EE6502 MICROPROCESSORS AND MICROCONTROLLERS L T P C 3 0 0 3

OBJECTIVES:

- To study the Architecture of uP8085 & uC 8051
- To study the addressing modes & instruction set of 8085 & 8051.
- To introduce the need & use of Interrupt structure 8085 & 8051.
- To develop skill in simple applications development with programming 8085 & 8051
- To introduce commonly used peripheral / interfacing
-

UNIT I 8085 PROCESSOR 9

Hardware Architecture, pinouts – Functional Building Blocks of Processor – Memory organization – I/O ports and data transfer concepts– Timing Diagram – Interrupts.

UNIT II PROGRAMMING OF 8085 PROCESSOR 9

Instruction -format and addressing modes – Assembly language format – Data transfer, data manipulation& control instructions – Programming: Loop structure with counting & Indexing – Look up table - Subroutine instructions - stack.

UNIT III 8051 MICRO CONTROLLER 9

Hardware Architecture, pintouts – Functional Building Blocks of Processor – Memory organization – I/O ports and data transfer concepts– Timing Diagram – Interrupts-Comparison to Programming concepts with 8085.

UNIT IV PERIPHERAL INTERFACING 9

Study on need, Architecture, configuration and interfacing, with ICs: 8255 , 8259 , 8254,8237,8251, 8279 ,- A/D and D/A converters &Interfacing with 8085& 8051.

UNIT V MICRO CONTROLLER PROGRAMMING & APPLICATIONS 9

Data Transfer, Manipulation, Control Algorithms& I/O instructions – Simple programming exercises key board and display interface – Closed loop control of servo motor- stepper motor control – Washing Machine Control.

TOTAL : 45 PERIODS

OUTCOMES:

- Ability to understand and analyse, linear and digital electronic circuits.
- To understand and apply computing platform and software for engineering problems.

TEXT BOOKS:

1. Krishna Kant, "Microprocessor and Microcontrollers", Eastern Company Edition, Prentice Hall of India, New Delhi , 2007.
2. R.S. Gaonkar, 'Microprocessor Architecture Programming and Application', with 8085, Wiley Eastern Ltd., New Delhi, 2013.
3. Soumitra Kumar Mandal, Microprocessor & Microcontroller Architecture, Programming & Interfacing using 8085,8086,8051,McGraw Hill Edu,2013.

REFERENCES:

1. Muhammad Ali Mazidi & Janice Gilli Mazidi, R.D.Kinely 'The 8051 Micro Controller and Embedded Systems', PHI Pearson Education, 5th Indian reprint, 2003.
2. N.Senthil Kumar, M.Saravanan, S.Jeevananthan, 'Microprocessors and Microcontrollers', Oxford,2013.
3. Valder – Perez, "Microcontroller – Fundamentals and Applications with Pic," Yeesdee Publishers, Tayler & Francis, 2013.

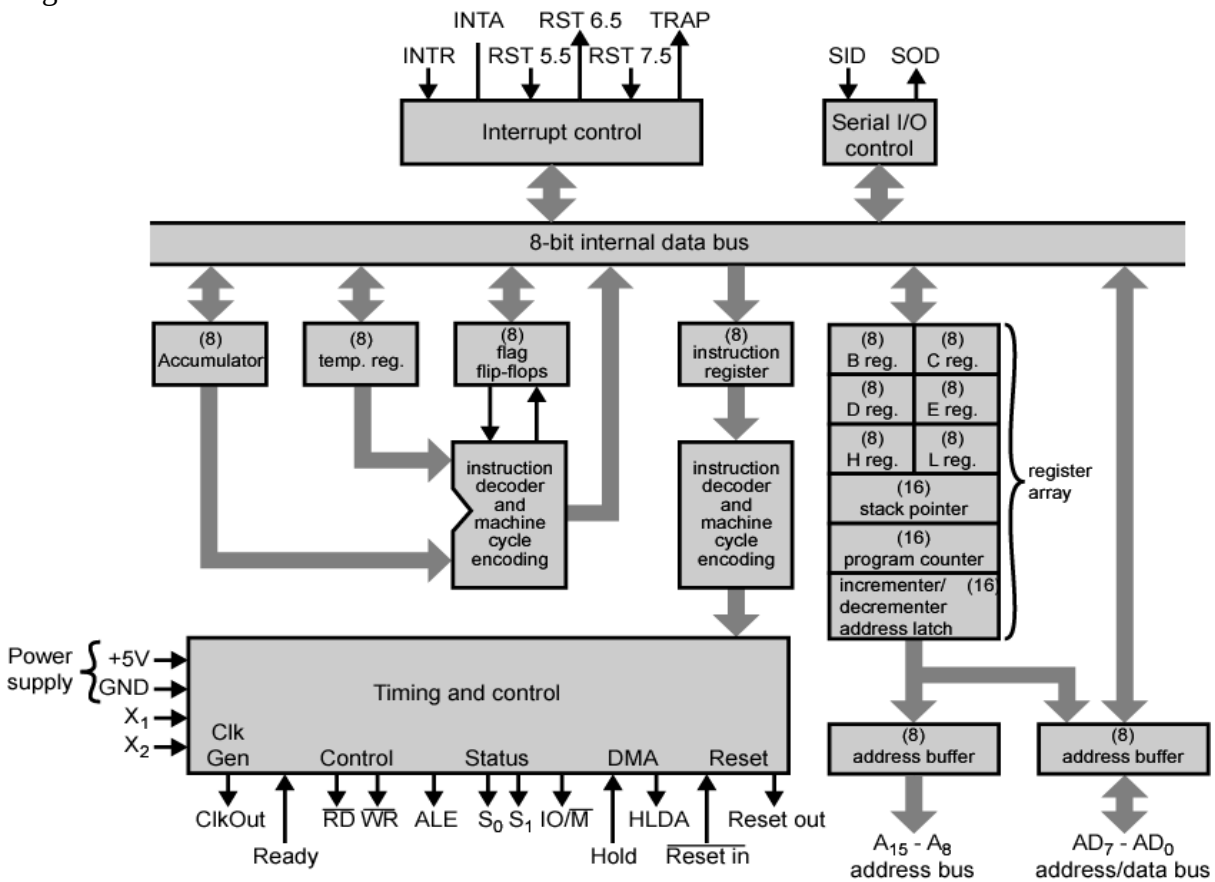
UNIT I

8085 PROCESSOR

1) 8085 Architecture:

The architecture of 8085 consist various components like:

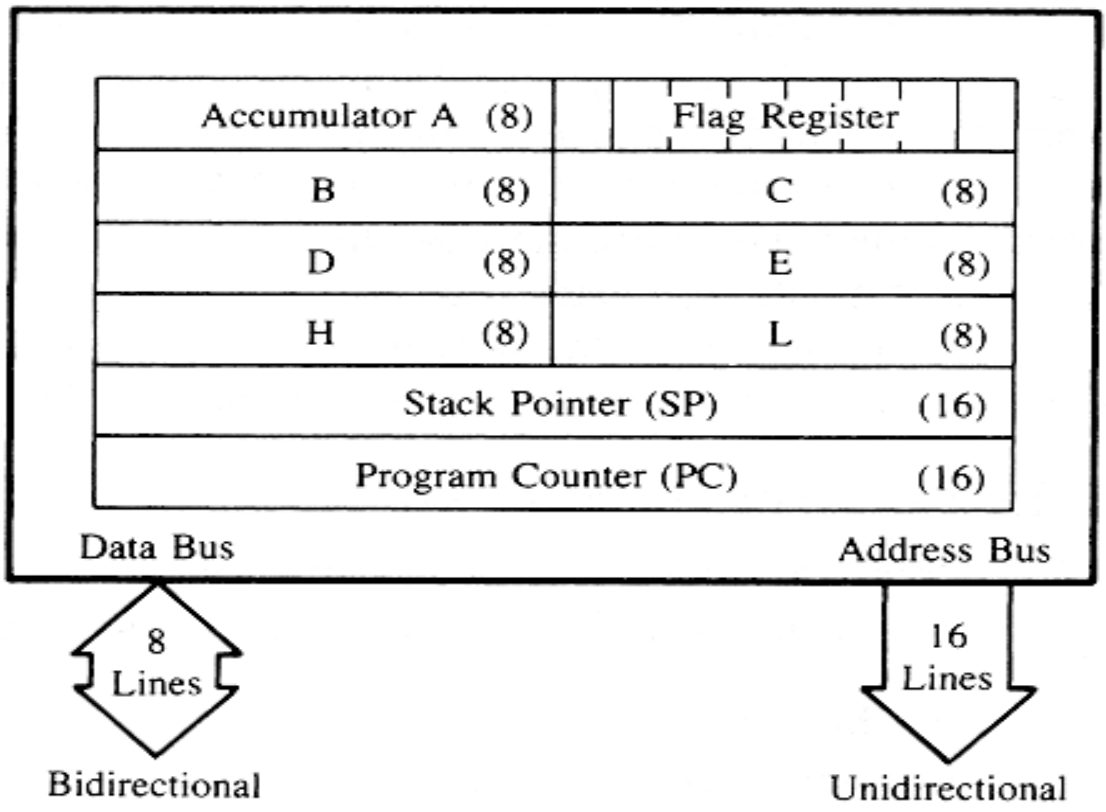
1. Accumulator & Register sets.
2. Program counter and stack pointer.
3. Flag Register.
4. ALU.
5. Instruction decoder and machine cycle encoder.
6. Address buffer.
7. Address/data buffer.
8. Increment/Decrement latch.
9. Interrupt control.
10. Serial I/O like SOD,SID.
11. Timing and Control circuit.



Accumulator:

- The accumulator is an 8-bit register then is part of the arithmetic/logic unit(ALU).
- This register is used to store to store 8-bit data this data is used to perform arithmetic & logical operation.
- The result of an operation is stored in the accumulator.

- The accumulator is also identified as register A.
- The accumulator is used for data transfer between an I/O port and memory location.



Registersets:

- The 8085 simulator has six general-purpose registers to store 8-bit data; these are identified as **B**, **C**, **D**, **E**, **H** and **L**. They can be combined as register pair like**BC**, **DE** and **HL** – to perform some 16-bit operations.
- The programmer can use these registers to store or copy data into the registers by using data copy instructions.
- Out of these six registers, four 8-bit registers are scratch pad registers which are accessible to the programmer and hence can be used to temporarily store data during a program execution.
- And the two registers **H** and **L** are utilized in indirect addressing mode. In this mode, the memory location i.e. the address is specified by the contents of the registers.

Program Counter (PC):

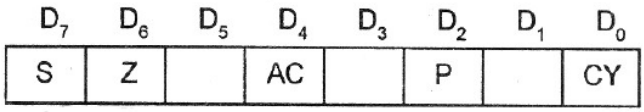
- 16 bit register which holds the memory address of the next instruction to be executed in the next step.
- This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a 16-bit register.
- The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next instruction is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.

Stack Pointer (SP):

- Stack pointer is used during subroutine calling and execution.
- The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack.

Flag or status register:

- The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero(Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags.
- The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.
- For example, after an addition of two numbers, if the sum in the accumulator is larger than eight bits, the flip-flop uses to indicate a carry -- called the Carry flag (CY) -- is set to one.
- When an arithmetic operation results is zero, the flip-flop called the Zero(Z) flag is set to one.
- The Figure shows an 8-bit register, called the flag register, adjacent to the accumulator..



Bit positions of various flags in the flag register of 8085

- **Flag** is an 8-bit register containing 5no.s of 1-bit flags:

1. Sign - If the result of the latest arithmetic operation is having MSB (most- significant bit) '1' (meaning it is a negative number), then the sign flag is set. Otherwise, it is reset to '0' which means it is a positive number.
2. Zero - If the result of the latest operation is zero, then zero flag will be set; otherwise it be reset.
3. Auxiliary carry - set if there was a carry out from bit 3 to bit 4 of the result.
4. Parity - set if the parity (the number of set bits in the result) is even. i.e., If the result of the latest operation is having even number of '1's, then this flag will be set. Otherwise this will be reset to '0'. This is used for error checking.
5. Carry - set if there was a carry during addition, or borrow during subtraction/comparison. Otherwise it will be reset.

Instruction register or Decoder:-

- Instruction register holds the instruction that is currently being processed.
- Once the instruction is fetch from the memory, it is reloaded in the instruction register for some time, after the decoder decode the instruction performing some event or task.

Address buffer:

- The remaining higher order address lines form the address buffer ranging from[A15-A8].This is having the unidirectional buffer

Address/data buffer:

- The address bus will be having 16 address lines[A15-A0] .In which A7-A0 are called as lower addressing lines and these are multiplexed with data lines[D7-D0] to form multiplexed address /data buffer .The address/data buffer is the bidirectional bus.

Increment/Decrement Address Latch:

- It increments/ decrements the address before sent to the address buffer

Interrupts:

The processor has 5 interrupts. They are presented below in the order of theirpriority (from lowest to highest):

- **INTR** is maskable 8080A compatible interrupt. When the interrupt occurs, the processor fetches from the bus one instruction, usually one of these instructions: One of the 8 RST instructions (RST0 - RST7). The processor saves current program counter into stack and branches to memory location N * 8 (where N is a 3-bit number from 0 to 7 supplied with the RST instruction).
- CALL instruction (3 byte instruction). The processor calls the subroutine, address of which is specified in the second and third bytes of the instruction.
- **RST5.5** is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 2CH (hexadecimal) address.
- **RST6.5** is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 34H (hexadecimal) address.
- **RST7.5** is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 3CH (hexadecimal) address.
- **TRAP** is a non-maskable interrupt. When this interrupt is received the processorsaves the contents of the PC register into stack and branches to 24H (hexadecimal) address.
- All maskable interrupts can be enabled or disabled using EI and DI instructions. RST 5.5, RST6.5 and RST7.5 interrupts can be enabled or disabled individually using SIM Instruction

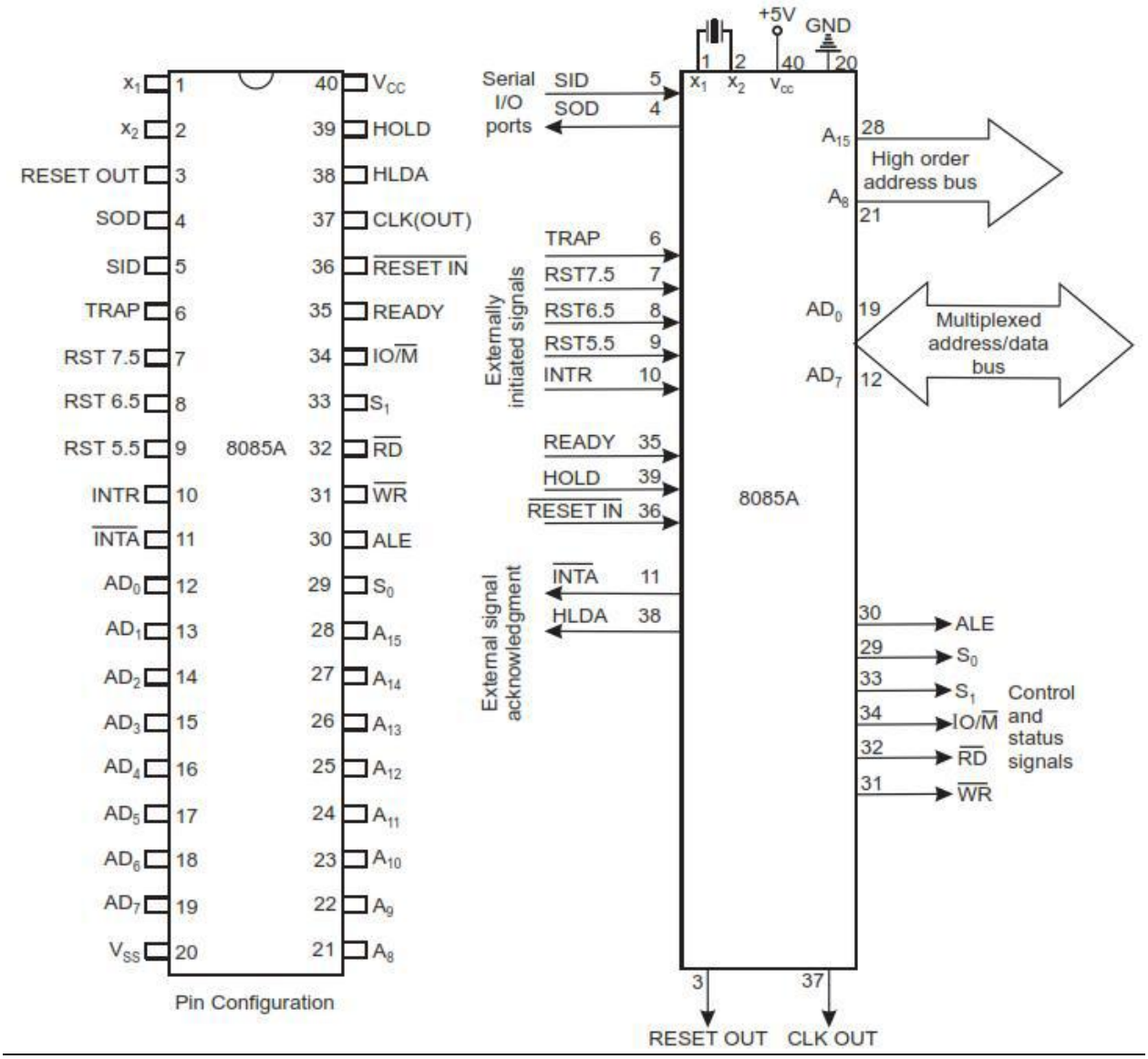
Serial I/O control

- There are control signals used for controlling 8085 these are subdivided into 2 types:

    1. SID(serial input data):
       This is used for transferring of data into the memory serially.
    2. SOD(serial output data):
       This is used for transferring of data from memory to external devices

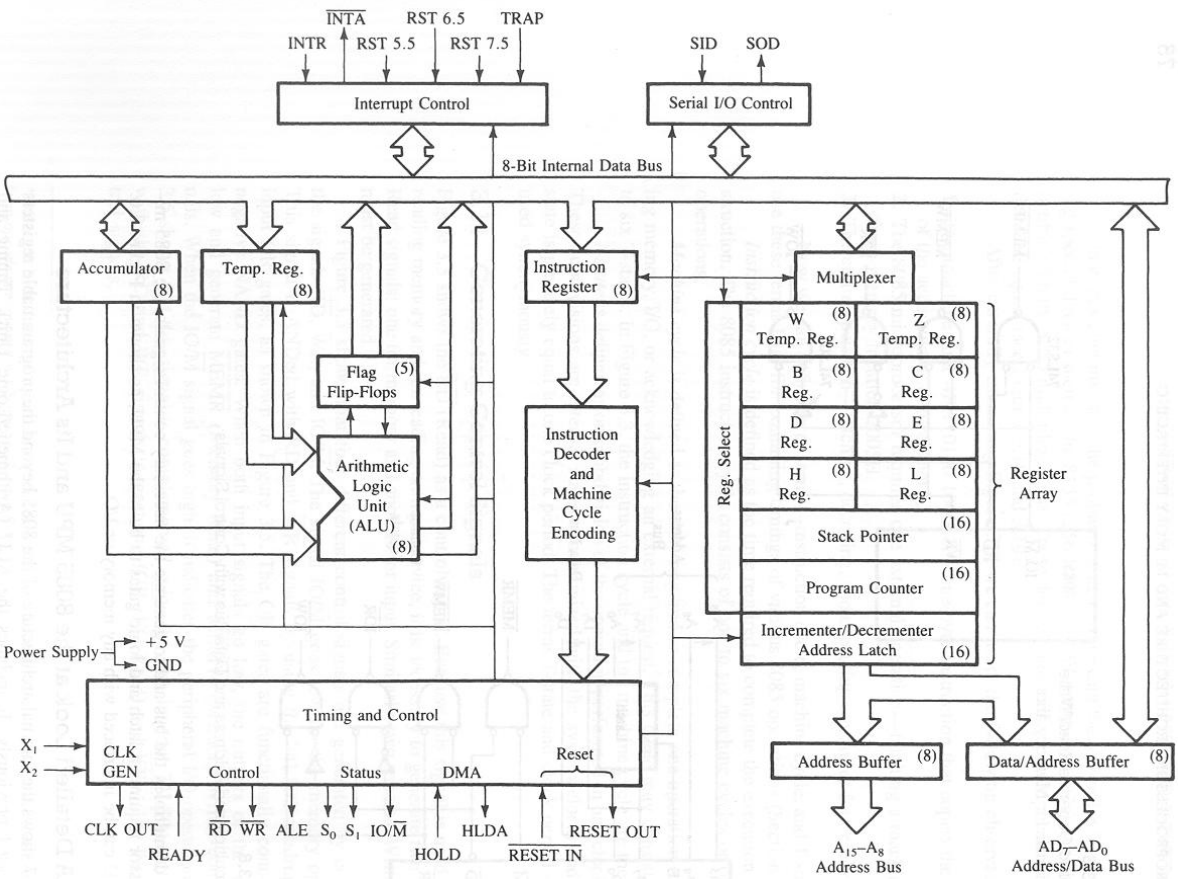
- Interrupt control is used to transfer the ISR to the CPU.

2) PIN DIAGRAM



Timing and Control Unit:

- The timing and control unit accepts information from the instruction decoder andgenerates different control signal. This unit synchronizes all the microprocessor operation and generates control and status signal necessary for communication between the microprocessor and peripherals.



A8 - A15 (Output 3 State):

Address Bus: The most significant 8 bits of the memory address or the 8 bits of the I/0 address,3 stated during Hold and Halt modes.

AD0 - AD7 (Input / Output 3 state):

- ✓ Multiplexed Address/Data Bus; Lower 8 bits of the memory address (or I/0 address) appear on the bus during the first clock cycle of a machine state.
- ✓ It then becomes the data bus during the second and third clock cycles. 3 stated during Hold and Halt modes.

ALE (Output):

- ✓ Address Latch Enable: It occurs during the first clock cycle of a machine state and enables the address to get latched into the on chip latch of peripherals.
- ✓ The falling edge of ALE is set to guarantee setup and hold times for the address information. ALE can also be used to strobe the status information. ALE is never 3stated.

SO, S1 (Output):

Data Bus Status. Encoded status of the bus cycle:

S1 S0
0 0 HALT
0 1 WRITE
1 0 READ
1 1 FETCH S1 can be used as an advanced R/W status.

RD (Output 3state):

READ: indicates the selected memory or I/0 device is to be read and that the Data Bus is available for the data transfer.

WR (Output 3state):

- ✓ WRITE: indicates the data on the Data Bus is to be written into the selected memory or I/0location.
- ✓ Data is set up at the trailing edge of WR. 3stated during Hold and Halt modes.

READY (Input):

- ✓ If Ready is high during a read or write cycle, it indicates that the memory or peripheral is readyto send or receive data.
- ✓ If Ready is low, the CPU will wait for Ready to go high before completing the read or write cycle.

HOLD (Input):

- ✓ HOLD: indicates that another Master is requesting the use of the Address and Data Buses.
- ✓ The CPU, upon receiving the Hold request will relinquish the use of buses as soon as the completion of the current machine cycle.
- ✓ Internal processing can continue. The processor can regain the buses only after the Hold is removed. When the Hold is acknowledged, the Address, Data, RD, WR, and IO/M lines are 3stated.

HLDA (Output):

- ✓ HOLD ACKNOWLEDGE: indicates that the CPU has received the Hold request and that it willrelinquish the buses in the next clock cycle.
- ✓ HLDA goes low after the Hold request is removed. The CPU takes the buses one half clock cycle after HLDA goes low.

INTR (Input):

- ✓ INTERRUPT REQUEST is used as a general purpose interrupt. It is sampled only during thenext to the last clock cycle of the instruction. If it is active, the Program Counter (PC) will be inhibited from incrementing and an INTA will be issued.
- ✓ During this cycle a RESTART or CALL instruction can be inserted to jump to the interrupt service routine. The INTR is enabled and disabled by software. It is disabled by Reset and immediately after an interrupt is accepted.

INTA (Output):

- ✓ INTERRUPT ACKNOWLEDGE: is used instead of (and has the same timing as) RD during theInstruction cycle after an INTR is accepted.
- ✓ It can be used to activate the 8259 Interrupt chip or some other interrupt port.

RESTART INTERRUPTS:

These three inputs have the same timing as INTR except they cause an internal RESTART to be automatically inserted.

- ➢ RST 7.5 Highest Priority
- ➢ RST 6.5
- ➢ RST 5.5 Lowest Priority

TRAP (Input):

- ✓ Trap interrupt is a non maskable restart interrupt. It is recognized at the same time as INTR. It is unaffected by any mask or Interrupt Enable. It has the highest priority of any interrupt.

RESET IN (Input):

- ✓ Reset sets the Program Counter to zero and resets the Interrupt Enable and HLDA flipflops.
- ✓ None of the other flags or registers (except the instruction register) are affected The CPU is held in the reset condition as long as Reset is applied.

RESET OUT (Output):

Indicates CPlJ is being reset. Can be used as a system RESET. The signal is synchronized to the processor clock.

X1, X2 (Input):

- ✓ Crystal or R/C network connections to set the internal clock generator X1 can also be an external clock input instead of a crystal. The input frequency is divided by 2 to give the internal operating frequency.

CLK (Output):

- ✓ Clock Output for use as a system clock when a crystal or R/ C network is used as an input to the CPU. The period of CLK is twice the X1, X2 input period.

IO/M (Output):

- ✓ IO/M indicates whether the Read/Write is to memory or l/O Tristated during Hold and Halt modes.

SID (Input):

- ✓ Serial input data line The data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.

SOD (output):

- ✓ Serial output data line. The output SOD is set or reset as specified by the SIM instruction.

Vcc: +5 volt supply.
Vss: Ground Reference.

3) MEMORY ORGANIZATION

Memory Interfacing

The memory is made up of semiconductor material used to store the programs and data. Three types of memory is,

- Process memory
- Primary or main memory
- Secondary memory

Typical EPROM and Static RAM

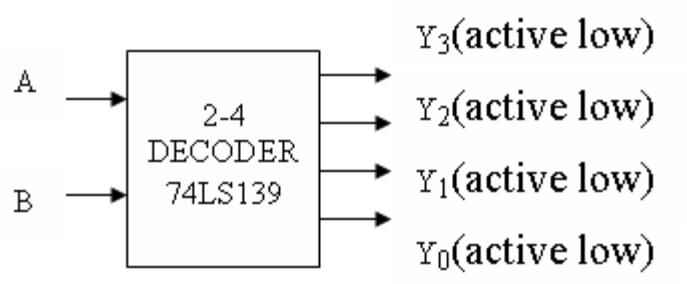
A typical semiconductor memory IC will have n address pins, m data pins (or output pins).

- Having two power supply pins (one for connecting required supply voltage (V and the other for connecting ground).
- The control signals needed for static RAM are chip select (chip enable), read control (output enable) and write control (write enable).
- The control signals needed for read operation in EPROM are chip select (chip enable) and read control (output enable).

Decoder

It is used to select the memory chip of processor during the execution of a program. No of IC's used for decoder is,

- 2-4 decoder (74LS139)
- 3-8 decoder (74LS138)



Block Diagram of 2-4 Decoder

Truth Table for 2-4 Decoder

Input		Output			
B	A	$\overline{Y}_3$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	1

Example for Memory Interfacing

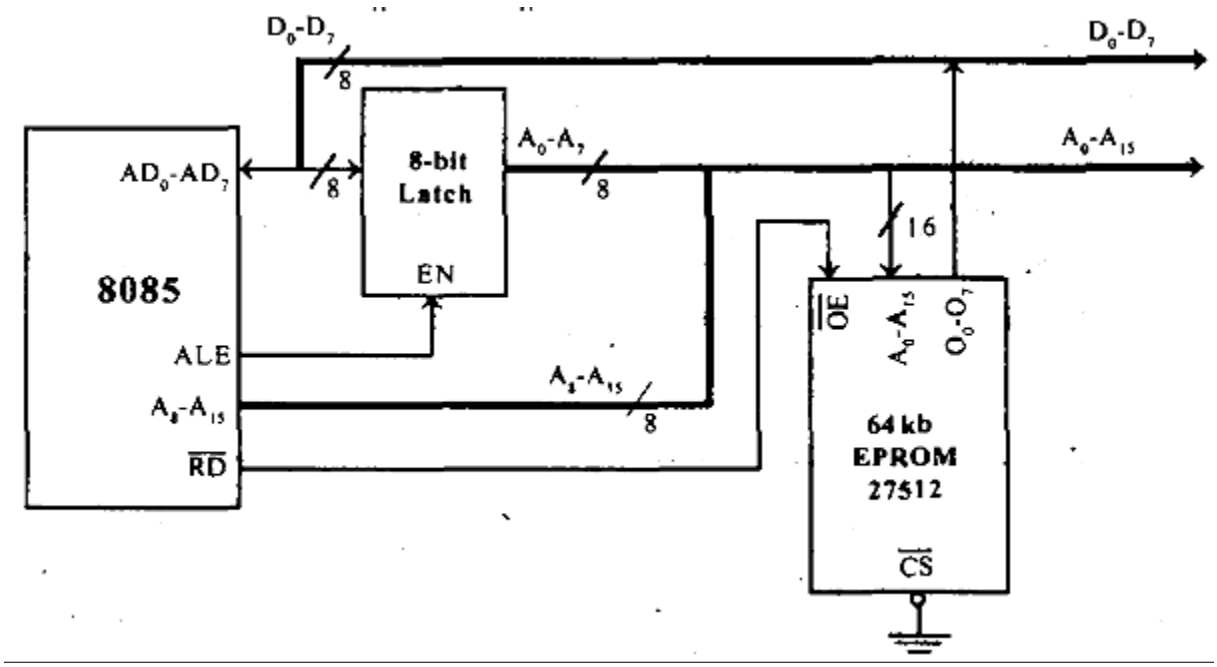
Consider a system in which the full memory space 64kb is utilized for EPROM memory. Interface the EPROM with 8085 processor.

- The memory capacity is 64 Kbytes. i.e
- 2^n = 64 x 1000 bytes where n = address lines.

- So, n = 16.

- In this system the entire 16 address lines of the processor are connected to address input pins of memory IC in order to address the internal locations of memory.
- The chip select (CS) pin of EPROM is permanently tied to logic low (i.e., tied to ground).

- Since the processor is connected to EPROM, the active low RD pin is connected to active low output enable pin of EPROM.
- The range of address for EPROM is 0000H to FFFFH.



Memory Interfacing

4) **TIMING DIAGRAM**

Timing Diagram is a graphical representation. It represents the execution time taken by each instruction in a graphical format. The execution time is represented in T-states.

- **Instruction Cycle**

The time required to execute an instruction is called instruction cycle.

- **Machine Cycle**

The time required to access the memory or input/output devices is called machine cycle.

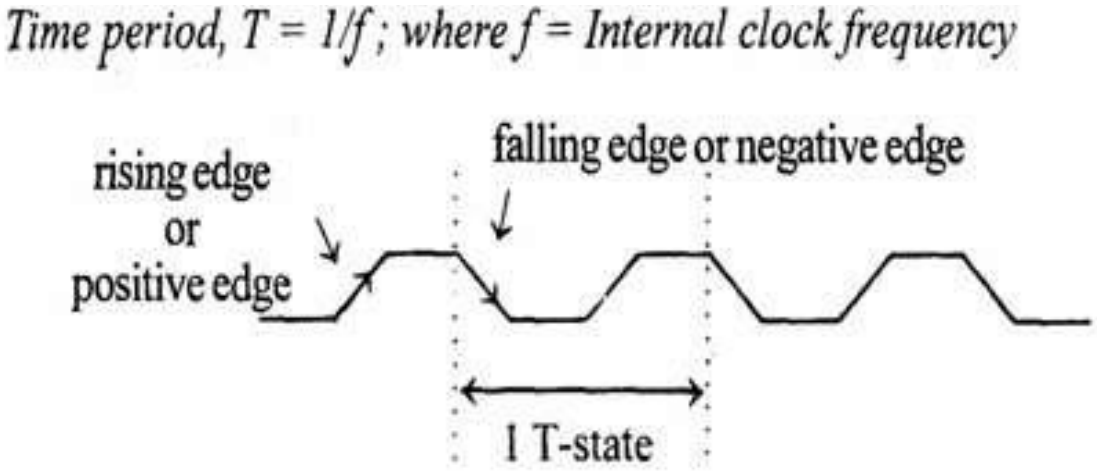
- **T-State**
  - The machine cycle and instruction cycle takes multiple clock periods.
  - A portion of an operation carried out in one system clock period is called as T-state.

