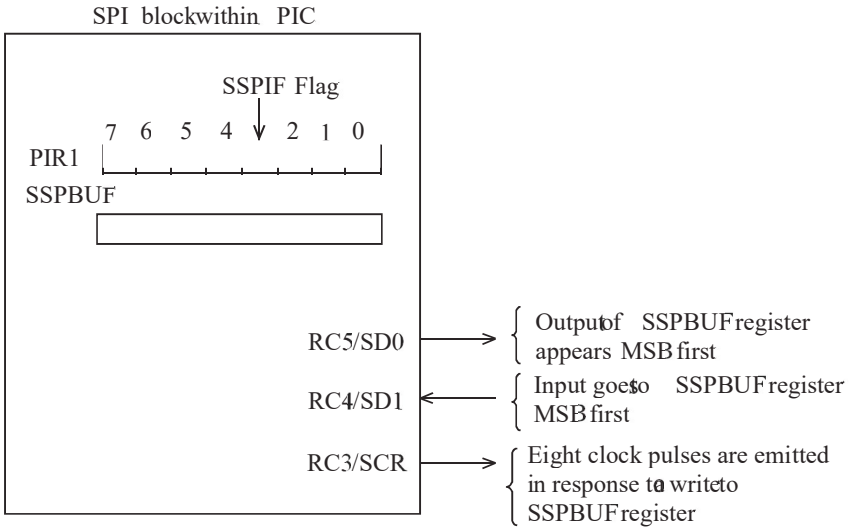
Synchronous Serial Port Module

Mid range PIC microcontroller includes a Synchronous Serial Port (SSP) module, which can be con- figured into either of two modes:

- Serial Peripheral Interface (SPI)
- Inter-Integrated Circuit (I^2C)

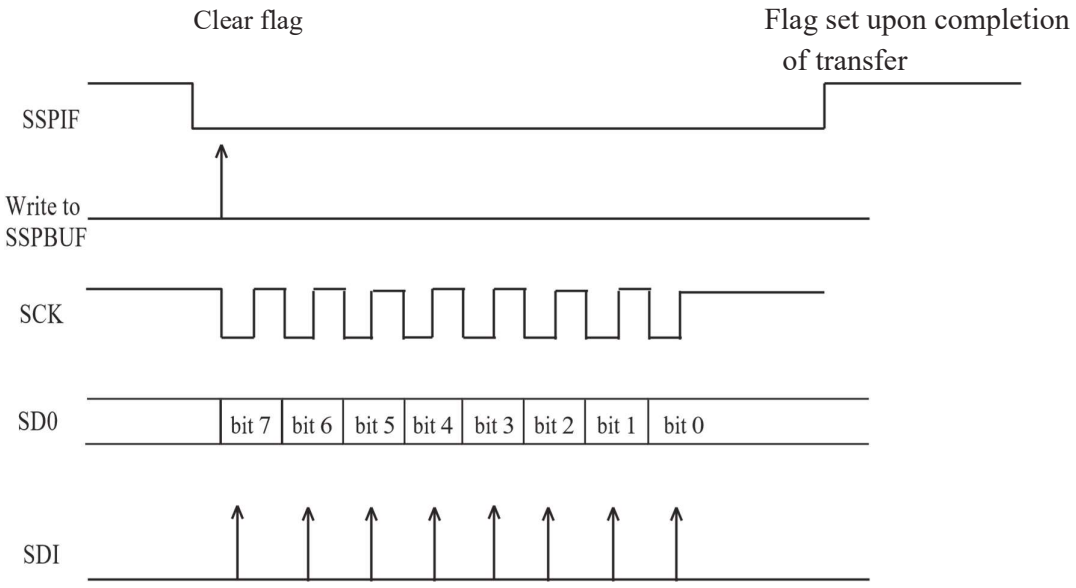
Either of these modes can be used to interconnect two oe more PIC chips to each other using a minimal number of wires for interconnection. Alternatively, either can be used to connect a PIC chip to a peripheral chip. In this case of the I^2 C mode, the peripheral chip must also include an I^2 C interface. In contrast, the SPI mode provides the clock and serial data lines for direct connection to shift registers, adding an arbitrary number of I/O pins to a PIC chips.

Serial Peripheral Interface



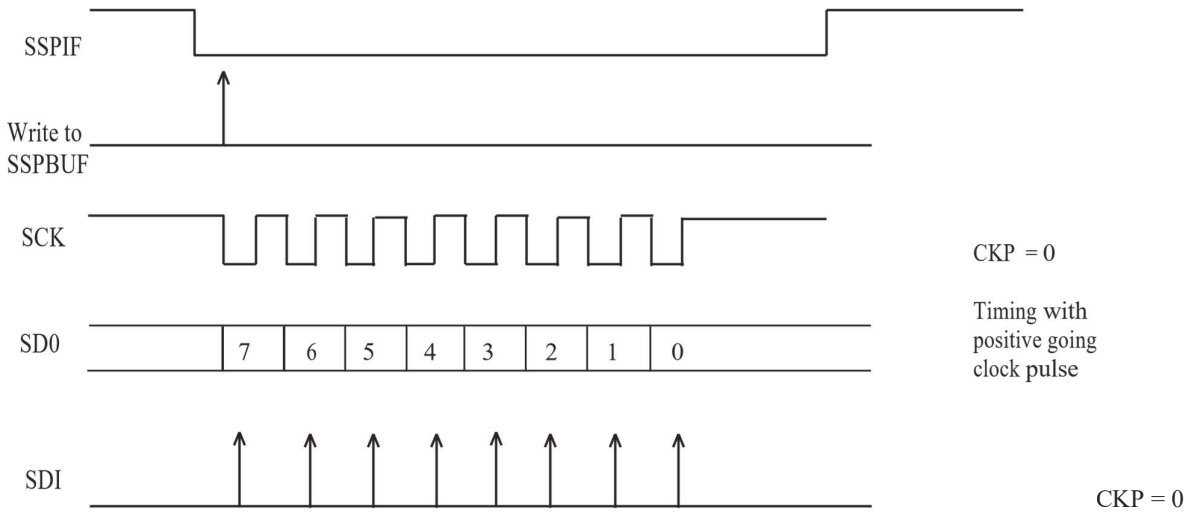
Portc three pins RC5, RC4 and RC3 are used for Synchronous Serial Interface. These pins revert to their normal general purpose I/O pins if neither of the two SSP modes is selected. The SPI port requires the RC3/SCK pin to be an output that generates the clock signal used by the external shift registers. This output line characterizes the SPI's master mode. In slave mode, RC3/SCK works as the input for the clock.

When a byte of data is written to SSPBUF register, it is shifted out the SDO pip in synchronism with the emitted pulses on the SCK pin. The MSB of SSPBUF is the first bit to appear on SDO pin. Simultaneously, the same write to SSPBUF also initiates the 8-bit data reception into SSPBUF of whatever appears on SDI pin at the time of rising edges of the clock on SCK pin.



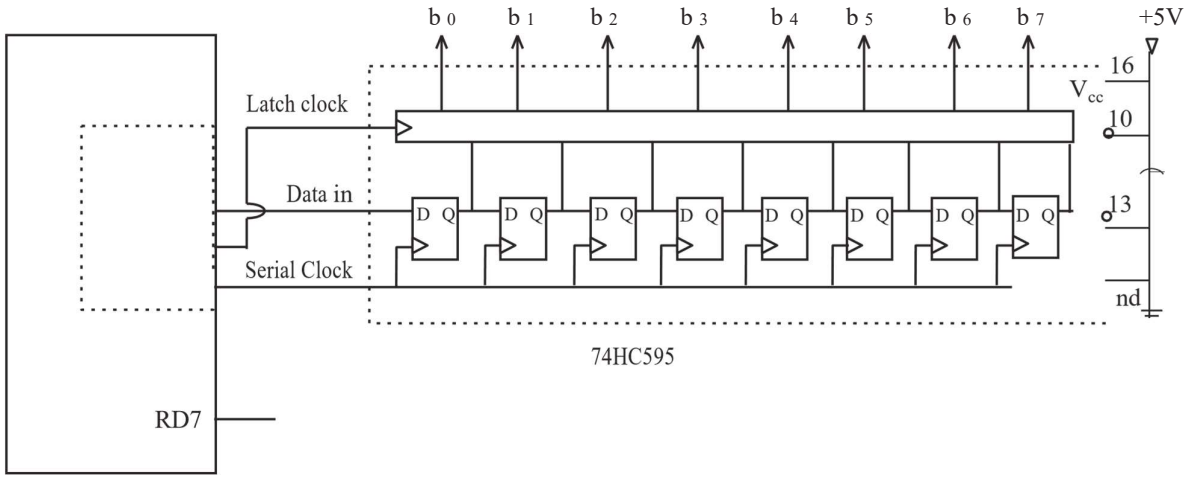
The SDI pin is read at these times
CKP = 1

Timing with negative going pulses.

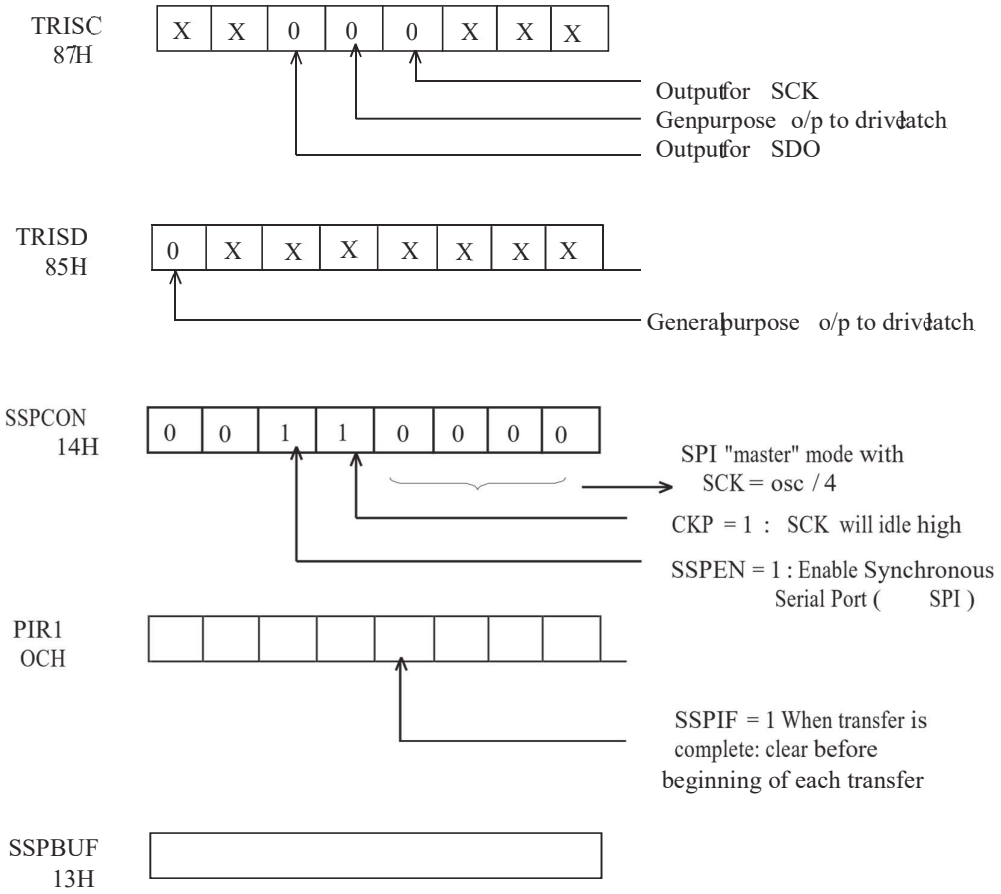


A read or write of one of the PIC's ports, such as PORTD takes one internal clock cycle to execute. In contrast, a read or write of an expansion port that is implemented with an SPI-connected octal shift register is slowed down by an order of magnitude by the eight clock pulses as seen before. If the SSPIF flag in the PIR1 register is cleared before the SPI transmission is initiated, then it will be automatically set at the completion of the transfer setting of SSSPIF flag indicated that the transferred data is in place and ready to be used.