Machine cycles of 8085

The 8085 microprocessor has 5 (seven) basic machine cycles. They are

1. Opcode fetch cycle (4T)

2. Memory read cycle (3 T)
3. Memory write cycle (3 T)
4. I/O read cycle (3 T)
5. I/O write cycle (3 T)

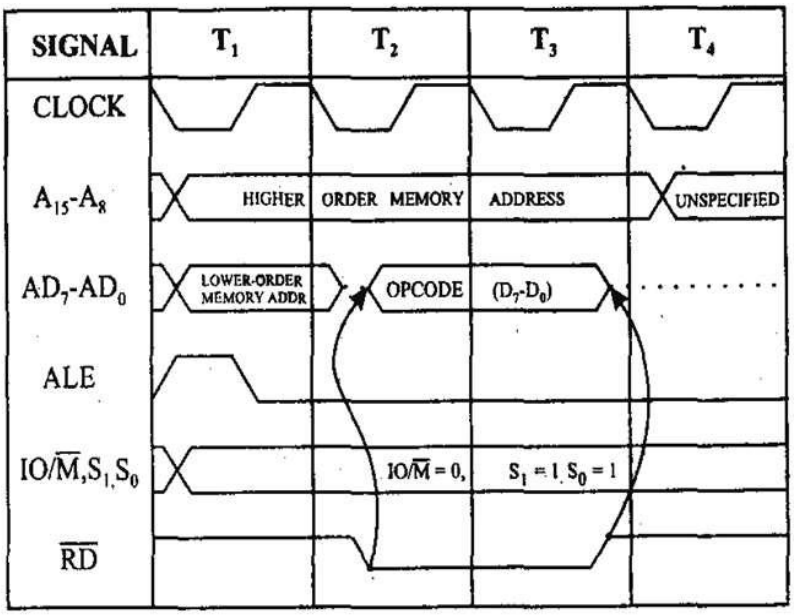


Clock Signal

1.Opcode fetch machine cycle of 8085 :

- Each instruction of the processor has one byte opcode.
- The opcodes are stored in memory. So, the processor executes the opcode fetch machine cycle to fetch the opcode from memory.
- Hence, every instruction starts with opcode fetch machine cycle.

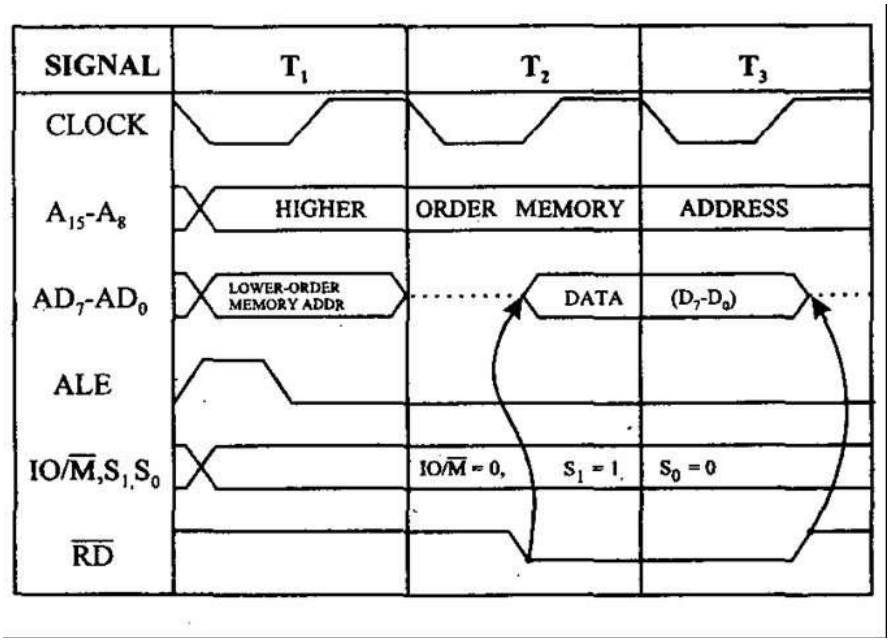
- The time taken by the processor to execute the opcode fetch cycle is 4T.
- In this time, the first, 3 T-states are used for fetching the opcode from memory and the remaining T-states are used for internal operations by the processor.



2.Memory Read Machine Cycle of 8085:

- The memory read machine cycle is executed by the processor to read a data byte from memory.
- The processor takes 3T states to execute this cycle.

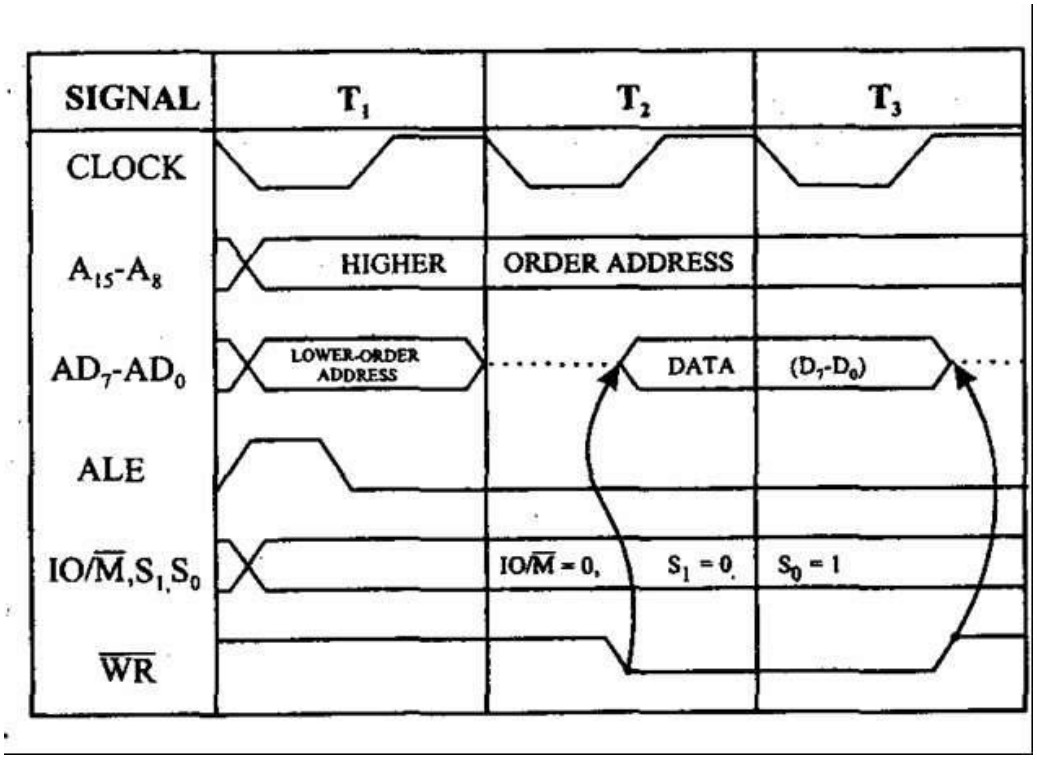
The instructions which have more than one byte word size will use the machine cycle after the opcode fetch machine cycle.



Memory Read Machine Cycle

3.Memory Write Machine Cycle of 8085

- The memory write machine cycle is executed by the processor to write a data byte in a memory location.
- The processor takes, 3T states to execute this machine cycle.

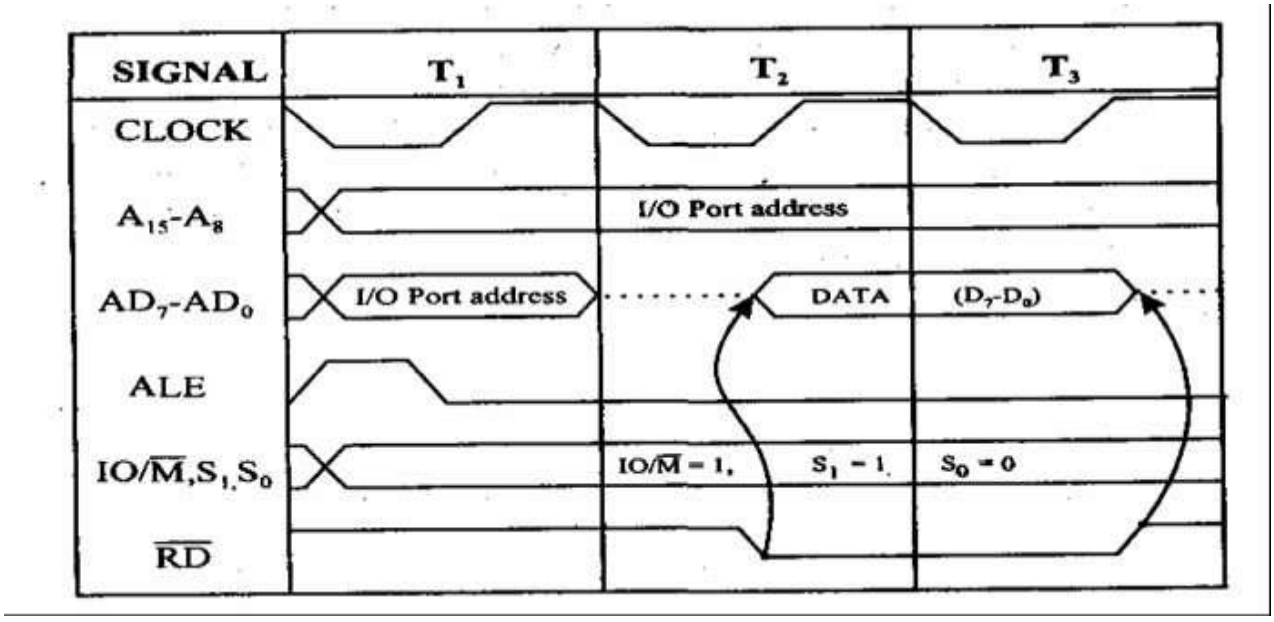


Memory Write Machine Cycle

4. I/O Read Cycle of 8085

- The I/O Read cycle is executed by the processor to read a data byte from I/O port or from the peripheral, which is I/O, mapped in the system.
- The processor takes 3T states to execute this machine cycle.

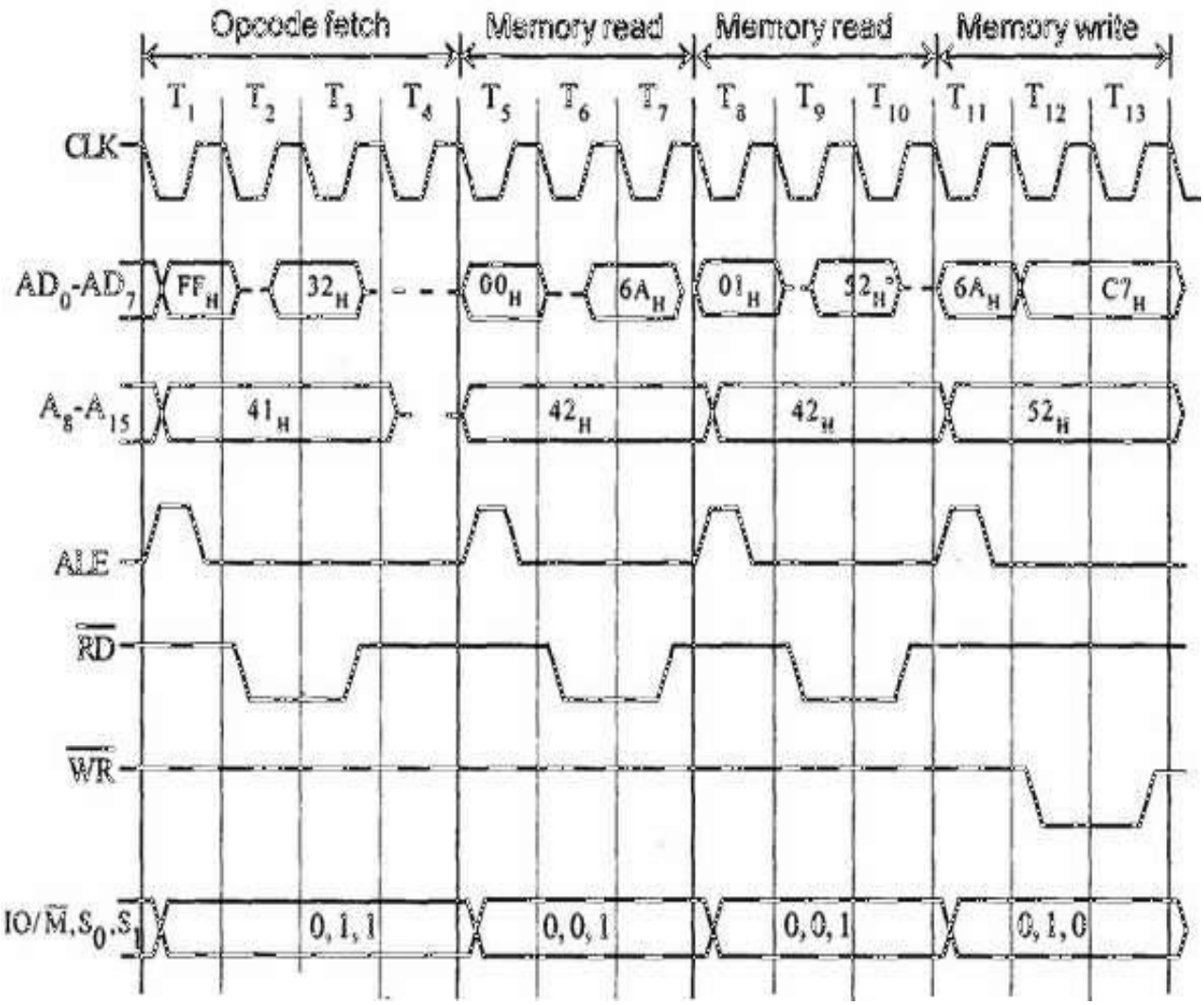
- The IN instruction uses this machine cycle during the execution.



I/O Read Cycle

Timing diagram for STA 526AH

- STA means Store Accumulator -The contents of the accumulator is stored in the specified address(526A).
- The opcode of the STA instruction is said to be 32H. It is fetched from the memory 41FFH(see fig). - OF machine cycle
- Then the lower order memory address is read(6A). - Memory Read Machine Cycle
- Read the higher order memory address (52).- Memory Read Machine Cycle
- The combination of both the addresses are considered and the content from accumulator is written in 526A. - Memory Write Machine Cycle
- Assume the memory address for the instruction and let the content of accumulator is C7H. So, C7H from accumulator is now stored in 526A.



Timing Diagram for STA 526A H

Timing diagram for INR M

- Fetching the Opcode 34H from the memory 4105H. (OF cycle)
- Let the memory address (M) be 4250H. (MR cycle -To read Memory address and data)
- Let the content of that memory is 12H.
- Increment the memory content from 12H to 13H. (MW machine cycle)

5) **INTERRUPTS:**

- Interrupt is signals send by an external device to the processor, to request the processor to perform a particular task or work.

- Mainly in the microprocessor based system the interrupts are used for data transfer between the peripheral and the microprocessor.

- The processor will check the interrupts always at the 2nd T-state of last machine cycle.

- If there is any interrupt it accept the interrupt and send the INTA (active low) signal to the peripheral.

- The vectored address of particular interrupt is stored in program counter.

- The processor executes an interrupt service routine (ISR) addressed in program counter.

- It returned to main program by RET instruction.

Types of Interrupts:

It supports two types of interrupts.

- Hardware

- Software

Software interrupts:

- The software interrupts are program instructions. These instructions are inserted at desired locations in a program.

- The 8085 has eight software interrupts from RST 0 to RST 7. The vector address for these interrupts can be calculated as follows.

- Interrupt number * 8 = vector address

- For RST 5,5 * 8 = 40 = 28H

Vector addresses of all interrupts.

Interrupt	Vector address
RST 0 RST 1	0000_H 0008_H
RST 2 RST 3	0010_H 0018_H
RST 4 RST 5	0020_H 0028_H
RST 6 RST 7	0030_H 0038_H

Hardware interrupts:

- An external device initiates the hardware interrupts and placing an appropriate signal at the interrupt pin of the processor.

- If the interrupt is accepted then the processor executes an interrupt service routine.

The 8085 has five hardware interrupts

(1) TRAP (2) RST 7.5 (3) RST 6.5 (4) RST 5.5 (5) INTR

(1)TRAP:

- This interrupt is a non-maskable interrupt. It is unaffected by any mask or interrupt enable.

- TRAP bas the highest priority and vectored interrupt.

- TRAP interrupt is edge and level triggered. This means hat the TRAP must go high and remain high until it is acknowledged.

- In sudden power failure, it executes a ISR and send the data from main memory to backup memory.
- The signal, which overrides the TRAP, is HOLD signal. (i.e., If the processor receives HOLD and TRAP at the same time then HOLD is recognized first and then TRAP is recognized).
- There are two ways to clear TRAP interrupt.

1.By resetting microprocessor (External signal)

2.By giving a high TRAP ACKNOWLEDGE (Internal signal)

(2)RST 7.5:

- The RST 7.5 interrupt is a maskable interrupt.
- It has the second highest priority.
- It is edge sensitive. ie. Input goes to high and no need to maintain high state until it recognized.
- Maskable interrupt. It is disabled by,

1.DI instruction
2.System or processor reset.
3.After reorganization of interrupt.

- Enabled by EI instruction**.**

(3)RST 6.5 and 5.5:

- The RST 6.5 and RST 5.5 both are level triggered. . ie. Input goes to high and stay high until it recognized.
- Maskable interrupt. It is disabled by,

1.DI, SIM instruction

2.System or processor reset.

3.After reorganization of interrupt.

- Enabled by EI instruction.

- The RST 6.5 has the third priority whereas RST 5.5 has the fourth priority.

- INTR is a maskable interrupt. It is disabled by,

  1.DI, SIM instruction
  2.System or processor reset.
  3.After reorganization of interrupt

- Enabled by EI instruction.

- Non- vectored interrupt. After receiving INTA (active low) signal, it has to supply the address of ISR.

- It has lowest priority.

- It is a level sensitive interrupts. ie. Input goes to high and it is necessary to maintain high state until it recognized.

The following sequence of events occurs when INTR signal goes high.

1. The 8085 checks the status of INTR signal during execution of each instruction.

2. If INTR signal is high, then 8085 complete its current instruction and sends active low interrupt acknowledge signal, if the interrupt is enabled.

3. In response to the acknowledge signal, external logic places an instruction OPCODE on the data bus. In the case of multibyte instruction, additional interrupt acknowledge machine cycles are generated by the 8085 to transfer the additional bytes into the microprocessor.

4. On receiving the instruction, the 8085 save the address of next instruction on

stack and execute received instruction.

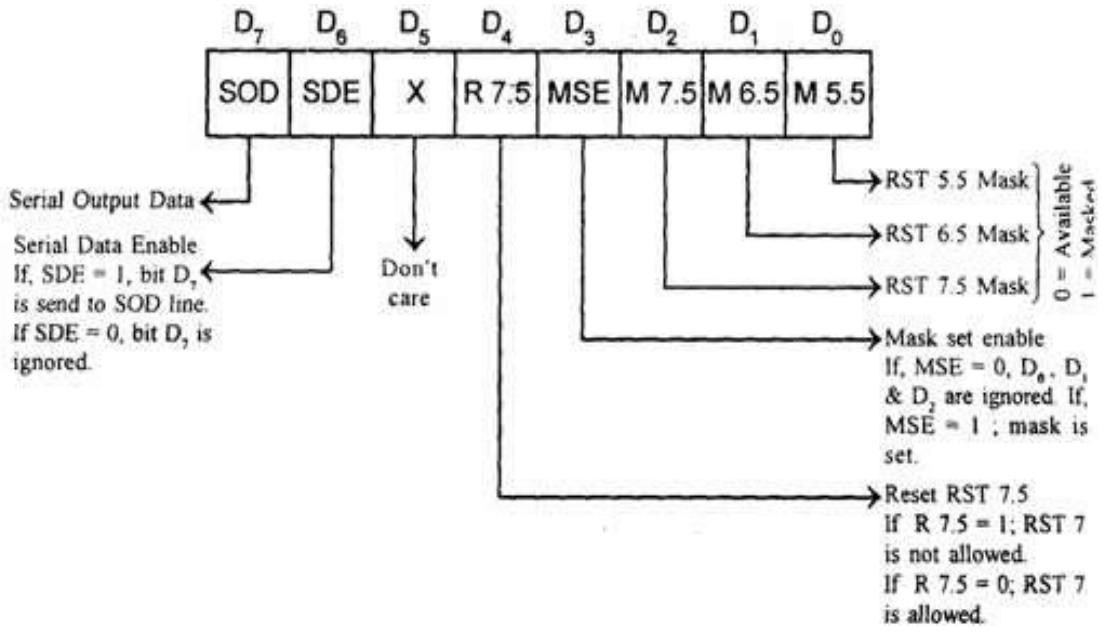
SIM and RIM for interrupts:

 The 8085 provide additional masking facility for RST 7.5, RST 6.5 and RST 5.5 using

SIM instruction.

 The status of these interrupts can be read by executing RIM instruction.

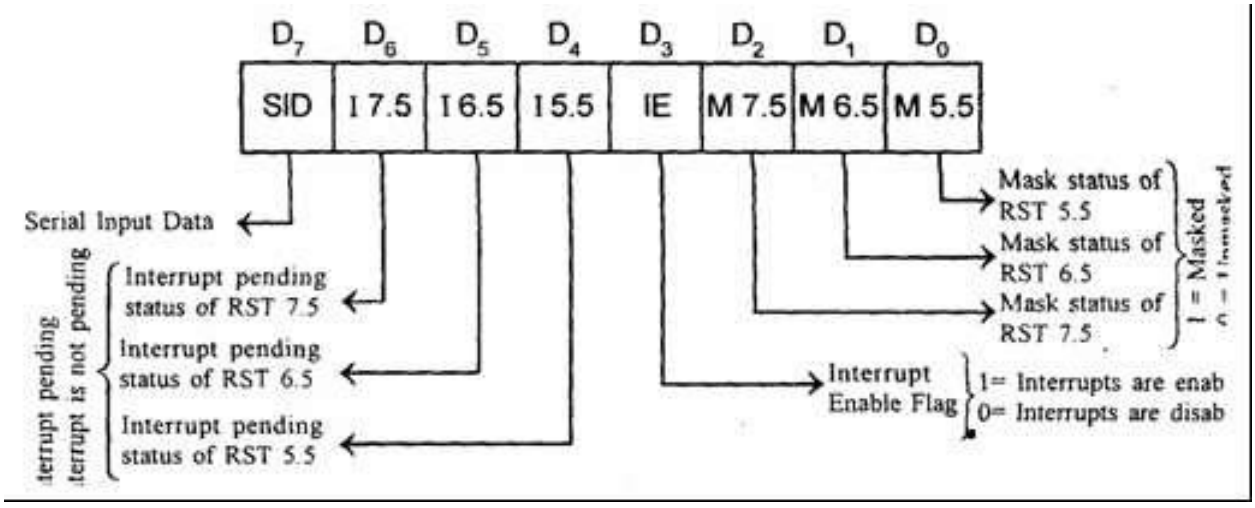
 The masking or unmasking of RST 7.5, RST 6.5 and RST 5.5 interrupts can be performed by moving an 8-bit data to accumulator and then executing SIM instruction.



8 bit data to be loaded into the Accumulator

 The status of pending interrupts can be read from accumulator after executing RIM instruction.

 When RIM instruction is executed an 8-bit data is loaded in accumulator, which can be interpreted as shown in fig.



Format of 8 bit data in Accumulator after executing RIM Instruction

8085 Bus Structure

• Address Bus :

- Consists of 16 address lines: A0 – A15
- Address locations: 0000 (hex) – FFFF (hex)
- Can access 64K (= 216) bytes of memory, each byte has 8 bits
- Can access 64K × 8 bits of memory
- Use memory to map I/O, Same instructions to use for
- accessing I/O devices and memory

• Data Bus :

- Consists of 8 data lines: D0 – D7
- Operates in bidirectional mode
- The data bits are sent from the MPU to I/O & vice versa
- Data range: 00 (hex) – FF (hex)

• Control Bus:

- – Consists of various lines carrying the control signals such as read / write enable, flag bits.

UNIT II

PROGRAMMING OF 8085 MICROPROCESSOR

1)INSTRUCTION FORMAT

An **instruction** is a command to the microprocessor to perform a given task on a specified data. Each instruction has two parts: one is task to be performed, called the **operation code** (opcode), and the second is the data to be operated on, called the **operand.** The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

Instruction word size

The 8085 instruction set is classified into the following three groups according to word size:

1. One-word or 1-byte instructions
2. Two-word or 2-byte instructions
3. Three-word or 3-byte instructions

In the 8085, "byte" and "word" are synonymous because it is an 8-bit microprocessor. However, instructions are commonly referred to in terms of bytes rather than words.

2.1.1 One-Byte Instructions

A 1-byte instruction includes the opcode and operand in the same byte. Operand(s) are internal register and are coded into the instruction

Task	Op code	Operand	Binary Code	Hex Code
Copy the contents of the accumulator in the register C.	MOV	C,A	0100 1111	4FH
Add the contents of register B to the contents of the accumulator.	ADD	B	1000 0000	80H
Invert (compliment) each bit in the accumulator.	CMA		0010 1111	2FH

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bit binary format in memory; each requires one memory location.

MOV rd, rs

rd <-- rs copies contents of rs into rd.

Coded as 01 ddd sss where ddd is a code for one of the 7 general registers which is

the destination of the data, sss is the code of the source register.

Example: MOV A,B

Coded as 01111000 = 78H = 170 octal (octal was used extensively in instruction design of such processors).

ADD r

A <-- A + r

Two-Byte Instructions

In a two-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. Source operand is a data byte immediately following the opcode. For example

Task	Opcode	Operand	Binary Code	Hex Code	
Load an 8-bit data byte in the accumulator.	MVI	A, Data	0011 1110 DATA	3E Data	First Byte Second Byte

The instruction would require two memory locations to store in memory.

MVI r,data

r <-- data

Example: MVI A,30H coded as 3EH 30H as two contiguous bytes. This is an example of immediate addressing.

ADI data

A <-- A + data

OUT port

0011 1110

DATA

where port is an 8-bit device address. (Port) <-- A. Since the byte is not the data but points directly to where it is located this is called direct addressing.

Three-Byte Instructions

In a three-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address.

opcode + data byte + data byte

Task	Opcode	Operand	Binary code	Hex Code	
Transfer the program sequence to the memory location 2085H.	JMP	2085H	1100 0011 1000 0101 0010 0000	C3 85 20	First byte Second Byte Third Byte

This instruction would require three memory locations to store in memory.

Three byte instructions - opcode + data byte + data byte

LXI rp, data16

rp is one of the pairs of registers BC, DE, HL used as 16-bit registers. The two data bytes are 16-bit data in L H order of significance.

rp <-- data16

LXI H,0520H coded as 21H 20H 50H in three bytes. This is also immediate addressing.

LDA addr

A <-- (addr) Addr is a 16-bit address in L H order.

Example: LDA 2134H coded as

3AH 34H 21H. This is also an example of direct addressing.

2) THE 8085 ADDRESSING MODES

The instructions MOV B, A or MVI A, 82H are to copy data from a source into a destination. In these instructions the source can be a register, an input port, or an 8-bit number (00H to FFH). Similarly, a destination can be a register or an output port. The sources and destination are operands. The various formats for specifying operands are called the ADDRESSING MODES. For 8085, they are:

1. Immediate addressing.
2. Register addressing.
3. Direct addressing.
4. Indirect addressing.

(1)Immediate addressing

Data is present in the instruction. Load the immediate data to the destination provided.

Example: MVI R,data

(2)Register addressing

Data is provided through the registers.

Example: MOV Rd, Rs

(3)Direct addressing

Used to accept data from outside devices to store in the accumulator or send the data stored in the accumulator to the outside device. Accept the data from the port 00H and store them into the accumulator or Send the data from the accumulator to the port 01H.

Example: IN 00H or OUT 01H

(4)Indirect Addressing

This means that the Effective Address is calculated by the processor. And the contents of the address (and the one following) is used to form a second address. The second address is where the data is stored. Note that this requires several memory accesses; two accesses to retrieve the 16-bit address and a further access (or accesses) to retrieve the data which is to be loaded into the register.

3) INSTRUCTION SET CLASSIFICATION

An **instruction** is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the **instruction set**, determines what functions the microprocessor can perform. These instructions can be classified into the following five functional categories: data transfer (copy) operations, arithmetic operations, logical operations, branching operations, and machine-control operations.

Data Transfer Croup

The data transfer instructions move data between registers or between memory and registers.

MOV Move

MVI Move Immediate

LDA Load Accumulator Directly from Memory

STA Store Accumulator Directly in Memory

LHLD Load H & L Registers Directly from Memory

SHLD Store H & L Registers Directly in Memory

An 'X' in the name of a data transfer instruction implies that it deals with a register pair (16-bits);

LXI Load Register Pair with Immediate data

LDAX Load Accumulator from Address in Register Pair

STAX Store Accumulator in Address in Register Pair

XCHG Exchange H & L with D & E

XTHL Exchange Top of Stack with H & L

Arithmetic Group

The arithmetic instructions add, subtract, increment, or decrement data in registers or memory.

ADD Add to Accumulator

ADI Add Immediate Data to Accumulator

ADC Add to Accumulator Using Carry Flag

ACI Add Immediate data to Accumulator Using Carry

SUB Subtract from Accumulator

SUI Subtract Immediate Data from Accumulator

SBB Subtract from Accumulator Using Borrow (Carry) Flag

SBI Subtract Immediate from Accumulator Using Borrow (Carry) Flag

INR Increment Specified Byte by One

DCR Decrement Specified Byte by One

INX Increment Register Pair by One

DCX Decrement Register Pair by One

DAD Double Register Add; Add Content of Register

Pair to H & L Register Pair

Logical Group

This group performs logical (Boolean) operations on data in registers and memory and on

condition flags.

The logical AND, OR, and Exclusive OR instructions enable you to set specific bits in the

accumulator ON or OFF.

ANA Logical AND with Accumulator

ANI Logical AND with Accumulator Using Immediate Data

ORA Logical OR with Accumulator

OR Logical OR with Accumulator Using Immediate Data

XRA Exclusive Logical OR with Accumulator

XRI Exclusive OR Using Immediate Data

The Compare instructions compare the content of an 8-bit value with the contents of the

accumulator;

CMP Compare

CPI Compare Using Immediate Data

The rotate instructions shift the contents of the accumulator one bit position to the left or right:

RLC Rotate Accumulator Left

RRC Rotate Accumulator Right

RAL Rotate Left Through Carry

RAR Rotate Right Through Carry

Complement and carry flag instructions:

CMA Complement Accumulator

CMC Complement Carry Flag

STC Set Carry Flag

Branch Group

The branching instructions alter normal sequential program flow, either unconditionally or

conditionally. The unconditional branching instructions are as follows:

JMP Jump

CALL Call

RET Return

Conditional branching instructions examine the status of one of four condition flags to determine

whether the specified branch is to be executed. The conditions that may be specified are as

follows:

NZ Not Zero (Z = 0)

Z Zero (Z = 1)

NC No Carry (C = 0)

C Carry (C = 1)

PO Parity Odd (P = 0)

PE Parity Even (P = 1)

P Plus (S = 0)

M Minus (S = 1)

Thus, the conditional branching instructions are specified as follows:

Jumps Calls Returns

C CC RC (Carry)

INC CNC RNC (No Carry)

JZ CZ RZ (Zero)

JNZ CNZ RNZ (Not Zero)

JP CP RP (Plus)

JM CM RM (Minus)

JPE CPE RPE (Parity Even)

JP0 CPO RPO (Parity Odd)

Two other instructions can affect a branch by replacing the contents or the program counter:

PCHL Move H & L to Program Counter

RST Special Restart Instruction Used

with Interrupts

POP Pop Two Bytes of Data off the Stack
XTHL Exchange Top of Stack with H & L
SPHL Move content of H & L to Stack Pointer

I/0 instructions

IN Initiate Input Operation
OUT Initiate Output Operation

Machine Control instructions

EI Enable Interrupt System
DI Disable Interrupt System
HLT Halt
NOP No Operation

SAMPLE PROGRAM

(1)Write an assembly program to add two numbers Program

```
MVI D, 8BH
MVI C, 6FH
MOV A, C
1100 0011
1000 0101
0010 0000
ADD D
OUT PORT1
HLT
```

(2)Write an assembly program to multiply a number by 8 Program

```
MVI A, 30H
RRC
RRC
RRC
OUT PORT1
HLT
```

(3)Write an assembly program to find greatest between two numbers Program

```
MVI B, 30H
MVI C, 40H
MOV A, B
CMP C
JZ EQU
JC GRT
OUT PORT1
HLT
EQU: MVI A, 01H
OUT PORT1
```

HLT
GRT: MOV A, C
OUT PORT1
HLT

2.5 Programming using Loop structure with Counting and Indexing

(i) 16 bit Multiplication

ADDRESS
LABEL
MNEMONICS
OPCODE
START
L1
L2
LHLD 4200
SPHL
LHLD 4202
XCHG
LXI H,0000
LXI B,0000
DAD SP
JNC L2
INX B
DCX D
MOV A,E
ORA D
JNZ L1
SHLD 4204
MOV L,C
MOV H,B
SHLD 4206
HLT

(ii)Finding the maximum number in the given array

ADDRESS
LABEL
MNEMONICS
OPCODE
START
L2
L3
L1
LDA 4500
MOV C, A
LXI H, 4501
MOV A, M
DCR C
INX H

JZ L1
CMP M
JC L2
JMP L3
STA 4520
HLT
(iii) To sort the array of data in ascending order
ADDRESS
LABEL
MNEMONICS
START
L3
L2
L1
MVI B, 00
LXI H, 4200
MOV C, M
DCR C
INX H
MOV A, M
INX H
CMP M
JC L1
MOV D, M
MOV M, A
DCX H
MOV M, D
INX H
MVI B, 01
DCR C
JNZ L2
DCR B
JZ L3
HLT

2.6 Programming using subroutine Instructions

Generation of Square waveform using DAC

ADDRESS
LABEL
MNEMONICS
START
DELAY
L2
L1
MVI A,00H
OUT C8
CALL DELAY

```
MVI A,FF
OUT C8
CALL DELAY
JMP START
MVI B,05H
MVI C,FF
DCR C
JNZ L1
DCR B
JNL L2
RET
```

2.7 Programming using Look up table

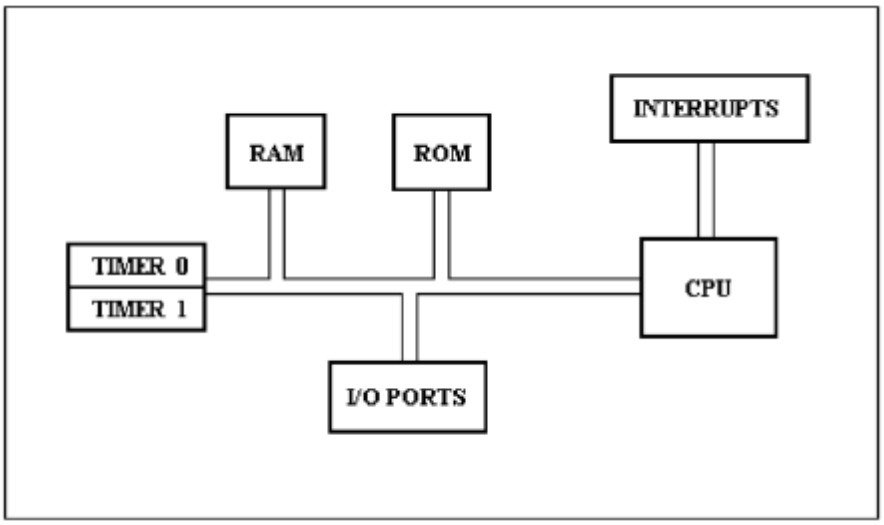
```
ADDRESS
LABEL
MNEMONICS
OPCODE
START
L2
MVI B,08
MVI A,00(DISPLAY MODE
SETUP)
OUT C2
MVI A,CC(CLEAR DISPLAY)
OUT C2
MVI A,90(WRITE DISPLAY
RAM)
OUT C2
MVI A, FF(CLEAR DISPLAY
RAM)
OUT C0
DCR B
JNZ L1
IN C2
ANI 07
JZ L2
MVI A, 40(SET TO READ FIFO
RAM)
OUT C2
IN C0
ANI 0F
MOV L, A
MVI H, 42
MOV A, M
OUT C0
JMP L2
LOOKUP TABLE
```

4200 0C 9F 4A 0B
4204 99 29 28 8F
4208 08 09 88 38
420C 6C 1A 68 E8

UNIT III

8051 MICROCONTROLLER

1)ARCHITECTURE OF 8051:



Memory Organization

- Logical separation of program and data memory
-Separate address spaces for Program (ROM) and Data (RAM) Memory

Allow Data Memory to be accessed by 8-bit addresses quickly and manipulated by 8-bit CPU

 Program Memory

-Only be read, not written to

-The address space is 16-bit, so maximum of 64K bytes

-Up to 4K bytes can be on-chip (internal) of 8051 core

-PSEN (Program Store Enable) is used for access to external Program Memory

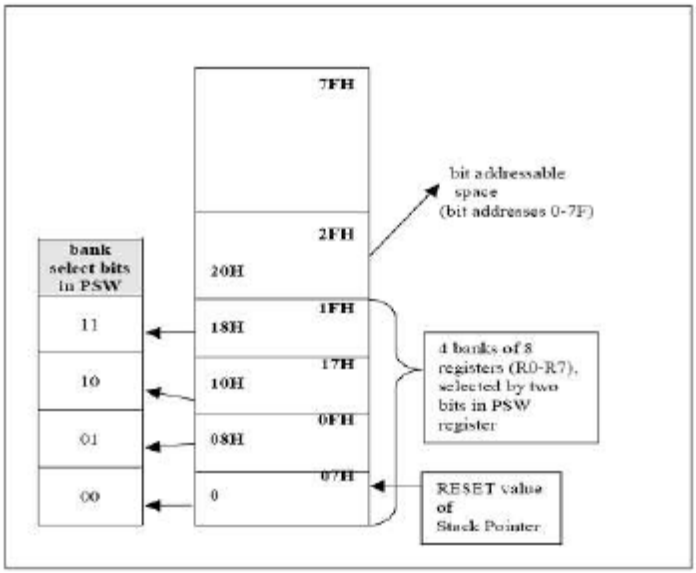
 Data Memory

-Includes 128 bytes of on-chip Data Memory which are more easily accessible directly by its instructions

-There is also a number of Special Function Registers (SFRs)

-Internal Data Memory contains four banks of eight registers and a special 32-byte long segment which is bit addressable by 8051 bit-instructions

-External memory of maximum 64K bytes is accessible by “movx”



Interrupt Structure

- The 8051 provides 4 interrupt sources
- Two external interrupts
- Two timer interrupts

Port Structure

- The 8051 contains four I/O ports
- All four ports are bidirectional
- Each port has SFR (Special Function Registers P0 through P3) which works like a latch, an output driver and an input buffer
- Both output driver and input buffer of Port 0 and output driver of Port 2 are used for accessing external memory
- Accessing external memory works like this
- Port 0 outputs the low byte of external memory address (which is timemultiplexed with the byte being written or read)
- Port 2 outputs the high byte (only needed when the address is 16 bits wide)
- Port 3 pins are multifunctional
- The alternate functions are activated with the 1 written in the corresponding bit in the port SFR

Port Pin	Alternate Function
P3.2	~INT0 (external interrupt)
P3.3	~INT1 (external interrupt)
P3.4	T0 (Timer/Counter 0 external input)
P3.5	T1 (Timer/Counter 1 external input)
P3.6	~WR (external Data Memory writestrobe)
P3.7	~RD (external DataMemory readstrobe)

Timer/Counter

- The 8051 has two 16-bit Timer/Counter registers
- Timer 0
- Timer 1
- Both can work either as timers or event counters
- Both have four different operating modes

2) INSTRUCTION FORMAT

An **instruction** is a command to the microprocessor to perform a given task on a specified data. Each instruction has two parts: one is task to be performed, called the **operation code** (opcode), and the second is the data to be operated on, called the **operand.** The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

Instruction word size

The 8051 instruction set is classified into the following three groups according to word size:

1. One-word or 1-byte instructions
2. Two-word or 2-byte instructions
3. Three-word or 3-byte instructions

One-Byte Instructions

A 1-byte instruction includes the opcode and operand in the same byte. Operand(s) are internal register and are coded into the instruction.

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bit binary format in memory; each requires one memory location.

Two-Byte Instructions

In a two-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. Source operand is a data byte immediately following the opcode.

Three-Byte Instructions

In a three-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address.

3) Addressing Modes of 8051

The 8051 provides a total of five distinct addressing modes.

(1) immediate
(2) register
(3) direct
(4) register indirect
(5) indexed

(1) Immediate Addressing Mode

- The operand comes immediately after the op-code.
- The immediate data must be preceded by the pound sign, "#".

```
MOV A,#25H          ;load 25H into A
MOV R4,#62          ;load the decimal value 62 into R4
MOV B,#40H          ;load 40H into B
MOV DPTR,#4521H     ;DPTR=4512H
```

(2) Register Addressing Mode

- Register addressing mode involves the use of registers to hold the data to be manipulated

```
MOV A,R0  ;copy the contents of R0 into A
MOV R2,A  ;copy the contents of A into R2
ADD A,R5  ;add the contents of R5 to contents of A
ADD A,R7  ;add the contents of R7 to contents of A
MOV R6,A  ;save accumulator in R6
```

(3)Direct Addressing Mode

- It is most often used to access RAM locations 30 - 7FH.

-This is due to the fact that register bank locations are accessed by the register names of R0 - R7.

-There is no such name for other RAM locations so must use direct addressing

-In the direct addressing mode, the data is in a RAM memory location whose address is known, and this address is given as a part of the instruction

```
MOV A,4        ;is same as
MOV A,R4       ;which means copy R4 into A

MOV A,7        ;is same as
MOV A,R7       ;which means copy R7 into A
```

***(4)Register Indirect* Addressing Mode**

- A register is used as a pointer to the data.
- If the data is inside the CPU, only registers R0 and R 1 are used for this purpose.
- R2 - R7 cannot be used to hold the address of an operand located in RAM when using indirect addressing mode.
- When RO and R 1 are used as pointers they must be preceded by the @ sign.

(5) Indexed Addressing Mode

- Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM space of the 8051.
- The instruction used for this purpose is :

MOVC A, @ A+DPTR

- The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM.
- Because the data elements are stored in the program (code) space ROM of the 8051, the instruction MOVC is used instead of MOV. The "C" means code.
- In this instruction the contents of A are added to the 16-bit register DPTR to form the 16-bit address of the needed data

4) Interrupt Structure

- 8051 provides 4 interrupt sources
- 2 external interrupts
- 2 timer interrupts
- They are controlled via two SFRs, IE and IP
- Each interrupt source can be individually enabled or disabled by setting or clearing a bit in IE (Interrupt Enable). IE also exists a global disable bit, which can be cleared to disable all interrupts at once
- Each interrupt source can also be individually set to one of two priority levels by setting or clearing a bit in IP (Interrupt Priority)
- A low-priority interrupt can be interrupted by high-priority interrupt, but not by another low-priority one
- A high-priority interrupt can‟t be interrupted by any other interrupt source
- If interrupt requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced, so within each priority lever there is a second priority structure
- This internal priority structure is determined by the polling sequence, shown in the following table

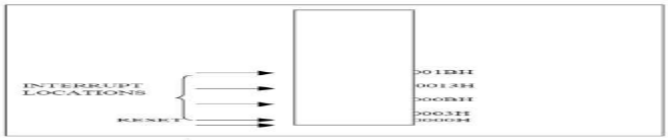
Source	Priority Within Level
IE0	highest
TF0	
IE1	
TF1	lowest

External Interrupt

- External interrupts ~INT0 and ~INT1 have two ways of activation
- Level-activated
- Transition-activated
- This depends on bits IT0 and IT1 in TCON
- The flags that actually generate these interrupts are bits IE0 and IE1 in TCON
- On-chip hardware clears that flag that generated an external interrupt when the service routine is vectored to, but only if the interrupt was transition-activated
- When the interrupt is level-activated, then the external requesting source is controlling the request flag, not the on-chip hardware

Handling Interrupt

- When interrupt occurs (or correctly, when the flag for an enabled interrupt is found to be set (1)), the interrupt system generates an LCALL to the appropriate location in Program Memory, unless some other conditions block the interrupt
- Several conditions can block an interrupt
- An interrupt of equal or higher priority level is already in progress
- The current (polling) cycle is not the final cycle in the execution of the instruction in progress
- The instruction in progress is RETI or any write to IE or IP registers
- If an interrupt flag is active but not being responded to for one of the above conditions, must be still active when the blocking condition is removed, or the
denied interrupt will not be serviced
- Next step is saving the registers on stack. The hardware-generated LCALL causes only the contents of the Program Counter to be pushed onto the stack, and reloads the PC with the beginning address of the service routine
- In some cases it also clears the flag that generated the interrupt, and in other cases it doesn‟t. It clears an external interrupt flag (IE0 or IE1) only if it was transitionavtivated.
- Having only PC be automatically saved gives programmer more freedom to decide how much time to spend saving other registers. Programmer must also be more careful with proper selection, which register to save.
- The service routine for each interrupt begins at a fixed location. The interrupt locations are spaced at 8-byte interval, beginning at 0003H for External Interrupt 0, 000BH for Timer 0, 0013H for External Interrupt 1 and 001BH for Timer 1.



I/O Ports

- The 8051 contains four I/O ports
- All four ports are bidirectional
- Each port has SFR (Special Function Registers P0 through P3) which works like a latch, an output driver and an input buffer
- Both output driver and input buffer of Port 0 and output driver of Port 2 are used for accessing external memory
- Accessing external memory works like this
- Port 0 outputs the low byte of external memory address (which is timemultiplexed with the byte being written or read)
- Port 2 outputs the high byte (only needed when the address is 16 bits wide)
- Port 3 pins are multifunctional
- The alternate functions are activated with the 1 written in the corresponding bit in the port SFR

Timers

The 8051 comes equipped with two timers, both of which may be controlled, set, read, and configured individually. The 8051 timers have three general functions: 1) Keeping time and/or calculating the amount of time between events, 2) Counting the events themselves, or 3) Generating baud rates for the serial port.

one of the primary uses of timers is to measure time. We will discuss this use of timers first and will subsequently discuss the use of timers to count events. When a timer is used to measure time it is also called an "interval timer" since it is measuring the time of the interval between two events.

Timer SFR

8051 has two timers which each function essentially the same way. One timer is TIMER0 and the other is TIMER1. The two timers share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to itself (TH0/TL0 and TH1/TL1).

Table 4.4 SFR

13-bit Time Mode (mode 0)

Timer mode "0" is a 13-bit timer. This is a relic that was kept around in the 8051 to maintain compatability with its predecesor, the 8048. Generally the 13-bit timer mode is not used in new development.

When the timer is in 13-bit mode, TLx will count from 0 to 31. When TLx is incremented from 31, it will "reset" to 0 and increment THx. Thus, effectively, only 13 bits of the two timer bytes are being used: bits 0-4 of TLx and bits 0-7 of THx. This also means, in essence, the timer can only contain 8192 values. If you set a 13-bit timer to 0, it will overflow back to zero 8192 Machine cycles later.

Again, there is very little reason to use this mode and it is only mentioned so you wont be surprised if you ever end up analyzing archaeic code which has been passed down through the generations (a generation in a programming shop is often on the order of about 3 or 4 months).

SFR Name Description SFR Address

TH0 Timer 0 High Byte 8Ch
TL0 Timer 0 Low Byte 8Ah
TH1 Timer 1 High Byte 8Dh
TL1 Timer 1 Low Byte 8Bh
TCON Timer Control 88h
TMOD Timer Mode 89h

16-bit Time Mode (mode 1)

Timer mode "1" is a 16-bit timer. This is a very commonly used mode. It functions just like 13-bit mode except that all 16 bits are used.

TLx is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536 distinct values. If you set a 16-bit timer to 0, it will overflow back to 0 after 65,536 machine cycles.

8-bit Time Mode (mode 2)

Timer mode "2" is an 8-bit auto-reload mode. What is that, you may ask? Simple. When a timer is in mode 2, THx holds the "reload value" and TLx is the timer itself. Thus, TLx starts counting up. When TLx reaches 255 and is subsequently incremented, instead of resetting to 0 (as in the case of modes 0 and 1), it will be reset to the value stored in THx.

Split Timer Mode (mode 3)

Timer mode "3" is a split-timer mode. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. That is to say, Timer 0 is TL0 and Timer 1 is TH0. Both timers count from 0 to 255 and overflow back to 0. All the bits that are related to Timer 1 will now be tied to TH0.

While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into modes 0, 1 or 2 normally--however, you may not start or stop the real timer 1 since the bits that do that are now linked to TH0. The real timer 1, in this case, will be incremented every machine cycle no matter what.

USING TIMERS AS EVENT COUNTERS

We've discussed how a timer can be used for the obvious purpose of keeping track of time. However, the 8051 also allows us to use the timers to count events. How can this be useful? Let's say you had a sensor placed across a road that would send a pulse every time a car passed over it. This could be used to determine the volume of traffic on the road. We could attach this sensor to one of the 8051's I/O lines and constantly monitor it, detecting when it pulsed high and then incrementing our counter when it went back to a low state. This is not terribly difficult, but requires some code. Let's say we hooked the sensor to P1.0; the code to count cars passing would look something like this: JNB P1.0,$;If a car hasn't raised the signal, keep waiting JB P1.0,$;The line is high which means the car is on the sensor right now INC COUNTER ;The car has passed completely, so we count it

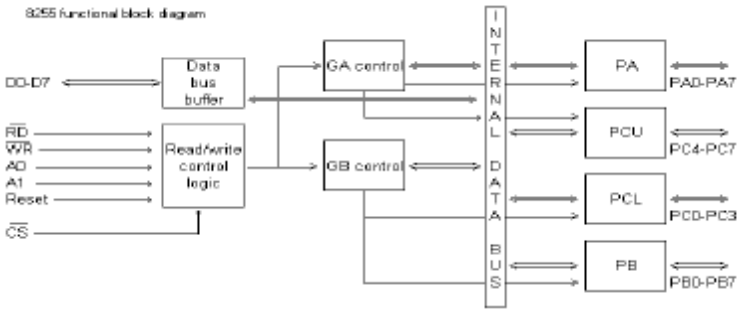
Serial Communication

Some of the external I/0 devices receive only the serial data.Normally serial communication is used in the Multi Processor environment.8051 has two pins for serial communication.
(1)SID- Serial Input data.
(2)SOD-Serial Output data.

UNIT IV
PERIPHERAL INTERFACING

1)Architecture of 8255

The parallel input-output port chip 8255 is also called as programmable peripheral input output port. The Intel‟s 8255 is designed for use with Intel‟s 8-bit, 16-bit and higher capability microprocessors. It has 24 input/output lines which may be individually programmed in two groups of twelve lines each, or three groups of eight lines. The two groups of I/O pins are named as Group A and Group B. Each of these two groups contains a subgroup of eight I/O lines called as 8-bit port and another subgroup of four lines or a 4-bit port. Thus Group A contains an 8-bit port A along with a 4-bit port. C upper.The port A lines are identified by symbols PA0-PA7 while the port C lines are identified as PC4-PC7. Similarly, Group B contains an 8-bit port B, containing lines PB0-PB7 and a 4-bit port C with lower bits PC0- PC3. The port C upper and port C lower can be used in combination as an 8-bit port C. Both the port C are assigned the same address. Thus one may have either three 8-bit I/O ports or two 8-bit and two 4-bit ports from 8255. All of these ports can function independently either as input or as output ports. This can be achieved by programming the bits of an internal register of 8255 called as control word register (CWR). This buffer receives or transmits data upon the execution of input or output instructions by the microprocessor. The control words or status information is also transferred through the buffer.



2) PIN DIAGRAM OF 8255

The signal description of 8255 are briefly presented as follows :

• **PA7-PA0**: These are eight port A lines that acts as either latched output or buffered input lines depending upon the control word loaded into the control word register.

• **PC7-PC4** : Upper nibble of port C lines. They may act as either output latches or input buffers lines.This port also can be used for generation of handshake lines in mode 1 or mode 2.

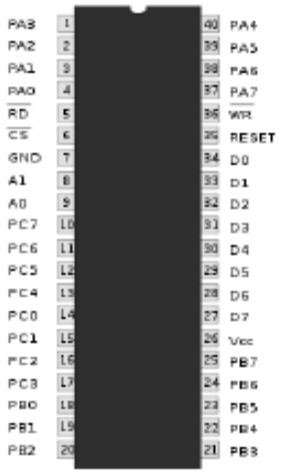
• **PC3-PC0** : These are the lower port C lines, other details are the same as PC7-PC4 lines.

• **PB0-PB7** : These are the eight port B lines which are used as latched output lines or buffered input lines in the same way as port A.

• **RD** : This is the input line driven by the microprocessor and should be low to indicate read operation to 8255.

• **WR** : This is an input line driven by the microprocessor. A low on this line indicates write operation.

CS : This is a chip select line. If this line goes low, it enables the 8255 to respond to RD and WR signals, otherwise RD and WR signal are neglected



A1-A0 : These are the address input lines and are driven by the microprocessor. These lines A1-A0 with RD, WR and CS from the following operations for 8255. These address lines are used for addressing any one of the four registers,i.e. three ports and a control word register as given in table below

• In case of 8086 systems, if the 8255 is to be interfaced with lower order data bus, the A0 and A1 pins of 8255 are connected with A1 and A2 respectively.

D0-D7 :

These are the data bus lines those carry data or control word to/from the Microprocessor.

RESET :

A logic high on this line clears the control word register of 8255. All ports are set as input ports by default after reset.

Operational Modes of 8255

There are two main operational modes of 8255:

1. Input/output mode 2. Bit set/reset mode

Input/Output Mode

There are three types of the input/output mode. They are as follows:

- **Mode 0**

In this mode, the ports can be used for simple input/output operations without handshaking. If both port A and B are initialized in mode 0, the two halves of port C can be either used together as an additional 8-bit port, or they can be used as individual 4-bit ports. Since the two halves of port C are independent, they may be used such that one-half is initialized as an input port while the other half is initialized as an output port. The input output features in mode 0 are as follows: 1. O/p are latched. 2. I/p are buffered not latched. 3. Port do not have handshake or interrupt capability.

- **Mode 1**

When we wish to use port A or port B for handshake (strobed) input or output operation, we initialise that port in mode 1 (port A and port B can be initilalised to operate in different modes,ie, for eg, port A can operate in mode 0 and port B in mode 1). Some of the pins of port C function as handshake lines.

For port B in this mode (irrespective of whether is acting as an input port or output port), PC0, PC1 and PC2 pins function as handshake lines.

If port A is initialised as mode 1 input port, then, PC3, PC4 and PC5 function as handshake signals. Pins PC6 and PC7 are available for use as input/output lines.

The mode 1 which supports handshaking has following features: 1. Two ports i.e. port A and B can be use as 8-bit i/o port. 2. Each port uses three lines of port c as handshake signal and remaining two signals can be function as i/o port. 3. interrupt logic is supported. 4. Input and Output data are latched.

- **Mode 2**

Only group A can be initialised in this mode. Port A can be used for bidirectional handshake data transfer. This means that data can be input or output on the same eight lines (PA0 - PA7). Pins PC3 - PC7 are used as handshake lines for port A. The remaining pins of port C (PC0 - PC2) can be used as input/output lines if group B is initialised in mode 0. In this mode, the 8255 may be used to extend the system bus to a slave microprocessor or to transfer data bytes to and from a floppy disk controller.

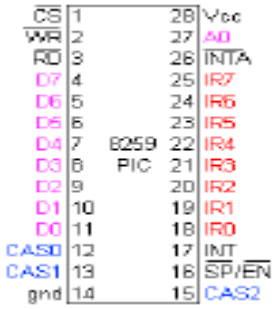
Bit Set/Reset (BSR) mode

In this mode only port b can be used (as an output port). Each line of port C (PC0 - PC7) can be set/reset by suitably loading the command word register.no effect occurs in input-output mode. The individual bits of port c can be set or reset by sending the signal OUT instruction to the control register.

3)PROGRAMMABLE INTERRUPT CONTROLLER(8259)

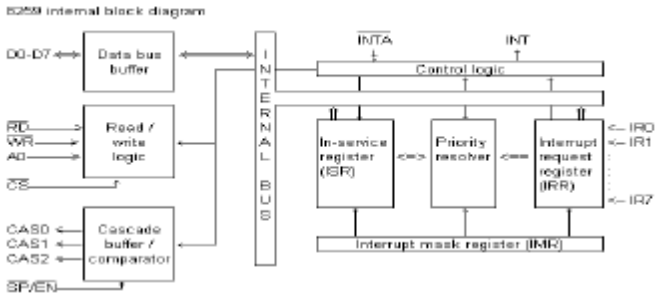
8 levels of interrupts.

- Can be cascaded in master-slave configuration to handle 64 levels of interrupts.
- Internal priority resolver.
- Fixed priority mode and rotating priority mode.
- Individually maskable interrupts.
- Modes and masks can be changed dynamically.
- Accepts IRQ, determines priority, checks whether incoming priority > current level being serviced, issues interrupt signal.
- In 8085 mode, provides 3 byte CALL instruction. In 8086 mode, provides 8 bit vector number.
- Polled and vectored mode.
- Starting address of ISR or vector number is programmable.
- No clock required



Pin diagram

D0-D7	Bi-directional, tristated, buffered data lines. Connected to data bus directly or through buffers
RD-bar	Active low read control
WR-bar	Active low write control
A0	Address input line, used to select control register
CS-bar	Active low chip select
CAS0-2	Bi-directional, 3 bit cascade lines. In master mode, PIC places slave ID no. on these lines. In slave mode, the PIC reads slave ID no. from master on these lines. It may be regarded as slave-select.
SP-bar / EN-bar	Slave program / enable. In non-buffered mode, it is SP-bar input, used to distinguish master/slave PIC. In buffered mode, it is output line used to enable buffers
INT	Interrupt line, connected to INTR of microprocessor
INTA-bar	Interrupt ack, received active low from microprocessor
IR0-7	Asynchronous IRQ input lines, generated by peripherals.



ICW1 (Initialisation Command Word One)

A0	D7	D6	D5	D4	D3	D2	D1	D0
0	A7	A6	A5	1	LTIM	ADI	SNGL	IC4

D0: IC4: 0=no ICW4, 1=ICW4 required
D1: SNGL: 1=Single PIC, 0=Cascaded PIC
D2: ADI: Address interval. Used only in 8085, not 8086. 1=ISR's are 4 bytes apart (0200, 0204, etc) 0=ISR's are 8 byte apart (0200, 0208, etc)
D3: LTIM: level triggered interrupt mode: 1=All IR lines level triggered. 0=edge triggered
D4-D7: A5-A7: 8085 only. ISR address lower byte segment. The lower byte is

A7	A6	A5	A4	A3	A2	A1	A0

of which A7, A6, A5 are provided by D7-D5 of ICW1 (if ADI=1), or A7, A6 are provided if ADI=0. A4-A0 (or A5-A0) are set by 8259 itself:

ADI=1 (spacing 4 bytes)

IRQ	A7	A6	A5	A4	A3	A2	A1	A0
IR0	A7	A6	A5	0	0	0	0	0
IR1	A7	A6	A5	0	0	1	0	0
IR2	A7	A6	A5	0	1	0	0	0
IR3	A7	A6	A5	0	1	1	0	0
IR4	A7	A6	A5	1	0	0	0	0
IR5	A7	A6	A5	1	0	1	0	0
IR6	A7	A6	A5	1	1	1	0	0
IR7	A7	A6	A5	1	1	1	0	0

ADI=0 (spacing 8 bytes)

IRQ	A7	A6	A5	A4	A3	A2	A1	A0
IR0	A7	A6	0	0	0	0	0	0
IR1	A7	A6	0	0	1	0	0	0
IR2	A7	A6	0	1	0	0	0	0
IR3	A7	A6	0	1	1	0	0	0
IR4	A7	A6	1	0	0	0	0	0
IR5	A7	A6	1	0	1	0	0	0
IR6	A7	A6	1	1	0	0	0	0
IR7	A7	A6	1	1	1	0	0	0

1	R	SL	EOI	0	0	L3	L2	L1

	R	SL	EOI	Action
EOI	0	0	1	Non specific EOI (L3L2L1=000)
	0	1	1	Specific EOI command (Interrupt to clear given by L3L2L1)
Auto rotation of priorities (L3L2L1=000)	1	0	1	Rotate priorities on non-specific EOI
	1	0	0	Rotate priorities in auto EOI mode set
	0	0	0	Rotate priorities in auto EOI mode clear
Specific rotation of priorities (Lowest priority ISR=L3L2L1)	1	1	1	Rotate priority on specific EOI command (resets current ISR bit)
	1	1	0	Set priority (does not reset current ISR bit)
	0	1	0	No operation

- **OCW3 (Operational Command Word Three)**

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	D7	ESMM	SMM	0	1	MODE	RIR	RIS

ESMM	SMM	Effect
0	X	No effect
1	0	Reset special mask
1	1	Set special mask

- **ICW2 (Initialisation Command Word Two)**

Higher byte of ISR address (8085), or 8 bit vector address (8086).

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	A15	A14	A13	A12	A11	A10	A9	A8

- **ICW3 (Initialisation Command Word Three)**

A0		D7	D6	D5	D4	D3	D2	D1	D0
1	Master	S7	S6	S5	S4	S3	S2	S1	S0
	Slave	0	0	0	0	0	ID3	ID2	ID1

- Master mode: 1 indicates slave is present on that interrupt, 0 indicates direct interrupt
- Slave mode: ID3-ID2-ID1 is the slave ID number. Slave 4 on IR4 has ICW3=04h (0000 0100)

- **ICW4 (Initialisation Command Word Four)**

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	SFNM	BUF	M/S	AEOI	Mode

- SFNM: 1=Special Fully Nested Mode, 0=FNM
- M/S: 1=Master, 0=Slave
- AEOI: 1=Auto End of Interrupt, 0=Normal
- Mode: 0=8085, 1=8086

- **OCW1 (Operational Command Word One)**

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

IRn is masked by setting Mn to 1; mask cleared by setting Mn to 0 (n=0..7)

- **OCW2 (Operational Command Word Two)**

A0	D7	D6	D5	D4	D3	D2	D1	D0

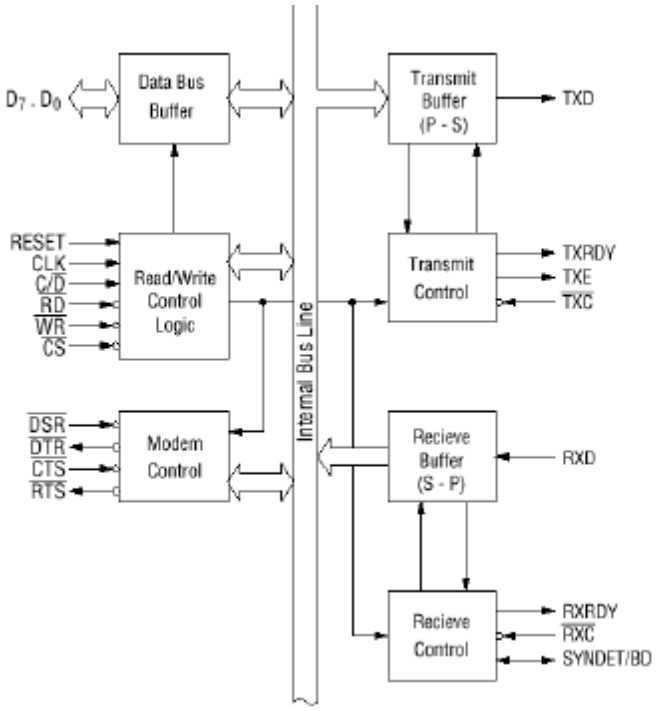
4) 8251 UNIVERSAL SYNCHRONOUS ASYNCHRONOUS RECEIVER TRANSMITTER (USART)

The 8251 is a USART (Universal Synchronous Asynchronous Receiver Transmitter) for serial data communication. As a peripheral device of a microcomputer system, the 8251receives parallel data from the CPU and transmits serial data after conversion. This device also receives serial data from the outside and transmits parallel data to the CPU after conversion

Control Words

There are two types of control word.

1. Mode instruction (setting of function)
2. Command (setting of operation)



1) Mode Instruction

Mode instruction is used for setting the function of the 8251. Mode instruction will be in "wait for write" at either internal reset or external reset. That is, the writing of a control word after resetting will be recognized as a "mode instruction."

Items set by mode instruction are as follows:

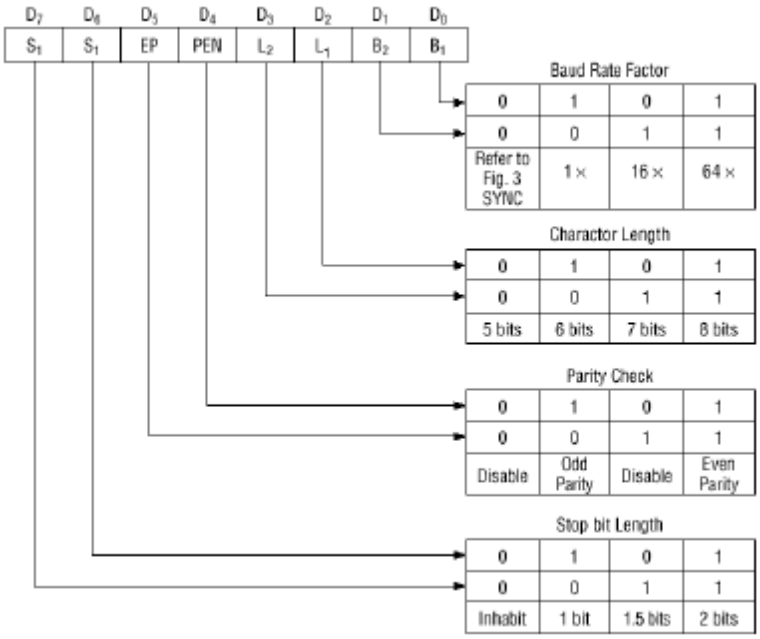
• Synchronous/asynchronous mode
• Stop bit length (asynchronous mode)
• Character length
• Parity bit
• Baud rate factor (asynchronous mode)
• Internal/external synchronization (synchronous mode)
• Number of synchronous characters (Synchronous mode)

The bit configuration of mode instruction is shown in Figure. In the case of synchronous mode, it is necessary to write one-or two byte sync characters. If sync characters were written, a function will be set because the writing of sync characters constitutes part of mode instruction.