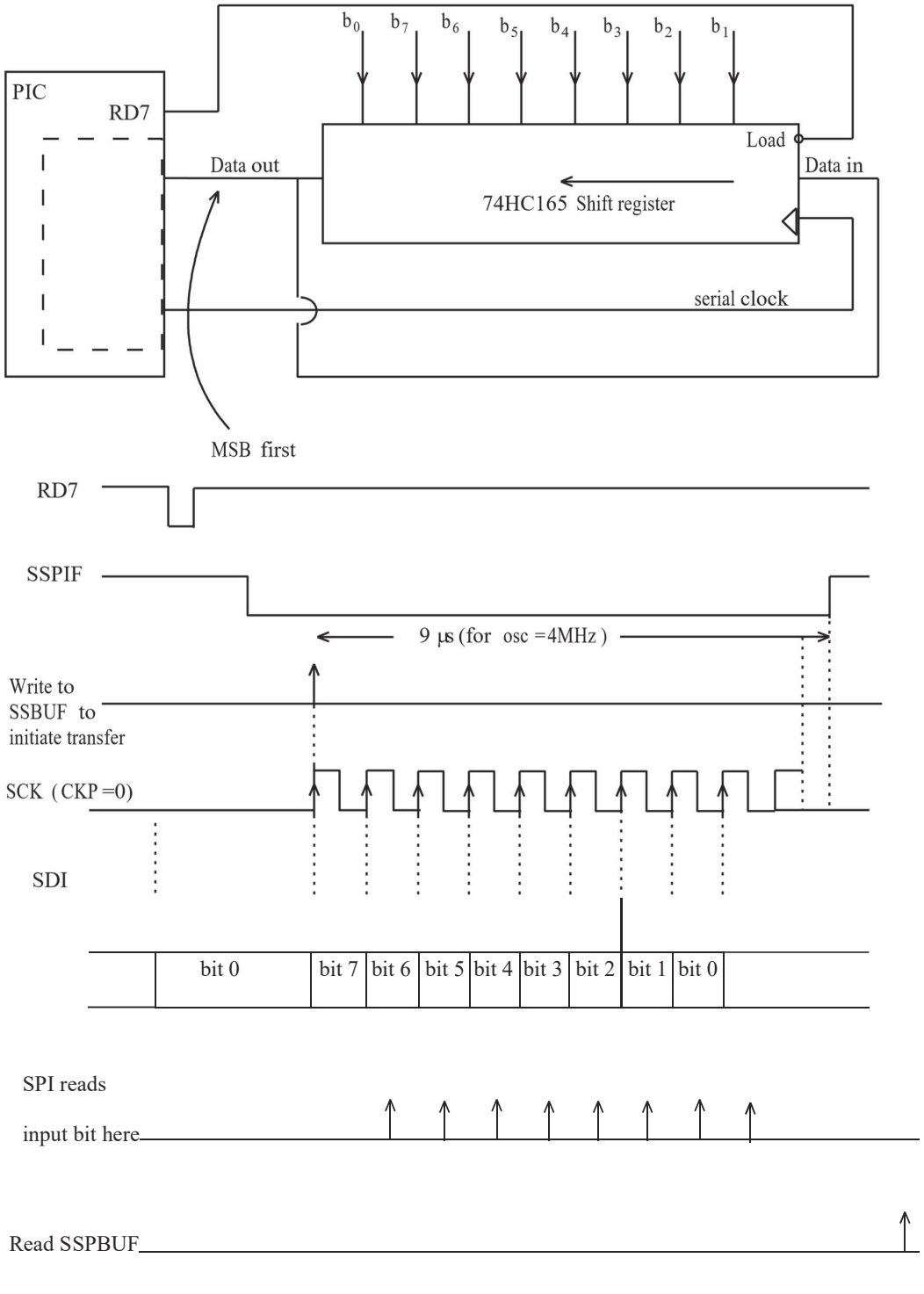
Output port expansion



Port configurations

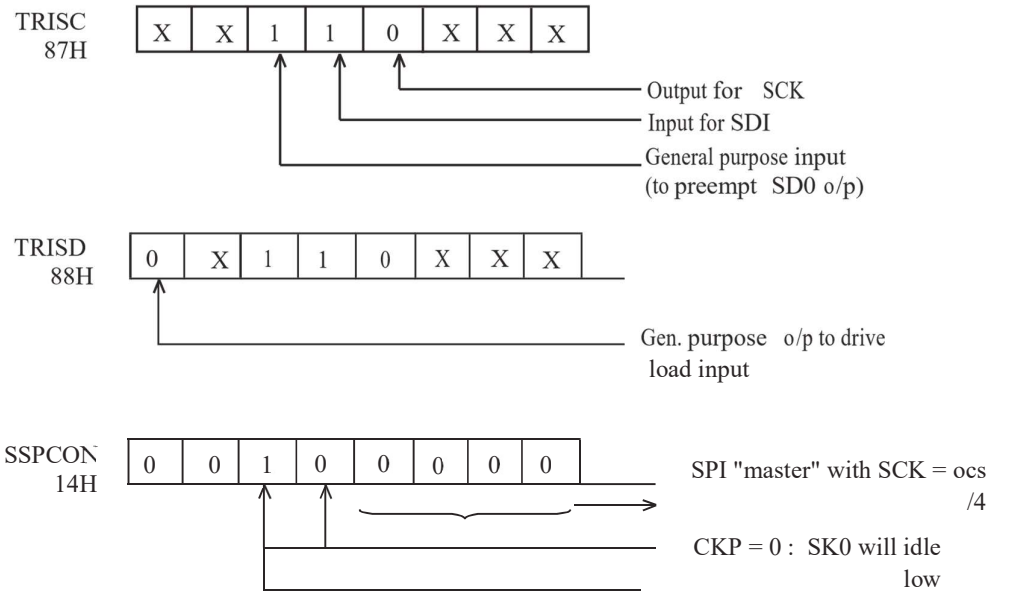


Input port expansion



Timing diagram

Port configurations

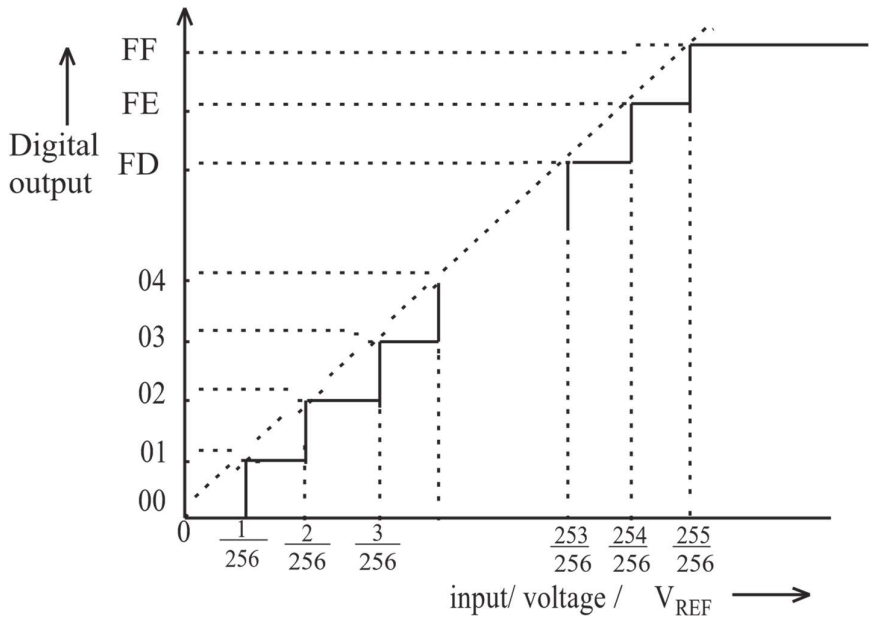


SSPENSynchronous Serial Port = 1 : enable (SPI)

Analog-to-Digital Converter

Fe atures (16C7X)

- Eight input channels
- An analog multiplexer
- A track and hold circuit for signal on the selected input channel
- Alternative clock sources for carrying out the conversion.
- An adjustable autonomous sampling rate.
- The choice of an internal or external ref. voltage.
- 8-bit conversion
- Interrupt response when conversion is completed.



Port A and Port E pins are used for analog inputs/ Reference voltage for ADC.

Port A pins

RA0/AN0 - Can be used as analog input-0 RA1/AN1 - Can be used as analog input-1 RA2/AN2 - Can be used as analog input-2

RA3/AN3/V_{REF} - RA3 can be used as analog input 3 or analog reference voltage RA4/TOCKI - RA4 can be used as clock input to Timer-0

RA5/$\overline{SS}$/AN4 - RA5 can be used as analog input 4 or the slave select for the sync serial port

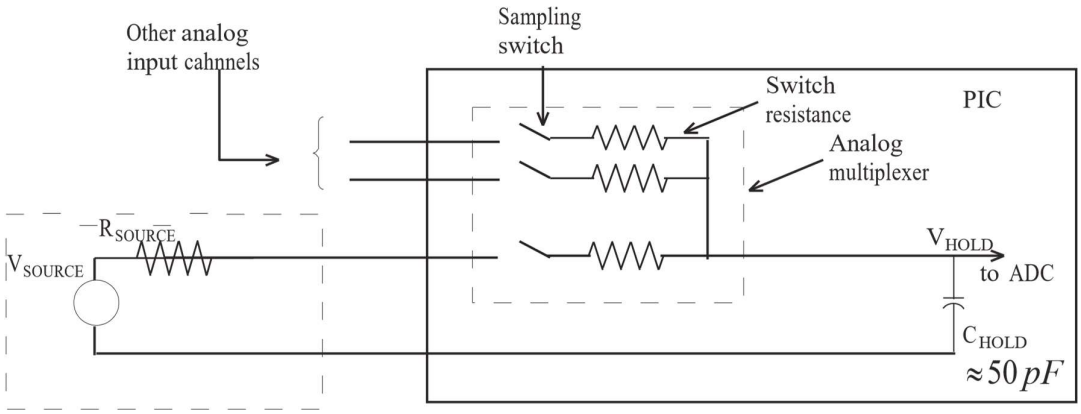
Port E pins

RE0/$\overline{RD}$/AN5 - Can be used as analog input 5

RE1/$\overline{WR}$/AN6 - Can be used as analog input 6

RE2/$\overline{CS}$/AN7 - Can be used as analog input 7

PIC microcontroller has internal sample and hold circuit. The input signal should be stable across the capacitor before the conversion is initiated



After waiting out the sampling time, a conversion can be initiated. The ADC circuit will open the sampling switch and carry out the conversion of the input voltage as it was at the moment of opening of the switch. Upon completion of the conversion, the sampling switch is closed and V_{HOLD} again tracks V_{SOURCE}.

Using the A/D Converter

Registers ADCON1, TRISA, and TRISE must be initialized to select the reference voltage and the input channels. The first step selects the ADC clock source from among four choices (OSC/2, OSC/8, OSC/32, and RC). The constraint for selecting clock frequency is that the ADC clock period must be 1.6 *mu*s or greater.

The A/D modules has three registers. These registers are

- A/D Result Register (ADRES)
- A/D Control Register 0 (ADCONO)
- A/D Control Register 1 (ADCON1)

The ADCONO register as shown here, controls the operation of A/D module.

7	6	5	4	3	2	1	0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	_	AD ON

bit 7 - 6

ADCS1 : ADCS 0

00 = $F_{osc}/2$ 01 =

$F_{osc}/8$

10 = $F_{osc}/32$

11 = F_{RC} (clock derived from an internal RC oscillator) bit 5 - 3

———CHS2: CHS0

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 bit 2 A/D Conversion

Status bit

GO/$\overline{DONE}$ if
ADON = 1
1 = A/D conversion is in progress (setting this bit starts the A/D conversion)
0 = A/D conversion is not in progress (this bit is automatically cleared by hardware when A/D conversion is complete.)

bit 1

Unimplemented

bit 0 ADON: A/D on bit
1 = A/D converter module is ON
0 = A/D converter module is OFF (not operating.)

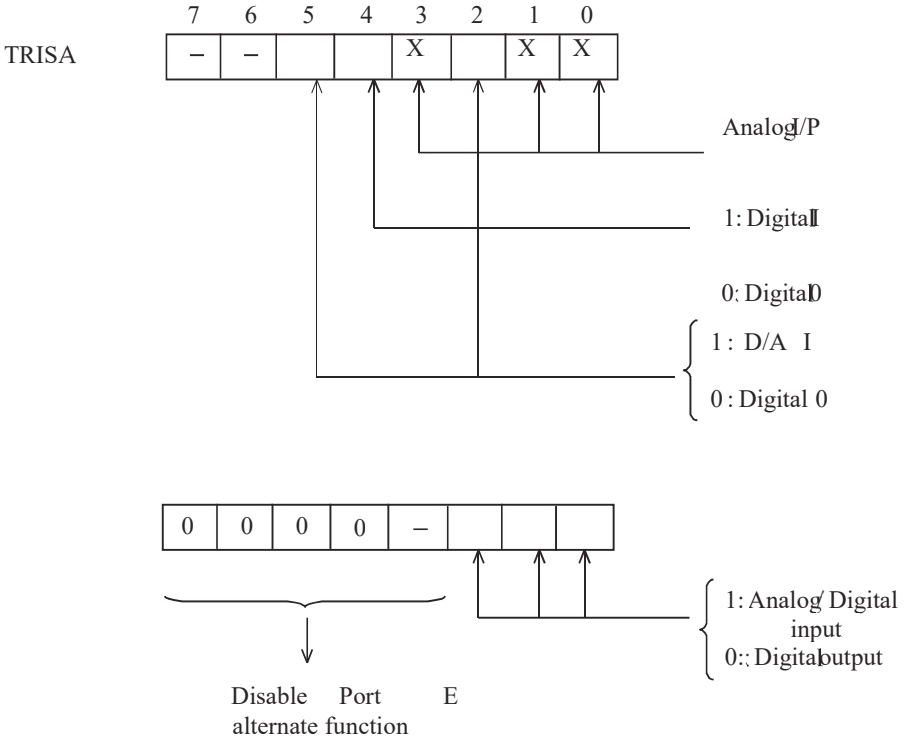
ADCON1 Register

bit 7							bit 0
–	–	–	–	–	PCFG2	PCFG1	PCFG0

PCFG2 : PCFG0 : A/D Port Configuration Control bits

PCFG2 : PCFG0 : A/D Port Configuration Control bits

PCFG2 : PCFG0	RA0	RA1	RA2	RA5	RA3	RE0	RE1	RE2	V_{REF}
000	A	A	A	A	A	A	A	A	V_{DD}
001	A	A	A	A	V_{REF}	A	A	A	RA3
010	A	A	A	A	A	D	D	D	V_{DD}
011	A	A	A	A	V_{REF}	D	D	D	RA3
100	A	A	D	D	A	D	D	D	V_{DD}
101	A	A	D	D	V_{REF}	D	D	D	RA3
11X	D	D	D	D	D	D	D	D	-



1. Configure A/D module
    - Configure analog pins/ voltage reference/ and digital I/O (ADCON1)
    - Select A/D channel (ADCON0)
    - Select A/D conversion clock (ADCONO)
    - Turn on A/D module (ADCONO)
2. Configure A/D interrupt (if required)
    - Clear AD—F bit in PIR 1 reg
    - Set AD—E bit in PIE 1 reg
    - Set G—E bit
3. Wait for required acquisitiion time
4. Start conversion
    - Set GO/~~*DON E*~~
5. Wait for A/D conversion to complete by either
    - polling dor GO/~~*DON E*~~ bit to be cleared
    - waiting for the A/D interrupt
6. Read A/D result register (ADRES) Clear AD—F if required.

Example Program

A/D Conversion with Interrupt

```
bsf STATUS, RPO ; Select Bank1 clrf ADCON 1 ;
Configure A/D input bsf PIE1, ADIE ; Enable A/D
interrupt
bcf STATUS, RPO ; Select Bank 0
movlw 0811+ ; Select fosc/32, channel 0, A/D on movwf ADCONO bcf
PIR1, ADIF bsf INTCON, PEIE
bsf INTCON, GIE
; Ensure that the required sampling time for the ;
selected input channel has elapsed.
; Then the conversion may be started
bsf ADCONO, GO ; start A/D conversion
; AD| F bit will be set
; and GO/DON E bit is cleared
; upon completion of A/D conversion
```

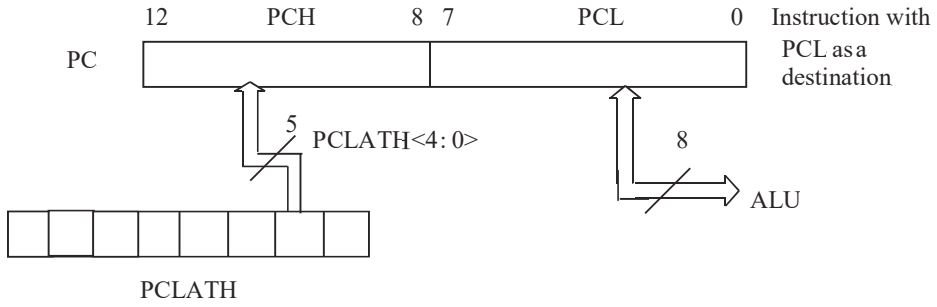
Code structure for large Programs

Memory paging is essential if the code exceeds 2k of program memory (2048). PIC 16C74A supports 4096 addresses and hence it is important to consider memory paging for this processor.

PCL and PCLATH

The program counter (PC) is 13-bit wide. The low byte comes from the PCL register, which is a readable and writable register. The upper bits (PC ¡12.8¿) are not readable, but are indirectly writable through the PCLATH register. On any reset , the upper bits of the PC will be cleared. PCL← 0 and PCLATH ← 0. Two situations for loading the PC following any reset are given here.

1. Any write to PCL register load the content of PCL to lower 8 bit of PC and content of PCLATH to higher 5 bits. mov wf PCL



2. PC is also loaded during a call or goto instruction

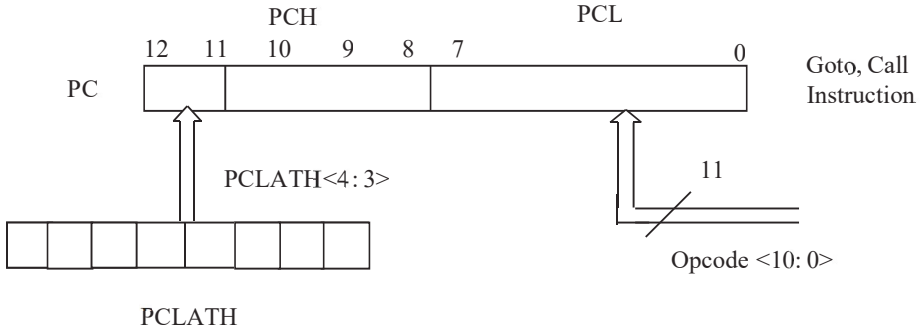
$$0 \leq k \leq 2047$$

Operation:

$$k \rightarrow PC\ <10:0>$$

$$PCLATH\ <4:3> \rightarrow PC\ <12:11>$$

Goto is an unconditional branch. The eleven bit immediate value is loaded into PC bits $<10:0>$. The upper bits of PC are loaded from PCLATH $<4:3>$.



STACK

The PIC16CXX family has an 8 level deep X 13-bit wide hardware stack. The stack space is not part of either program or data memory and the stack pointer is not readable or writable. The PC is pushed onto the stack when a CALL instruction is executed or an interrupt causes a branch. The stack os POPed in the event of a RETURN, RETLW or a RETIE instruction execution. PCLATH is not affected by a PUSH or a POP operation. The stack operates on a circular buffer.