Command

Command is used for setting the operation of the 8251. It is possible to write a command whenever necessary after writing a mode instruction and sync characters.Items to be set by command are as follows:

• Transmit Enable/Disable
• Receive Enable/Disable

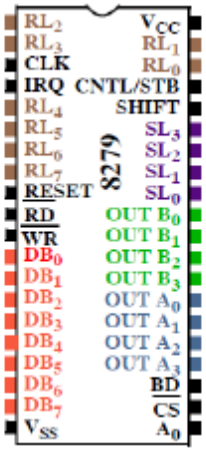


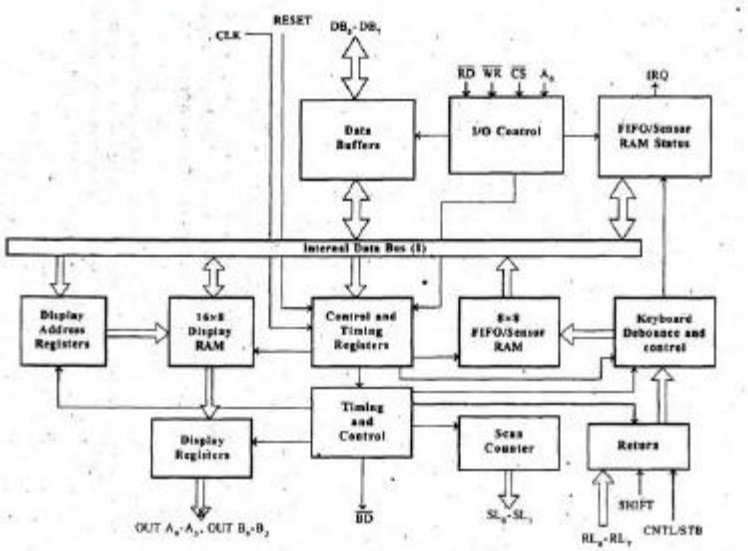
5)Programmable Keyboard/Display Interface - 8279

A programmable keyboard and display interfacing chip.Scans and encodes up to a 64-key keyboard.Controls up to a 16-digit numerical display.Keyboard section has a built-in FIFO 8 character buffer.The display is controlled from an internal 16x8 RAM tha stores the coded display information.

A0: Selects data (0) or control/status (1) for reads and writes between micro and 8279.

- **BD:** Output that blanks the displays.
- **CLK:** Used internally for timing. Max is 3 MHz.
- **CN/ST:** Control/strobe, connected to the control key on the keyboard
- **CS**: Chip select that enables programming, reading the keyboard, etc.
- **DB7-DB0:** Consists of bidirectional pins that connect to data bus on micro.
- **IRQ:** Interrupt request, becomes 1 when a key is pressed, data is available.
- **OUT A3-A0/B3-B0**: Outputs that sends data to the most significant/least significant nibble of display.
- **RD(WR):** Connects to micro's IORC or RD signal, reads data/status registers.
- **RESET:** Connects to system RESET.
- **RL7-RL0:** Return lines are inputs used to sense key depression in the keyboard matrix.
- **Shift:** Shift connects to Shift key on keyboard





The display section has eight output lines divided into two groups A0-A3 and B0-B3.

- The output lines can be used either as a single group of eight lines or as two groups of four lines, in conjunction with the scan lines for a multiplexed display.
- The output lines are connected to the anodes through driver transistor in case of common cathode 7-segment LEDs.
- The cathodes are connected to scan lines through driver transistors.
- The display can be blanked by BD (low) line.
- The display section consists of 16 x 8 display RAM. The CPU can read from or write into any location of the display RAM.

Scan section:

- The scan section has a scan counter and four scan lines, SL0 to SL3.
- In decoded scan mode, the output of scan lines will be similar to a 2-to-4 decoder.
- In encoded scan mode, the output of scan lines will be binary count, and so an external decoder should be used to convert the binary count to decoded output.
- The scan lines are common for keyboard and display.
- The scan lines are used to form the rows of a matrix keyboard and also connected to digit drivers of a multiplexed display, to turn ON/OFF.

CPU interface section:

- The CPU interface section takes care of data transfer between 8279 and the processor.
- This section has eight bidirectional data lines DB0 to DB7 for data transfer between 8279 and CPU.
- It requires two internal address A =0 for selecting data buffer and A = 1 for selecting control register of8279.
- The control signals WR (low), RD (low), CS (low) and A0 are used for read/write to 8279.
- It has an interrupt request line IRQ, for interrupt driven data transfer with processor.
- The 8279 require an internal clock frequency of 100 kHz. This can be obtained by dividing the input clock by an internal prescaler.
- The RESET signal sets the 8279 in 16-character display with two -key lockout keyboard modes.

6)Keyboard Interface of 8279

The keyboard matrix can be any size from 2x2 to 8x8.Pins SL2-SL0 sequentially scan each column through a counting operation.The 74LS138 drives 0's on one line at a time.The 8279 scans RL pins synchronously with the scan.RL pins incorporate internal pull-ups, no need for external resistor pull-ups.The 8279 must be programmed first. First three bits given below select one of 8 control registers (opcode).

- 000DDMMM

Mode set: Opcode 000.
DD sets displays mode.
MMM sets keyboard mode.
DD field selects either:

- 8- or 16-digit display
- Whether new data are entered to the rightmost or leftmost display position.

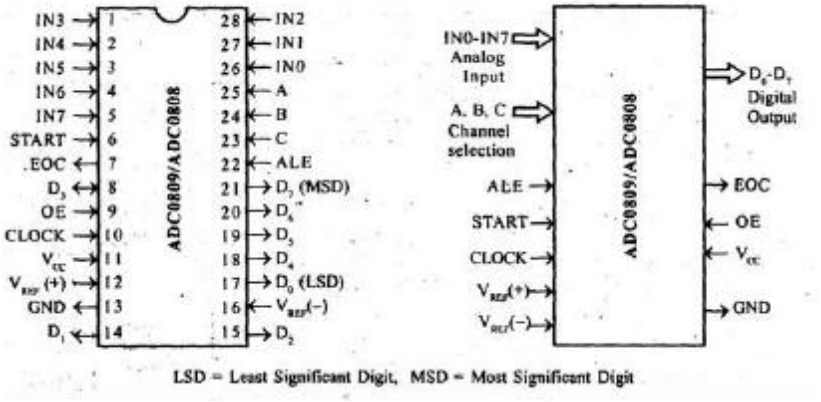
DD	*Function*
000	Encoded keyboard with 2-key lockout
001	Decoded keyboard with 2-key lockout
010	Encoded keyboard with N-key rollover
011	Decoded keyboard with N-key rollover
100	Encoded sensor matrix
101	Decoded sensor matrix
110	Strobed keyboard, encoded display scan
111	Strobed keyboard, decoded display scan

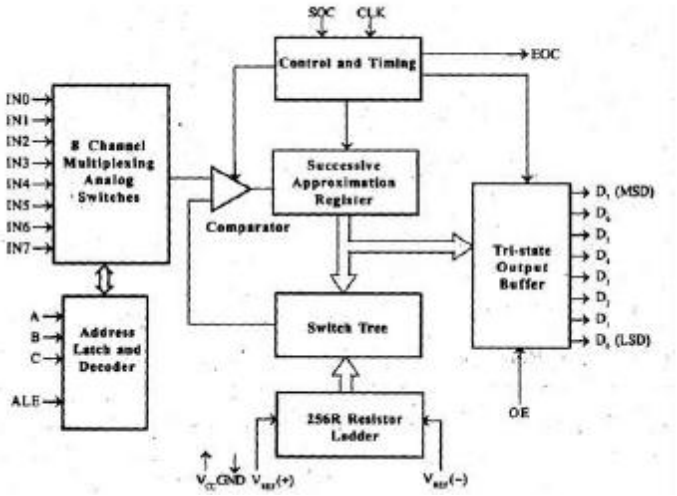
Encoded: SL outputs are active-high, follow binary bit pattern 0-7 or 0-15.
Decoded: SL outputs are active-low (only one low at any time). Pattern output: 1110, 1101, 1011, 0111.
Strobed: An active high pulse on the CN/ST input pin strobes data from the RL pins into an internal FIFO for reading by micro later.
2-key lockout/N-key rollover: Prevents 2 keys from being recognized if pressed simultaneously/Accepts all keys pressed from 1st to last.

7)ADC Interfacing with 8085 Microprocessor

Features

- The ADC0809 is an 8-bit successive approximation type ADC with inbuilt 8-channel multiplexer.
- The ADC0809 is suitable for interface with 8086 microprocessor.
- The ADC0809 is available as a 28 pin IC in DIP (Dual Inline Package).
- The ADC0809 has a total unadjusted error of ±1 LSD (Least Significant Digit).
- The ADC0808 is also same as ADC0809 except the error. The total unadjusted error in ADC0808 is ± 1/2 LSD.





The successive approximation register (SAR) performs eight iterations to determine the digital code for input value. The SAR is reset on the positive edge of START pulse and start the conversion process on the falling edge of START pulse. A conversion process will be interrupted on receipt of new START pulse. The End-Of-Conversion (EOC) will go low between 0 and 8 clock pulses after the positive edge of START pulse. The ADC can be used in continuous conversion mode by tying the EOC output to START input. In this mode an external START pulse should be applied whenever power is switched ON. The 256R ladder network has been provided instead of conventional R/2R ladder because of its inherent monotonic, which guarantees no missing digital codes. Also the 256R resistor network does not cause load

variations on the reference voltage. The comparator in ADC0809/ADC0808 is a chopper-stabilized comparator. It converts the DC input signal into an AC signal, and amplifies the AC sign using high gain AC amplifier Then it converts AC signal to DC signal. This technique limits the drift component of the amplifier, because the drift is a DC component and it is not amplified/passed by the AC amp1ifier. This makes the ADC extremely insensitive to temperature, long term drift and input offset errors. In ADC conversion process the input analog value is quantized and each quantized analog value will have a unique binary equivalent. The quantization step in ADC0809/ADC0808 is given by,

$$Q_{step} = \frac{V_{REF}}{2^8} = \frac{V_{REF}(+) - V_{REF}(-)}{256_{10}}$$

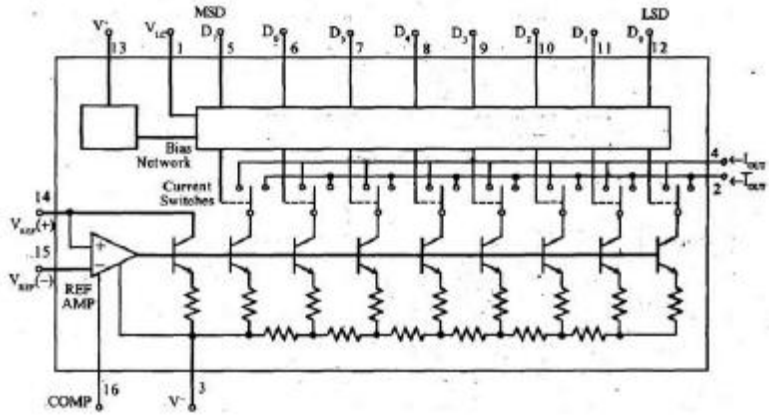
The digital data corresponding to an analog input (V_{in}) is given by,

$$\text{Digital data} = \left(\frac{V_{in}}{Q_{step}} - 1\right)_{10}$$

8) DAC Interfacing with 8085 Microprocessor

To convert the digital signal to analog signal a Digital-to-Analog Converter (DAC) has to be employed.

- The DAC will accept a digital (binary) input and convert to analog voltage or current.
- Every DAC will have "n" input lines and an analog output.
- The DAC require a reference analog voltage (Vref) or current (Iref) source.
- The smallest possible analog value that can be represented by the n-bit binary code is called resolution.
- The resolution of DAC with n-bit binary input is 1/2nof reference analog value.



The DAC0800 is an 8-bit, high speed, current output DAC with a typical settling time (conversion time) of 100 ns.

It produces complementary current output, which can be converted to voltage by using simple resistor load.

- The DAC0800 require a positive and a negative supply voltage in the range of ± 5V to ±18V.
- It can be directly interfaced with TTL, CMOS, PMOS and other logic families.

- For TTL input, the threshold pin should be tied to ground (VLC = 0V).
- The reference voltage and the digital input will decide the analog output current, which can be converted to a voltage by simply connecting a resistor to output terminal or by using an op-amp I to V converter.
- The DAC0800 is available as a 16-pin IC in DIP.

UNIT V
MICRO CONTROLLER PROGRAMMING & APPLICATIONS

1)ARITHMETIC INSTRUCTIONS

Mnemonic	Operation	Addressing modes	Execution time
ADD A, <byte>	A = A + <byte>	Dir, Ind, Reg, Imm	1
ADDC A, <byte>	A = A + <byte> + C	Dir, Ind, Reg, Imm	1
SUBB A, <byte>	A = A - <byte> - C	Dir, Ind, Reg, Imm	1
INC A	A = A + 1	Accumulator only	1
INC <byte>	<byte> = <byte> + 1	DPTR only	1
INC DPTR	DPTR = DPTR + 1	Dir, Ind, Reg	2
DEC A	A = A – 1	Accumulator only	1
DEC <byte>	<byte> = <byte> - 1	Dir, Ind, Reg	1
MUL AB	B:A=B x A	ACC and B only	4
DIV AB	A = Int [A/B] B = Mod [A/Bl	ACC and B only	4
DA A	Decimal Adjust	Accumulator only	1

2)LOGICAL INSTRCUTIONS

Mnemonic	Operation	Addressing modes	Execution time
ANL A, <byte>	A = A .AND. <byte>	Dir, Ind, Reg, Imm	1
ANL A, <byte>	<byte> = <byte> .AND. A	Dir	1
ANL <byte>, #data	<byte> = <byte> .AND. #data	Dir	2
ORL A, <byte>	A = A .OR. <byte>	Dir, Ind, Reg, Imm	1
ORL A, <byte>	<byte> = <byte> .OR. A	Dir	1
ORL <byte>, #data	<byte> = <byte> .OR. #data	Dir	2
XRL A, <byte>	A = A .XOR. <byte>	Dir, Ind, Reg, Imm	1
XRL A, <byte>	<byte> = <byte> .XOR. A	Dir	1
XRL <byte>, #data	<byte> = <byte> .XOR. #data	Dir	2
CRL A	A = 00H	Accumulator only	1
CPL A	A = .NOT. A	Accumulator only	1
RL A	Rotate ACC Left 1 bit	Accumulator only	1
RLC A	Rotate Left through Carry	Accumulator only	1
RR A	Rotate ACC Right 1 bit	Accumulator only	1
RRC A	Rotate Right through Carry	Accumulator only	1
SWAP A	Swap Nibbles in A	Accumulator only	1

3)DATA TRANSFER INSTRUCTION THAT ACCESS THE INTERNAL MEMORY

Mnemonic	Operation	Addressing modes	Execution time
MOV A, <src>	A = <src>	Dir, Ind, Reg, Imm	1
MOV <dest>, A	<dest> = A	Dir, Ind, Reg	1
MOV <dest>, <src>	<dest> = <src>	Dir, Ind, Reg, Imm	2
MOV DPTR, #data 16	DPTR = 16-bit immediate constant	Imm	2
PUSH <src>	INCSP: MOV "@'SP', <src>	Dir	2
POP <dest>	MOV <dest>, "@SP": DECSP	Dir	2
XCH A, <byte>	ACC and <byte> exchange data	Dir, Ind, Reg	1
XCHD A, @Ri	ACC and @Ri exchange low nibbles	Ind	1

4))DATA TRANSFER INSTRUCTION THAT ACCESS THE EXTERNAL MEMORY

Address width	Mnemonic	Operation	Execution time
8 bits	MOVX A, @Ri	Read external RAM @Ri	2
8 bits	MOVX @Ri, A	Write external RAM @Ri	2
16 bits	MOVX A, @DPTR	Read external RAM @DPTR	2
16 bits	MOVX @DPTR, A	Write external RAM @DPTR	2

5)LOOK UP TABLES

Mnemonic	Operation	Execution time
MOVC A, @A+DPTR	Read Program Memory at (A + DPTR)	2
MOVC A, @A+PC	Read Program Memory at (A + PC)	2

6)BOOLEAN INSTRUCTION

Mnemonic	Operation	Execution time
ANL C, bit	C = C .AND. bit	2
ANL C, /bit	C = C .AND. .NOT. bit	2
ORL C, bit	C = C .OR. bit	2
ORL C, /bit	C = C .OR. .NOT. bit	2
MOV C, bit	C = bit	1
MOV bit, C	bit = C	2
CRL C	C = 1	1
CRL bit	bit = 0	1
SETB C	C = 1	1
SETB bit	bit = 1	1
CPL C	C = .NOT. C	1
CPL bit	bit = .NOT. bit	1
JC rel	Jump if C = 1	2
JNC rel	Jump if C = 0	2
JB bit, rel	Jump if bit = 1	2
JNB bit, rel	Jump if bit = 0	2
JBC bit, rel	Jump if bit = 1; CLR bit	2

7)JUMP INSTRUCTIONS

Mnemonic	Operation	Execution time
ANL C, bit	C = C .AND. bit	2
ANL C, /bit	C = C .AND. .NOT. bit	2
ORL C, bit	C = C .OR. bit	2
ORL C, /bit	C = C .OR. .NOT. bit	2
MOV C, bit	C = bit	1
MOV bit, C	bit = C	2
CRL C	C = 1	1
CRL bit	bit = 0	1
SETB C	C = 1	1
SETB bit	bit = 1	1
CPL C	C = .NOT. C	1
CPL bit	bit = .NOT. bit	1
JC rel	Jump if C = 1	2
JNC rel	Jump if C = 0	2
JB bit, rel	Jump if bit = 1	2
JNB bit, rel	Jump if bit = 0	2
JBC bit, rel	Jump if bit = 1; CLR bit	2

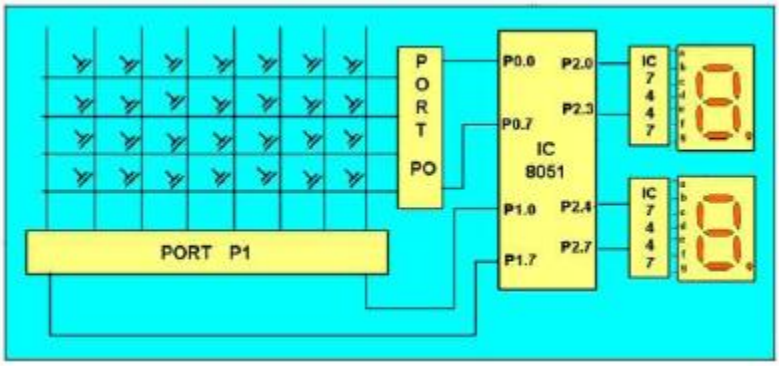
8)Interfacing Keyboard to 8051 Microcontroller

The key board here we are interfacing is a matrix keyboard. This key board is designed with a particular rows and columns. These rows and columns are connected to the microcontroller through its ports of the micro controller 8051. We normally use 8*8 matrix key board. So only two ports of 8051 can be easily connected to the rows and columns of the key board. When ever a key is pressed, a row and a column gets shorted through that pressed key and all the other keys are left open. When a key is pressed only a bit in the port goes high. Which indicates microcontroller that the key is pressed. By this high on the bit key in the corresponding column is identified. Once we are sure that one of key in the key board is pressed next our aim is to identify that key. To do this we firstly check for particular row and then we check the corresponding column the key board.

To check the row of the pressed key in the keyboard, one of the row is made high by making one of bit in the output port of 8051 high . This is done until the row is found out. Once we get the row next out job is to find out the column of the pressed key. The column is detected by contents in the input ports with the help of a counter. The content of the input port is rotated with carry until the carry bit is set.

The contents of the counter is then compared and displayed in the display. This display is designed using a seven segment display and a BCD to seven segment decoder IC 7447.

The BCD equivalent number of counter is sent through output part of 8051 displays the number of pressed key.



SEVEN SEGMENT DISPLAY

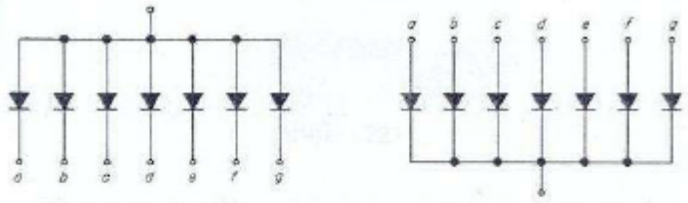
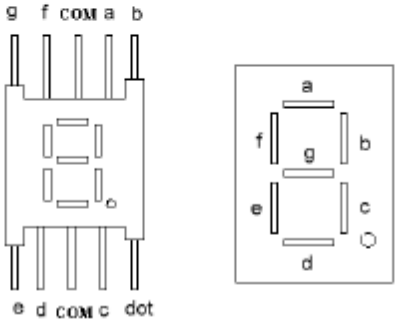


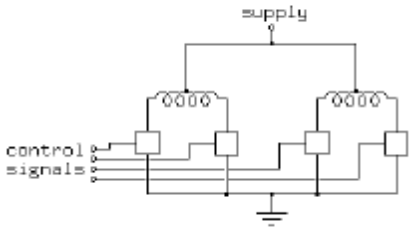
Fig 5.3 Interfacing LEDS to 8051 Microcontroller



9)Interfacing Stepper Motor with 8051Microcontroller

Step motor is the easiest to control. It's handling simplicity is really hard to deny – all there is to do is to bring the sequence of rectangle impulses to one input of step controller and direction information to another input. Direction information is very simple and comes down to "left" for logical one on that pin and "right" for logical zero. Motor control is also very simple every impulse makes the motor operating for one step and if there is no impulse the motor won't start. Pause between impulses can be shorter or longer and it defines revolution rate. This rate cannot be infinite because the motor won't be able to "catch up" with all the impulses (documentation on specific motor should contain such information). The picture below represents the scheme for connecting the step motor to microcontroller and appropriate program code follows.

The key to driving a stepper is realizing how the motor is constructed. A diagram shows the representation of a 4 coil motor, so named because 4 coils are used to cause the revolution of the drive shaft. Each coil must be energized in the correct order for the motor to spin.



Step angle

It is angle through which motor shaft rotates in one step. step angle is different for different motor . selection of motor according to step angle depends on the application , simply if you require small increments in rottion choose motor having smaller step angle. No of steps require to rotate one complete rotation = 360 deg. / step angle in deg.

INTERFACING TO 8051.

To cause the stepper to rotate, we have to send a pulse to each coil in turn. The 8051 does not have sufficient drive capability on its output to drive each coil, so there are a number of ways to drive a stepper, Stepper motors are usually controlled by transistor or driver IC like ULN2003. Driving current for each coil is then needed about 60mA at +5V supply.

Servo Motor

Servos are DC motors with built in gearing and feedback control loop circuitry. And no motor drivers required. They are extremely popular with robot, RC plane, and RC boat builders. Most servo motors can rotate about 90 to 180 degrees. Some rotate through a full 360degreesormore.

However, servos are unable to continually rotate, meaning they can't be used for driving wheels, unless they are modified (how to modify), but their precision positioning makes them ideal for robot legs and arms, rack and pinion steering, and sensor scanners to name a few. Since servos are fully self contained, the velocity and angle control loops are very easy to impliment, while prices remain very affordable. To use a servo, simply connect the black wire to ground, the red to a 4.8-6V source, and the yellow/white wire to a signal generator (such as from your microcontroller). Vary the square wave pulse width from 1-2 ms and your servo is now position/velocity controlled.

Pulse width modulation (PWM) is a powerful technique for controlling analog circuits with a processor's digital outputs. PWM is employed in a wide variety of applications, ranging from measurement and communications to power control and conversion. The general concept is to simply send an ordinary logic square wave to your servo at a specific wave length, and your servo goes to a particular angle (or velocity if your servo is modified).The wavelength directly maps to servo angle.

Controlling the Servo Motor

- **PWM**

Pulse width modulation (PWM) is a powerful technique for controlling analog circuits with a processor's digital outputs. PWM is employed in a wide variety of applications, ranging from measurement and communications to power control and conversion. The general concept is to simply send an ordinary logic square wave to your servo at a specific wave length, and your servo goes to a particular angle (or velocity if your servo is modified). The wavelength directly maps to servo angle.

- **Programmable Counter Array (PCA)**

The PCA is a special modules in Philips P89V51RD2 which includes a special 16-bit Timer that has five 16-bit capture/compare modules associated with it. Each of the modules can be programmed to operate in one of four modes: rising and/or falling edge capture, software timer, high-speed output, or pulse width modulator. Each module has a pin associated with it in port 1.
Module 0 is connected to P1.3 (CEX0), module 1 to P1.4 (CEX1), etc. Registers CH and CL contain current value of the free running up counting 16-bit PCA timer. The PCA timer is a common time base for all five modules and can be programmed to run at: 1/6 the oscillator frequency, 1/2 the oscillator frequency, the Timer 0 overflow, or the input on the ECI pin (P1.2). The timer count source is determined from the CPS1 and CPS0 bits in the CMOD SFR. In the CMOD SFR there are three additional bits associated with the PCA. They are CIDL which allows the PCA to stop during idle mode, WDTE which enables or disables the Watchdog function on module 4, and ECF which when set causes an interrupt and the PCA overflow flag CF (in the CCON SFR) to be set when the PCA timer overflows. The Watchdog timer function is implemented in module 4 of PCA. Here, we are interested only PWM mode.

- **8051 Pulse width modulator mode**

All of the PCA modules can be used as PWM outputs. Output frequency depends on the source for the PCA timer. All of the modules will have the same frequency of output because they all share one and only PCA timer. The duty cycle of each module is independently variable using the module's capture register CCAPnL.When the value of the PCA CL SFR is less than the value in the module's CCAPnL SFR the output will be low, when it is equal to or greater than the output will be high. When CL overflows from FF to 00, CCAPnL is reloaded with the value in CCAPnH. this allows updating the PWM without glitches. The PWM and ECOM bits in the module's CCAPMn register must be set to enable the PWM mode. For more details see P89V51RD2 datasheet. This is an example how to control servos with 8051 by using PWM. The schematic is shown below. I use P1.4 (CEX1) to control the left servo and P1.2 (CEX2) to control the right servo. Here, I use GWS servo motor model S03T STD. I need three states of duty cycle:

- 20 ms to Stop the servo
- 1 ms to Rotate Clockwise

Calculation for duty cycle (for XTAL 18.432 MHz with 6 Clock/Machine cycle)

- Initial PWM Period = 20mS (18.432MHz /6-Cycle Mode)
- Initial PCA Count From Timer0 Overflow
- 1 Cycle of Timer0 = (1/18.432MHz)x6 = 0.326 uS
- Timer0 AutoReload = 240 Cycle = 78.125 uS
- 1 Cycle PCA = [(1/18.432MHz)x6]x240 = 78.125 uS
- Period 20mS of PCA = 20ms/78.125us = 256 (CL Reload)
- CL (20mS) = 256 Cycle Auto Reload
- Load CCAPxH (1.0mS) = 256-13 = 243 (243,244,...,255 = 13 Cycle)
- Load CCAPxH (2.0mS) = 255-26 = 230 (230,231,...,255 = 26 Cycle)
- 2 ms to Rotate Counter-clockwise

Washing Machine Control

Many washing m/c shell in the market has mechanical controlled sequence for activated the timer and the sequence back and forth for their motor; washing motor or spinning motor. Spinning motor control only has one direction only, and its simple could be changed to the discrete mechanical timer which sell on the market. But washing motor control has 2 direction for this purpose, it means to squeeze the clothes, it must go to forward and then reversed. The sequence is like this :

- First, go to forward direction for about a few seconds
- Than stop, while the chamber is still rotate
- Second, go back to reverse direction for about a few seconds
- Than stop, while the chamber is still rotate
- And so on, back and forth, until the the timer elapsed

SCHEMATIC

Timing sequence like the above description, can be implemented with many way, by using discrete electronic components, timer, using a program or a microcontroller or microprocessor, etc. Because I am learning the PIC microcontroller for right now, I will implement this function using this microcontroller, but for you who familiar with another kind of microcontroller my adapted it to your purpose. By using PIC micro, it can be made more compact. First I plan to make 2 buttons, 1 for set the timer and another for reset the timer or for the emergency stop push button. Then to know the timer works or not, I need a visual display. For this purpose I will use 7-segmen display showing the rest of the timer. To run the motor sequence of course I need a pair of relays (power relays, about 3 Amperes output), one for forward and another for reverse option. I will use the very common family of PIC micro, ie : 16F84A, because this is the most popular type and very simples used and very much used. Also can be obtained easily in the market. But this is the medium type of PIC micro family. It has 1kByte of memory (EEPROM type) and 13 I/O pins. It can be reprogrammable thousands times. Because the I/O just only 13 pins, I used a BCD to 7-segmen chip. So it will left a few I/O pins for expanded in the future. You can omitted this chip for timing sequence purpose and save one IC price, because the I/O just exactly enough.

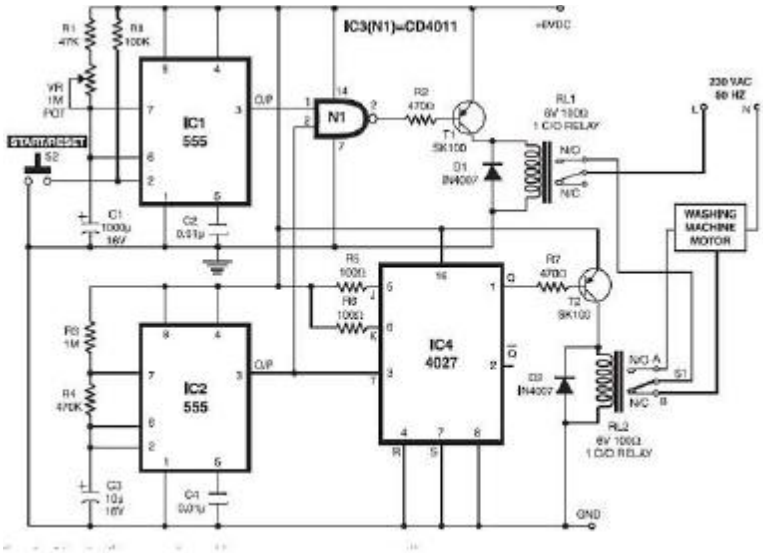
- I/O port A-0 = SET push button

- I/O port A-1 = RST push button
- I/O port A-2 = Reserved
- I/O port A-3 = Reserved
- I/O port A-4 = Reserved
- I/O port B-0 = Forward Relay (Run motor forward)
- I/O port B-1 = Reverse Relay (Run motor reverse)
- I/O port B-2 = Activated unit 7-segmen (multiplexed)
- I/O port B-3 = Activated ten 7-segmen (multiplexed)
- I/O port B-4 = BCD data A (for 7-segmen)
- I/O port B-5 = BCD data B (for 7-segmen)
- I/O port B-6 = BCD data C (for 7-segmen)
- I/O port B-7 = BCD data D (for 7-segmen)
- Also integrated power supply to run it modularly

The I/O can be configured as input pin or output pin bit-ly. It is up to you to choose the I/O pin number goes to what function, but it infect the program firmware of course. Once you choose, then it is just like that, except you also change both, the program and the hardware.

Working of Washing Machine

The direction of rotation can be controlled When switchS1 is in position A, coil L1 of the motor receives the current directly, whereas coil L2 receives the current with a phase shiftdue to capacitor C. So the rotor rotates in clockwise direction (see Fig. 2(a)). Whenswitch S1 is in position B, the reverse happens and the rotor rotates in anti-clockwisedirection Thus switch S1 can change the rotation direction.The motor cannot be reversed instantly. It needs abrief pause between switching directions, or else it mayget damaged. For this purpose, another spin direction control timer (IC2) is employed. It is realised with an IC 555. This timer gives an alternate „on‟ and „off‟ time duration of 10 seconds and 3 seconds, respectively.So after every l0 seconds of running (either in clockwise or anticlockwisedirection), the motor stops for a brief duration of 3 seconds. The values of R3 and R4 are calculated accordingly.The master timer is realised with monostable IC555 (IC1) and its „on‟ time is decided by the resistance of 1-megaohm potmeterVR. A 47-kilo-ohm resistor is added in series so thateven when the VR knob in zero resistance position ,the net series resistanceis not zero.


IC3(N1)=CD4011
+9VDC
R1
47K
R6
100K
VR
1M
POT
IC1
555
O/P
N1
R2
470Ω
T1
SK100
D1
IN4007
RL1
6V 100Ω
1 C/O RELAY
N/O
N/C
230 VAC
50 HZ
L
N
START/RESET
S2
C1
1000µ
16V
C2
0.01µ
WASHING
MACHINE
MOTOR
R5
100Ω
R8
100Ω
IC4
4027
R7
470Ω
T2
SK100
R3
1M
R4
470K
IC2
555
D2
IN4007
N/O A
S1
N/C B
RL2
6V 100Ω
1 C/O RELAY
C3
10µ
16V
C4
0.01µ
GND

UNIT 1

TWO MARKS

1. What is the function of program counter in 8085 microprocessor? [may 2013]

Program counter stores the address of next instruction to be fetched. Thus it is used as pointer to the instruction.

2. List the control and status signal of 8085 microprocessor and mention its need?[dec 12,08,06]

ALE (Output):

- ✓ Address Latch Enable: It occurs during the first clock cycle of a machine state and enables the address to get latched into the on chip latch of peripherals.
- ✓ The falling edge of ALE is set to guarantee set up and hold times for the address information.ALE can also be used to store the status information.

RD(Output3state):

READ: indicates the selected memory or I/0 device is to be read and that the Data Bus is available for the data transfer.

WR (Output3state):

WRITE: indicates the data on the Data Bus is to be written into the selected memory or I/0 location. Data is set up at the trailing edge of WR. 3stated during Hold and Halt modes.

IO/M,S_0,S_1:

- ✓ IO/M indicates whether input output operation or memory operation is being carried out.
- ✓ S_0 and S_1 indicated the type os machine cycle in progress.