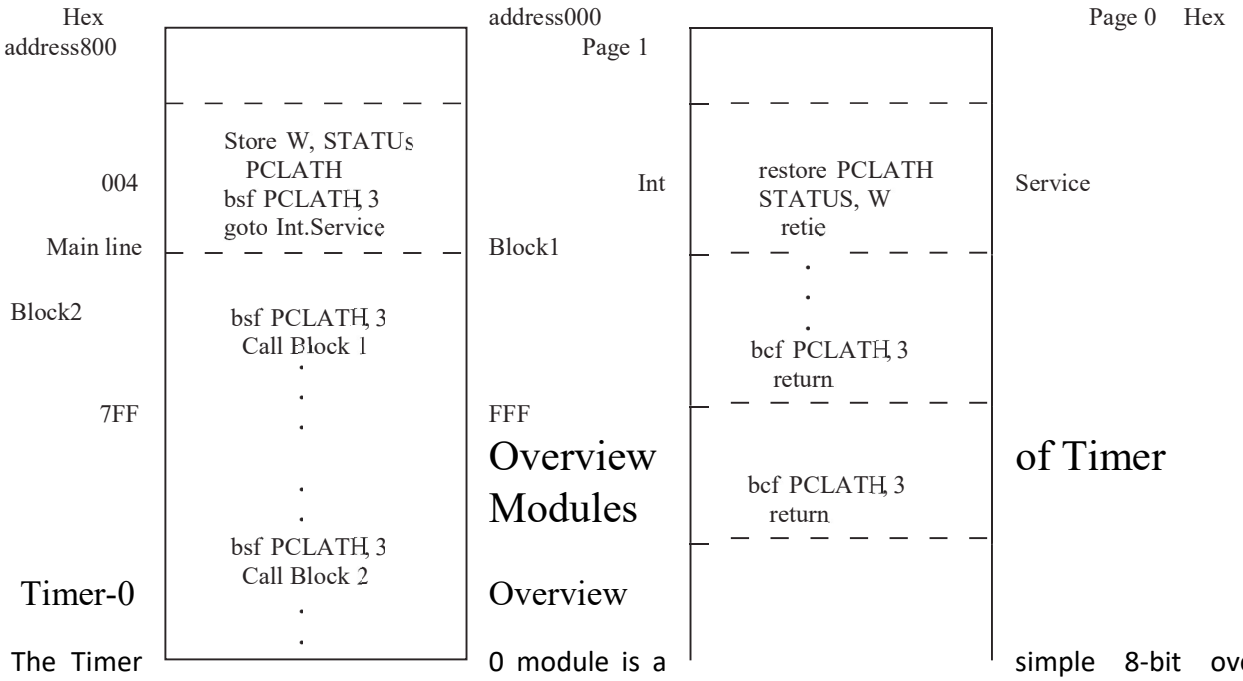
Paging:

Following any reset PCL and PCLATH are cleared to 0. For a 4k .... program memory, the address range is from 0000H to 0FFFH. Hence each call and goto instruction will actually reach the desired address only if bit 3 of PCLATH is set or cleared correctly. However even for 4k PIC controllers, there is no need to take care of PCLATH bit 3, if the code size fits into 2k address space. Bit 3 of PCLATH will come out of reset in the zero state and there will never be a need to change it. Consequently, every call and goto instruction will go to the correct place.

For large programs, it is helpful to break out blocks of code that are reached by a single call instruction and that terminates in a single return instruction. Such a block of code can be placed on program memory's page 1. Then, before executing the call instruction to reach the block, the following in- struction can be executed. bsf PCLATH, 3 ; Switch to program memory's Page 1.

When it is finally time to exit from the block to return to the mainline program in Page 0, the return instruction is preceded by the instruction bcf PCLATH, 3

Program memory allocation for large programs



Overview of Timer Modules

Timer-0 Overview

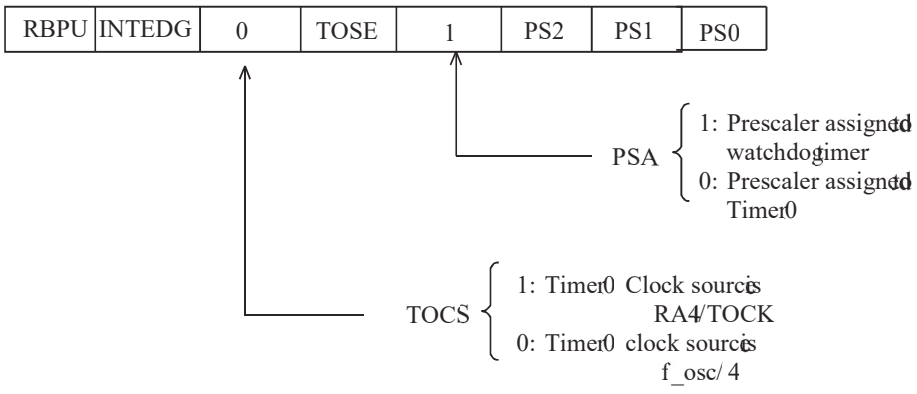
The Timer 0 module is a simple 8-bit overflow counter. The clock source can be either the internal clock ($f_{osc}/4$) or an external clock. When the clock source is an external clock, the Timer-0 module can be selected to increment on either the rising or falling edge.

Timer-0 module also has a programmable prescalar option. This prescalar can be assigned either to Timer 0 or the watchdog Timer.

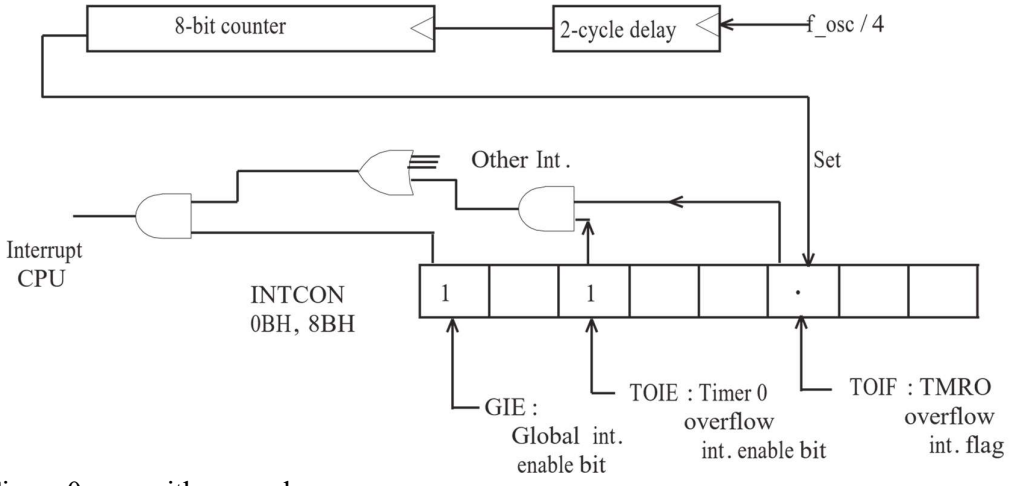
The counter sets a flag TOIF when it overflows and can cause an interrupt at that time if that interrupt source has been enabled (TOIF=1). Timer 0 can be assigned an 8-bit prescalar that can divide the input by 2,4,8,16,...,256. Writing to TMRO resets the prescalar assigned to it. Timer-0, or its prescalar can be connected to either of two input sources.

1. $f_{osc}/4$

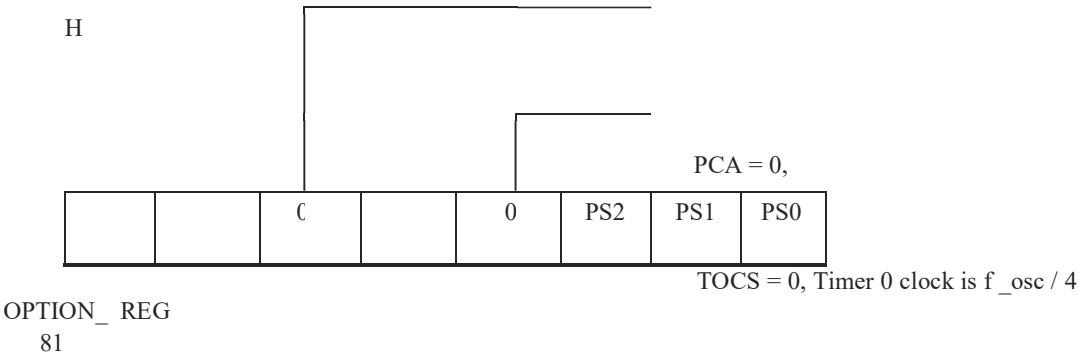
2. RA4/ TOCKI, the input connected to bit 4 of PORTA.



TMRO 01H



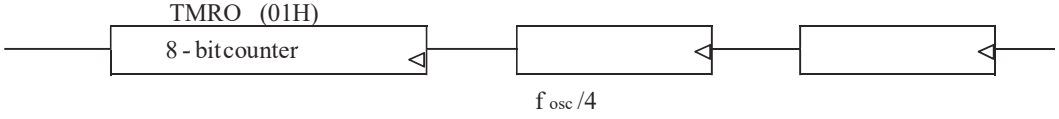
Timer-0 use with prescalar



Prescaler assigned to Timer0

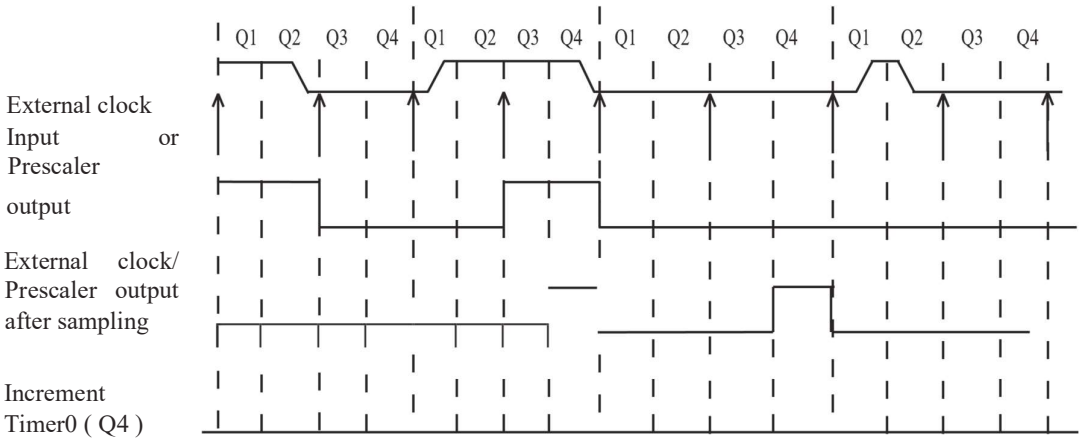
Prescaler .....

0 0 0 2
0 0 1 4
0 1 0 8
0 1 1 16
1 0 0 32
1 0 1 64
1 1 0 128
1 1 1 256



Overflow

External clock synchronization



Timer 0

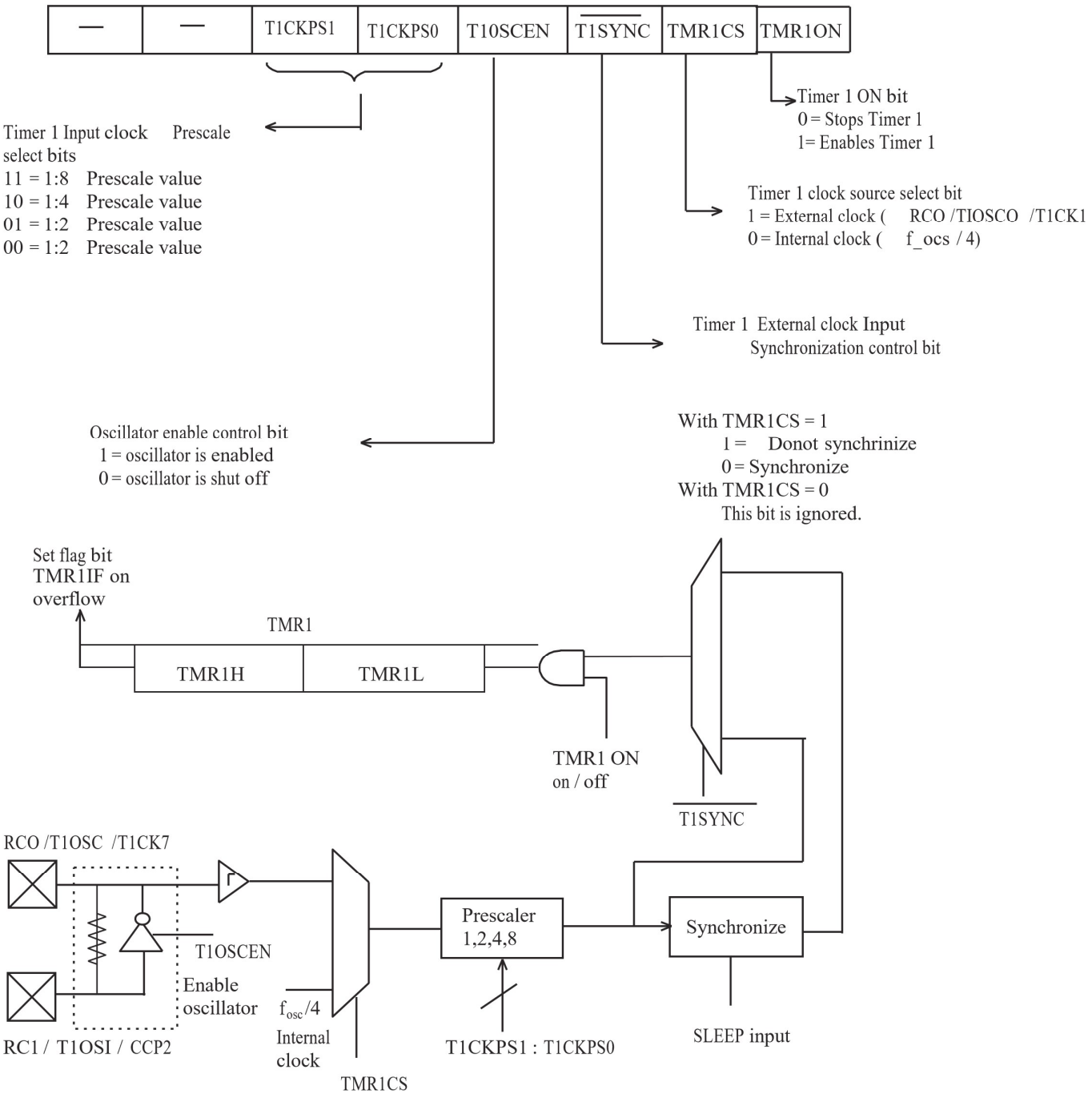
Timer-1 Module

The Timer1 module is a 16-bit timer/counter consisting of two 8-bit registers (TMR1H and TMR1L) which are readable and writable. The TMR1 register pair (TMR1H: TMR1L) increments from 0000H to FFFFH and rolls over to 0000H. The TMR1 interrupt, if enabled, is generated on overflow which sets the interrupt flag bit TMR1IF-(PIR< 0 >). This interrupt can be enabled/disabled by set- ting/clearing TMR1 interrupt enable bit TMR1IE-(PIE < 0 >)

The operating and control modes of Timer 1 is determined by the special purpose register T1CON.

T1CON (10H)

bit 7 0 bit 0


—
—
T1CKPS1
T1CKPS0
T10SCEN
T1SYNC
TMR1CS
TMR1ON
Timer 1 ON bit
0 = Stops Timer 1
1= Enables Timer 1
Timer 1 Input clock Prescale select bits
11 = 1:8 Prescale value
10 = 1:4 Prescale value
01 = 1:2 Prescale value
00 = 1:2 Prescale value
Timer 1 clock source select bit
1 = External clock (RCO /TIOSCO /T1CK1
0 = Internal clock (f_ocs / 4)
Timer 1 External clock Input Synchronization control bit
Oscillator enable control bit
1 = oscillator is enabled
0 = oscillator is shut off
With TMR1CS = 1
1 = Donot synchrinize
0 = Synchronize
With TMR1CS = 0
This bit is ignored.
Set flag bit TMR1IF on overflow
TMR1
TMR1H
TMR1L
TMR1 ON on / off
T1SYNC
RCO /T1OSC /T1CK7
T1OSCEN
Enable oscillator
RC1 / T1OSI / CCP2
fosc/4
Internal clock
TMR1CS
Prescaler 1,2,4,8
T1CKPS1 : T1CKPS0
Synchronize
SLEEP input

Timer 1 can operate in one of the two modes.

- As a timer. (TMR1CS = 0)
  In timer mode, Timer 1 increments in every instruction cycle. The Timer 1 clock source is $f_{osc}/4$. Since the internal clock is selected, the timer is always synchronized and there is no further need of synchronization.
- As a counter (TMR1CS = 1)
  In counter mode, external clock input from the pin RCO/T1OSC/T1CKI is selected.

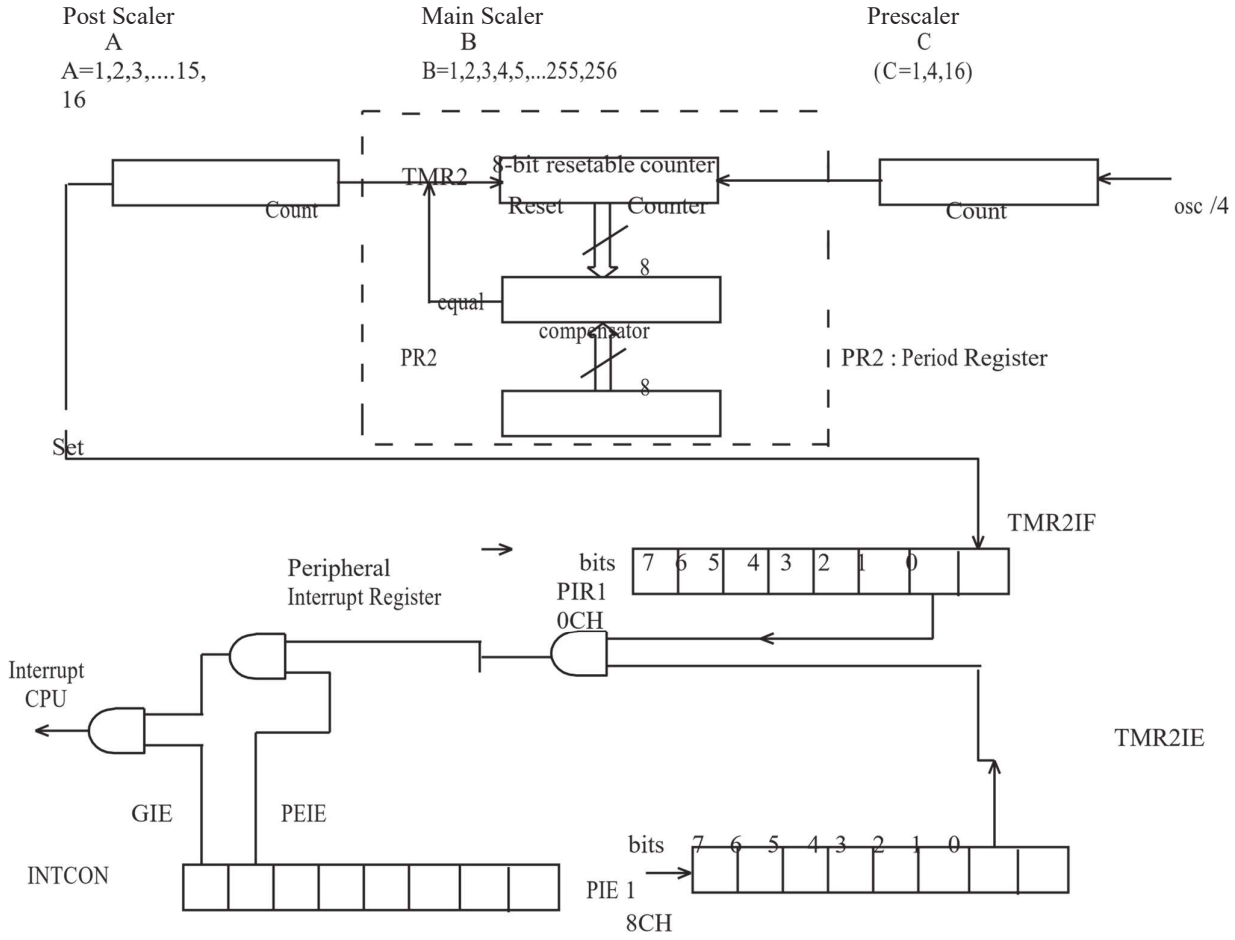
Use of Timer-2

Timer 0: 8-bit timer/counter with 8-bit prescalar

Timer 1: 16-bit timer/counter with prescalar, can be incremented during sleep via external crystal/clock.

Timer 2: 8-bit timer/counter with 8-bit period register, prescalar, post scalar.

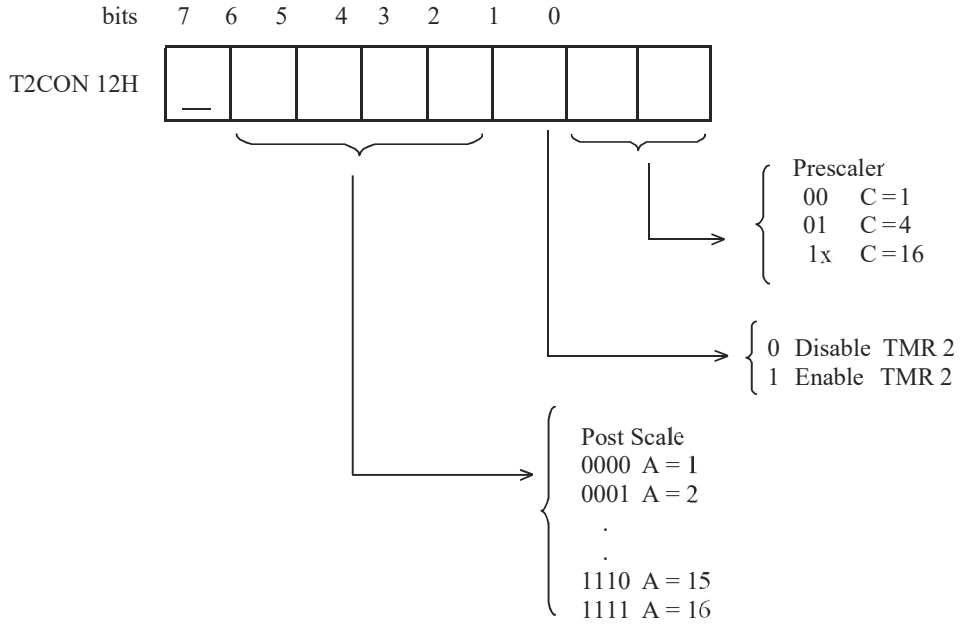
Timer 2 Circuitry



Timer 2 is an 8-bit timer with a prescalar and a port sclar. It can be used on the PWM mode of CCP modules. The TMR2 register is readable and writable and is cleared on any device reset. The input clock ($f_{osc}/4$) has a prescalar option of 1:1, 1:4 or 1:16 selected by bits 0 and 1 of T2CON register.

The timer 2 module has a 8-bit period register (PR2). timer 2 increments from 00H until it matches PR2 and then resets to 00H on the next increment cycle. PR2 is a readable and a writable register. PR2 is initialized to FFH on reset.

The output of TMR2 goes through a 4-bit post scalar (1:1, 1:2 to 1:16) to generate a TMR2 interrupt by setting TMR2IF flag.



CCP overview

The CCP module(s) can operate in one of the three modes: 16-bit capture, 16-bit compare, or upto 1-bit Pulse Width Modulation (PWM).

Capture mode captures the 16-bit value of TMR1 into CCPRxH: CCPRxL register pair. The capture event can be programmed for either the falling edge, rising edge, fourth rising edge, or the sixteenth rising edge of the CCPx pair.

Compare mode compares the TMR1H: TMR1L register pair to the CCPRxH: CCPRxL register pair. When a match occurs an interrupt can be generated, and the output pin CCPx can be forced to given state (High or Low), TMR1 can be reset (CCP1) or TMR1 reset and start A/D conversion (CCP2).

This depends on the control bits < CCPxM3 : CCPxM0>

PWM mode compares the TMR2 register to a 10 bit duty cycle register (CCPRxH : CCPRxL<5:4>) as well as an 8-bit period register (PR2). When the TMR2 register= Duty cycle register, the CCPx pin will be forcred low. When TMR2=PR2, TM2 is cleared to 00H, an interrupt can be generated, and the CCPx pin, if programmed in the O/P mode, will be forced high.

Compare Mode

Timer 1 is a 16-bit counter which can be used with CCP (Capture/compare/PWM) module to drive a pin high or low at precisely controlled time, independent of what the CPU is doing at that time.
The pins are Port-C RC1/CCP2 and RC2/CCP1 pins.
Which Timer1 includes a prescalar to divide the internal clock by 1,2,4 or 8, the choice of divide-by- one gives the finest resolution in setting the time of an output edge.

Capture/Compare/PWM modules

Each CCP (Capture/compare/PWM) module contains a 16-bit register which can operate as a 16bit capture register, as a 16-bit compare register or as a PWM master/slave duty cycle register. Both CCP1 and CCP2 are identical in operation, with the exception of the operation of the special event trigger.
The following shows the CCP mode timer resources.

CCP Mode	Timer Resource
Capture	Timer 1
Compare	Timer 1
PWM	Timer 2

CCP1 Module:

Capture/Compare/PWM Register 1 consists of two 8-bit register: CCPR1L (low byte) and CCPR2H (high byte). THe CCP1CON register controls the operation of CCP1. All are readable and writable.

CCP2 Module:

Capture/Compare/PWM Register 2 consists of two 8-bit registers: CCPR2L (low byte) and CCPR2H (high byte). The CCP2CON register controls the operation of CCP2. Al are readable and writable.

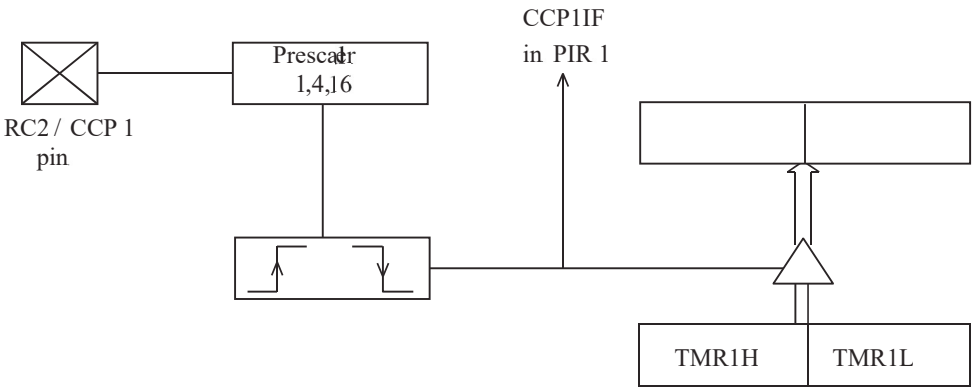
CCP1CON Register / CCP2CON Register

bit 7 bit 0

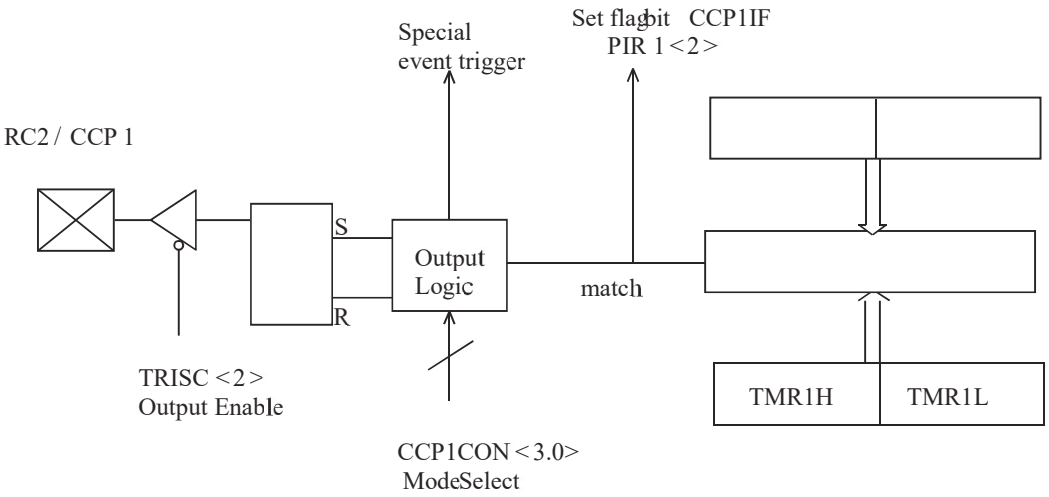
—	—	CCPxX	CCPxY	CCPxM3	CCPxM2	CCPxM1	CCPxM0

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. *Capture Mode*

Set flag bit



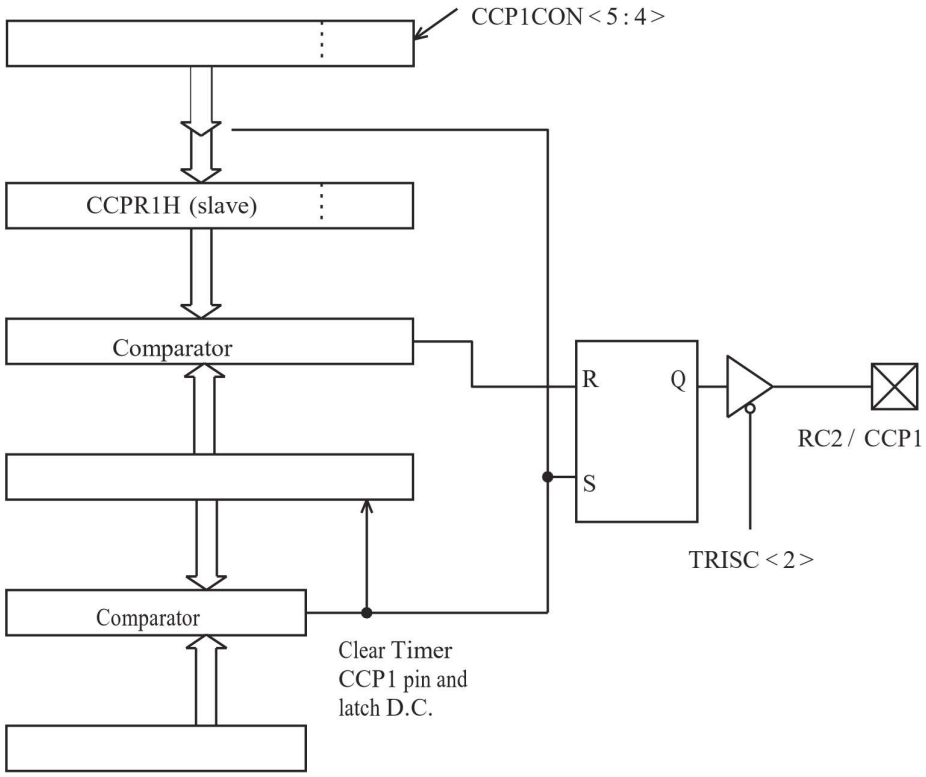
Compare Mode



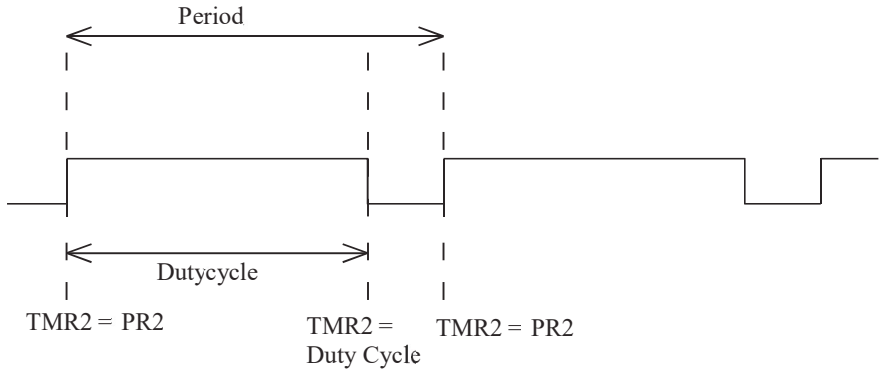
PWM Mode

In Pulse WIdth Modulation (PWM) mode, the CCPx pin produced upto a 10-bit resolution PWM output. Since CCP1 pin is multiplexed with PORT C data latch, the TRISC < 2 > pin must be cleared to make CCP1 pin an output.