READY(Input):

If Ready is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data .

If Ready is low, the CPU will wait for Ready to go high before completing the read or write cycle.

3. Specify the size of data, address, memory word and capacity of 8085 microprocessor[may 2011]

- ✓ Size of data bus =8 bits
- ✓ Size of address bus=16bit
- ✓ Size of memory word=8 bit
- ✓ Memory capacity=64kb

4. What is the need of ALE signal in 8085 microprocessor?[may 2010]

- ✓ This is used to demultiplex AD0 to AD7 lines to A0-A7 and D0-D7.
- ✓ The separation of address line and data line is achieved by connecting a external latch to AD0-AD7 lines and enabling the latch when ALE signal is active.

5. What is tristate logic?[june 2009]

- ✓ Logic output has two states LOW and HIGH corresponding to the logic value 0 and 1.
- ✓ However some output have a third electrical state that is not logical called high impedance or floating state.
- ✓ In this state the output behaves as if it is not even connected to circuit except small leakage current that may flow into or out of the output pin. This state is tristate logic.

6. List the five interrupt pins available in 8085[may 2010]

The five interrupt pins available in 8085 are

- TRAP
- RST 7.5
- RST 6.5
- RST 5.5
- INTR

7. What is stack and what is the function of stack pointer[Dec 07]

- ✓ The stack is a reserved area of memory in the RAM where temporary information may be stored.
- ✓ A 16 bit stack pointer is used to hold the address of most recent stack entry.

8.Define the function of parity flag and zero flag n 8085[may 12]

- Parity Flag (PF): This flag is used to indicate the parity of result. If lower order 8-bits of the result contains even number of 1‟s, the Parity Flag is set and for odd number of1‟s,the Parity Flag is reset.
- Zero Flag(ZF) :It is set; if the result of arithmetic or logical operation is zero else it is reset.

9. What is the important control signal in 8085 microprocessor[Dec08]

The important signal in 8085 microprocessor are: ALE,IO/M,RD/WR.

10. Why interfacing is needed for 1/0 devices?

Generally I/O devices are slow devices. Therefore the speed of I/O devices does not match with the speed of microprocessor. And so an interface is provided between system bus and I/O devices.

11. Why address bus is unidirectional?

The address is an identification number used by the microprocessor to identify or access a memory location or I / O device. It is an output signal from the processor. Hence the address bus is unidirectional.

12. What are machine language and assembly language programs?

The software developed using 1's and 0's are called machine language, programs. The software developed using mnemonics are called assembly language programs.

13. What are the basic units of a microprocessor ?

The basic units or blocks of a microprocessor are ALU, an array of registers and control unit

14. Steps involved to fetch a byte in 8085?

- The pc places the 16-bit memory address on the address bus
- The control unit sends the control signal RD to enable the memory chip
- The byte from the memory location is placed on the data bus

- The byte is placed in the instruction decoder of the microprocessor and the task is carried out according to the instruction.

15. Define instruction cycle, machine cycle and T-state?
- ✓ Instruction cycle is defined as the time required completing the execution of an instruction.
- ✓ Machine cycle is defined as the time required completing one operation of accessing memory, I/O or acknowledging an external request.
- ✓ T –cycle is defined as one subdivision of the operation performed in one clock period.

16. How many machine cycles does 8085 have, mention them?

The 8085 have seven machine cycles they are
- Opcode fetch
- Memory read
- Memory write
- I/O read
- I/O write
- Interrupt acknowledge

UNIT 2

1.What is an instruction?

An instruction is a binary pattern entered through an input device to command the icroprocessor to perform that specific function.

2.How many operations are there in the instruction set of 8085 microprocessor?

There are 74 operations in the 8085 microprocessor

3.List out the five categories of the 8085 instructions.Give e.g. of the instructions for each group?[December 2011]
- ✓ Data transfer group–MOV,MVI,LXI
- ✓ Arithmetic group–ADD,SUB,INR.
- ✓ Logical group-ANA,XRA,CMP.
- ✓ Branch group–JMP,JNZ,CALL.
- ✓ StackI/O and machine control group–PUSH,POP,IN,HLT.

4.Explain the use of branch instruction and give example.[may 2012]

JMP instruction permanently changes the program counter. A CALL instruction leaves Information on the stacks that the original program execution sequence can be resumed.

5.Explain the purpose of the I/O instructions IN and OUT[may 10]
- ✓ The IN instruction is used to move data from an I/O port into the accumulator.
- ✓ The OUT instruction is used to move data from the accumulator to an I/Oport.
- ✓ The IN and OUT instructions are used only on microprocessor, which usea separate address space for interfacing.

6.What is the difference between the shift and rotate instructions?

A rotate instruction is a closed loop instruction.that is the data moved out at one end is put back in at the other end.The shift instruction loses the data that is moved out of the last bit locations.

7. List the four instructions which control the interrupt structure of the 8085 microprocessor?[may 2013]

- ✓ DI(disable interrupts)
- ✓ EI(enable interrupts)
- ✓ RIM(read interrupt masks)
- ✓ SIM(set interrupt masks)

8.Mention the categories of machine control group of instruction [may 2013]

The instructions of 8085 can be categorized into the following five

1. EI
2. DI
3. NOP
4. HLT
5. SIM
6. RIM

9.Explain LDA, STAANDDAA instructions

LDA copies the data byte into the accumulator from the memory location specified by the 16-bit address.STA copies the data byte from the accumulator in the memory location specified by 16-bitaddress. DAA changes the content of the accumulator from binary to 4-bit BCD digits

10.Explain the different instruction format set [june 09]

The instruction set is grouped into the following formats One by the instruction MOV C,A Two byte instruction MVI A,39H Three byte instruction JMP2345H

11.What is the use of addressing modes, mention the different types?[may 2012]

The various formats of specifying the operands are called as addressing modes, it is used to access the operands or data. The different types are as follows

1. Immediate addressing
2. Register addressing
3. Direct addressing
4. Indirect addressing
5. Implicit addressing

12.Define stack and stack related instructions?[may 2013 and December 2012]

The stack is a group of memory locations in the R/W memory that is used for the temporary storage of binary information during the execution of the program. The stack related instructions are PUSH and POP

13.Why do we use XRAA instruction?

The XRAA instruction is used to clear the contents of the accumulator and store the value 00H

14.Compare CALL and PUSH instructions

CALL	PUSH
When CALL is executed the microprocessor automatically stores the16-bit address of the instruction next to CALL on the stack	The program uses the instruction PUSH to save the contents of the register pair on the stack
When CALL is executed the stack pointer is decremented by two	When PUSH is executed the stack pointer register is decremented by two

15.What are subroutine?

- ✓ Procedures are group of instructions stored as a separate program in memory and it is called from the main program in memory and it is called from the main program when ever required.
- ✓ The type of procedure depends on where the procedures are stored in memory. If it is in the same code segment as that of the main program then it is an ear procedure other wise it is a far procedure.

UNIT 3

1.What is mean by microcontroller

A device which contains the microprocessor with integrated peripherals like memory ,serial ports, parallel ports, timer/counter, interrupt controller, data acquisition interfaces like ADC, DAC is called microcontroller.

2.Explain DJNZ instruction of Intel8051 microcontroller ?[June 2013]

DJNZ Rn lDecrement the content of the register Rn and jump if not zero.
DJNZ direct, rel Decrement the content of direct8 –bit address and jump if not zero

3.State the function of RS1 and RS0 bits in the flag register of Intel8051microcontroller?

- RS1,RS0-Register bank select bits
- RS1,RS0-Bank
- Bank0
- Bank1
- Bank2
- Bank3

4.Give the alternate functions for the port pins of port3?[Dec 11 May 11]

- RDWRT1T0
- INT1INT0TXDRXD
- RD–Readdatacontroloutput
- WR–Writedatacontroloutput
- T1– Timer/counter1 external input or test pin T0– Timer/counter0 external input or test pin INT1–Interrupt1 input pin
- INT0– interrupt0 input pin
- TXD–Transmit data pin for serial port in UART mode
- RXD–Receive data pin for serial port in UART mode

5.Specify the single instruction, which clears the most significant bit of Bregister of 8051,without affecting the remaining bits.

Single instruction, which clears the most significant bit of Bregister of 8051,without affecting the remaining bits, is CLRB.7.

6.What are the applications of microcontroller[may 2012]

- Calculators
- Traffic light control system
- Game machine
- Military applications**.**

7.Give the memory size of 8051 microcontroller[may 2010]

The 8051 can access upto 64 kilobyte of program memory and 64 kilo byte of data memory.

8.Name the special functions registers available in 8051[Dec10]

- Accumulator
- BRegister
- ProgramstatusWord.
- Stackpointer.
- Datapointer
- Port0
- Port1
- Port2
- Port3
- Interrupt priority control register. Interrupt enable control register.

9.Explain the register IE format of 8051.[dec 2011]

- EAET2ES
- ET1EX1ET0EX0
- EA-Enableallcontrolbit.
- ET2-Timer2 interruptenablebit.
- ES- Enableserialport controlbit.
- ET1-EnableTimer1controlbit.
- EX1-Enableexternalinterrupt1controlbit.
- ET0-EnableTimer0controlbit.
- EX0-Enableexternalinterrupt0controlbit.

10.CompareMicroprocessorandMicrocontroller.

Microprocessor Microcontroller

1.Microprocessor contains ALU, general Microcontroller contains the circuitry of Purpose register counter,clock timing microprocessor and in addition it has Circuit and interrupt circuit.

Built-in ROM,RAM,I/O devices and counter

2.It has many instructions to move data It has many instructions to move between memory and CPU. Data between memory and CPU.

11.Name the five interrupt sources of 8051?[may 08 dec08]

- The interrupt are: Vector address
- External interrupt0:IE0:0003H
- Timers interrupt0: TF0:000BH
- External interrupt1:IE1:0013
- T imersinterrupt1:TF1:001BH
- Serial interrupt
- Receiveinterrupt:RI:0023H Transmitinterrupt:TI:0023H

12.Write a program to subtract the contents of RI of Bank0 from the contents of R00f Bank2.

```
MOVPSW,#10
MOVA,R0
MOVPSW,#00
SUBBA,R1
```

13.List the features of 8051 microcontroller?[may 04]

The features are

- Single supply +5 volt operation using HMOS technology.
- 4096 bytes program memory on chip (not on 8031)
- 128 data register banks
- Four register mode,16-bit timer/counter.
- Extensive Boolean processing capabilities.
- 64 KB external RAM size
- 32 bi-directional individually addressable I/O lines.
- 8 bit CPU optimized for control applications.

14.List the addressing modes of 8051?[dec09 may 13]

- Direct addressing
- Register addressing
- Register in direct addressing Implicit addressing Immediate addressing
- Index addressing
- Bit addressing

15. What are the modes of operation used in 8253?

Each of the three counters of 8253 can be operated in one of the following six modes of operation.

1. Mode 0 (Interrupt on terminal count) 2. Mode 1 (Programmable monoshot)
3. Mode 2 (Rate generator) 4. Mode 3 (Square wave generator)

UNIT 4

1.What is the use of 8051 chip?

- ✓ Intel's 8251A is a universal synchronous asynchronous receiver and transmitter compatible with Intel's Processors. This may be programmed to operate in any of the serial communication modes built into it.

- ✓ This chip converts the parallel data in to a serial stream of bits suitable for serial transmission. It is also able to receive a serial stream of bits and converts it in to parallel data bytes to be read by a microprocessor.

2.What is the various programmed data transfer method?

- ✓ Synchronous data transfer
- ✓ Asynchronous data transfer
- ✓ Interrupt driven data transfer

3.What is synchronous data transfer?

It is a data method which is used when the I/O device and the microprocessor match in speed. The transfer a data to or from the device ,the user program issues a suitable instruction addressing the device. The data transfer is completed at the end of the execution of this instruction.

4.What are the features used mode1 in 8255?[MAY 09]

- ✓ Two groups A and group B are available for stored Data transfer.
- ✓ Each group contains one 8-bit data I/O port an done4-bit control/data port.
- ✓ The8-bit data port can be either used as input or output port. The inputs and outputs both are latched.

5.What are the modes of operation used in 8253?[MAY 08 MAY 09]

Each of the three counters of 8253 can be operated in one of the following six modes of operation.

- ✓ Mode0 (Interrupt on terminal count)
- ✓ Mode1 (Programmable mono shot)
- ✓ Mode2 (Rate generator)
- ✓ Mode3 (Square wave generator)
- ✓ Mode4 (Software triggered strobe)
- ✓ Mode5 (Hardware triggered strobe)

6.Give the different types of command words used in 8259A

The command words of 8259A are classified in two groups

- ✓ Initialization command words(ICWs)
- ✓ Operation command words(OCWs)

7.Define scan counter?

- ✓ The scan counter has two modes to scan the key matrix and refresh the display. In the encoded mode ,the counter provides binary count that is to be externally decoded to provide the scan lines for keyboard and display
- ✓ .In the decoded scan mode, the counter internally decodes the least significant 2bit and provides a decoded1 out of 4scan on SL3-SL3.The keyboard and display both are in the same mode at a time.

8.What are the modes used in keyboard modes?

- ✓ Scanned Keyboard mode with 2Key Lockout

- ✓ Scanned keyboard with N-Key Rollover.
- ✓ Scanned Keyboard Special Error Mode.
- ✓ Scanned Matrix Mode.

9.What are the modes used in display modes?

1.Left Entry Mode

In the left entry mode,the data is entered from the left side of the display unit.

2.Right Entry Mode

In the right entry mode,the first entry to be displayed is entered on the right most display.

10.What is the use of modem control unit in 8251?

The modem control unit handles the modem handshake signals to coordinate the communication between them on demand the USART.

11.List the operation modes of 8255?

- ✓ Mode0-Simple Input/Output.
- ✓ Mode1-Strobe Input/Output (handshakemode)
- ✓ Mode2-Strobe bi-directional mode.

12.What is a control word?

It is a word stored in a register (control register) used to control the operation of a program digital device.

13.What is the purpose of control word written to control register in 8255?

The control words written to control register specify an I/O function for each I/O port. The bit D7of the control word determines either the I/O functions of the BSR function.

14.What is the size of ports in 8255?

- ✓ Port-A:8-bits
- ✓ Port-B:8-bits
- ✓ Port-CU :4-bits
- ✓ Port-CL:4-bits

15.What is an USART?

USART stands for universal Synchronous/Asynchronous Receiver/Transmitter.It is a programmable communication interface that can communicate by using either synchronous or asynchronous serial data.

UNIT 5

1.Write a program using 8051 assembly language to change the data 55h stored in the lower byte of the data pointer register to AAH using rotate instruction?

```
MOV DPL,#55H
MOV A,DPLRL A
LABEL :SJMP Label
```

2.Explain the content of the accumulator after the execution of the following program segments?

MOV A,#3CH
MOVR4,#66H ANLA,R4
A3C R466
A24

3.Write a program to load accumulator a, DPH and DPL with 30H?

MOVA,#30
MOVDPH
AMOVDPL,A

4.Write a program to perform multiplication of 2 Nos using 8051?

MOVA,#data1
MOVB,#data2
MULAB
MOVDPTR,#5000
MOV@DPTR,A(lower value)
INCDPTR
MOVA,B
MOVX@DPTR,A

5.Write a program to mask the 0th&7th bit using 8051?

MOVA,#data
ANLA,#81
MOVDPTR,#4500
MOVX@DPTR
,A LOOPS JMP LOOP

6.Write about CALL statement in 8051?

- ✓ There are two subroutine CALL instructions. they are
  *LCALL(Long CALL)
  *ACALL(Absolute CALL)
- ✓ Each increments the pc to the 1stbyte of the instruction & pushes them into this tack.

7. Write about the jump statement?

There are three forms of jump.they are

LJMP(Long-jump)-address16
AJMP(Absolute jump)-address11
Sjmp(shortjump)-relative address

8.Write a program to load accumulator DPH&DPL using 8051?

MOVA,#30
MOVDPH,
AMOVDPL,A

9.Write a program to find 2's complement using 8051?

```
MOVA,R0
CPLA INCA
```

10.Write a program to add two8-bit numbers using8051?

```
MOVA,#30H
ADDA,#50H
```

11.Write a program to swap two numbers using8051?

```
MOVA,#data
SWAPA
```

12.Write a program to subtract two8-bit numbers & exchange the digits using 8051?

```
MOVA,#9F
MOVR0,#40
SUBBA,R0
SWAPA
```

13.Write a program to subtract the contents of R1 of bank0 from the contents of R0 of bank2 using 8051?

```
MOVPSW,#10
MOVA,R0
MOVPSW,#00
SUBBA,R1
```

14.Explain the operating mode0 of 8051 serial ports?

In this mode serial enters & exits through RXD ,TXD output the shift clock8 bits are transmitted or received 8 data bits(LSB first).the baud rate is fixed at 1/12 the oscillator frequency.

15.Explain the operating mode 2 of 8051 serial ports?

In this mode 11bits are transmitted (through TXD)or received(through RXD)A start bit(0), 8 data bits(LSB first),a programmable 9thdata bit & a stop bit(1) ON transmit the 9thdata bit (TB*in SCON) can be assigned the value of 0or1.or fo reg :the parity bit (p,in the PSW)could be movedintoTB8.

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

III B.Tech II Sem (ECE)

15A04601 MICROPROCESSORS AND MICROCONTROLLERS

Course Outcomes:

After completion of this subject the students will be able to :

1. Do programming with 8086 microprocessors
2. Understand concepts of Intel x86 series of processors
3. Program MSP 430 for designing any basic Embedded System
4. Design and implement some specific real time applications Using MSP 430 low power microcontroller.

UNIT I

Introduction-8086 Architecture-Block Diagram, Register Organization, Flag Register, Pin Diagram, Timing and Control Signals, System Timing Diagrams, Memory Segmentation, Interrupt structure of 8086 and Interrupt Vector Table. Memory organization and memory banks accessing.

UNIT II

Instruction Formats -Addressing Modes-Instruction Set of 8086, Assembler Directives-Macros and Procedures.- Sorting, Multiplication, Division and multi byte arithmetic code conversion. String Manipulation instructions-Simple ALPs.

UNIT III

Low power RISC MSP430 – block diagram, features and architecture,Variants of the MSP430 family viz. MSP430x2x, MSP430x4x, MSP430x5x and their targeted applications, MSP430x5x series block diagram, Addressing modes, Instruction set Memory address space, on-chip peripherals (analog and digital), and Register sets.Sample embedded system on MSP430 microcontroller.

UNIT-IV

I/O ports pull up/down resistors concepts, Interrupts and interrupt programming. Watchdog timer. System clocks. Low Power aspects of MSP430: low power modes, Active vs Standby current consumption, FRAM vs Flash for low power & reliability.

Timer & Real Time Clock (RTC), PWM control, timing generation and measurements. Analog interfacing and data acquisition: ADC and Comparator in MSP430, data transfer using DMA.

UNIT-V

Serial communication basics, Synchronous/Asynchronous interfaces (like UART, USB, SPI, and I2C). UART protocol, I2C protocol, SPI protocol. Implementing and programming UART, I2C, SPI interface using MSP430, Interfacing external devices. Implementing Embedded Wi-Fi using CC3100

Text Books:

1. "Microprocessor and Microcontrollers", N. Senthil Kumar, M. Saravanan, S. Jeevanathan, Oxford Publishers. 1 st Edition, 2010
2. "The X86 Microprocessors , Architecture, Programming and Inerfacing" , Lyla B. Das, Pearson Publications, 2010
3. MSP430 microcontroller basics. John H. Davies, Newnes Publication, I st Edition, 2008

References:

http://processors.wiki.ti.com/index.php/MSP430_LaunchPad_Low_Power_Mode
http://processors.wiki.ti.com/index.php/MSP430_16-Bit_Ultra-Low_Power_MCU_Training

UNIT-I

UNIT-1

INTRODUCTION:

Microprocessor acts as a CPU in a microcomputer. It is present as a **single ICchip** in a

microcomputer.

Microprocessor is the heart of the machine.

A Microprocessor is a device, which is capable of

1. Receiving Input 2 Performing Computations 3. Storing data and instructions
4. Display the results 5. Controlling all the devices that perform the above 4 functions.

The device that performs tasks is called Arithmetic Logic Unit (ALU). A single chip called Microprocessor performs these tasks together with other tasks.

"A MICROPROCESSOR is a multipurpose programmable logic device that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to those instructions and provides results as output."

EVOLUTION OF MICROPROCESSORS:

The microprocessor age began with the advancement in the IC technology to put all necessary functions of a CPU into a single chip.

Intel started marketing its first microprocessor in the name of Intel 4004 in 1971. This was a4-bit microprocessor having 16-pins in a single chip of PMOS technology. This was called the **first generation microprocessor**. The Intel 4004 along with few other devices was used for making calculators. The 4004 instruction set contained only 45 instructions. Later in 1971, INTEL Corporation released the 8008 – an extended 8-bit version of the 4004 microprocessor. The 8008 addressed an expanded memory size (16KB) and 48 instructions.

Limitations of first generation microprocessors is small memory size, slow speed and instruction set limited its usefulness.

Second generation microprocessors:

The second generation microprocessor using NMOS technology appeared in the market in the year 1973. The Intel 8080, an 8-bit microprocessor, of NMOS technology was developed in the year 1974 which required only two additional devices to design a functional CPU. The advantages of second generation microprocessors were

- Large chip size (170x200 mil) with 40-pins. More chips on decoding circuits.
- Ability to address large memory space (64-K Byte) and I/O ports(256).
- More powerful instruction sets. Dissipate less power.

- Better interrupt handling facilities. Cycle time reduced to half (1.3 to 9 m sec.)
- Sized 70x200 mil) with 40-pins. Less Support Chips Required
- Used Single Power Supply Faster Operation

The 8080 microprocessor addresses more memory and execute additional instructions, but executes them 10 times faster than 8008.The 8080 has memory of 64 KB whereas for 8008 16 KB only. In 1977, INTEL, introduced 8085 which was an updated version of 8080 last 8-bit processor.

The main advantages of 8085 were its internal clock generator, internal system controller and higher clock frequency.

Third Generation Microprocessor:

In 1978, INTEL released the 8086 microprocessor, a year later it released 8088. Both devices were 16 bit microprocessors, which executed instructions in less than 400ns.The 8086 and 8088 addresses 1MB of memory and rich instruction set to 246.16-bit processors were designed using HMOS technology. The Intel 80186 and 80188 were the improved versions of Intel 8086 and8088, respectively. In addition to 16-bit CPU, the 80186 and 80188 had programmable peripheral devices integrated on the same package.

Fourth Generation Microprocessor:

The single chip 32-bit microprocessor was introduced in the year 1981 by Intel as iAPX 432. The other 4thgeneration microprocessors were; Bell Single Chip Bellmac-32, Hewlett-Packard, National NSl 6032,Texas Instrument99000. Motorola 68020 and 68030. The Intel in the year 1985 announced the 32-bit microprocessor(80386). The 80486 has already been announced and is also a 32-bit microprocessor.

The 80486 is a combination 386 processor a math coprocessor, and a cache memory controller on a single chip.

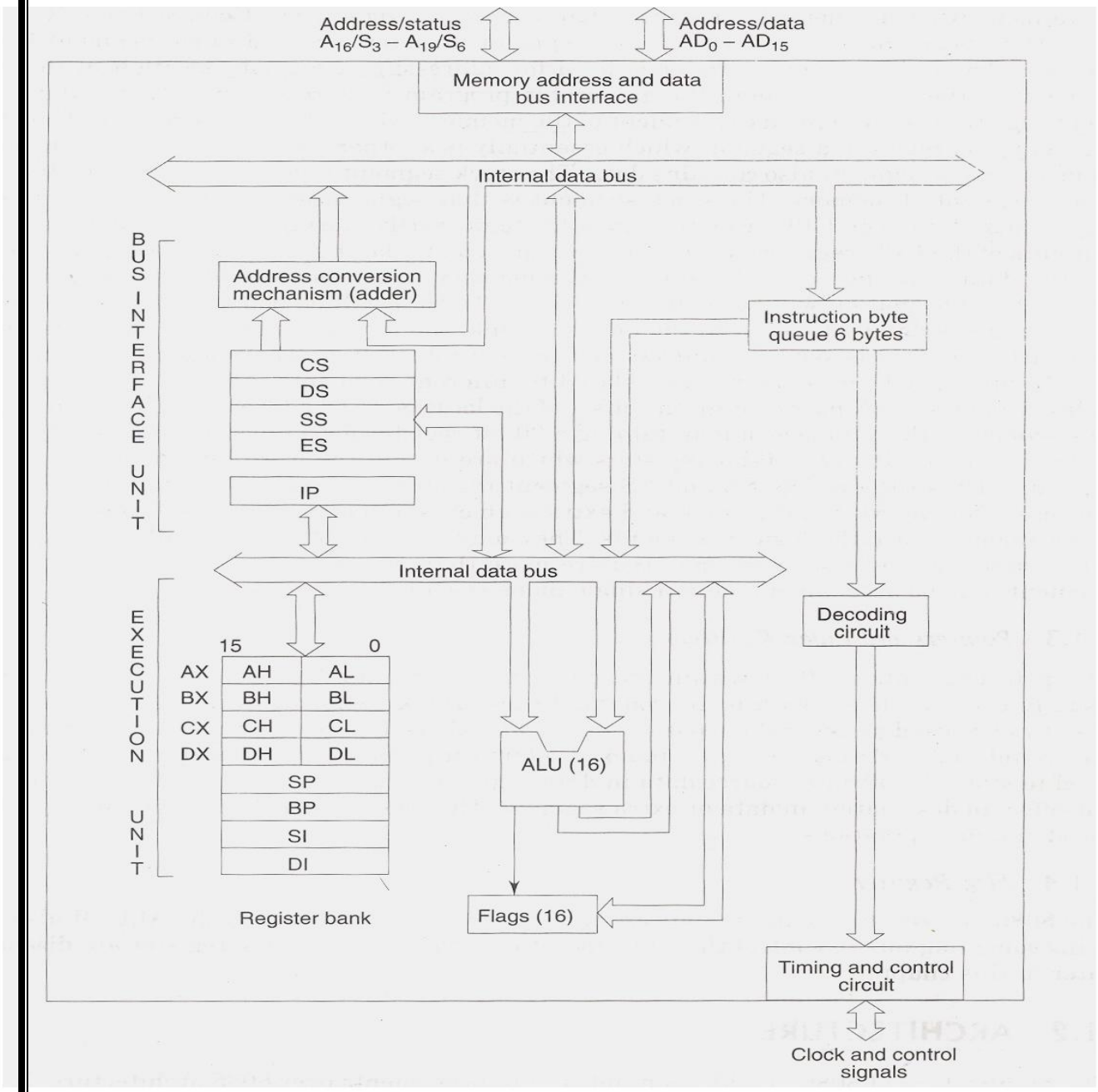
The Pentium is a 64-bit superscalar processor. It can execute more than one instruction at a time and has a full 64-bit data bus and 32-bit address bus. Its performance is double than 80486.

Features of 8086:

•It is a 16-bit μp.
•8086 has a 20 bit address bus can access up to 2^20 memory locations (1 MB).
•It can support up to 64K I/O ports. •It provides 14, 16 -bit registers.
•It has multiplexed address and data bus AD0- AD15 and A16 – A19.
•It requires single phase clock with 33% duty cycle to provide internal timing
•8086 is designed to operate in two modes, Minimum anMaximum.
•It can pre-fetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.
•It requires +5V power supply. •A 40 pin dual in line package.

Architecture of 8086:

- 8086 has two blocks BIU and EU.
- The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.
- EU executes instructions from the instruction byte queue.
- Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as **Pipelining.** This results in efficient use of the system bus and system performance.
- BIU contains Instruction queue, Segment registers, IP, address adder.
- EU contains control circuitry, Instruction decoder, ALU, Flag register.



Bus Interface Unit:

- It provides full 16 bit bidirectional data bus and 20 bit address bus.
- The BIU is responsible for performing all external bus operations.

Specifically it has the following functions:

- Instructions fetch Instruction queuing, Operand fetch and storage, Address relocation and Bus control.
- The BIU uses a mechanism known as an instruction stream queue to implement pipeline architecture.
- This queue permits pre-fetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by pre-fetching the next sequential instruction.
- These pre-fetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.
- The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory.
- These intervals of no bus activity, which may occur between bus cycles, are known as **idle state**.
- If the bus is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.
- The BIU also contains a dedicated adder which is used to generate the 20 bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.
- For example: The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.
- The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

Execution Unit:

- The EU extracts instructions from top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands.
- During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.
- If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue.
- When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.
- When ever this happens, the BIU automatically resets the queue and then begins to fetch instructions from

this new location to refill the queue.

Register organization of 8086:

The 8086 has four groups of the user accessible internal registers. They are the instruction pointer, four data registers, four pointer and index register, four segment registers. The 8086 has a total of fourteen 16-bit registers including a 16 bit register called the ***status register***, with 9 of bits implemented for status and control flags.

There are four different 64 KB segments for instructions, stack, data and extra data. To Specify where in 1 MB of processor memory these 4 segments are located the processor uses four segment registers:

•**Code segment** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

•**Stack segment** (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

•**Data segment** (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment.DS register can be changed directly using POP and LDS instructions.

•**Accumulator** register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX. AL in this case contains the low order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

•**Base** register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

•**Count** register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low order byte of the word, and CH contains the high-order byte. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation,.

•**Data** register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

•**The following registers are both general and index registers:**

•**Stack Pointer** (SP) is a 16-bit register pointing to program stack.

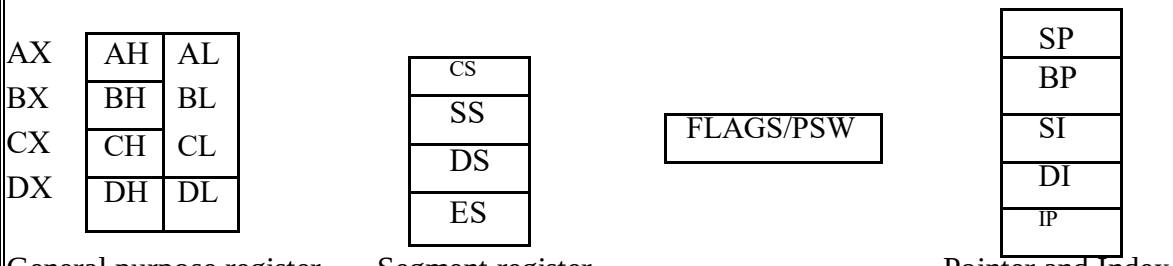
•**Base Pointer** (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

•**Source Index** (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions.

•**Destination Index** (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Instruction Pointer (IP) register acts as a program counter for 8086. It points to the address of the next instruction to be executed Its content is automatically incremented when the program execution of a program proceeds further. The contents of IP and CS register are used to compute the memory address of the instruction code to be fetched.

General data registers:



General purpose register Segment register Pointer and Index

Flag register of 8086: It is a 16-bit register, also called flag register or Program Status Word (PSW). Seven bits remain unused while the rest nine are used to indicate the conditions of flags. The status flags of the register are shown below in Fig.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	1
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AC	X	PF	X	CY

X=Defined

- Out of nine flags, six are condition flags and three are control flags. The control flags
- are TF (Trap), IF (Interrupt) and DF (Direction) flags, which can be set/reset by the
- programmer, while the **condition flags [OF (Overflow), SF (Sign), ZF (Zero), AF (Auxiliary**
- **Carry), PF (Parity) and CF (Carry)]** are set/reset depending on the results of some arithmetic or logical operations during program execution.
- **CF is set** if there is a carry out of the MSB position resulting from an addition operation or if a borrow is needed out of the MSB position during subtraction.
- **PF is set** if the lower 8-bits of the result of an operation contains an even number of 1's. AF is set if there is a carry out of bit 3 resulting from an addition operation or borrow required from bit 4 into bit 3 during subtraction operation.
- **ZF is set** if the result of an arithmetic or logical operation is zero.
- **SF is set** if the MSB of the result of an operation is 1. SF is used with unsigned numbers.
- OF is used only for signed arithmetic operation and is set if the result is too large to be fitted in the number of bits available to accommodate it.

  The three control flags of 8086 are TF, IF and DF. These three flags are programmable, i.e., can be set/reset by the programmer so as to control the operation of the processor.

- **When TF (trap flag) is set (=1),** the processor operates in **single stepping mode**—i.e., pausing after each

instruction is executed. This mode is very useful during program development or program debugging.

- When an interrupt is recognized, TF flag is cleared. When the CPU returns to the main program from ISS (interrupt service subroutine), by execution of IRET in the last line of ISS, TF flag is restored to its value that it had before interruption.
- TF cannot be directly set or reset. So indirectly it is done by pushing the flag register on the stack, changing TF as desired and then popping the flag register from the stack.
- **When IF (interrupt flag) is set,** the maskable interrupt INTR is enabled otherwise disabled (i.e., when IF = 0).
- **IF can be set by executing STI instruction and cleared by CLI instruction.** Like TF flag, when an interrupt is recognized, IF flag is cleared, so that INTR is disabled. In the last line of ISS when IRET is encountered, IF is restored to its original value. When 8086 is reset, IF is cleared, i.e., resetted.
- DF (direction flag) is used in string (also known as block move) operations. **It can be set by STD instruction and cleared by CLD.** If DF is set to 1 and MOVS instruction is executed, the contents of the index registers DI and SI are automatically decremented to access the string from the highest memory location down to the lowest memory location.