Simplified PWM Block Diagram



PWM Output



A PWM output as ashown has a time period. The time for which the output stays high is called duty cycle.

PWM Period

The PWM period is specified by writing to PR2 register. The PWM period can be calculated using the following formula:

$$\text{PWM period} = [(PR2) + 1] \times 4 \times T_{osc} \times (\text{TMR2 prescale value})$$

$$\text{PWM frequency} = 1/\text{ PWM period}$$

When TMR2 is equal to PR2, the following events occur on the next increment cycle.

- TMR2 is cleared

- the CCP1 pin is set (if PWM duty cycle is 0
- The PWM duty cycle is latched from CCPR1L into CCPR1H

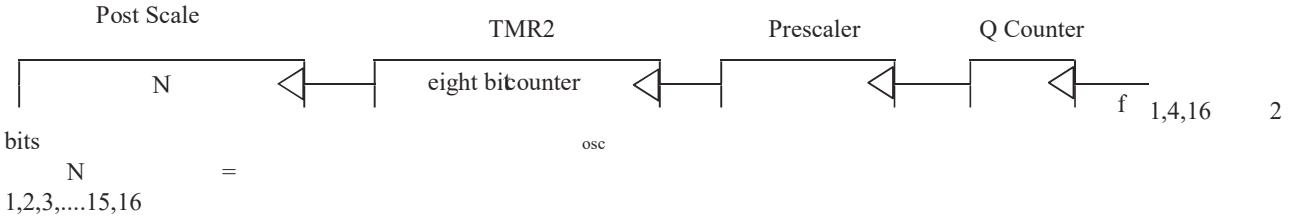
PWM duty cycle

The PWM duty cycle is specified by writing to the CCPR1L register and to CCP1CON < 5 : 4 > bits. Up to 10-bit resolution is available where CCPR1L contains the eight MSBs and CCP1CON < 5 : 4 > contains the two LSB's. The 10-bit value is represented by CCPR1L : CCP1CON < 5 : 4 >.

The PWM duty cycle is given by

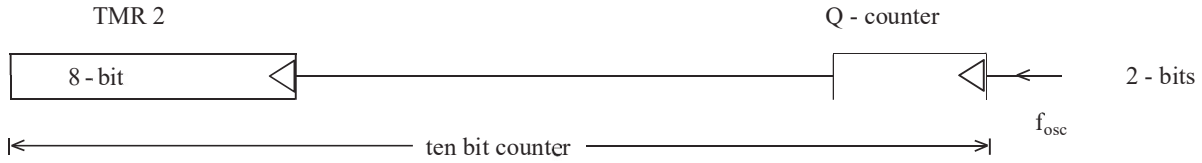
PWM duty cycle = (CCPR1L : CCP1CON < 5 : 4 >). T_{osc} . (TMR2 prescale value)

Although CCPR1L and CCP1CON < 5 : 4 > can be written to at anytime, the duty cycle value is not latched

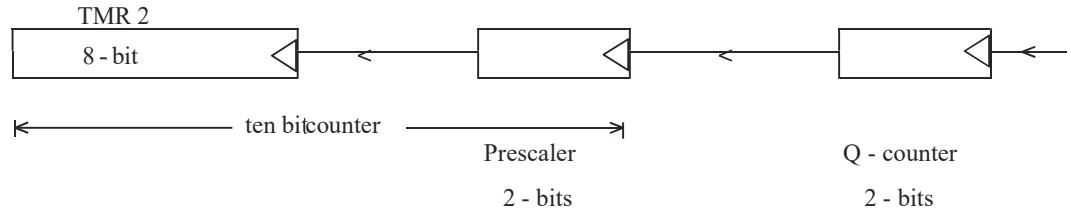


Counting mechanism in Timer 2

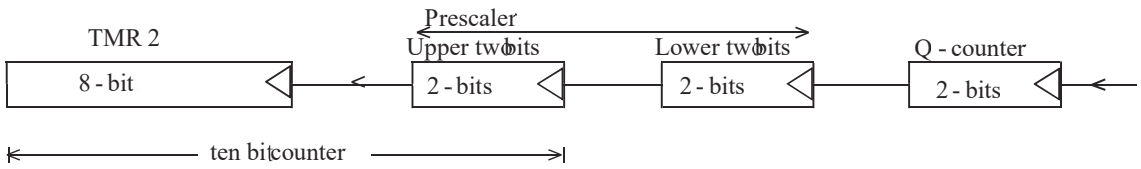
PWM Mode



Prescaler set to divide by one



Prescaler programed to divide by four



into CCPR1H until a match between PR2 and TMR2 occurs. In PWM mode, CCPR1H is a read-only register.

The CCPR1H register and a 2-bit internal latch are used to double buffer the PWM duty cycle. This double buffering is essential for glitchless PWM operation. When the CCPR1H and 2-bit latch match TMR2 concatenated with an internal 2-bit Q clock or 2-bits of prescalar, the CCP1 pin is cleared. Maximum PWM resolution (bits) for a given PWM frequency can be calculated as

$$\frac{\log(f_{PWM}/f_{osc})}{\log 2}$$

If the PWM duty cycle is longer than the PWM period, then the CCP1 pin will not be cleared.

PWM Period and duty cycle calculation Example

Desired PWM frequency = 78.125 kHz f_{osc} = 20MHz

TMR2 Prescalar = 1

$\frac{1}{78.125} \times 10^3 = (PR2 + 1)4 \times \frac{1}{20 \times 10^6}$ PR2 = 63

Find the maximum resolution of duty cycle that can be used with a 78.124 kHz frequency and 20 MHz oscillator.

$\frac{1}{78.125 \times 10^3}$ = 2PWM Resolution. $\frac{1}{20 \times 10^6}.1$

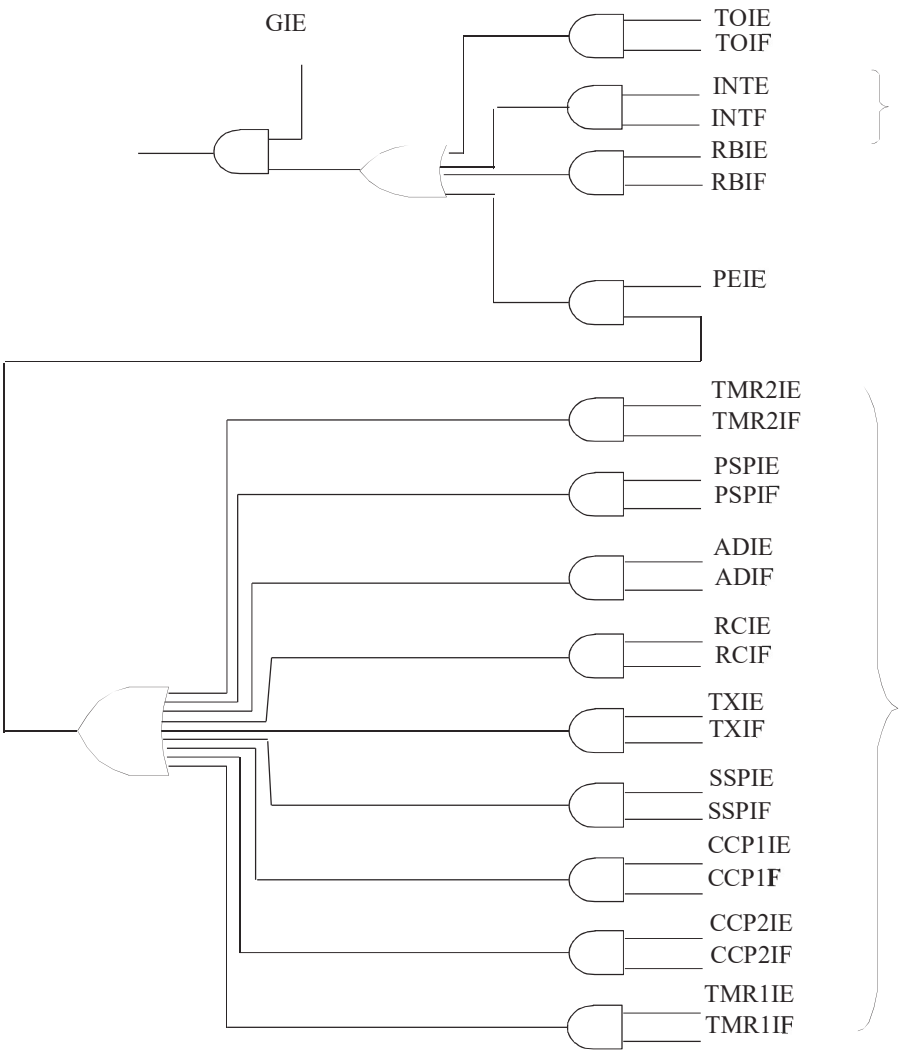
256 = 2PWM Resolution

PWM Resolution = 8

At most, an 8-bit resolution duty cycle can be obtained from a 78.125 kHz frequency and 20 MHz oscillator, ie, $0 \leq$ CCPR1L : CCP1CON $< 5 : 4 > \leq 255$. Any value greater than 255 will result in a 100 $^o/_o$ duty cycle. The following table gives the PWM frequency f_{PWM} if f_{osc} =20MHz

Duty cycle resolution	10-bit counter scale	PR2 value	Prescalar 1	Prescalar 4	Prescalar 16
10 bit	1024	255	19.53 KHz	4.88 kHz	1.22 kHz
≈ 10 bit	1000	249	20kHz	5kHz	1.25kHz
8 bit	256	63	78.125kHz	19.53kHz	4.88kHz
6 bit	64	15	312.5kHz	78.125kHz	19.53kHz

Interrupt Logic



RBO / INT

Peripheral
Interrupts

Four of PORTB's pins RB7 : RB4 have an interrupt on change feature. Only pins configured on inputs can cause this interrupt to occur. The input pins (of RB7 : RB4) are compared with the old values on the last read of Port B. the "mismatch" outputs of RB7 : RB4 are used

together to generate the RB port change interrupt flag bit RB1F.

References

1.NPTEL lecture notes ---www.nptel.com

2.PIC data sheet-----www.microchip.com

Advanced

RISC (Reduced Instruction Set Computer)

Machine
[ARM]

Introduction

[Credit: The definitive guide to the ARM by Josep Yiu]

An ARM processor is one of a family of CPUs based on the RISC (Reduced Instruction Set Computer) architecture developed by Advanced RISC Machines (ARM). ARM makes 32-bit and 64-bit RISC multicore processors. RISC processors are designed to perform a smaller number of types of computer instructions so that they can operate at a higher speed, performing more millions of instructions per second (MIPS). By stripping out unneeded instructions and optimising pathways, RISC processors provide outstanding performance at a fraction of the power demand of CISC (complex instruction set computing) devices.

ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee. Subsequently, additional companies, including Texas Instruments, NEC, Sharp, and ST Microelectronics, licensed the ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products.

Nowadays, ARM partners ship in excess of 2 billion ARM processors each year. Unlike many semiconductor companies, ARM does not manufacture processors or sell the chips directly. Instead,

ARM licenses the processor designs to business partners, including a majority of the world's leading semiconductor companies. Based on the ARM low-cost and power-efficient processor designs, these partners create their processors, microcontrollers, and system-on-chip solutions. This business model is commonly called intellectual property (IP) licensing.

In addition to processor designs, ARM also licenses systems-level IP and various software IPs. To support these products, ARM has developed a strong base of development tools, hardware, and software products to enable partners to develop their own products.

Architecture versions

Over the years, ARM has continued to develop new processors and system blocks. These include the popular ARM7TDMI processor and, more recently, the ARM1176TZ(F)-S processor, which is used in high-end applications such as smart phones. The evolution of features and enhancements to the processors over time has led to successive versions of the ARM architecture. Note that architecture version numbers are independent from processor names. For example, the ARM7TDMI processor is based on the ARMv4T architecture (the T is for Thumb® instruction mode support).

The ARMv5E architecture was introduced with the ARM9E processor families, including the ARM926E-S and ARM946E-S processors. This architecture added "Enhanced" Digital Signal Processing (DSP) instructions for multimedia applications.

With the arrival of the ARM11 processor family, the architecture was extended to the ARMv6. New features in this architecture included memory system features and Single Instruction–Multiple Data (SIMD) instructions. Processors based on the ARMv6 architecture include the ARM1136J(F)-S, the ARM1156T2(F)-S, and the ARM1176JZ(F)-S.

Following the introduction of the ARM11 family, it was decided that many of the new technologies, such as the optimized Thumb-2 instruction set, were just as applicable to the lower cost markets of micro- controller and automotive components. It was also decided that although the architecture needed to be con- sistent from the lowest MCU to the highest performance application processor, there was a need to deliver processor architectures that best fit applications, enabling very deterministic and low gate count processors for cost-sensitive markets and feature-rich and highperformance ones for high-end applications.

Over the past several years, ARM extended its product portfolio by diversifying its CPU develop- ment, which resulted in the architecture version 7 or v7. In this version, the architecture design is divided into three profiles:

- The A profile is designed for high-performance open application platforms.
- The R profile is designed for high-end embedded systems in which real-time performance is needed.
- The M profile is designed for deeply embedded microcontroller-type systems.

Let's look at these profiles in a bit more detail:

- A Profile (ARMv7-A): Application processors which are designed to handle complex applications such as high-end embedded operating systems (OSs) (e.g., Symbian, Linux, and Windows Embedded). These processors requiring the highest processing power, virtual memory system support with memory management units (MMUs), and, optionally, enhanced Java support and a secure program execution environment. Example products include high-end mobile phones and electronic wallets for financial transactions.
- R Profile (ARMv7-R): Real-time, high-performance processors targeted primarily at the higher end of the real-time1 market—those applications, such as high-end breaking systems and hard drive controllers, in which high processing power and high reliability are essential and for which low latency is important.
- M Profile (ARMv7-M): Processors targeting low-cost applications in which processing efficiency is important and cost, power consumption, low interrupt latency, and ease of use are critical, as well as industrial control applications, including real-time control systems.

The Cortex processor families are the first products developed on architecture v7, and the CortexM3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for microcontroller products.