ADDRESSING MODES OF 8086:

Addressing modes indicates way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction.

According to the flow of instruction execution, the instruction may be categorized as:
Sequential Control flow instructions Control Transfer instructions

- **Sequential Control flow instructions:** In this type of instruction after execution control can be transferred to the next immediately appearing instruction in the program.

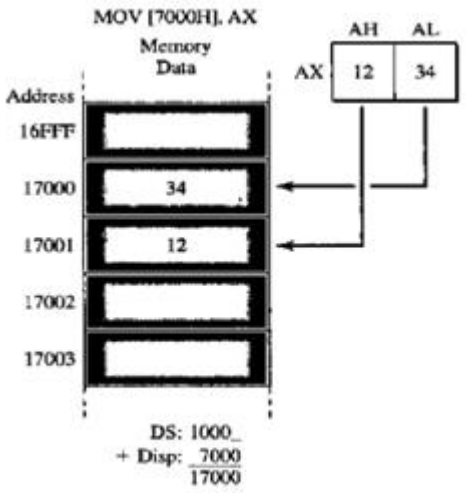
  The addressing modes for sequential control transfer instructions are as follows:
- **Immediate addressing mode:** In this mode, immediate is a part of instruction and appears in the form of successive byte or bytes.

  Example: MOV CX, 0007H; Here 0007 is the immediate data

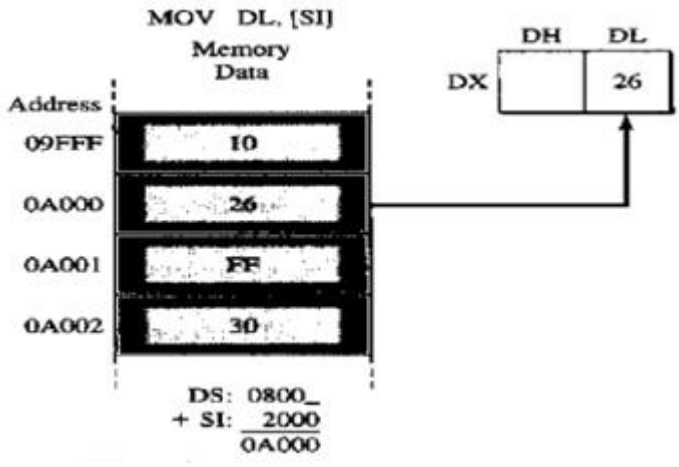
- **Direct Addressing mode:** In this mode, the instruction operand specifies the memory address where data is located.

  Example: MOV AX, [5000H]; Data is available in 5000H memory location
  Effective Address (EA) is computed using 5000H as offset address and content of DS as segment address.

  EA=10H*DS+5000H



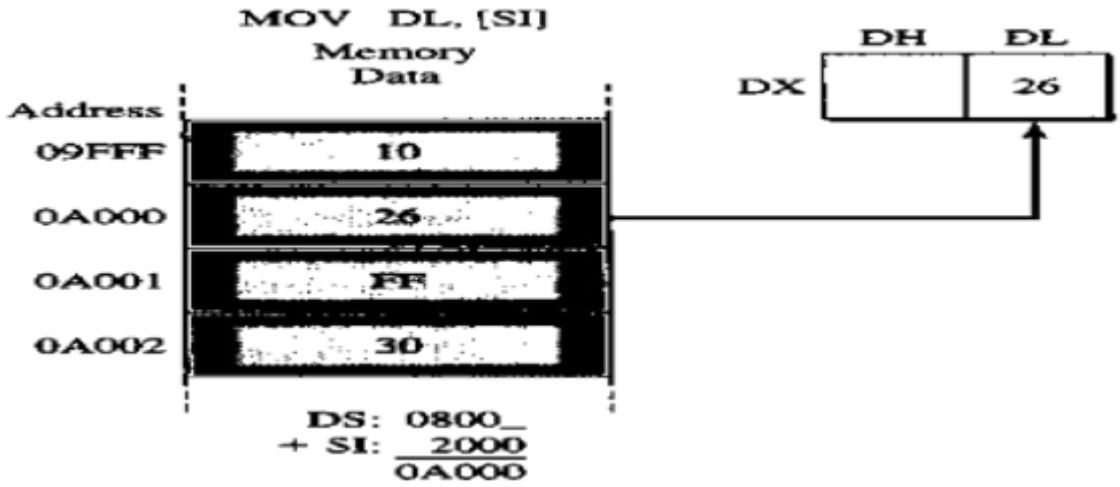
- **Register Addressing mode:** In this mode, the data is stored in a register and it is referred using particular register. All the registers except IP may be used in this mode.
  Example: MOV AX, BX;
- **Register Indirect addressing mode:** In this mode, instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers.
  Example: MOV AX, [BX];
  EA=10H * DS +[BX]
- **Indexed Addressing mode:** 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides. DS and ES are default segments for index registers SI and DI.
  DS=0800H, SI=2000H,
  MOV DL, [SI]
  Example: MOV AX, [SI];
  EA=10H * DS + [SI]



- **Register Relative Addressing mode:** In this mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI, DI in the default segments.

Example: MOV AX, 50H [BX];

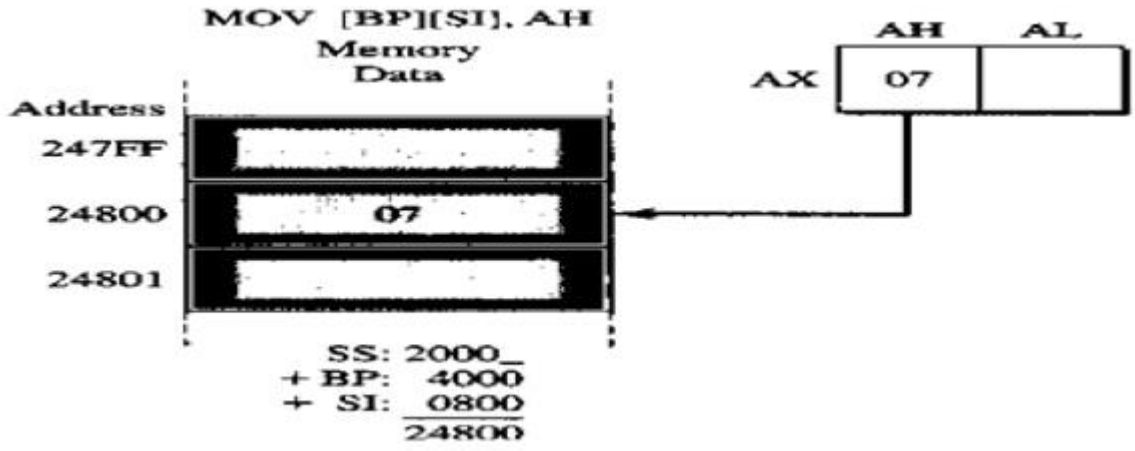
EA=10H * DS + 50H +[BX]



- **Based Indexed Addressing mode:** In this mode, the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

  Example: MOV AX, [BX] [SI];

EA=10H * DS + [BX] + [SI]



- **Relative Based Indexed Addressing mode:** In this mode, 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

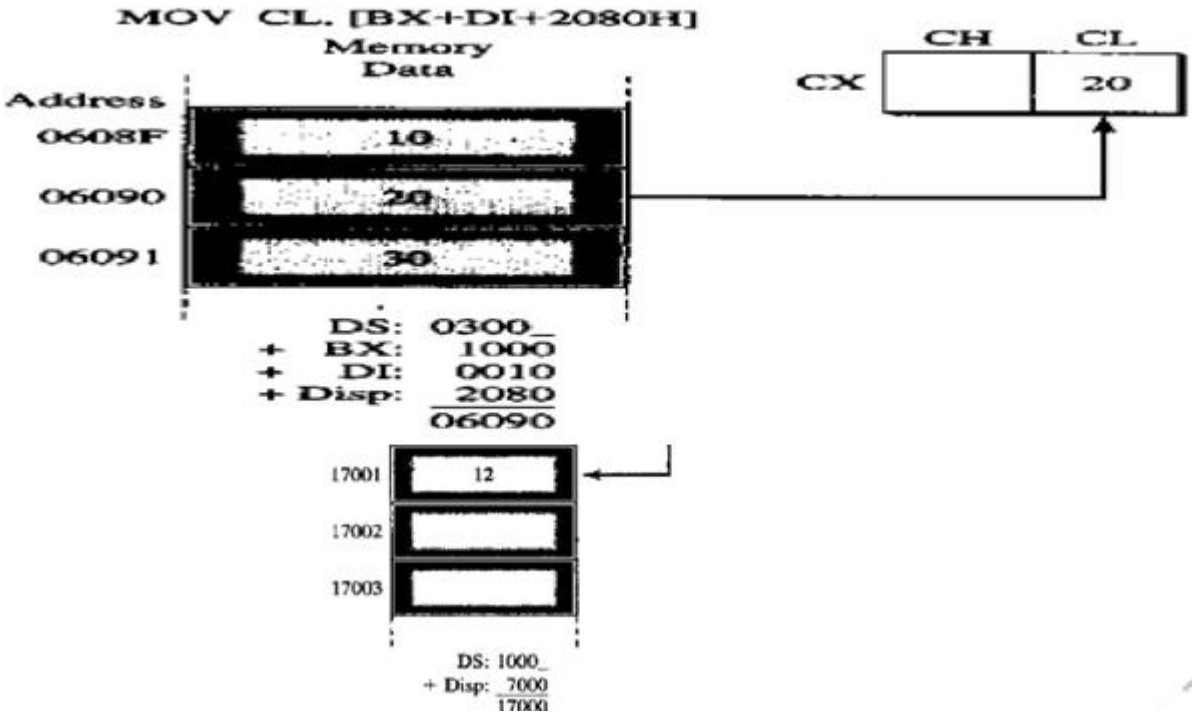
  Example: MOV AX, 50H [BX] [SI];

- **Control Transfer Instructions:** In control transfer instruction, the control can be transferred to some predefined address or the address somehow specified in the instruction after their execution.

  For the control transfer instructions, the addressing modes depend upon whether the destination location is within the segment or different segments. It also depends upon the method of passing the destination address to the processor. Depending on this control transfer instructions are categorized as follows:

EA=10H * DS + 50H + [BX] +[SI]

- **Intra**



segment Direct mode: In this mode, the address to which control is to be transferred lies in the same segment in which control transfer instruction lies and appears directly in the instruction as an immediate displacement value.

- **Intra segment Indirect mode:** In this mode, the address to which control is to be transferred lies in the same segment in which control transfer instruction lies but it is passed to the instruction indirectly.
- **Inter segment Direct mode:** In this mode, the address to which control is to be transferred lies in a different segment in which control transfer instruction lies and appears directly in the instruction as an immediate displacement value.
- **Inter segment Indirect mode:** In this mode, the address to which control is to be transferred lies in a different segment in which control transfer instruction lies but it is passed to the instruction indirectly.

Memory Segmentation for 8086:

8086, via its 20-bit address bus, can address 220 = 1,048,576 or 1 MB of different memory locations. Thus the memory space of 8086 can be thought of as consisting of 1,048,576 bytes or 524,288 words. The memory map of 8086 is shown in Figure where the whole memory space starting from 00000 H to FFFFF H is divided into 16 blocks—each one consisting of 64KB.

1 MB memory of 8086 is partitioned into 16 segments—each segment is of 64 KB length. Out of these 16 segments, only 4 segments can be active at any given instant of time— these are code segment, stack

segment, data segment and extra segment. The four memory segments that the CPU works with at any time are called currently active segments. Corresponding to these four segments, the registers used are Code Segment Register (CS), Data Segment Register (DS), Stack Segment Register (SS) and Extra Segment Register (ES) respectively. Each of these four registers is 16-bits wide and user accessible—i.e., their content scan be changed by software.

The code segment contains the instruction codes of a program, while data, variables and constants are held in data segment. The stack segment is used to store interrupt and subroutine return addresses. The extra segment contains the destination of data for certain string instructions. Thus 64 KB are available for program storage (in CS) as well as for stack (in SS) while128 KB of space can be utilized for data storage (in DS and ES).One restriction on the base address (starting address) of a segment is that it must reside on a 16-byte address memory—examples being 00000 H, 00010 H or 00020 H, etc.

Memory segmentation, as implemented for 8086, gives rise to the following advantages:

- Although the address bus is 20-bits in width, memory segmentation allows one towork with registers having width 16-bits only.
- It allows instruction code, data, stack and portion of program to be more than 64 KBlong by using more than one code, data, extra segment and stack segment.
- In a time-shared multitasking environment when the program moves over from oneuser's program to another, the CPU will simply have to reload the four segmentregisters with the segment starting addresses assigned to the current user's program.
- User's program (code) and data can be stored separately.
- Because the logical address range is from 0000 H to FFFF H, the same can be loadedat any place in the memory.

Instruction Set of 8086:

There are 117 basic instructions in the instruction set of 8086.The instruction set of 8086 can be divided into the following number of groups, namely:

1. Data copy / Transfer instructions
2. Arithmetic and Logical instructions
3. Branch instructions
4. Loop instructions
5. Machine control instructions
6. Flag Manipulation instructions
7. Shift and Rotate instructions
8. String instructions

Data copy / Transfer instructions: The data movement instructions copy values from one location to another. These instructions include **MOV, XCHG, LDS, LEA, LES, PUSH, PUSHF, PUSHFD, POP,POPF, LAHF, AND SAHF.**

MOV The MOV instruction copies a word or a byte of data from source to a destination. The destination can be a register or a memory location. The source can be a register, or memory location or immediate data. MOV instruction does not affect any flags. The mov instruction takes several different forms:

The MOV instruction cannot:

1. Set the value of the CS and IP registers.
2. Copy value of one segment register toa nother segment register (should copy to general register first). MOV CS, DS (Invalid)

3. Copy immediate value to segment register (should copy to general register first). MOV CS, 2000H (Invalid)

Example:

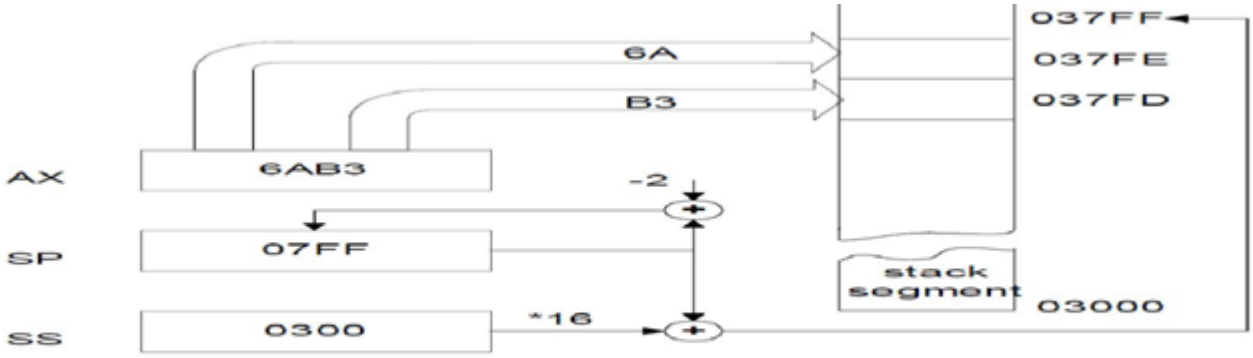
```
ORG 100h
MOV AX, 0B800h ;        set AX = B800h
MOV DS, AX ;            copy value of AX to DS.
MOV CL, 'A' ;           CL = 41h (ASCII code).
```

The XCHG Instruction: Exchange This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them, may be a memory location. However, exchange of data contents of two memory locations is not permitted.

Example: MOV AL, 5 ; AL = 5 MOV
BL, 2 ; BL = 2
XCHG AL,BL ; AL = 2, BL = 5

PUSH: Push to stack; this instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

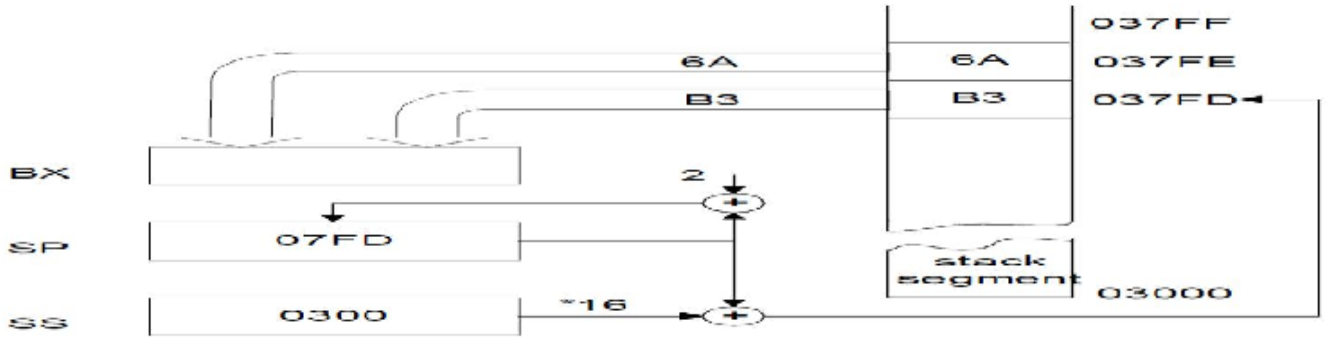
1. PUSH AX
2. PUSH DS
3. PUSH [500OH] ; Content of location 5000H and 5001 H in DS are pushed onto the stack.



The effect of PUSH AX instruction

POP: Pop from Stack this instruction when executed loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

1. POP BX
2. POP DS
3. POP [5000H]



The effect of POP BX instruction

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2for each pop operation.

LAHF: Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except overflow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in

AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

LEA: Load Effective Address The load effective address instruction loads the offset of an operand in the specified register. This instruction is similar to MOV, MOV is faster than LEA.

LEA cx, [bx+si] ; CX (BX+SI) mod 64K **If bx=2f00 H; si=10d0H cx 3fd0H**

The LDS AND LES instructions:

• LDS and LES load a 16-bit register with offset address retrieved from a memory location then load either DS or ES with a segment address retrieved from memory.

This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers.

• LDS and LES instructions obtain a new far address from memory.

– offset address appears first, followed by the segment address

• This format is used for storing all 32-bit memory addresses.

• A far address can be stored in memory by the assembler.

LDS BX,DWORD PTR[SI] BL
[SI];

BH [SI+1]

DS [SI+3:SI+2]; in the data segment

LES BX,DWORD PTR[SI]

BL [SI];

BH [SI+1]

ES [SI+3:SI+2]; in the extra segment

I/O Instructions: The 80x86 supports two I/O instructions: in and out15. They take the forms: In ax, port

in ax, dx out port,

ax out dx, ax

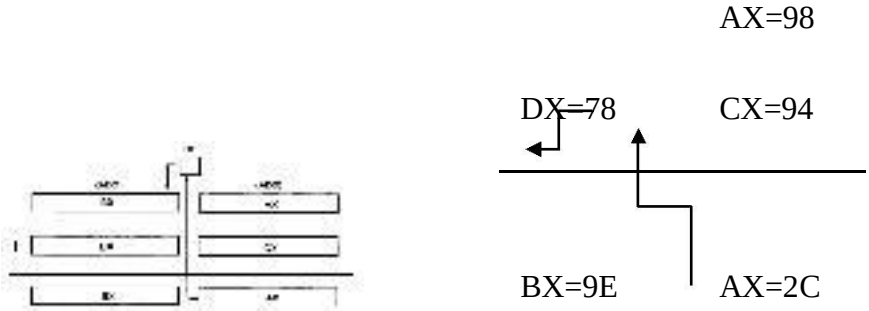
port is a value between 0 and 255.

The in instruction reads the data at the specified I/O port and copies it into the accumulator. The out instruction writes the value in the accumulator to the specified I/O port.

Arithmetic instructions: These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions.

The ADD and ADC instructions: The add instruction adds the contents of the source operand to the destination operand. For example, **add ax, bx** adds bx to ax leaving the sum in the ax register. **Add computes dest :=dest+source while adc computes dest :=dest+source+C where C represents the value in the carry flag**. Therefore, if the carry flag is clear before execution, adc behaves exactly like the add instruction.

Example: CF=1



Both instructions affect the flags identically. They set the flags as follows:

- The overflow flag denotes a signed arithmetic overflow.
- The carry flag denotes an unsigned arithmetic overflow.
- The sign flag denotes a negative result (i.e., the H.O. bit of the result is one).
- The zero flag is set if the result of the addition is zero.
- The auxiliary carry flag contains one if a BCD overflow out of the L.O. nibble occurs.
- The parity flag is set or cleared depending on the parity of the L.O. eight bits of the result. If there is even number of one bits in the result, the ADD instructions will set the parity flag to one (to denote even parity). If there is an odd number of one bits in the result, the ADD instructions clear the parity flag (to denote odd parity).

The INC instruction: The increment instruction adds one to its operand. Except for carry flag, inc sets the flags the same way as Add ax, 1 same as inc ax. The inc operand may be an eight bit, sixteen bit. The inc instruction is more compact and often faster than the comparable add reg, 1 or add mem, 1 instruction.

The AAA and DAA Instructions

The aaa (ASCII adjust after addition) and daa (decimal adjust for addition) instructions support BCD

arithmetic. BCD values are decimal integer coded in binary form with one decimal digit(0..9) per nibble. ASCII (numeric) values contain a single decimal digit per byte, the H.O. nibble of the byte should contain zero (30 ….39).

The aaa and daa instructions modify the result of a binary addition to correct it for ASCII or decimal arithmetic. For example, to add two BCD values, you would add them as though they were binary numbers and then execute the daa instruction afterwards to correct the results.

Note: These two instructions assume that the add operands were proper decimal or ASCII values. If you add binary(non-decimal or non-ASCII) values together and try to adjust them with these instructions, you will not produce correct results.

Aaa (which you generally execute after an add, adc, or xadd instruction) checks the value in al for BCD overflow. It works according to the following basic algorithm:

if ((al and 0Fh) > 9 or (AuxC =1)) then al := al + 6

else

ax := ax +6 endif

ah := ah + 1

AuxC := 1 ;Set auxilliary carry Carry := 1 ; and carry flags. Else

AuxC := 0 ;Clear auxilliary carry Carry := 0 ; and carry flags.

add al=08 +06; al=0E >9 al=0E+06=04
ah=00+01=01
al=04+03=08, now al<9,
so only clear ah=0

endif

al := al and 0Fh

The aaa instruction is mainly useful for adding strings of digits where there is exactly one decimal digit per byte in a string of numbers.

The **daa instruction** functions like aaa except it handles packed BCD values rather than the one digit per byte unpacked values aaa handles. As for aaa, daa's main purpose is to add strings of BCD digits (with two digits per byte). The algorithm for daa is

if ((AL and 0Fh) > 9 or (AuxC = 1)) then
al := al + 6
AuxC := 1 ;Set Auxilliary carry.
Endif
if ((al > 9Fh) or (Carry = 1)) then
al := al + 60h
Carry := 1; ;Set carry flag.
Endif

al=24+77=9B, as B>9 add 6 to al
al=9B+06=A1, as higher nibble A>9, add 60 to al, al=A1+60=101
Note: if higher or lower nibble of AL <9 then no need to add 6 to AL

EXAMPLE:

Assume AL = 0 0 1 1 0 1 0 1, ASCII 5 BL = 0 0 1 1 1 0 0 1, ASCII 9

ADDAL,BL Result: AL= 0 1 1 0 1 1 1 0 = 6EH,which is incorrect BCD
AAA Now AL = 00000100, unpacked BCD 4.

CF = 1 indicates answer is 14 decimal

NOTE: OR AL with 30H to get 34H, the ASCII code for 4. The AAA instruction works only on the AL register. The AAA instruction updates AF and CF, but OF, PF, SF, and ZF are left undefined.

EXAMPLES:

AL = 0101 1001 = 59 BCD ; BL = 0011 0101 = 35 BCD
ADD AL, BL AL = 1000 1110 = 8EH

DAA Add 01 10 because 1110 > 9 AL = 1001 0100 = 94 BCD AL
= 1000 1000 = 88 BCD BL = 0100 1001 = 49 BCD

ADD AL, BL AL = 1101 0001, AF=1

DAA Add 0110 because AF =1, AL = 11101 0111 = D7H
1101 > 9 so add 0110 0000

AL = 0011 0111= 37 BCD, CF =1

The DAA instruction updates AF, CF, PF, and ZF. OF is undefined after a DAA instruction.

The SUBTRACTION instructions: SUB, SBB, DEC, AAS, and DAS

The sub instruction computes the value dest :=dest - src. The sbb instruction computes dest :=dest src - C.

The sub, sbb, and dec instructions affect the flags as follows:

• They set the zero flag if the result is zero. This occurs only if the operands are equal for sub and sbb. The dec instruction sets the zero flag only when it decrements the value one.

• These instructions set the sign flag if the result is negative.

• These instructions set the overflow flag if signed overflow/underflow occurs.

• They set the auxiliary carry flag as necessary for BCD/ASCII arithmetic.

• They set the parity flag according to the number of one bits appearing in the result value.

• The sub and sbb instructions set the carry flag if an unsigned overflow occurs. Note that the dec instruction does not affect the carry flag.

The aas instruction, like its aaa counterpart, lets you operate on strings of ASCII numbers with one decimal digit (in the range 0..9) per byte. This instruction uses the following algorithm:

if ((al and 0Fh) > 9 or AuxC = 1) then al
:= al - 6

ah := ah - 1

AuxC := 1 ;Set auxilliary carry
Carry := 1 ; and carry flags. else

AuxC := 0 ;Clear Auxilliary carry
Carry := 0 ; and carry flags.

endif

al := al and 0Fh

The das instruction handles the same operation for BCD values, it uses the following algorithm:

if ((al and 0Fh) > 9 or (AuxC = 1)) then al

:= al -6

AuxC = 1
endif

if (al > 9Fh or Carry = 1) then al
:= al - 60h

Carry := 1 ;Set the Carry flag.
Endif

EXAMPLE:

ASCII 9-ASCII 5 (9-5)

AL = 00111001 = 39H = ASCII 9 BL =
001 10101 = 35H = ASCII 5

SUB AL, BL Result: AL = 00000100 = BCD 04 and CF = 0
AAS Result: AL = 00000100 = BCD 04 and CF = 0

no borrow required ASCII
5-ASCII 9 (5-9)

Assume AL = 00110101 = 35H ASCII 5 and
BL = 0011 1001 = 39H = ASCII 9

SUB AL, BL Result: AL = 11111100 = - 4 in 2s complement and CF =1
AAS Result: AL = 00000100 = BCD 04 and CF = 1, borrow needed

EXAMPLES:

AL 1000 0110 86 BCD ; BH 0101 0111 57 BCD

SUB AL,BH AL 0010 1111 2FH, CF = 0

DAS Lower nibble of result is 1111, so DAS automatically

subtracts 0000 0110 to give AL = 00101001 29 BCD AL
0100 1001 49 BCD BH 0111 0010 72 BCD SUB AL,BH
AL 1101 0111 D7H, CF = 1

DAS Subtracts 0110 0000 (- 60H) because 1101 in upper nibble > 9 AL =
01110111= 77 BCD, CF=1 CF=1 means borrow was needed

The CMP Instruction: The cmp (compare) instruction is identical to the sub instruction with one crucial difference– it does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub, the generic form is **cmpdest, src**

Consider the following cmp instruction: cmp ax, bx

This instruction performs the computation ax-bx and sets the flags depending upon the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if ax = bx. This is the only time ax-bx produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative.

O: The overflow flag is set after a cmp operation if the difference of ax and bx produced an overflow or underflow.

C: The carry flag is set after a cmp operation if subtracting bx from ax requires a borrow. This occurs only

when ax is less than bx where ax and bx are both unsigned values.

The Multiplication Instructions: MUL, IMUL, and AAM: This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general-purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX.

The mul instruction, with an **eight bit operand**, multiplies the al register by the operand and **stores the 16 bit result in ax.** So

mul operand	(Unsigned)	MUL BL	i.e. AL * BL; Al=25 * BL=04; AX=00 (AH) 64 (AL)
imul operand	(Signed)	IMUL BL	i.e. AL * BL; AL=09 * BL=-2; AL * 2's comp(BL) AL=09 * BL (0EH)=7E; 2's comp (7e)=-82

The aam (ASCII Adjust after Multiplication) instruction, adjust an unpacked decimal value after multiplication. This instruction operates directly on the ax register. It assumes that you've multiplied two eight bit values in the range 0..9 together and the result is sitting in ax (actually, the result will be sitting in al since 9*9 is 81,the largest possible value; ah must contain zero). This instruction divides ax by 10 and leaves the quotient in ah and the remainder in al: mul bl; al=9, bl=9 al*bl=9*9=51H; AX=00(AH) 51(AL); AAM ; first hexadecimal value is converted to decimal value i.e. 51 to 81; al=81D; second convert packed BCD to unpacked BCD, divide AL content by 10 i.e. 81/10 then AL=01, AH =08; AX = 0801

EXAMPLE:

AL 00000101 unpacked BCD 5 BH
00001001 unpacked BCD 9 MUL BH
AL x BH; result in AX
AX = 00000000 00101101 = 002DH
AAM AX = 00000100 00000101 = 0405H, which is unpacked BCD for 45.

If ASCII codes for the result are desired, use next instruction OR AX, 3030H Put 3 in upper nibble of each byte.
AX = 0011 0100 0011 0101 = 3435H, which is ASCII code for 45

The Division Instructions: DIV, IDIV, and AAD

The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16division or a 16/8 division. These instructions take the form:

Div reg	For unsigned division
Div mem	
Idiv reg	For signed division
Idiv mem	

The div instruction computes an unsigned division. If the operand is an eight bit operand ,div divides the ax register by the operand leaving the quotient in al and the remainder(modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx ax by the operand leaving the quotient in ax and the remainder in .

Note: If an overflow occurs (or you attempt a division by zero) then the80x86 executes an INT 0 (interrupt zero).

The aad (ASCII Adjust before Division) instruction is another unpacked decimal operation. It splits apart unpacked binary coded decimal values before an ASCII division operation. The aad instruction is useful for

other operations. The algorithm that describes this instruction is
al := ah*10 + al AX=0905H; BL=06; AAD; AX=AH*10+AL=09*10+05=95D;
convert decimal to hexadecimal; 95D=5FH; al=5f;
DIV BL; AL/BL=5F/06; AX=05(AH)0F(AL) ah := 0

EXAMPLE:

AX = 0607H unpacked BCD for 67 decimal CH = 09H, now adjust to binary

AAD Result: AX = 0043 = 43H = 67 decimal DIV
CH Divide AX by unpacked BCD in CH Quotient:
AL = 07 unpacked BCD Remainder:
AH = 04 unpacked BCD Flags undefined after DIV
NOTE: If an attempt is made to divide by 0, the 8086 will do a type 0 interrupt.

CBW-Convert Signed Byte to Signed Word: This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. The CBW operation must be done before a signed byte in AL can be divided by another signed byte with the IDIV instruction. CBW affects no flags.

EXAMPLE:

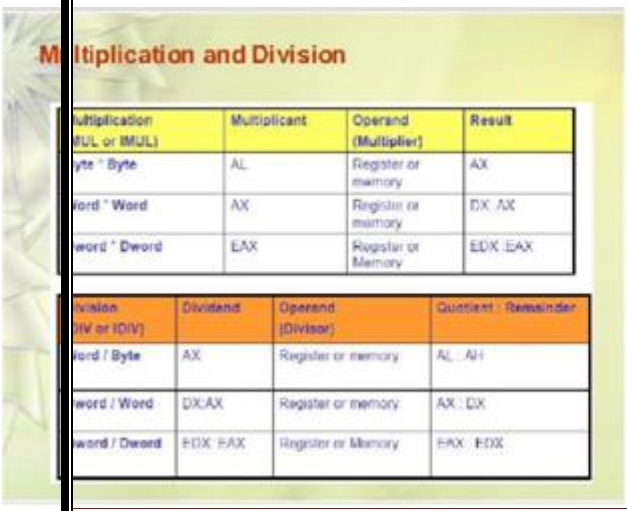
AX = 00000000 10011011 155 decimal

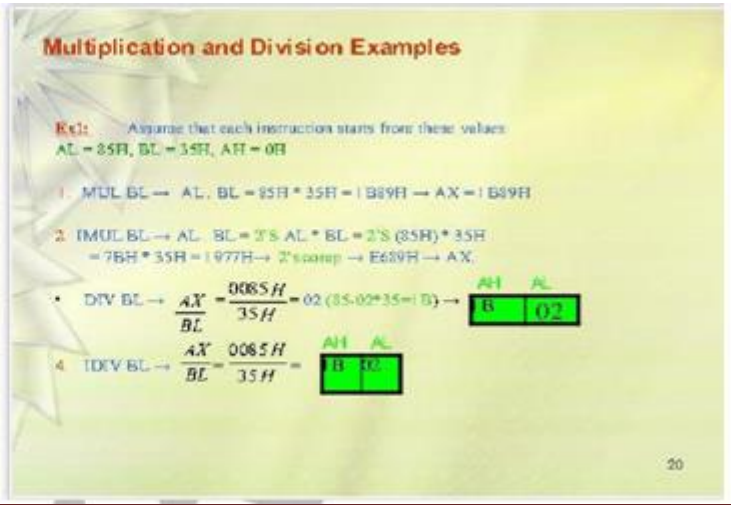
CBW Convert signed byte in AL to signed word in AX
Result: AX = 11111111 10011011 155 decimal

CWD-Convert Signed Word to Signed Double word: CWD copies the sign bit of a word in AX to all the bits of the DX register. In other words it extends the sign of AX into all of DX. The CWD operation must be done before a signed word in AX can be divided by another signed word with the IDIV instruction. CWD affects no flags.

EXAMPLE:

DX = 00000000 00000000
AX = 11110000 11000111 3897 decimal
CWD Convert signed word in AX to signed double word in DX:AX
Result DX = 11111111 11111111
AX = 11110000 11000111 3897 decimal





Logical, Shift, Rotate and Bit Instructions: The 80x86 family provides five logical instructions, four rotate instructions, and three shift instructions. The logical instructions are and, or, xor, test, and not; the rotates are ror, rol, rcr, and rcl; the shift instructions are shl/sal, shr, and sar.

The Logical Instructions: AND, OR, XOR, and NOT: The 80x86 logical instructions operate on a bit-by-bit basis. Except not, these instructions affect the flags as follows:

• They clear the carry flag.

• They clear the overflow flag.

• They set the zero flag if the result is zero, they clear it otherwise.

• They copy the H.O. bit of the result into the sign flag.

• They set the parity flag according to the parity (number of one bits) in the result.

• They scramble the auxiliary carry flag.

The not instruction does not affect any flags.

The **AND** instruction sets the zero flag if the two operands do not have any ones in corresponding bit positions. **AND AX, BX**

The **OR** instruction will only set the zero flag if both operands contain zero. **OR AX, BX**

The **XOR** instruction will set the zero flag only if both operands are equal. Notice that the xor operation will produce a zero result if and only if the two operands are equal. Many programmers commonly use this fact to clear a sixteen bit register to zero since an instruction of the form

xor reg16, reg16; XOR AX, AX is shorter than the comparable mov reg,0 instruction.

You can use the and instruction to set selected bits to zero in the destination operand. This is known as *masking out* data; Likewise, you can use the or instruction to force certain bits to one in the destination operand;

The Shift Instructions: SHL/SAL, SHR, SAR: The 80x86 supports three different shift instructions (shl and sal are the same instruction):shl (shift left), sal (shift arithmetic left), shr (shift right), and sar (shift arithmetic right).

SHL/SAL: These instructions move each bit in the destination operand one bit position to the left the number of times specified by the count operand. Zeros fill vacated positions at the L.O. bit; the H.O. bit shifts into the carry flag.

The shl/sal instruction sets the condition code bits as follows:

• If the shift count is zero, the shl instruction doesn't affect any flags.

• The carry flag contains the last bit shifted out of the H.O. bit of the operand.

• The overflow flag will contain one if the two H.O. bits were differen tprior to a single bit shift. The overflow flag is undefined if the shift count is not one.

• The zero flag will be one if the shift produces a zero result.

• The sign flag will contain the H.O. bit of the result.

• The parity flag will contain one if there are an even number of one bits inthe L.O. byte of the result.

• The A flag is always undefined after the shl/sal instruction.

The shift left instruction is especially useful for packing data. For example, suppose you have two nibbles in al and ah that you want to combine. You could use the following code to do this:

shl ah, 4 ;

or al, ah ;Merge in H.O. four bits.

Of course, al must contain a value in the range 0..F for this code to work properly (the shift left operation automatically clears the L.O. four bits of ah before the or instruction).

SHL OPERATION

H.O. four bits of al are not zero before this operation, you can easily clear them with an and instruction:

shl ah, 4 ;Move L.O. bits to H.O. position. and
al, 0Fh ;Clear H.O. four bits.
or al, ah ;Merge the bits.

Since shifting an integer value to the left one position is equivalent to multiplying that value by two, you can also use the **shift left instruction for multiplication by powers of two**:

shl ax, 1 ;Equivalent to AX*2 shl
ax, 2 ;Equivalent to AX*4 shl ax, 3
;Equivalent to AX*8

SAR: Thesar instruction shifts all the bits in the destination operand to the right one bit, replicating the H.O. bit.

The sar instruction's main purpose is to perform a signed division by some power of two. Each shift to the right divides the value by two. Multiple right shifts divide the previous shifted result by two, so multiple shifts produce the following results:

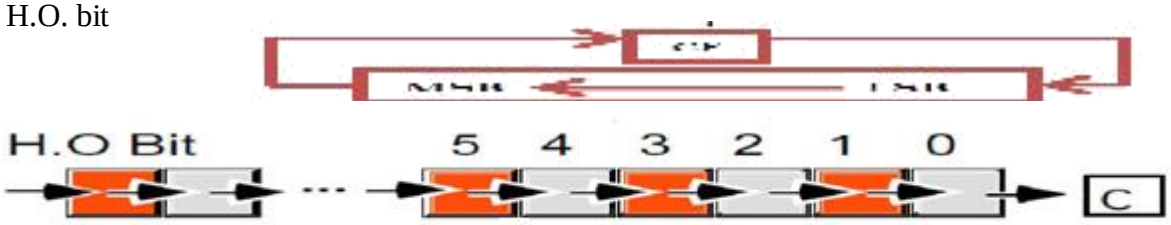
SAR OPERATION

sar ax, 1 ;Signed division by 2 sar ax,
2 ;Signed division by 4 sar ax, 3
;Signed division by 8 sar ax, 4 ;Signed
division by 16 sar ax, 5 ;Signed
division by 32 sar ax, 6 ;Signed
division by 64 sar ax, 7 ;Signed
division by 128 sar ax, 8 ;Signed
division by 256

There is a very important difference between the sar and idiv instructions. The idiv instruction always truncates towards zero while sar truncates results toward the smaller result. For positive results, an arithmetic shift right by one position produces the same result as an integer division by two. However, if the quotient is

negative, idiv truncates towards zero while sar truncates towards negative infinity.

SHR: The shr instruction shifts all the bits in the destination operand to the right one bit shifting a zero into the H.O. bit



SHR

OPERATION

The shift right instruction is especially useful for unpacking data. shifting an unsigned integer value to the right one position is equivalent to dividing that value by two, you can also use the shift right instruction for division by powers of two:

shr ax, 1 ;Equivalent to AX/2 shr ax,
2 ;Equivalent to AX/4 shr ax, 3
;Equivalent to AX/8 shr ax, 4
;Equivalent to AX/16

The Rotate Instructions: RCL, RCR, ROL, and ROR

The rotate instructions shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions re-circulate through the operand. They include rcl(rotate through carry left), rcr(rotate through carry right), rol(rotate left), and ror(rotate right). These instructions all take the forms :rcldest, count roldest, count rcr dest, count ror dest, count

RCL: The rcl(rotate through carry left), as its name implies, rotates bits to the left, through the carry flag, and back into bit zero on the right. The rcl instruction sets the flag bits as follows:

- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- If the shift count is one, rcl sets the overflow flag if the sign changes as a result of the rotate. If the count is not one, the overflow flag is undefined.
- The rcl instruction does not modify the zero, sign, parity, or auxiliary carry flags.

RCL OPERATION

RCR: The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit. This instruction sets the flags in a manner analogous to rcl:

- The carry flag contains the last bit shifted out of the L.O. bit of the operand.
- The rcr instruction does not affect the zero, sign, parity, or auxiliary carry flags.

RCR

ROL:

rcl

OPERATION

The rol instruction is similar to the instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit ,rather than the carry, into bit zero.

Rol also copies the output of the H.O. bit into the carry flag . The rol instruction sets the flags identically to rcl. Other than the source of the value shifted into bit zero, this instruction behaves exactly like the rcl instruction.

Like shl, the rol instruction is often useful for packing and unpacking data.

ROL OPERATION

ROR: The ror instruction relates to the rcr instruction in much the same way that the rol instruction relates to rcl. That is, it is almost the same operation other than the source of the input bit to the operand. Rather than shifting the previous carry flag into the H.O. bit of the destination operation, ror shifts bit zero into the H.O. bit.

ROR OPERATION

String Instructions: A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes or words on 8086.**All members of the 80x 86 families support five different string instructions: MOVS, CMPS, SCAS, LODS, AND STOS.**

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the movs instruction moves a sequence of bytes from one memory location to another. The cmps instruction compares two blocks of memory. The scas instruction scansa block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the movs instruction to copy a string, you need a source address, a destination address, and a count (the number of string elements to move).The operands for the string instructions include:

- **theSI (source index) register,**
- **the DI (destination index) register,**
- **the CX (count) register,**
- **theAX register, and**
- **the direction flag in the FLAGS register.**

The REP/REPE/REPZ and REPNZ/REPNE Prefixes: The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

Field:

Label repeat mnemonic operand ; comment

For MOVS:

```
Rep movs {operands}
```

For CMPS:

Repe cmps {operands} Repz cmps {operands} Repne cmps {operands} Repnz cmps {operands}

For SCAS:

Repe scas {operands} Repz scas {operands} Repne scas {operands} repnz scas {operands}

For STOS:

repstos {operands}

When specifying the repeat prefix before a string instruction, the string instruction repeats cx times. Without the repeat prefix, the instruction operates only on a single byte,word, or double word.

If the direction flag is clear, the CPU increments si and di after operating upon eachstring element. If the direction flag is set, then the 80x86 decrements si and di after processing eachstring element. The direction flag may be set or cleared using the cld (clear direction flag) and std (setdirection flag) instructions.

The MOVS Instruction: The movsb (move string, bytes) instruction fetches the byte at address ds:si, stores it at address es:di, and then increments or decrements the si and di registers by one. If the rep prefix is present, the CPU checks cx to see if it contains zero. If not, then it moves the byte from ds:si to es:di and decrements the cx register. This process repeats until cx becomes zero. The syntax is :

{REP} MOVSB {REP} MOVSW

The CMPS Instruction: The cmps instruction compares two strings. The CPU compares the string referenced by es:di to the string pointed at by ds:si. Cx contains the length of the two strings (whenusing the rep prefix). The syntax is: **{REPE} CMPSB {REPE} CMPSW**

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match or length of the string cx=0. The**repe** prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and cx is greater than zero.

The SCAS Instruction: The scas instruction, by itself, compares the value in the accumulator (al or ax) against the value pointed at by es:di and then increments (or decrements) di by one or two. The CPU sets the flags according to the result of the comparison. When using the repne prefix (repeat while not equal), scas scans the string searching for the first string element which is equal to the value in the accumulator. The scas instruction takes the following forms:**{REPNE} SCASB {REPNE} SCASW The STOS**

Instruction: The stos instruction stores the value in the accumulator at the location specified byes:di. After storing the value, the CPU increments or decrements di depending upon the state of the direction flag. Its primary use is to initialize arrays and strings to a constant value. **{REP} STOSB**

{REP} STOSW

The LODS Instruction: The lods instruction copies the byte or word pointed at by ds:si into the al or ax register, after which it increments or decrements the si register by one or two.**{REP} LODSB**

{REP} LODSW

Flag Manipulation and Processor Control Instructions: These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions.

The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

CLC - Clear carry flag **CMC - Complement carry flag** **STC - Set carry flag**
CLD - Clear direction flag **STD - Set direction flag** **CLI - Clear interrupt flag**
STI - Set interrupt flag

These instructions modify the carry(CF), direction(DF) and interrupt(IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and auto-increment or auto-decrement modes.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

WAIT - Wait for Test input pin to go low **HLT - Halt the processor** **NOP - No Operation** **ESC - Escape to external device like NDP (numeric co-processor)** **LOCK - Bus lock instruction prefix.**

After executing the **HLT instruction**, the processor enters the halt state. The two ways to pull it out of the halt state are to reset the processor or to interrupt it.

When **NOP instruction** is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP by one. It then continues with further execution after 4 clock cycles.

ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices.

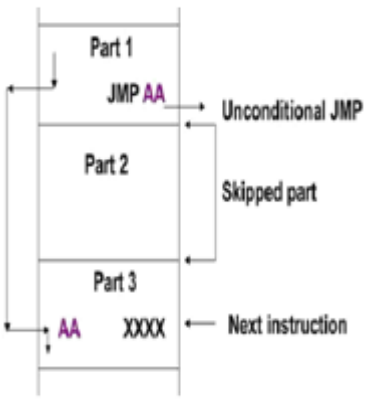
The **LOCK prefix** may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems.

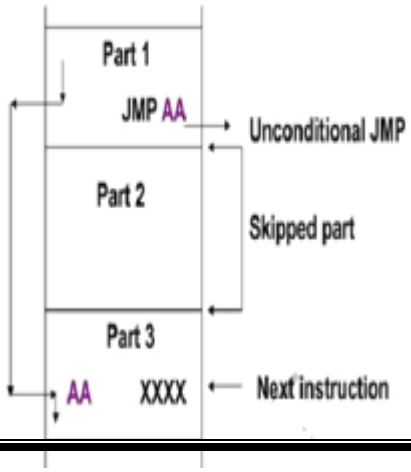
The **WAIT instruction** when executed holds the operation of processor with the current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further execution.

Program Flow Control Instructions: The control transfer instructions are used to transfer the control from one memory location to another memory location. In 8086 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions.

Unconditional Jumps: The jmp (jump) instruction unconditionally transfers control to another point in the program. Intra segment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment.

JMP Address





Unconditional jump **Conditional jump**

Conditional Jump: The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like if…..then statement. The conditional jumps test one or more bits in the status register to see if they match some particular pattern. If the pattern matches, control transfers to the target location. If the condition fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some instructions, for example, test the conditions of the sign, carry, overflow and zero flags.

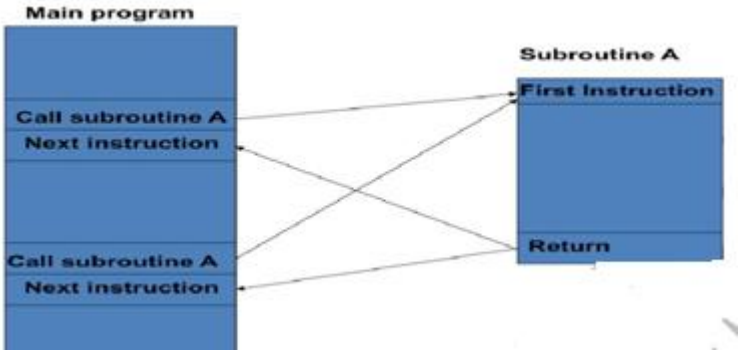
Definition	Description	Condition[1]
	Jump Based on Unsigned Data	
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JA / JNBE	Jump above or jump not below/ equal	C=0 & Z=0
JAE / JNB	Jump above/ equal or jump not below	C=0
JB / JNAE	Jump below or jump not above/ equal	C=1
JBE / JNA	Jump below/ equal or jump not above	C=1 or Z=1
	Jump Based on Signed Data	
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JG / JNLE	Jump greater or jump not less/ equal	N=0 & Z=0
JGE / JNL	Jump greater/ equal or jump not less	N=0
JL / JNGE	Jump less or jump not greater/ equal	N=1
JLE / JNG	Jump less/ equal or jump not greater	N=1 or Z=1
	Arithmetic Jump	
JS	Jump sign set	N=1
JNS	Jump no sign set	N=0
JC	Jump carry set	C=1
JNC	Jump no carry set	C=0
JO	Jump overflow set	O=1
JNO	Jump not overflow set	O=0
JP / JPE	Jump parity even	P=1
JNP / JPO	Jump parity odd	P=0

Loop Instruction:

- These instructions are used to repeat a set of instructions several times.
- Format: LOOP Short-Label
- Operation: (CX) ← (CX)-1
- Jump is initialized to location defined by short label if CX≠0. Otherwise, execute next sequential instruction.
- Instruction LOOP works with respect to contents of CX. CX must be preloaded with a count that represents the number of times the loop is to be repeat.
- Whenever the loop is executed, contents at CX are first decremented then checked to determine if they are equal to zero.
- If CX=0, loop is complete and the instruction following loop is executed.
- If CX ≠ 0, content return to the instruction at the label specified in the loop instruction.

- **LOOP AGAIN is <u>almost same</u> as: DEC CX, JNZ AGAIN**

SUBROUTINE & SUBROUTINE HANDILING INSTRUCTIONS: CALL, RET



- A subroutine is a special segment of program that can be called for execution from any point in a program.
- An assembly language subroutine is also referred to as a "procedure".
- Whenever we need the subroutine, a single instruction is inserted in to the main body of the program to call subroutine.
- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL saves a return address on the
  - stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.
- To branch a subroutine the value in the IP or CS and IP must be modified.
- After execution, we want to return the control to the instruction that immediately follows the one called the subroutine i.e., the original value of IP or CS and IP must be preserved.
- Execution of the instruction causes the contents of IP to be saved on the stack. (this time (SP) ← (SP) -2)
- A new 16-bit (near-proc, mem16, reg16 i.e., Intra Segment) value which is specified by the instructions operand is loaded into IP.
- Examples: CALL 1234H

  CALL BX

  CALL [BX]

Return Instruction: RET instruction removes an address from the stack so the program returns to the instruction following the CALL

- Every subroutine must end by executing an instruction that returns control to the main program. This is the return (RET) instruction.
- By execution the value of IP or IP and CS that were saved in the stack to be returned back to their corresponding registers. (this time (SP) ← (SP)+2)

MACROS: The macro directive allows the programmer to write a named block of source statements, then use that name in the source file to represent the group of statements. During the assembly phase, the assembler automatically replaces each occurrence of the macro name with the statements in the macro definition.

Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if used repeatable. Procedures or subroutines take up less space, but the increased overhead of saving and restoring addresses and parameters can make them slower. In summary, the advantages and disadvantages of macros are,

Advantages

- Repeated small groups of instructions replaced by one macro
- Errors in macros are fixed only once, in the definition
- Duplication of effort is reduced
- In effect, new higher level instructions can be created
- Programming is made easier, less error prone
- Generally quicker in execution than subroutines

Disadvantages

In large programs, produce greater code size than procedures

When to use Macros

- To replace small groups of instructions not worthy of subroutines
- To create a higher instruction set for specific applications
- To create compatibility with other computers
- To replace code portions which are repeated often throughout the program

Minimum Mode 8086 System

•In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.

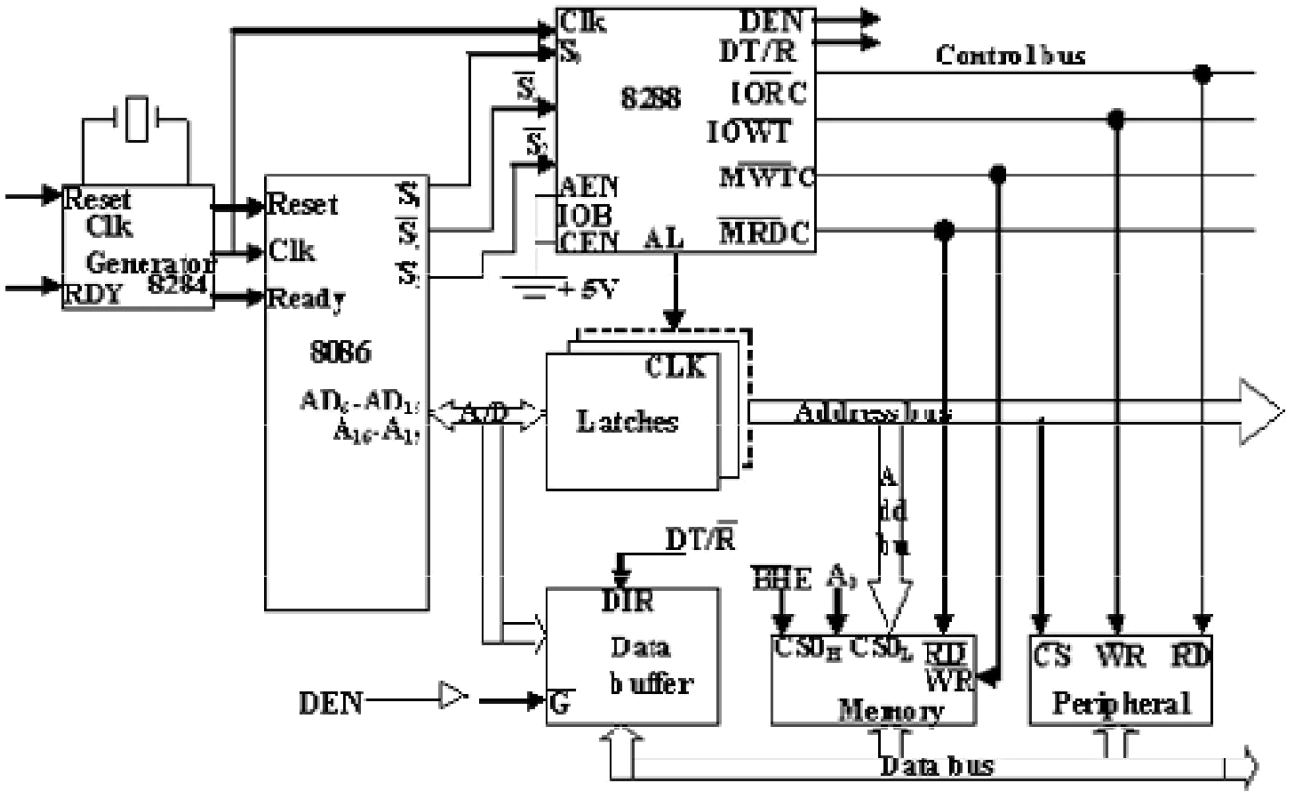
•In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.

•The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

•Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.

•Transreceivers are the bidirectional buffers and sometimes they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signals.

•They are controlled by two signals namely, DEN and DT/R.



Maximum Mode 8086 System.

The DEN signal indicates the direction of data, i.e. from or to the processor. The system contains memory for the monitor and users program storage.

•Usually, EPROMs are used for monitor storage, while RAM for users program storage. A system may contain I/O devices.

•The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.

•The read cycle begins in T1 with the assertion of address latch enable (ALE) signal and also M / IO signal. During the negative going edge of this signal, the valid address is latched on the local bus.

•The BHE and A0 signals address low, high or both bytes. From T1 to T4 , the M/IO signal indicates a memory or I/O operation.

•At T2, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD) control signal is also activated in T2.

•The read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus.

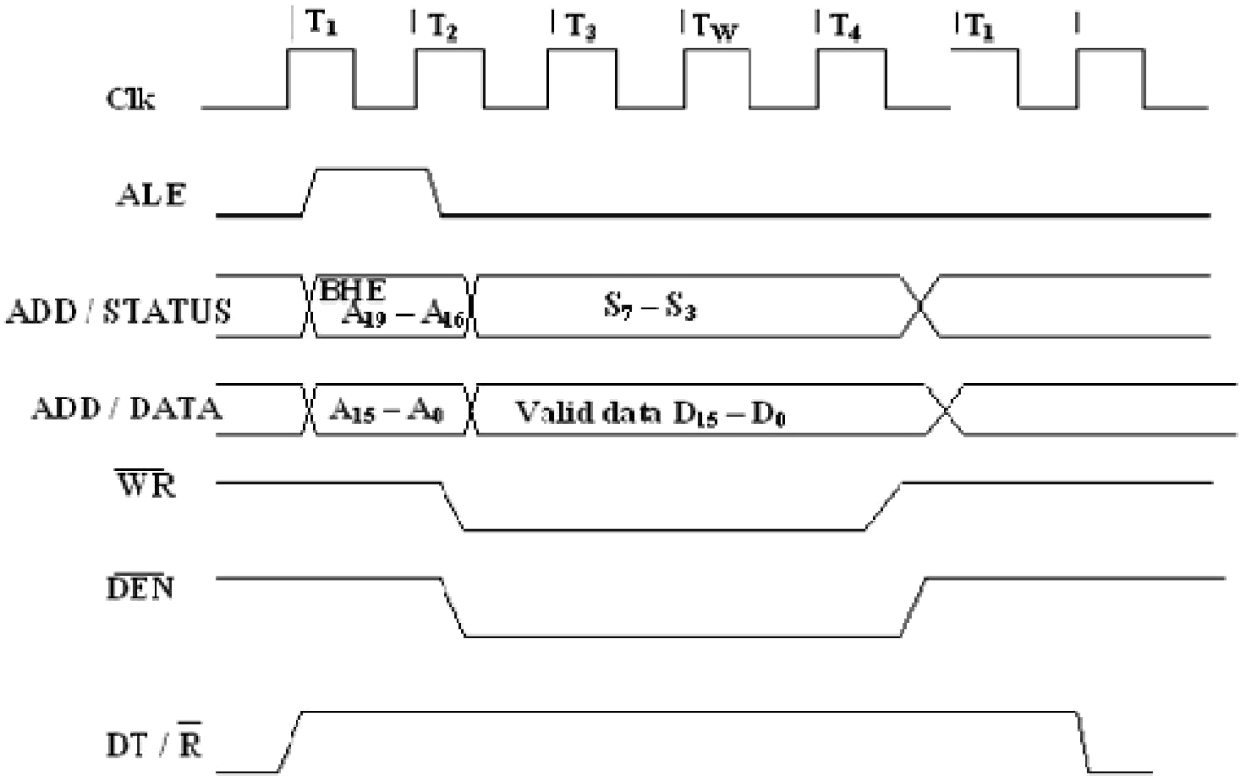
•The addressed device will drive the READY line high. When the processor returns the read signal to

high level, the addressed device will again tristate its bus drivers.

•A write cycle also begins with the assertion of ALE and the emission of the address. The M/IO signal is again asserted to indicate a memory or I/O operation. In T2, after sending the address in T1, the processor sends the data to be written to the addressed location.

•The data remains on the bus until middle of T4 state. The WR becomes active at the beginning of T2 (unlike RD is somewhat delayed in T2 to provide time for floating).

•The BHE and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.



Write Cycle Timing Diagram for Minimum Mode

Maximum Mode 8086 System

•In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.

In this mode, the processor derives the status signal S2, S1, S0. Another chip called bus controller derives the control signal using this status information.

•In the maximum mode, there may be more than one microprocessor in the system configuration.

•The components in the system are same as in the minimum mode system.

•The basic function of the bus controller chip IC8288, is to derive control signals like RD and WR (for memory and I/O devices), DEN, DT/R, ALE etc. using the information by the processor on the status lines.

•The bus controller chip has input lines S2, S1, S0 and CLK. These inputs to 8288 are driven by CPU.

•It derives the outputs ALE, DEN, DT/R, MRDC, MWTC, AMWC, IORC, IOWC and AIOWC. The AEN, IOB and CEN pins are especially useful for multiprocessor systems

•AEN and IOB are generally grounded. CEN pin is usually tied to +5V. The significance of the

MCE/PDEN output depends upon the status of the IOB pin.

•If IOB is grounded, it acts as master cascade enable to control cascade 8259A, else it acts as peripheral data enable used in the multiple bus configurations.

•INTA pin used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

•IORC, IOWC are I/O read command and I/O write command signals respectively.

These signals enable an IO interface to read or write the data from or to the address port. •The MRDC, MWTC are memory read command and memory write command signals respectively and may be used as memory read or write signals.

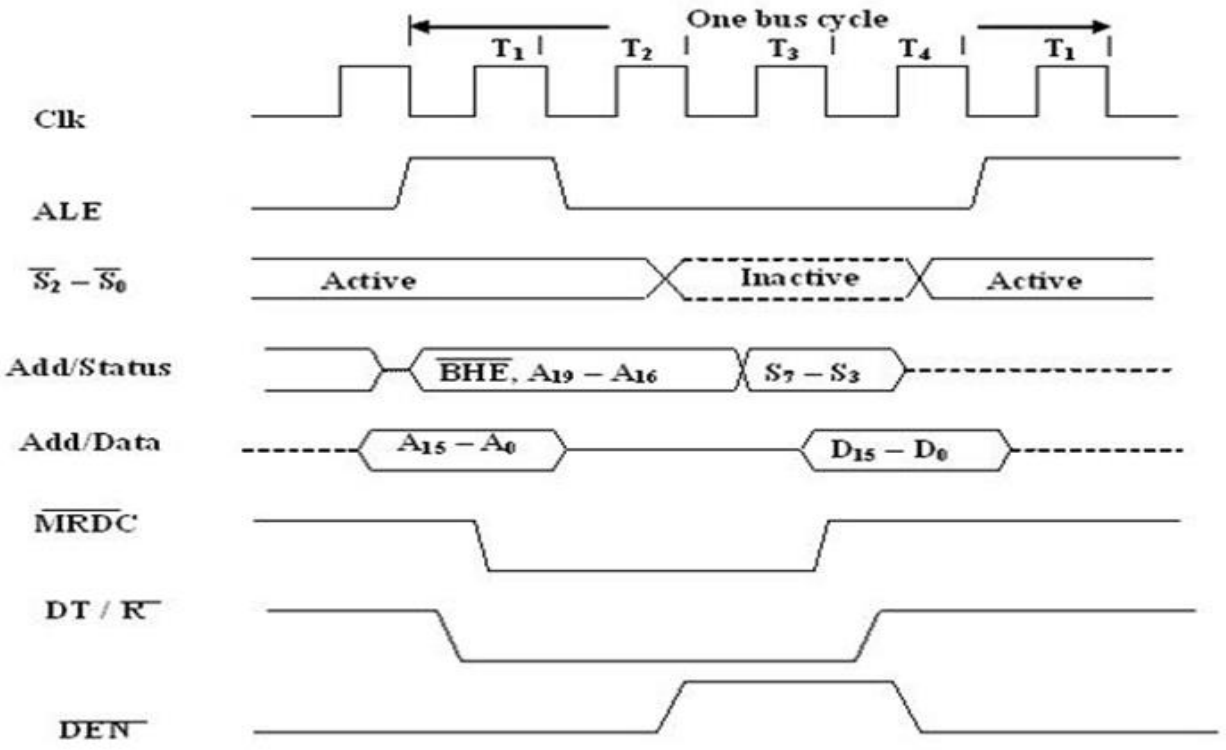
•All these command signals instructs the memory to accept or send data from or to the bus. •For both of these write command signals, the advanced signals namely AIOWC and AMWTC are available.

•Here the only difference between in timing diagram between minimum mode and maximum mode is the status signals used and the available control and advanced command signals. •R0, S1, S2 are set at the beginning of bus cycle.8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT / R pin during T1.

•In T2, 8288 will set DEN=1 thus enabling transceivers, and for an input it will activate MRDC or IORC. These signals are activated until T4. For an output, the AMWC or

AIOWC is activated from T2 to T4 and MWTC or IOWC is activated from T3 to T4. •The status bit S0 to S2 remains active until T3 and become passive during T3 and T4.

•If reader input is not activated before T3, wait state will be inserted between T3 and T4.



Memory Read Timing in Maximum Mode

UNIT -II

UNIT -II

Modular Programming: Instead of writing a large program in a single unit, it is better to write small programs—which are parts of the large program. Such small programs are called program modules or simply modules. Each such module can be separately written, tested and debugged. Once the debugging of the small programs is over, they can be linked together. Such methodology of developing a large program by linking the modules is called modular programming.

Assembler Directives:

Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the segment directive, equ, assume, and end. These mnemonics are not valid 80x86 instructions. They are messages to the assembler, to generate address.

A pseudo-opcode is a message to the assembler, just like an assembler directive, how ever a pseudo-opcode will emit object code bytes. Examples of pseudo-opcodes include byte, word, dword, qword, and byte. These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

ASSUME: The ASSUME directive tell the assembler the name of the logical segment it should use for a specified segment. Ex: ASSUME CS: Code, DS : Data, SS : Stack; or ASSUME CS: Code

Data Directives: The directives DB, DW, DD, DR and DT are used to (a) define different types of variables or (b) to set aside one or more storage locations in memory-depending on the data type:

DB — Define Byte DW — Define Word DD — Define Double word
DQ — Define Quadword DT — Define Ten Bytes

The **DB directive** is used to declare a byte-type variable or to set aside one or more storage locations of type byte in memory (Define Byte)

Example: Temp DB 42H; Temp is a variable allotted 1byte of memory location assigned with data 42H
The **DW directive** is used to declare a variable of type word or to reserve memory locations which can be accessed as type double word (Define word)

Example: N2 DW 427AH; N2 variable is initialized with value 427AH when it is loaded into memory to run.

The **DD directive** is used to declare a variable of type double word or to reserve memory locations which can be accessed as type double word (Define double word)

Example: Big DD 2456756CH; Big variable is initialized with 4 bytes

The **DQ directive** is used to tell the assembler to declare a variable 4 words in length or to reverse 4 words of storage in memory (Define Quadword)

Example: Big DQ 2456756C88464567H; Big variable is initialized with 4 words (8 bytes)

The **DT directive** is used to tell the assembler to declare a variable 10 bytes in length or to reverse 10bytes of storage in memory (Define Ten bytes)

Example: Packed BCD DT 112233445566778899OOH; 10 byte data is initialized to variable packed BCD
DUP: This directive operator is used to initialize several locations and to assign values to these locations. Its format is: Name Data-Type Num DUP (value)

Example: TABLE DB 20 DUP(0) ; Reserve an array of 20 bytes of memory and initialize all
20 bytes with 0. Array is named TABLE

END: The **END** directive is placed after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statement after an end directive.

The **ENDP** directive is used with the name of the procedure to indicate the end of a procedure to the

assembler.

SQUARE NUM PROC

….

….

SQUARE NUM ENDP

The **ENDS** directive is used with the name of the segment to indicate the end of a segment to the assembler.