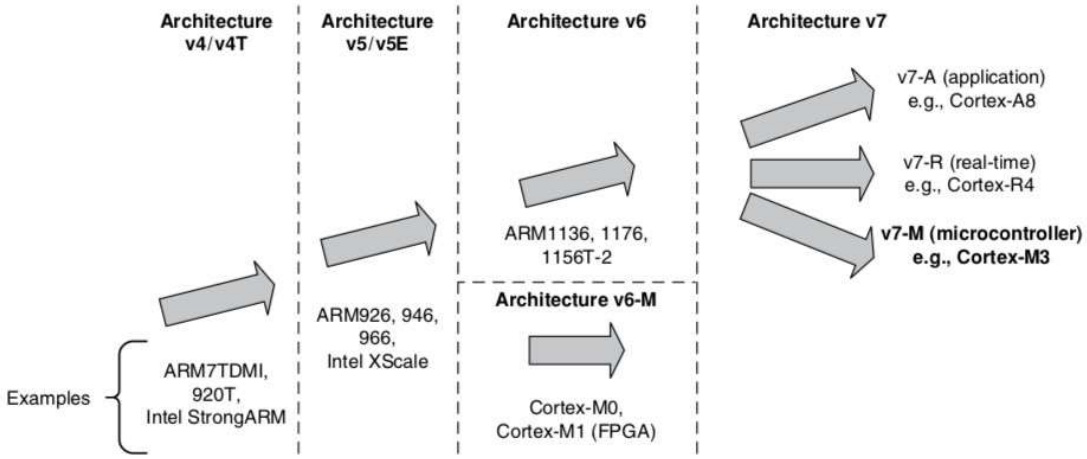


Fig:1 The Evolution of ARM Processor Architecture.

The ARM Architecture

[Credit: ARM systemon-chip architecture by Steve Furber]

In 1990 ARM Limited was established as a separate company specifically to widen the exploitation of ARM technology, since when the ARM has been licensed to many semiconductor manufacturers around the world. It has become established as a market-leader for low-power and cost-sensitive embedded applications.

No processor is particularly useful without the support of hardware and software development tools. The ARM is supported by a toolkit which includes an instruction set emulator for hardware modelling and software testing and benchmarking, an assembler, C and C++ compilers, a linker and a symbolic debugger.

The ARM architecture has evolved to a point where it supports implementations across a wide spectrum of performance points. Over two billion parts have shipped, establishing it as the dominant architecture across many market segments. The architectural simplicity of ARM processors has traditionally led to very small implementations, and small implementations allow devices with very low power consumption. Implementation size, performance, and very low power consumption remain key attributes in the development of the ARM architecture.

The ARM is a *Reduced Instruction Set Computer* (RISC), as it incorporates these typical RISC architecture features:

- a large uniform register file
- a *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents

- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only

- uniform and fixed-length instruction fields, to simplify instruction decode. In addition, the ARM architecture provides:

- control over both the *Arithmetic Logic Unit* (ALU) and shifter in most data-processing instructions to maximize the use of an ALU and a shifter

- auto-increment and auto-decrement addressing modes to optimize program loops

- Load and Store Multiple instructions to maximize data throughput

- conditional execution of almost all instructions to maximize execution throughput. These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area.

ARM registers

[Credit: ARM architecture reference manual]

ARM has 31 general-purpose 32-bit registers. At any one time, 16 of these registers are visible. The other registers are used to speed up exception processing. All the register specifiers in ARM instructions can address any of the 16 visible registers.

The main bank of 16 registers is used by all unprivileged code. These are the User mode registers. User mode is different from all other modes as it is unprivileged, which means:

- User mode can only switch to another processor mode by generating an exception. The SWI instruction provides this facility from program control.

- Memory systems and coprocessors might allow User mode less access to memory and coprocessor functionality than a privileged mode. Three of the 16 visible registers have special roles: **Stack pointer Link register**

Program counter

Software normally uses R13 as a *Stack Pointer* (SP). R13 is used by the PUSH and POP instructions in T variants, and by the SRS and RFE instructions from ARMv6.

Register 14 is the *Link Register* (LR). This register holds the address of the next instruction after a Branch and Link (BL or BLX) instruction, which is the instruction used to make a subroutine call. It is also used for return address information on entry to exception modes. At all other times, R14 can be used as a general-purpose register.

Register 15 is the *Program Counter* (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed. In ARM state, all ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 nonconstant bits. Two other processor states are supported by some versions of the architecture. Thumb® state is supported on T variants, and Jazelle® state on J variants. The PC can be halfword (16-bit) and byte aligned respectively in these states.

The remaining 13 registers have no special hardware purpose.

Exceptions

ARM supports seven types of exception, and a privileged processing mode for each type. The seven types of exception are:

- reset
- attempted execution of an Undefined instruction
- software interrupt (SWI) instructions, can be used to make a call to an operating system
- Prefetch Abort, an instruction fetch memory abort
- Data Abort, a data access memory abort
- IRQ, normal interrupt
- FIQ, fast interrupt. When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exception modes have replacement *banked* registers for R13 and R14. The fast interrupt mode has additional banked registers for fast interrupt processing. When an exception handler is entered, R14 holds the return address for exception processing. This is used to return after the exception is processed and to address the instruction that caused the exception. Register 13 is banked across exception modes to provide each exception handler with a private stack pointer. The fast interrupt mode also banks registers 8 to 12 so that interrupt processing can begin without the need to save or restore these registers. There is a sixth privileged processing mode, System mode, which uses the User mode registers. This is used to run tasks that require privileged access to memory and/or coprocessors, without limitations on which exceptions can occur during the task.

Status registers

All processor state other than the general-purpose register contents is held in *status registers*. The current operating processor status is in the *Current Program Status Register* (CPSR). The CPSR holds:

- four condition code flags (Negative, Zero, Carry and oVerflow).

- one sticky (Q) flag (ARMv5 and above only). This encodes whether saturation has occurred in saturated arithmetic instructions, or signed overflow in some specific multiply accumulate instructions.

- four GE (Greater than or Equal) flags (ARMv6 and above only). These encode the following conditions separately for each operation in parallel instructions:

  — whether the results of signed operations were non-negative

  — whether unsigned operations produced a carry or a borrow.

- two interrupt disable bits, one for each type of interrupt (two in ARMv5 and below).

- one (A) bit imprecise abort mask (from ARMv6)

- five bits that encode the current processor mode.

- two bits that encode whether ARM instructions, Thumb instructions, or Jazelle opcodes are being executed.

- one bit that controls the endianness of load and store operations (ARMv6 and above only). Each exception mode also has a *Saved Program Status Register* (SPSR) which holds the CPSR of the task immediately before the exception occurred. The CPSR and the SPSRs are accessed with special instructions.

Field	Description	Architecture
N Z C V	Condition code flags	All
J	Jazelle state flag	5TEJ and above
GE[3:0]	SIMD condition flags	6
E	Endian Load/Store	6
A	Imprecise Abort Mask	6
I	IRQ Interrupt Mask	All
F	FIQ Interrupt Mask	All
T	Thumb state flag	4T and above
Mode[4:0]	Processor mode	All

Table:1 Status register summary

Instruction set

[Credit: Jin-Fu Li, National Chiao-Tung University]

ARM processor is a 32-bit architecture.
Most ARM's implement two instruction sets –

- 32-bit ARM instruction set
- 16-bit Thumb instruction set

ARM processor supports 6 data types

- – 8-bits signed and unsigned bytes
- – 16-bits signed and unsigned half-word, aligned on 2-byte boundaries
- – 32-bits signed and unsigned words, aligned on 4-byte boundaries
- ARM instructions are all 32-bit words, word-aligned; Thumb instructions are half-words, aligned on 2- byte boundaries
- ARM coprocessor supports floating-point values

ARM has 37 registers, all of which are 32 bits long

- 1 dedicated program counter
- 1 dedicated current program status register
- 5 dedicated saved program status registers
- 31 general purpose registers

The current processor mode governs which bank is accessible

– User mode can access

- A particular set of r0 – r12 registers
- A particular r13 (stack pointer, SP) and r14 (link register. LR)
- The program counter, r15 (PC)

• The curent program status register, CPSR

– Privileged modes (except system) can access

• A particular SPSR (Saved Program Status Register)

Register banking

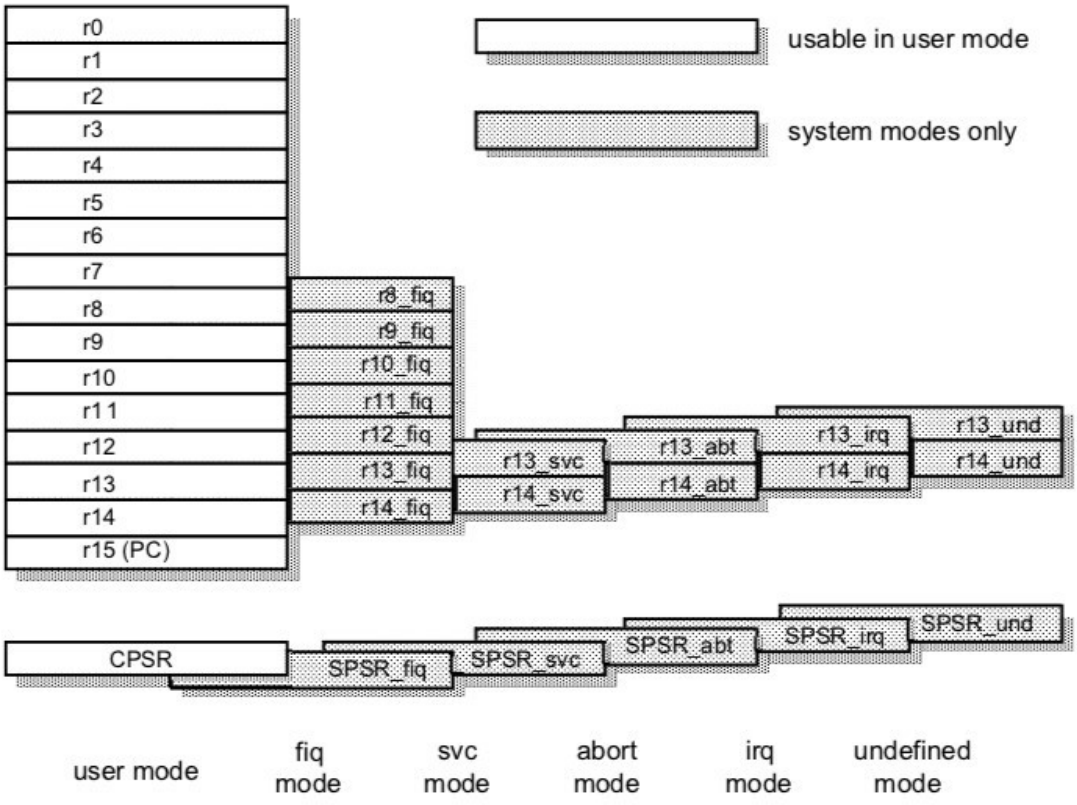


Fig:2 Register banking

Program Counter

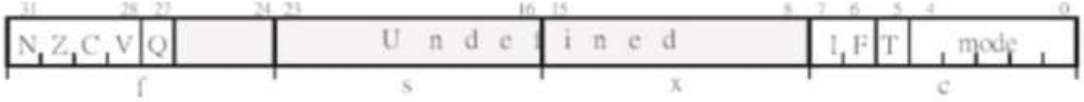
When the processor is executing in ARM state:

- All instructions are 32 bits wide
- All instructions must be word-aligned
- Therefore, the PC value is stored in bits [32:2] with bits [1:0] undefined (as instruction cannot be half word)

When the processor is executing in Thumb state:

- All instructions are 16 bits wide
- All instructions must be half word-aligned
- Therefore, the PC value is stored in bits [32:1] with bits [0] undefined (as instruction cannot be byte-aligned)

Current Program Status Registers (CPSR)



Condition code flags:

– N: Negative result form ALU

– Z: Zero result from ALU

– C: ALU Operation Carried out

– V: ALU operation overflowed

Interrupt disable bits:

– I = 1, disable the IRQ

– F = 1, disable the FIQ

T Bit

– Architecture xT only

– T = 0, processor in ARM state

– T = 1, processor in Thumb state

Mode bits

– Specify the processor mode

Sticky overflow flag – Q flag

– Architecture 5TE only

– Indicates if saturation has occurred during certain operations state

Processor Modes

- ARM has seven basic operation modes
- Mode changes by software control or external interrupts

CPRS[4:0]	Mode	Use	Registers
10000	User	Normal user code	User
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irp
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system	user

Features of the ARM Instruction Set

- Load-store architecture
  - Process values which are in registers
  - Load, store instructions for memory data accesses
- 3-address data processing instructions
- Conditional execution of every instruction
- Load and store multiple registers
- Shift, ALU operation in a single instruction
- Open instruction set extension through the coprocessor instruction
- Very dense 16-bit compressed instruction set (Thumb)

Thumb

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code – Improved performance form narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has two execution states – ARM and Thumb – Switch between them using BX instruction
- Thumb has characteristic features:
  – Most Thumb instruction are executed unconditionally
  – Many Thumb data process instruction use a 2-address format
  – Thumb instruction formats are less regular than ARM instruction formats, as a result of the dense encoding.

32-bit instruction set

- Data processing instructions
- Data transfer instructions
- Control flow instructions

ARM Instruction Set Summary

Mnemonic	Instruction	Action
ADC	Add with carry	Rd:=Rn+Op2+Carry
ADD	Add	Rd:=Rn+Op2
AND	AND	Rd:=Rn AND Op2
B	Branch	R15:=address
BIC	Bit Clear	Rd:=Rn AND NOT Op2
BL	Branch with Link	R14:=R15 R15:=address
BX	Branch and Exchange	R15:=Rn T bit:=Rn[0]
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags:=Rn+Op2
CMP	Compare	CPSR flags:=Rn-Op2

Mnemonic	Instruction	Action
EOR	Exclusive OR	Rd:=Rn^Op2
LDC	Load Coprocessor from memory	(Coprocessor load)
LDM	Load multiple registers	Stack Manipulation (Pop)
LDR	Load register from memory	Rd:=(address)
MCR	Move CPU register to coprocessor register	CRn:=rRn{<op>cRm}
MLA	Multiply Accumulate	Rd:=(Rm*Rs)+Rn
MOV	Move register or constant	Rd:=Op2
MRC	Move from coprocessor register to CPU register	rRn:=cRn{<op>cRm}
MRS	Move PSR status/flags to register	Rn:=PSR
MSR	Move register to PSR status/flags	PSR:=Rm

Mnemonic	Instruction	Action
MUL	Multiply	Rd:=Rm*Rs
MVN	Move negative register	Rd:=~Op2
ORR	OR	Rd:=Rn OR Op2
RSB	Reverse Subtract	Rd:=Op2-Rn
RSC	Reverse Subtract with Carry	Rd:=Op2-Rn-1+Carry
SBC	Subtract with Carry	Rd:=Rn-Op2-1+Carry
STC	Store coprocessor register to memory	address:=cRn
STM	Store Multiple	Stack manipulation (Push)

Mnemonic	Instruction	Action
STR	Store register to memory	<address>:=Rd
SUB	Subtract	Rd:=Rn-Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd:=[Rn] [Rn]:=Rm
TEQ	Test bitwise equality	CPSR flags:=Rn EOR Op2
TST	Test bits	CPSR flags:=Rn AND Op2

Data Processing Instruction

- Consist of
  - Arithmetic (ADD, SUB, RSB)
  - Logical (BIC, AND)
  - Compare (CMP, TST)
  - Register movement (MOV, MVN)
- All operands are 32-bit wide; come from registers or specified as literal in the instruction itself
- Second operand sent to ALU via barrel shifter
- 32-bit result placed in register; long multiply instruction produces 64-bit result
- 3-address instruction format

Conditional Execution

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Data Processing Instructions

- Simple register operands
- Immediate operands
- Shifted register operands
- Multiply

Simple Register Operands

```
ADD r0,r1,r2     ;r0:=r1+r2

ADC r0,r1,r2     ;r0:=r1+r2+C

SUB r0,r1,r2     ;r0:=r1-r2

SBC r0,r1,r2     ;r0:=r1-r2+C-1

RSB r0,r1,r2     ;r0:=r2-r1, reverse subtraction

RSC r0,r1,r2     ;r0:=r2-r1+C-1
```

By default, data processing operations do no affect the condition flags

Bit-wise Logical Operations

```
AND r0,r1,r2 ;r0:=r1ANDr2 ORR

r0,r1,r2 ;r0:=r1ORr2

EOR r0,r1,r2 ;r0:=r1XORr2

BIC r0,r1,r2 ;r0:=r1AND (NOT r2), bit clear
```

Simple Register Operands

Register Movement Operations

– Omit 1st source operand from the format

```
MOV r0,r2      ;r0:=r2

MVN r0,r2      ;r0:=NOT r2, move 1's complement
```

Comparison Operations

– Not produce result; omit the destination from the format

– Just set the condition code bits (N, Z, C and V) in CPSR

```
CMP r1,r2 ;set cc on r1 - r2, compare

CMN r1,r2 ;set cc on r1 + r2, compare negated

TST r1,r2 ;set cc on r1 AND r2, bit test

TEQ r1,r2 ;set cc on r1 XOR r2, test equal
```

Immediate Operands

Replace the second source operand with an immediate

operand, which is a literal constant, preceded

by "#"

```
ADD r3,r3,#1      ;r3:=r3+1
```

`AND r8,r7,#&FF ;r8:=r7[7:0], &:hexadecimal` Since the immediate value is coded within the 32 bits of the instruction, it is not possible to enter every possible 32-bit value as an immediate.

Shift Register Operands

`- ADD r3,r2,r2,LSL#3 ;r3 := r2 + 8 * r1`

- A single instruction executed in a single cycle

- LSL: Logical Shift Left by 0 to 31 places, 0 filled at the lsb end
- LSR, ASL (Arithmetic Shift Left), ASR, ROR (Rotate Right), RRX (Rotate Right eXtended by 1 place)

Data Processing Instructions

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

Single Register Data Transfer

- Word transfer – LDR / STR
- Byte transfer – LDRB / STRB
- Halfword transfer – LDRH / STRH
- Load singled byte or halfword-load value and sign extended to 32 bits – LDRSB / LDRSH

- All of these can be conditionally executed by inserting the appropriate condition code after STR/LDR

Addressing

Register-indirect addressing

Base-plus-offset addressing

– Base register • r0–r15

– Offset, and or subtract an unsigned number • Immediate

• Register (not PC)

• Scaled register (only available for word and unsigned byte instructions)

Stack addressing

Block-copy addressing

Control Flow Instructions

- Branch instructions
- Conditional branches
- Conditional execution
- Branch and link instructions
- Subroutine return instructions
- Supervisor calls
- Jump tables

Conditional Branch

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not Equal	Comparison equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave give carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

ARM Processor Programming in C using ARM development tools

[Credit: ARM systemon-chip architecture by Steve Furber]

Software development for the ARM is supported by a coherent range of tools devel- oped by ARM Limited, and there are also many third party and public domain tools available, such as an ARM backend for the gcc C compiler.

The ARM C compiler is compliant with the ANSI (American National Standards Institute) standard for C and is supported by the appropriate library of standard functions. It uses the ARM Procedure Call Standard (see Section 6.8 on page 175) for all externally available functions. It can be told to produce assembly source output instead of ARM object format, so the code can be inspected, or even hand optimized, and then assembled subsequently. The compiler can also produce Thumb code.

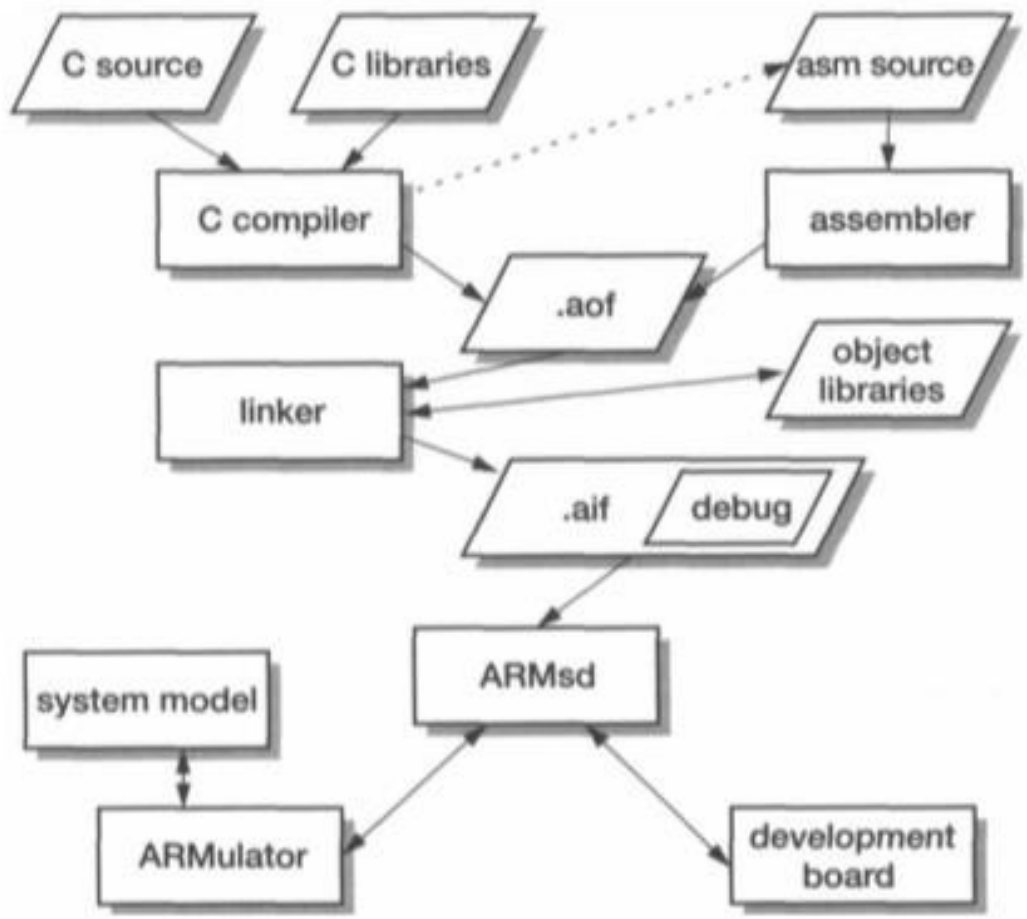


Fig:3 The structure of the ARM cross-development toolkit.

The ARM Development Board is a circuit board incorporating a range of compo- nents and interfaces to support the development of ARM-based systems. It includes an ARM core (for example, an ARM7TDMI), memory components which can be configured to match the performance and buswidth of the memory in the target sys- tem, and electrically programmable devices which can be configured to emulate application-specific peripherals. It can support both hardware and software development before the final application-specific hardware is available.

ARM Limited supplies the complete set of tools described above, with some support utility programs and documentation, as the 'ARM Software Development Toolkit'. The Toolkit CD-ROM includes a PC version of the toolset that runs under most ver- sions of the Windows operating system and includes a full Windows-based project manager. The toolkit is updated as new versions of the ARM become available.

References

- ARM System-on-Chip Architecture by S.Furber, Addison Wesley Longman: ISBN 0-20167519-6.
- http://twins.ee.nctu.edu.tw/courses/ip_core_02/index.html
- ARM Architecture Reference Manual
- The Definitive Guide to the ARM Cortex-M3 by Joseph Yiu, Elsevier, DOI: 10.1016/B978-185617-963-8.00004-1

Embedded system

What is Embedded Systems?

Embedded system is defined as a way of working, performing or organizing one or many tasks according to a fixed set of rules (or) an arrangement in which all the units assemble and work together according to the program or plan. Examples of embedded systems are a watch and washing machine.

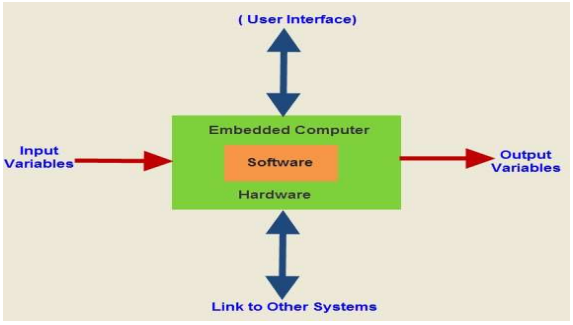


Fig. 4.1

Embedded System

An embedded system is a system that has software embedded in computer hardware. It makes a system dedicated to a specific part of an application or product of a larger system. Depending on the application, embedded system may be programmable or nonprogrammable. Examples of embedded systems include various products such as washing machine, microwave ovens, cameras, printers and automobiles. They use microprocessors and microcontrollers as well as specially designed processors such as digital signal processors (DSP).

Basics of Embedded Systems

The embedded systems basics include the components of embedded system hardware, embedded system types and several characteristics. An embedded system has three main components: Embedded system hardware, Embedded system software and Operating system.

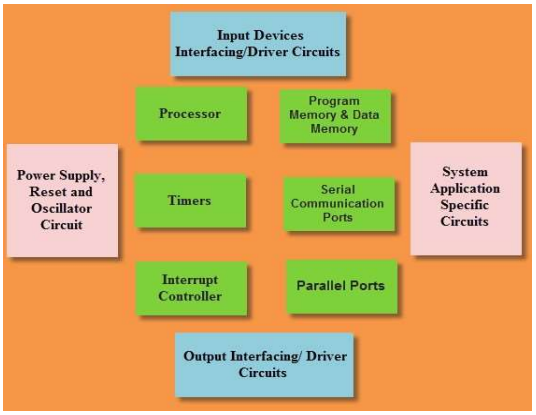


Fig 4.2

Embedded System Block Diagram *Embedded*

System Hardware:

As with any electronic system, an embedded system requires a hardware platform on which it performs the operation. Embedded system hardware is built with a microprocessor or microcontroller. The embedded system hardware has elements like input output (I/O) interfaces, user interface, memory and the display. Usually, an embedded system consists of:

- Power Supply
- Processor
- Memory
- Timers
- Serial communication ports
- Output/Output circuits
- System application specific circuits *Embedded System Software:*

The embedded system software is written to perform a specific function. It is typically written in a high level format and then compiled down to provide code that can be lodged within a non-volatile memory within the hardware. An embedded system software is designed to keep in view of the three limits:

- Availability of system memory
- Availability of processor's speed
- When the system runs continuously, there is a need to limit power dissipation for events like stop, run and wake up.

Real Time Operating System

A system is said to be real time, if it is essential to complete its work and deliver its service on time. Real time operating system manages the application software and affords a mechanism to let the processor run. The Real Time operating system is responsible for handling the hardware resources of a computer and host applications which run on the computer.

An RTOS is specially designed to run applications with very precise timing and a high amount of reliability. Especially, this can be important in measurement and industrial automation systems wherein downtime is costly or a program delay could cause a safety hazard.

DIFFERENCE BETWEEN MICROPROCESSOR AND MICROCONTROLLER

Microprocessor is an IC which has only the CPU inside them i.e. only processing power such as Intel's Pentium 1,2,3,4 or core 2 due, i3, i5 etc..

These microprocessors don't have RAM, ROM and other peripheral on the chip. A system designer has to add them externally to make them functional.

Application of microprocessor includes Desktop PC's Laptops, notepads, any computation systems , defense systems and network communications.

But this is not the case with Microcontrollers. Microcontroller has a CPU, in addition with a fixed amount of RAM, ROM and other peripherals all embedded on a single chip. It is also termed as a mini computer or a computer on a single chip. Microcontrollers are designed to perform specific tasks. Specific means applications where the relationship of input and output is defined.

Depending on the input, some processing needs to be done and output is delivered. For example: keyboard, mouse, washing machine, digicam, pendrive, remote, microwave, cars, bikes, telephone, mobiles, watches, etc..

Since the applications are very specific, they need small resources like RAM, ROM, I/O ports etc and hence can be embedded on a single chip. This in turn reduces the size and the cost.

Microprocessor find applications where tasks are unspecified like developing software, games, websites, photo editing, creating documents etc.. In such cases the relationship between input and output is not defined. They need high amount of resources like RAM, ROM, I/O ports etc.

The clock speed of the Microprocessor is quite high as compared to Microcontroller. Whereas Microcontrollers operates from a few MHz to 30-50 MHz, today's Microprocessors operates above 1GHz as they perform complex tasks.

Comparing Microprocessors and Microcontrollers in terms of cost is not justified.

Undoubtedly a microcontroller is far cheaper than a microprocessor.

However Microcontroller cannot be used in place of microprocessor and doing that is not advisable. As it makes the application quite costly.

Microprocessor cannot be used stand alone. They need other peripherals like RAM, ROM, Buffer, I/O ports etc.. and hence a system designed around a microprocessor is quite costly.

https://www.quora.com/What-is-the-difference-between-a-microprocessor-andmicrocontroller

Embedded Real-Time Systems vs. General-Purpose Computers Embedded real-time systems have two main characteristics:

1. They have a computer buried inside, but the users don't perceive them as computers.

2. They often must respond to external events in a timely fashion, which means that for all practical purposes, a late computation is just as bad as an outright wrong computation.

Vague as it is, this definition can gain the most strength by contrasting real-time embedded systems with general-purpose computers (such as desktop PCs), in which the two main characteristics are either nonexistent or far less important. So, you can read embedded to mean "not for general-purpose computing" and real-time to mean "dedicated to an application with timeliness requirements." Either way, the definition emphasizes that embedded systems pose different challenges and require different programming strategies than general-purpose computers. I strongly disagree with the opinion that embedded real-time developers face all the challenges of "regular" software development plus the complexities inherent in embedded real-time systems. Although each domain has its fair share of difficulties, each also offers unique opportunities for simplification, so embedded-systems programmers specifically do not have to cope with many problems encountered in programming general-purpose computers.

Consider for example the challenges of programming a desktop PC. As far as hardware is concerned, no desktop application can rely on a specific amount of memory available to it or on how many and what kind of disk drives, network cards, graphics adapters, and other peripherals are present and available at the moment. The software environment is even less predictable. Users frequently install and remove applications and application components from all possible sources (remember the Windows DLL Hell?). All the time, users launch, close, or crash their applications -- drastically changing the CPU load and availability of memory and other resources. The desktop operating system has the tough job of allocating CPU cycles, memory, and other resources among constantly changing tasks in such a way that each receives a fair share of the resources and no single task can hog the CPU. To succeed in this harsh environment, the desktop OS has no other option but to drastically limit the applications. All applications must strictly comply with a specific API (such as Win32 or a Unix API). Interrupt handling is black magic reserved for device drivers that common mortals (application programmers) better not touch. Fiddling directly with external hardware is prohibited.

This scheme is diametrically opposed to the needs of embedded real-time systems, in which a specific task must gain control right now and run until it produces the appropriate output. Fairness isn't part of real-time programming -- meeting the deadlines is. To achieve this, however, embedded software must have full control over the CPU, memory and all the external hardware. Restricted to a desktop-style API, an embedded developer not only loses control that he so badly needs, but must bend backwards just to flash an LED, let alone to service an interrupt. The increased security of a desktop API in the embedded domain is bogus too. In an embedded system, the specific application code is at least as critical as the generic OS (many

embedded systems don't use an OS at all), so a failure in the application renders the system useless regardless of the security mechanisms built into the OS.

Characteristics of an Embedded System

- **Single-functioned** − An embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.
- **Tightly constrained** − All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.
- **Reactive and Real time** − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or deaccelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.
- **Microprocessors based** − It must be microprocessor or microcontroller based.
- **Memory** − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.
- **Connected** − It must have connected peripherals to connect input and output devices.
- **HW-SW systems** − Software is used for more features and flexibility. Hardware is used for performance and security.

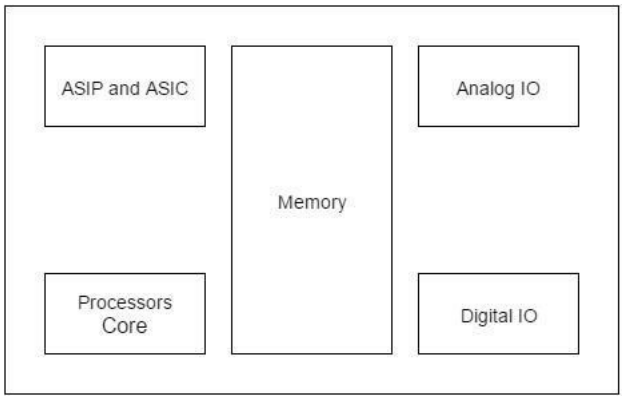


Fig 4.3

Advantages

- Easily Customizable
- Low power consumption
- Low cost
- Enhanced performance

CLASSIFICATIONS OF EMBEDDED SYSTEM

Embedded systems are classified into four categories based on their performance and functional requirements:

- Standalone embedded systems
- Real time embedded systems
- Networked embedded systems
- Mobile embedded systems

Embedded Systems are classified into three types based on the performance of the microcontroller such as

- Small scale embedded systems
- Medium scale embedded systems
- Sophisticated embedded systems

Stand Alone Embedded Systems

Standalone embedded systems do not require a host system like a computer, it works by itself. It takes the input from the input ports either analog or digital and processes, calculates and converts the data and gives the resulting data through the connected device-Which either controls, drives or displays the connected devices. Examples for the stand alone embedded systems are mp3 players, digital cameras, video game consoles, microwave ovens and temperature measurement systems.

Real Time Embedded Systems

A real time embedded system is defined as, a system which gives a required o/p in a particular time.These types of embedded systems follow the time deadlines for completion of a task. Real time embedded systems are classified into two types such as soft and hard real time systems.

Networked Embedded Systems

These types of embedded systems are related to a network to access the resources. The connected network can be LAN, WAN or the internet. The connection can be any wired or wireless. This type of embedded system is the fastest growing area in embedded system applications. The embedded web server is a type of system wherein all embedded devices are connected to a web server and accessed and controlled by a web browser. Example for the LAN networked embedded system is a home security system wherein all sensors are connected and run on the protocol TCP/IP

Mobile Embedded Systems

Mobile embedded systems are used in portable embedded devices like cell phones, mobiles, digital cameras, mp3 players and personal digital assistants, etc. The basic limitation of these devices is the other resources and limitation of memory.

Small Scale Embedded Systems

These types of embedded systems are designed with a single 8 or 16-bit microcontroller, that may even be activated by a battery. For developing embedded software for small scale embedded systems, the main programming tools are an editor, assembler, cross assembler and integrated development environment (IDE).

Medium Scale Embedded Systems

These types of embedded systems design with a single or 16 or 32 bit microcontroller, RISCs or DSPs. These types of embedded systems have both hardware and software complexities. For developing embedded software for medium scale embedded systems, the main programming tools are C, C++, JAVA, Visual C++, RTOS, debugger, source code engineering tool, simulator and IDE.

Sophisticated Embedded Systems

These types of embedded systems have enormous hardware and software complexities, that may need ASIPs, IPs, PLAs, scalable or configurable processors. They are used for cuttingedge applications that need hardware and software Co-design and components which have to assemble in the final system.

APPLICATIONS OF EMBEDDED SYSTEMS:

Embedded systems are used in different applications like automobiles, telecommunications, smart cards, missiles, satellites, computer networking and digital consumer electronics.

Embedded Systems in Automobiles and in telecommunications

- Motor and cruise control system
- Body or Engine safety
- Entertainment and multimedia in car
- E-Com and Mobile access
- Robotics in assembly line

- Wireless communication
- Mobile computing and networking

Embedded Systems in Smart Cards, Missiles and Satellites

- Security systems
- Telephone and banking
- Defense and aerospace
- Communication

Embedded Systems in Peripherals & Computer Networking

- Displays and Monitors
- Networking Systems
- Image Processing
- Network cards and printers

Embedded Systems in Consumer Electronics

- Digital Cameras
- Set top Boxes
- High Definition TVs
- DVDs

REFERENCE :

1. www.efxkits.us

2. www.tutorialspoint.com

3. www.edgefx.in

4. www.quora.com

5. Embedded Systems – Raj Kamal

6. Embedded Systems – Shivu

Question :

1. Mention some application of Embedded Systems.
2. Differentiate between microprocessor and microcontroller.

MODULE – VI

Real-Time Operating System (RTOS)

What Is a Real-Time Operating System (RTOS)
The heightened reliance on technology to execute crucial tasks led to the development of high-performance and deterministic operating systems, including real-time operating systems(RTOS). An operating system is software that facilitates hardware to receive and execute user commands. Operating systems are needed for scheduling tasks, memory and file management, and for access to hardware resources.

A **real-time operating system (RTOS)** is an operating system that works in real time, with deterministic constraints that require efficient time usage and power to process incoming data and relay the expected results without any unknown or unexpected delays. RTOS software is time dependent, meaning that it should process input and offer output within a short predetermined deterministic period. However the key to an RTOS, and the most important demand of RTOS software is that a request and response for data is guaranteed to occur. If a Windows OS has request and response calls that are fast 90% of the time, yet the remaining 10% of the time an input/output request takes too long, then the real-time application is not performing correctly. Thus an RTOS is not meant to be only fast, it is more importantly meant to be dependable

Components of a RTOS

A real-time operating system includes multiple components:

The scheduler: This is the main RTOS element that determines the order of execution of tasks or threads usually based on a priority scheme, and either in a run to completion or round robin fashion. Some RTOS may try to load balance thread across processors but most require developers to assign process affinity to cores to optimize real-time application resource usage.
Symmetric Multiprocessing (SMP): An RTOS has the ability to handle and separate multiple tasks or threads so that they can be run on multiple cores to allow for parallel processing of code (i.e. multitasking).
Function library: Is a standard interface that can contain an application program interface (API) to call routines within it, this is the interface that connects that application code and the kernel. Application code entities direct requests to the kernel via the function library to prompt the application to give the desired programmatic behavior.
Fast dispatch latency/context switch time: Dispatch latency represents the time from when the operating system identifies that a task has finishedanuntil a ready to run thread is started or when an event is triggered that causes a higher priority tasks to preempt a currently running task. The context switch is the time it takes for the scheduled to switch from one running thread to another thread, this involved saving off the context of the current task and replacing it with the context of the new thread to run.In an RTOS, the switching time should remain deterministic and minimal.
User-defined data objects and classes: An RTOS relies on programming languages with data structures that are organized based on their type of operation. The user defines object sets through a specified programming language like C++ that the RTOS will use in to control the specified application.
Memory Management: Memory management is required to allocate memory for every program to be run or object to be referenced in memory. In an RTOS this is important, since unlike General Purpose OSes like Windows it can't afford to have memory paged in or out since it leads to non-deterministic behavior.

Types of Real-Time Operating Systems

Real-time operating systems are classified into three types:

Soft real-time systems

Meeting command deadlines in soft real-time operating systems is not compulsory for every task. However, the systems should always give the expected results. A soft RTOS requires that a response be logically correct and occur before a certain deadline or the result becomes increasingly inaccurate. Essentially the result can still hold some value even though it occurred after the required deadline.

Hard real-time systems

A hard real-time system is a time constrained and deterministic system that responds within a specified time frame. They are dictated by deadlines, latency and time constraints. For instance, if an output is expected within 10 seconds, the system should process the input and give out the output by the 10th second. Note that the output should not be released by the 9th or 11th second to prevent the system from failing.

Applications of Real-Time Operating Systems

An RTOS can be flexible but is usually designed for set purposes. Most RTOS subsystems are assigned certain tasks and leave anything and everything else not designated to it for the Windows OS itself to handle. An RTOS offers mostly operational solutions, including applications such as:

Control systems: The RTOS is used to monitor and execute control system commands. Realtime systems are used to control actuators and sensors for functions like digital controllers. Controlled systems include aircraft, brakes, and engines. Controlled systems are monitored with the help of sensors and altered by actuators. The RTOS reads the data from sensors and then performs calculations and moves the actuators so that movement in a flight can be simulated.

Image processing: Computers, mobile gadgets, and cameras must achieve their intended duty in realtime, which means that visual input is needed in real-time with the utmost precision so that industrial automation, for instance can control what is happening on conveyors or an assembly line when an item is moving down its path and there is a defect, or the item has moved

its location. Real-time image processing is essential for making real-time adjustments for moving objects.

Voice over IP (VoIP)*:* VoIP relies on Internet protocols to transmit voices in real time. As such, VoIP can be implemented on any IP network like intranets, local area networks, and the Internet. The voice is digitalized, compressed and converted to IP packets in real time before being transmitted over an IP network.

Considerations for Choosing an RTOS

Performance is a core factor that must be considered when choosing an RTOS. **Real-time operating systems** are different and perform differently. Key aspect for an RTOS is that its determinism guarantees that request and responses of data happen within a set period of time no matter what else is happening in the PC system. When determining the best RTOS, ask questions such as whether the system is showing any jitter within your tolerance range and thereby providing the determinism that you need. RTOS performance should be determined by a system's dependable in executing calls within a specified period, regardless of anything else happening on the system.

A **real-time operating system** should be of premium quality and easy to navigate. Developing embedded projects is hard and time-consuming; developers should not have to struggle with real-time system-related issues that can be distracting. An RTOS should be a trusted component that any developer can count on.

Good examples of real-time operating systems are the RTX (32-bit) and RTX64 (64-bit) solutions that allow you and your team to focus on adding value to your applications. This software is designed to serve as a hard real-time system that delivers output within a specified time frame to improve embedded systems' quality.

PROGRAMS, PROCESSES, TASKS AND THREADS

The above four terms are often found in literature on OS in similar contexts. All of them refer to a unit of computation. A program is a general term for a unit of computation and is typically used in the context of programming. A process refers to a program in execution. A process is an independently executable unit handled by an operating system. Sometimes, to ensure better utilization of computational resources, a process is further broken up into threads. Threads are sometimes referred to as lightweight processes because many threads can be run in parallel, that is, one at a time, for each process, without incurring significant additional overheads. A task is a generic term, which, refers to an independently schedulable unit of computation, and is used typically in the context of scheduling of computation on the processor. It may refer either to a process or a thread.

MULTITASKING

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The intertask communication facilities allow these tasks to synchronize and coordinate their activity. Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications. Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilization of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power, silicon area and other constraints. In a multi-tasking operating system it is assumed that the various tasks are to cooperate to serve the requirements of the overall system. Co-operation will require that the tasks communicate with each other and share common data in an orderly an disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed.

To realize such a system, the following major functions are to be carried out.

A. Process Management

• Interrupt handling

• Task scheduling and dispatch

• Create/delete, suspend/resume task

• Manage scheduling information – priority, scheduling policy etc

B. Interprocess Communication and Synchronization

• Code, data and device sharing

• Synchronization, coordination and data exchange mechanisms

• Deadlock and Livelock detection

Interrupt in RTOS

In systems programming, an interrupt is a signal to the processor.

It can be emitted either by hardware or software indicating an event that needs immediate attention.

1. **Hardware interrupt**

A hardware interrupt is a signal which can tell the CPU that something happen in hardware device, and should be immediately responded. Hardware interrupts are triggered by peripheral devices outside the microcontroller. An interrupt causes the processor to save its state of execution and begin execution of an interrupt service routine.

Unlike the software interrupts, hardware interrupts are **asynchronous** and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an **interrupt request (IRQ)**.

2. **Software interrupt**

Software interrupt is an instruction which cause a context switch to an interrupt handler similar to a hardware interrupt. Usually it is an interrupt generated within a processor by executing a special instruction in the instruction set which causes an interrupt when it is executed.

Another type of software interrupt is triggered by an exceptional condition in the processor itself. This type of interrupt is often called a **trap** or **exception**.

Unlike the hardware interrupts where the number of interrupts is limited by the number of interrupt request (IRQ) lines to the processor, software interrupt can have hundreds of different interrupts.

Interrupt latency

Interrupt latency refers primarily to the software interrupt handling latencies. In other words, the amount of time that elapses from the time that an external **interrupt arrives** at the processor until the time that the **interrupt processing begins**. One of the most important aspects of kernel real-time performance is the ability to service an interrupt request (IRQ) within a specified amount of time.

Here are the sources contributing the interrupt latency

Operating system (OS) interrupt latency

An RTOS must sometimes disable interrupts while accessing critical OS data structures. The maximum time that an RTOS disables interrupts is referred to as the OS interrupt latency. Although this overhead will not be incurred on most interrupts since the RTOS disables interrupts relatively infrequently, developers must always factor in this interrupt latency to understand the worst-case scenario.

Low-level interrupt-related operations

When an interrupt occurs, the context must be initially saved and then later restored after the interrupt processing has been completed. The amount of context that needs to be saved depends on how many registers would potentially be modified by the ISR (Interrupt Service Routine).

Enabling the ISR to interact with the RTOS

An ISR will typically interact with an RTOS by making a system call such as a semaphore post. To ensure the ISR function can complete and exit before any context switch to a task is made, the RTOS interrupt dispatcher must disable preemption before calling the ISR function. Once the ISR function completes, preemption is re-enabled and the application will context switch to the highest priority thread that is ready to run. If there is no need for an ISR to make an RTOS system call, the disable/enable kernel preemption operations would again add overhead. It is logical to handle such an ISR outside of the RTOS.

Context switching

When an ISR defers processing to an RTOS task or other thread, a context switch needs to occur for the task to run. Context switching will still typically be the largest part of anyRTOS related interrupt processing overhead.

Embedded Software Testing

Embedded Software Testing is testing of embedded systems. Embedded software testing is similar to other testing types. The embedded software is tested for their performance, consistency and validated as per the requirements of the client of the software development team.

Embedded Software testing checks and ensure the concerned software is of good quality and complies with all the requirements it should meet. Embedded software testing is an excellent approach to guarantee security in critical applications like medical equipment, railways, aviation, vehicle industry, etc. Strict and careful testing is crucial to grant software certification.

How to perform Embedded Software Testing

In general, you test for four reasons:

- To find bugs in software
- Helps to reduce risk to both users and the company
- Cut down development and maintenance costs
- To improve performance

In Embedded Testing, the following activities are performed:

1. The software is provided with some inputs.

2. A Piece of the software is executed.
3. The software state is observed, and the outputs are checked for expected properties like whether the output matches the expected outcome, conformance to the requirements and absence of system crashes.

Embedded Software Testing Types

Fundamentally, there are five levels of testing that can be applied to embedded software

Software Unit Testing

The unit module is either a function or class. Unit Testing is performed by the development team, primarily the developer and is usually carried out in a peer-review model. Based on the specification of the module test cases are developed.

Integration Testing

Integration testing can be classified into two segments:

1. Software integration testing
2. Software/hardware integration testing.

In the end, the interaction of the hardware domain and software components is tested. This can incorporate examining the interaction between built-in peripheral devices and software.

Embedded software development has a unique characteristic which focuses on the actual environment, in which the software is run, is generally created in parallel with the software. This causes inconvenience for testing since comprehensive testing cannot be performed in a simulated condition.

System Unit Testing

Now the module to be tested is a full framework that consists of complete software code additionally all real-time operating system (RTOS) and platform-related pieces such as interrupts, tasking mechanisms, communications and so on. The Point of Control protocol is not anymore a call to a function or a method invocation, but rather a message sent/got utilizing the RTOS message queues.

System resources are observed to evaluate the system's ability to support embedded system execution. For this aspect, gray-box testing is the favored testing method. Depending on the organization, system unit testing is either the duty of the developer or a dedicated system integration team.

System Integration Testing

The module to be tested begins from a set of components within a single node. The Points of Control and Observations (PCOs) are a mix of network related communication protocols and RTOS, such as network messages and RTOS events. Additionally to a component, a Virtual Tester can likewise play the role of a node.

System Validation Testing

The module to be tested is a subsystem with a complete implementation or the complete embedded system. The objective of this final test is to meet external entity functional requirements.

REFERENCE :-

1. www.guru99.com

2. www.nptel.com

3. www.intervalzero.com

4. Embedded Systems – Raj Kamal

5. Embedded Systems - Shivu