CODE SEGMENT

…

…

CODE ENDS

EQU: The **EQU** directive is used to give a name to some value or to a symbol. Each time assembler finds the name in the program it will replace the name with the value.

FACTOR EQU 03H; This statement should be written at the start

ADD AL, FACTOR; The assembler converts this instruction as ADD AL, 03H

EVEN: The **EVEN** directive instructs the assembler to increment the location of the counter to the next even address if it is not already in the even address. If the word starts at an odd address, 8086 will take 2 bus cycles to get the 2 byte of the word. *"A series of words can read much more quickly if they are at even address".*

```
DATA HERE SEGMENT    ; Location counter will point to 0009H after assembler reads next statement
SALES DB 9 DUP (?)   ;Declare an array of 9 bytes
EVEN                 ; Increment location counter to 000AH
RECORD DW 100 DUP (?) ; Array of 100 words starting on even address for quicker read
DATA HERE ENDS       ;
```

GLOBAL: This **GLOBAL** directive can be used in place of PUBLIC directive or in place of an EXTRN directive. The GOLBAL directive is used to make the symbol available to other modules.

PUBLIC: The **PUBLIC** directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive.

EXTRN: This **EXTRN** directive is used to tell the assembler that the names or labels following the directive are in some other assembly module.

GROUP: This **GROUP** directive is used to tell the assembler to group the logical segments named after the directive into one logical group segment.

Example: SMALL SYSTEM GROUP CODE, DATA, STACK

ASSUME CS:SMALL SYSTEM, DS: SMALL SYSTEM, SS: SMALL SYSTEM OFFSET—

Is an operator which tells the assembler to determine the offset or the displacement of a named data item (variable) or procedure from start of the segment which contains it. This operator is used to load the offset of a variable into a register so that the variable can be accessed with one of the indexed addressing modes.

MOV AL, OFFSET N1

ORG – This **ORG** directive allows to set the location counter to a desired value at any point in the program. The statement ORG 100H tells the assembler to set the location counter to 0100H.

PROCEDURE: A PROC directive is used to define a label and to delineate a sequence of instructions that are usually interpreted to be a subroutine, that is, called either from within the same physical segment (near) or from another physical segment (far).

Syntax:
name PROC [type]

…..
name ENDP

Labels: A label, a symbolic name for a particular location in an instruction sequence, maybe defined in one of three ways. The first way is the most common. The format is shown below: **label: [instruction]** where "label" is a unique ASM86 identifier and "instruction" is an*8086/8087/8088* instruction. This label will have the following attributes:

1. Segment-the current segment being assembled.
2. Offset-the current value of the location counter.
3. Type-will be NEAR.

An example of this form of label definition is: ALAB: MOV AX, COUNT

Instruction Set of 8086:

There are 117 basic instructions in the instruction set of 8086.The instruction set of 8086 can be divided into the following number of groups, namely:

1. Data copy / Transfer instructions
2. Arithmetic and Logical instructions
3. Branch instructions
4. Loop instructions
5. Machine control instructions
6. Flag Manipulation instructions
7. Shift and Rotate instructions
8. String instructions

Data copy / Transfer instructions: The data movement instructions copy values from one location to another. These instructions include **MOV, XCHG, LDS, LEA, LES, PUSH, PUSHF, PUSHFD, POP,POPF, LAHF, AND SAHF.**

MOV The MOV instruction copies a word or a byte of data from source to a destination. The destination can be a register or a memory location. The source can be a register, or memory location or immediate data. MOV instruction does not affect any flags. The mov instruction takes several different forms:

The MOV instruction cannot:

4. Set the value of the CS and IP registers.
5. Copy value of one segment register toa nother segment register (should copy to general register first). MOV CS, DS (Invalid)
6. Copy immediate value to segment register (should copy to general register first). MOV CS, 2000H (Invalid)

Example:

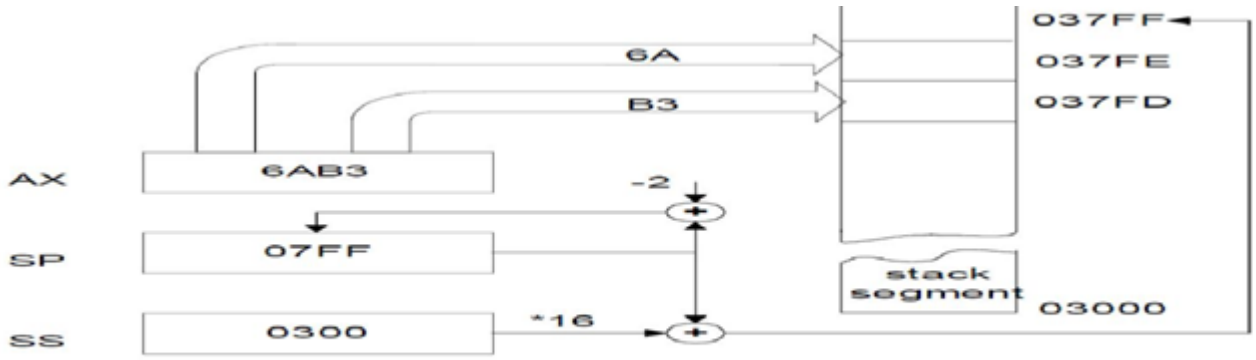
```
ORG 100h
MOV AX, 0B800h ;        set AX = B800h
MOV DS, AX ;            copy value of AX to DS.
MOV CL, 'A' ;       CL = 41h (ASCII code).
```

The XCHG Instruction: Exchange This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them, may be a memory location. However, exchange of data contents of two memory locations is not permitted.

Example: MOV AL, 5 ; AL = 5
MOV BL, 2 ; BL = 2
XCHG AL,BL ; AL = 2, BL = 5

PUSH: Push to stack; this instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

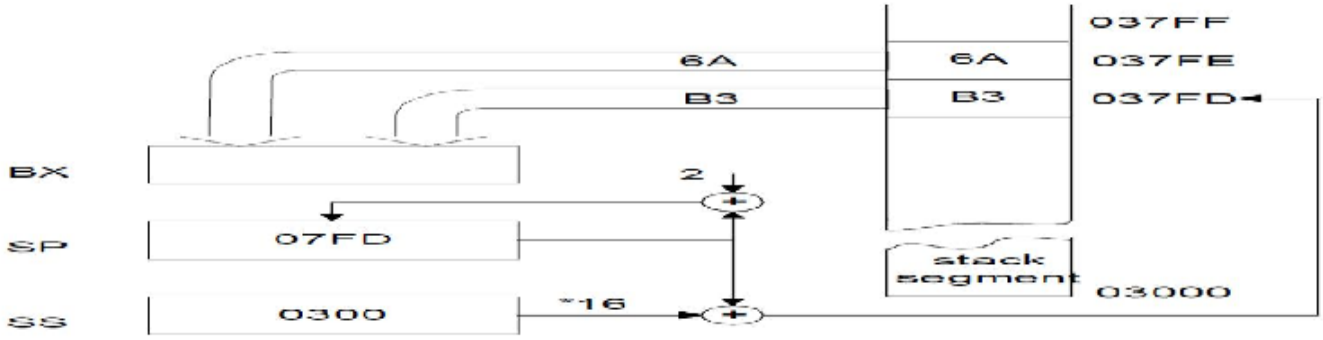
4. PUSH AX
5. PUSH DS
6. PUSH [500OH] ; Content of location 5000H and 5001 H in DS are pushed onto the stack.



The effect of PUSH AX instruction

POP: Pop from Stack this instruction when executed loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

4. POP BX
5. POP DS
6. POP [5000H]



The affect of POP BX instruction

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2for each pop operation.

LAHF: Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except overflow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

LEA: Load Effective Address The load effective address instruction loads the offset of an operand in the specified register. This instruction is similar to MOV, MOV is faster than LEA.

LEA cx, [bx+si] ; CX (BX+SI) mod 64K **If bx=2f00 H; si=10d0H cx 3fd0H**

The LDS AND LES instructions:

• LDS and LES load a 16-bit register with offset address retrieved from a memory location then load either DS or ES with a segment address retrieved from memory.

This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers.

• LDS and LES instructions obtain a new far address from memory.

– offset address appears first, followed by the segment address

• This format is used for storing all 32-bit memory addresses.

• A far address can be stored in memory by the assembler.

LDS BX,DWORD PTR[SI]
BL [SI];
BH [SI+1]
DS [SI+3:SI+2]; in the data segment

LES BX,DWORD PTR[SI]
BL [SI];
BH [SI+1]
ES [SI+3:SI+2]; in the extra segment

I/O Instructions: The 80x86 supports two I/O instructions: in and out15. They take the forms: In

ax, port

in ax, dx out

port, ax out dx,

ax

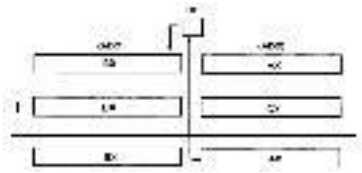
port is a value between 0 and 255.

The in instruction reads the data at the specified I/O port and copies it into the accumulator. The out instruction writes the value in the accumulator to the specified I/O port.

Arithmetic instructions: These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions.

The ADD and ADC instructions: The add instruction adds the contents of the source operand to the destination operand. For example, **add ax, bx** adds bx to ax leaving the sum in the ax register. **Add computes dest :=dest+source while adc computes dest :=dest+source+C where C represents the value in the carry flag**. Therefore, if the carry flag is clear before execution, adc behaves exactly like the add instruction.

Example:



CF=1

AX=98

DX=78 CX=94

BX=9E AX=2C

Both instructions affect the flags identically. They set the flags as follows:

• The overflow flag denotes a signed arithmetic overflow.

• The carry flag denotes an unsigned arithmetic overflow.

• The sign flag denotes a negative result (i.e., the H.O. bit of the result is one).

• The zero flag is set if the result of the addition is zero.

• The auxiliary carry flag contains one if a BCD overflow out of the L.O. nibble occurs.

• The parity flag is set or cleared depending on the parity of the L.O. eight bits of the result. If there is even number of one bits in the result, the ADD instructions will set the parity flag to one (to denote even parity). If there is an odd number of one bits in the result, the ADD instructions clear the parity flag (to denote odd parity).

The INC instruction: The increment instruction adds one to its operand. Except for carry flag, inc sets the flags the same way as Add ax, 1 same as inc ax. The inc operand may be an eight bit, sixteen bit. The inc instruction is more compact and often faster than the comparable add reg, 1 or add mem, 1 instruction.

The AAA and DAA Instructions

The aaa (ASCII adjust after addition) and daa (decimal adjust for addition) instructions support BCD arithmetic. BCD values are decimal integer coded in binary form with one decimal digit(0..9) per nibble. ASCII (numeric) values contain a single decimal digit per byte, the H.O. nibble of the byte should

contain zero (30 ….39).

The aaa and daa instructions modify the result of a binary addition to correct it for ASCII or decimal arithmetic. For example, to add two BCD values, you would add them as though they were binary numbers and then execute the daa instruction afterwards to correct the results.

Note: These two instructions assume that the add operands were proper decimal or ASCII values. If you add binary(non-decimal or non-ASCII) values together and try to adjust them with these instructions, you will not produce correct results.

Aaa (which you generally execute after an add, adc, or xadd instruction) checks the value in al for BCD overflow. It works according to the following basic algorithm:

if ((al and 0Fh) > 9 or (AuxC =1)) then al := al + 6
else

ax := ax +6 endif
ah := ah + 1

AuxC := 1 ;Set auxilliary carry Carry := 1 ; and carry flags. Else

AuxC := 0 ;Clear auxilliary carry Carry := 0 ; and carry flags.

add al=08 +06; al=0E >9 al=0E+06=04
ah=00+01=01
al=04+03=08, now al<9,
so only clear ah=0
endif

al := al and 0Fh

The aaa instruction is mainly useful for adding strings of digits where there is exactly one decimal digit per byte in a string of numbers.

The **daa instruction** functions like aaa except it handles packed BCD values rather than the one digit per byte unpacked values aaa handles. As for aaa, daa's main purpose is to add strings of BCD digits (with two digits per byte). The algorithm for daa is

if ((AL and 0Fh) > 9 or (AuxC = 1)) then
al := al + 6
AuxC := 1 ;Set Auxilliary carry.
Endif
if ((al > 9Fh) or (Carry = 1)) then
al := al + 60h
Carry := 1; ;Set carry flag.
Endif

al=24+77=9B, as B>9 add 6 to al
al=9B+06=A1, as higher nibble A>9, add 60 to al, al=A1+60=101
Note: if higher or lower nibble of AL <9 then no need to add 6 to AL

EXAMPLE:

Assume AL = 0 0 1 1 0 1 0 1, ASCII 5 BL = 0 0 1 1 1 0 0 1, ASCII 9

ADDAL,BL Result: AL= 0 1 1 0 1 1 1 0 = 6EH,which is incorrect
BCD AAA Now AL = 00000100, unpacked BCD 4.

CF = 1 indicates answer is 14 decimal

NOTE: OR AL with 30H to get 34H, the ASCII code for 4. The AAA instruction works only on the AL register. The AAA instruction updates AF and CF, but OF, PF, SF, and ZF are left undefined.

EXAMPLES:

AL = 0101 1001 = 59 BCD ; BL = 0011 0101 = 35

BCD ADD AL, BL AL = 1000 1110 = 8EH

DAA Add 01 10 because 1110 > 9 AL = 1001 0100 = 94
BCD AL = 1000 1000 = 88 BCD BL = 0100 1001 = 49 BCD

ADD AL, BL AL = 1101 0001, AF=1

DAA Add 0110 because AF =1, AL = 11101 0111 =
D7H 1101 > 9 so add 0110 0000

AL = 0011 0111= 37 BCD, CF =1

The DAA instruction updates AF, CF, PF, and ZF. OF is undefined after a DAA instruction.

The SUBTRACTION instructions: SUB, SBB, DEC, AAS, and DAS

The sub instruction computes the value dest :=dest - src. The sbb instruction computes dest :=dest src - C.

The sub, sbb, and dec instructions affect the flags as follows:

• They set the zero flag if the result is zero. This occurs only if the operands are equal for sub and sbb. The dec instruction sets the zero flag only when it decrements the value one.

• These instructions set the sign flag if the result is negative.

• These instructions set the overflow flag if signed overflow/underflow occurs.

• They set the auxiliary carry flag as necessary for BCD/ASCII arithmetic.

• They set the parity flag according to the number of one bits appearing in the result value.

• The sub and sbb instructions set the carry flag if an unsigned overflow occurs. Note that the dec instruction does not affect the carry flag.

The aas instruction, like its aaa counterpart, lets you operate on strings of ASCII numbers with one decimal digit (in the range 0..9) per byte. This instruction uses the following algorithm:

if ((al and 0Fh) > 9 or AuxC = 1)
then al := al - 6

ah := ah - 1

AuxC := 1 ;Set auxilliary
carry Carry := 1 ; and carry
flags. else

AuxC := 0 ;Clear Auxilliary
carry Carry := 0 ; and carry flags.

endif

al := al and 0Fh

The das instruction handles the same operation for BCD values, it uses the following algorithm:

if ((al and 0Fh) > 9 or (AuxC = 1))
then al := al -6

AuxC =
1 endif

if (al > 9Fh or Carry = 1)
then al := al - 60h

Carry := 1 ;Set the Carry
flag. Endif

EXAMPLE:

ASCII 9-ASCII 5 (9-5)

AL = 00111001 = 39H = ASCII 9
BL = 001 10101 = 35H = ASCII 5

SUB AL, BL Result: AL = 00000100 = BCD 04 and CF =
0 AAS Result: AL = 00000100 = BCD 04 and CF = 0

no borrow required
ASCII 5-ASCII 9 (5-9)

Assume AL = 00110101 = 35H ASCII
5 and BL = 0011 1001 = 39H = ASCII
9

SUB AL, BL Result: AL = 11111100 = - 4 in 2s complement and CF
=1 AAS Result: AL = 00000100 = BCD 04 and CF = 1, borrow needed

EXAMPLES:

AL 1000 0110 86 BCD ; BH 0101 0111 57 BCD

SUB AL,BH AL 0010 1111 2FH, CF = 0

DAS Lower nibble of result is 1111, so DAS automatically

subtracts 0000 0110 to give AL = 00101001 29
BCD AL 0100 1001 49 BCD BH 0111 0010 72
BCD SUB AL,BH AL 1101 0111 D7H, CF = 1

DAS Subtracts 0110 0000 (- 60H) because 1101 in upper nibble > 9
AL = 01110111= 77 BCD, CF=1 CF=1 means borrow was needed

The CMP Instruction: The cmp (compare) instruction is identical to the sub instruction with one crucial difference– it does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub, the generic form is **cmpdest, src**

Consider the following cmp instruction: cmp ax, bx

This instruction performs the computation ax-bx and sets the flags depending upon the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if ax = bx. This is the only time ax-bx produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative.

O: The overflow flag is set after a cmp operation if the difference of ax and bx produced an overflow or underflow.

C: The carry flag is set after a cmp operation if subtracting bx from ax requires a borrow. This occurs only when ax is less than bx where ax and bx are both unsigned values.

The Multiplication Instructions: MUL, IMUL, and AAM: This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general-purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX.

The mul instruction, with an **eight bit operand**, multiplies the al register by the operand and **stores the 16 bit result in ax.** So

mul operand	(Unsigned)	MUL BL	i.e. AL * BL; Al=25 * BL=04; AX=00 (AH) 64 (AL)
imul operand	(Signed)	IMUL BL	i.e. AL * BL; AL=09 * BL=-2; AL * 2's comp(BL)

AL=09 * BL (0EH)=7E; 2's comp (7e)=-82

The aam (ASCII Adjust after Multiplication) instruction, adjust an unpacked decimal value after multiplication. This instruction operates directly on the ax register. It assumes that you've multiplied two eight bit values in the range 0..9 together and the result is sitting in ax (actually, the result will be sitting in al since 9*9 is 81,the largest possible value; ah must contain zero). This instruction divides ax by 10 and leaves the quotient in ah and the remainder in al: mul bl; al=9, bl=9 al*bl=9*9=51H; AX=00(AH) 51(AL); AAM ; first hexadecimal value is converted to decimal value i.e. 51 to 81; al=81D; second convert packed BCD to unpacked BCD, divide AL content by 10 i.e. 81/10 then AL=01, AH =08; AX = 0801

EXAMPLE:

AL 00000101 unpacked BCD 5
BH 00001001 unpacked BCD 9
MUL BH AL x BH; result in AX
AX = 00000000 00101101 = 002DH
AAM AX = 00000100 00000101 = 0405H, which is unpacked BCD for 45.

If ASCII codes for the result are desired, use next instruction OR AX, 3030H Put 3 in upper nibble of each byte.
AX = 0011 0100 0011 0101 = 3435H, which is ASCII code for 45

The Division Instructions: DIV, IDIV, and AAD

The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16division or a 16/8 division. These instructions take the form:

Div reg For unsigned division
Div mem
Idiv reg For signed division
Idiv mem

The div instruction computes an unsigned division. If the operand is an eight bit operand ,div divides the ax register by the operand leaving the quotient in al and the remainder(modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx ax by the operand leaving the quotient in ax and the remainder in .

Note: If an overflow occurs (or you attempt a division by zero) then the80x86 executes an INT 0 (interrupt zero).

The aad (ASCII Adjust before Division) instruction is another unpacked decimal operation. It splits apart unpacked binary coded decimal values before an ASCII division operation. The aad instruction is useful for other operations. The algorithm that describes this instruction is

al := ah*10 + al AX=0905H; BL=06; AAD; AX=AH*10+AL=09*10+05=95D;
convert decimal to hexadecimal; 95D=5FH; al=5f;

DIV BL; AL/BL=5F/06; AX=05(AH)0F(AL) ah := 0

EXAMPLE:

AX = 0607H unpacked BCD for 67 decimal CH = 09H,
now adjust to binary

AAD Result: AX = 0043 = 43H = 67 decimal
DIV CH Divide AX by unpacked BCD in CH
Quotient: AL = 07 unpacked BCD Remainder:

AH = 04 unpacked BCD Flags undefined after DIV

NOTE: If an attempt is made to divide by 0, the 8086 will do a type 0 interrupt.

CBW-Convert Signed Byte to Signed Word: This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. The CBW operation must be done before a signed byte in AL can be divided by another signed byte with the IDIV instruction. CBW affects no flags.

EXAMPLE:

AX = 00000000 10011011 155 decimal

CBW Convert signed byte in AL to signed word in
AX Result: AX = 11111111 10011011 155 decimal

CWD-Convert Signed Word to Signed Double word: CWD copies the sign bit of a word in AX to all the bits of the DX register. In other words it extends the sign of AX into all of DX. The CWD operation must be done before a signed word in AX can be divided by another signed word with the IDIV instruction. CWD affects no flags.

EXAMPLE:

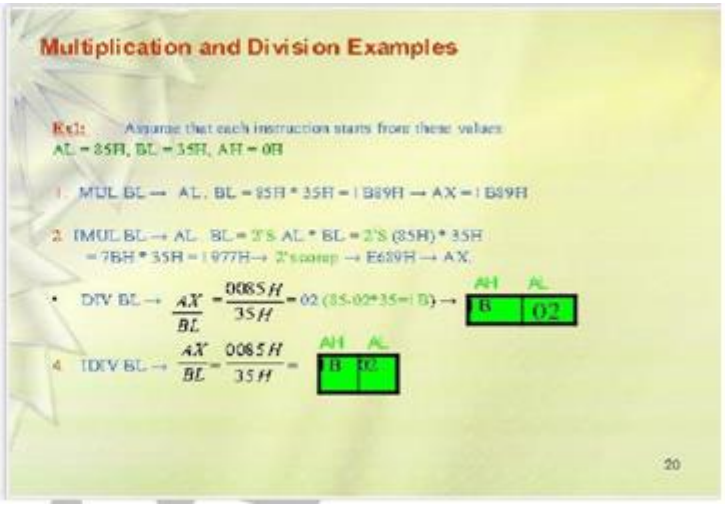
DX = 00000000 00000000

AX = 11110000 11000111 3897 decimal

CWD Convert signed word in AX to signed double word in DX:AX

Result DX = 11111111 11111111

AX = 11110000 11000111 3897 decimal



Logical, Shift, Rotate and Bit Instructions: The 80x86 family provides five logical instructions, four rotate instructions, and three shift instructions. The logical instructions are and, or, xor, test, and not; the rotates are ror, rol, rcr, and rcl; the shift instructions are shl/sal, shr, and sar.

The Logical Instructions: AND, OR, XOR, and NOT: The 80x86 logical instructions operate on a bit-by-bit basis. Except not, these instructions affect the flags as follows:

- They clear the carry flag.
- They clear the overflow flag.
- They set the zero flag if the result is zero, they clear it otherwise.
- They copy the H.O. bit of the result into the sign flag.
- They set the parity flag according to the parity (number of one bits) in the result.
- They scramble the auxiliary carry flag.

The not instruction does not affect any flags.

The **AND** instruction sets the zero flag if the two operands do not have any ones in corresponding bit positions. **AND AX, BX**

The **OR** instruction will only set the zero flag if both operands contain zero. **OR AX, BX**

The **XOR** instruction will set the zero flag only if both operands are equal. Notice that the xor operation will produce a zero result if and only if the two operands are equal. Many programmers commonly use this fact to clear a sixteen bit register to zero since an instruction of the form

xor reg16, reg16; XOR AX, AX is shorter than the comparable mov reg,0 instruction.

You can use the and instruction to set selected bits to zero in the destination operand. This is known as *masking out* data; Likewise, you can use the or instruction to force certain bits to one in the destination operand;

The Shift Instructions: SHL/SAL, SHR, SAR: The 80x86 supports three different shift instructions (shl and sal are the same instruction):shl (shift left), sal (shift arithmetic left), shr (shift right), and sar (shift arithmetic right).

SHL/SAL: These instructions move each bit in the destination operand one bit position to the left the number of times specified by the count operand. Zeros fill vacated positions at the L.O. bit; the H.O. bit shifts into the carry flag.

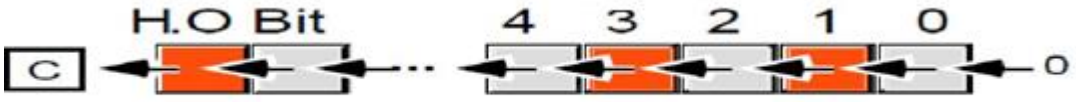
The shl/sal instruction sets the condition code bits as follows:

- If the shift count is zero, the shl instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- The overflow flag will contain one if the two H.O. bits were differen tprior to a single bit shift. The overflow flag is undefined if the shift count is not one.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.
- The parity flag will contain one if there are an even number of one bits inthe L.O. byte of the result.
- The A flag is always undefined after the shl/sal instruction.

The shift left instruction is especially useful for packing data. For example, suppose you have two nibbles in al and ah that you want to combine. You could use the following code to do this:

```
shl ah, 4 ;
or al, ah ;Merge in H.O. four bits.
```

Of course, al must contain a value in the range 0..F for this code to work properly (the shift left operation automatically clears the L.O. four bits of ah before the or instruction).



SHL OPERATION

H.O. four bits of al are not zero before this operation, you can easily clear them with an and instruction:

```
shl ah, 4 ;Move L.O. bits to H.O. position.
and al, 0Fh ;Clear H.O. four bits.
or al, ah ;Merge the bits.
```

Since shifting an integer value to the left one position is equivalent to multiplying that value by two, you can also use the **shift left instruction for multiplication by powers of two**:

```
shl ax, 1 ;Equivalent to AX*2
shl ax, 2 ;Equivalent to AX*4
shl ax, 3 ;Equivalent to AX*8
```

SAR: Thesar instruction shifts all the bits in the destination operand to the right one bit, replicating the H.O. bit.

The sar instruction's main purpose is to perform a signed division by some power of two. Each shift to the right divides the value by two. Multiple right shifts divide the previous shifted result by two, so multiple shifts produce the following results:

SAR OPERATION

```
sar ax, 1 ;Signed division by 2 sar
ax, 2 ;Signed division by 4 sar ax,
3 ;Signed division by 8 sar ax, 4
;Signed division by 16 sar ax, 5
;Signed division by 32 sar ax, 6
;Signed division by 64 sar ax, 7
;Signed division by 128 sar ax, 8
;Signed division by 256
```

There is a very important difference between the sar and idiv instructions. The idiv instruction always truncates towards zero while sar truncates results toward the smaller result. For positive results, an arithmetic shift right by one position produces the same result as an integer division by two. However, if the quotient is negative, idiv truncates towards zero while sar truncates towards negative infinity.

SHR: The shr instruction shifts all the bits in the destination operand to the right one bit shifting a zero into the H.O. bit

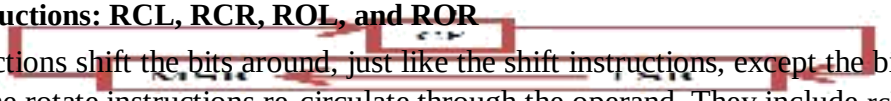
SHR OPERATION

The shift right instruction is especially useful for unpacking data. shifting an unsigned integer value to the right one position is equivalent to dividing that value by two, you can also use the shift right instruction for division by powers of two:

```
shr ax, 1 ;Equivalent to AX/2
shr ax, 2 ;Equivalent to AX/4
```

shr ax, 3 ;Equivalent to AX/8
shr ax, 4 ;Equivalent to AX/16

The Rotate Instructions: RCL, RCR, ROL, and ROR



The rotate instructions shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions re-circulate through the operand. They include rcl(rotate through carry left), rcr(rotate through carry right), rol(rotate left), and ror(rotate right). These instructions all take the forms :rcldest, count roldest, count rcr dest, count ror dest, count

RCL: The rcl(rotate through carry left), as its name implies, rotates bits to the left, through the carry flag, and back into bit zero on the right. The rcl instruction sets the flag bits as follows:

- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- If the shift count is one, rcl sets the overflow flag if the sign changes as a result of the rotate. If the count is not one, the overflow flag is undefined.
- The rcl instruction does not modify the zero, sign, parity, or auxiliary carry flags.

RCL OPERATION

RCR: The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit. This instruction sets the flags in a manner analogous to rcl:

- The carry flag contains the last bit shifted out of the L.O. bit of the operand.
- The rcr instruction does not affect the zero, sign, parity, or auxiliary carry flags.

RCR OPERATION

ROL: The rol instruction is similar to the rcl instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit ,rather than the carry, into bit zero. Rol also copies the output of the H.O. bit into the carry flag . The rol instruction sets the flags identically to rcl. Other than the source of the value shifted into bit zero, this instruction behaves exactly like the rcl instruction.

Like shl, the rol instruction is often useful for packing and unpacking data.

ROL OPERATION

ROR: The ror instruction relates to the rcr instruction in much the same way that the rol instruction relates to rcl. That is, it is almost the same operation other than the source of the input bit to the operand. Rather than shifting the previous carry flag into the H.O. bit of the destination operation, ror shifts bit zero into the H.O. bit.

ROR OPERATION

String Instructions: A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes or words on 8086.**All members of the 80x 86 families support five different string instructions: MOVS, CMPS, SCAS, LODS, AND STOS.**

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the movs instruction moves a sequence of bytes from one memory location to another. The cmps instruction compares two blocks of memory. The scas instruction scansa block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the movs instruction to copy a string, you need a source address, a destination address, and a count (the number of string elements to move).The operands for the string instructions include:

- **theSI (source index) register,**
- **the DI (destination index) register,**
- **the CX (count) register,**
- **theAX register, and**
- **the direction flag in the FLAGS register.**

The REP/REPE/REPZ and REPNZ/REPNE Prefixes: The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

Field:

Label repeat mnemonic operand ; comment

For MOVS:

Rep movs {operands}

For CMPS:

Repe cmps {operands} Repz cmps {operands} Repne cmps {operands} Repnz cmps {operands}

For SCAS:

Repe scas {operands} Repz scas {operands} Repne scas {operands} repnz scas {operands}

For STOS:

repstos {operands}

When specifying the repeat prefix before a string instruction, the string instruction repeats cx times. Without the repeat prefix, the instruction operates only on a single byte,word, or double word.

If the direction flag is clear, the CPU increments si and di after operating upon eachstring element. If the direction flag is set, then the 80x86 decrements si and di after processing eachstring element. The direction flag may be set or cleared using the cld (clear direction flag) and std (setdirection flag) instructions.

The MOVS Instruction: The movsb (move string, bytes) instruction fetches the byte at address ds:si, stores it at address es:di, and then increments or decrements the si and di registers by one. If the rep prefix is present, the CPU checks cx to see if it contains zero. If not, then it moves the byte from ds:si to es:di and decrements the cx register. This process repeats until cx becomes zero. The syntax is :

{REP} MOVSB **{REP} MOVSW**

The CMPS Instruction: The cmps instruction compares two strings. The CPU compares the string referenced by es:di to the string pointed at by ds:si. Cx contains the length of the two strings (whenusing the rep prefix). The syntax is: **{REPE} CMPSB {REPE} CMPSW**

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match or length of the string cx=0. The**repe** prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and cx is greater than zero.

The SCAS Instruction: The scas instruction, by itself, compares the value in the accumulator (al or ax) against the value pointed at by es:di and then increments (or decrements) di by one or two. The CPU sets the flags according to the result of the comparison. When using the repne prefix (repeat while not equal), scas scans the string searching for the first string element which is equal to the value in the accumulator. The scas instruction takes the following forms:**{REPNE} SCASB {REPNE} SCASW The STOS**

Instruction: The stos instruction stores the value in the accumulator at the location specified byes:di. After storing the value, the CPU increments or decrements di depending upon the state of the direction flag. Its primary use is to initialize arrays and strings to a constant value. **{REP} STOSB**
{REP} STOSW

The LODS Instruction: The lods instruction copies the byte or word pointed at by ds:si into the al or ax register, after which it increments or decrements the si register by one or two.**{REP} LODSB**
{REP} LODSW

Flag Manipulation and Processor Control Instructions: These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions.

The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

CLC - Clear carry flag **CMC - Complement carry flag** **STC - Set carry flag**
CLD - Clear direction flag **STD - Set direction flag** **CLI - Clear interrupt flag**
STI - Set interrupt flag

These instructions modify the carry(CF), direction(DF) and interrupt(IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and auto-increment or auto-decrement modes.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

WAIT - Wait for Test input pin to go low **HLT - Halt the processor** **NOP - No Operation ESC - Escape to external device like NDP (numeric co-processor)** **LOCK - Bus lock instruction prefix.**

After executing the **HLT instruction**, the processor enters the halt state. The two ways to pull it out of the halt state are to reset the processor or to interrupt it.

When **NOP instruction** is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP by one. It then continues with further execution after 4 clock cycles.

ESC instruction when executed, frees the bus for an external master like a coprocessor or

peripheral devices.

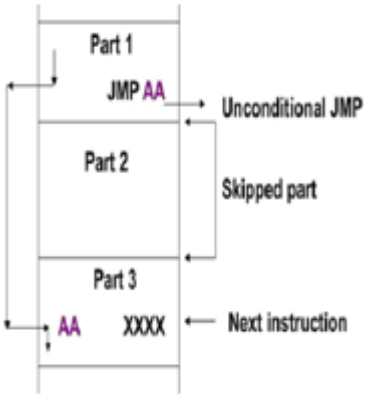
The **LOCK prefix** may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems.

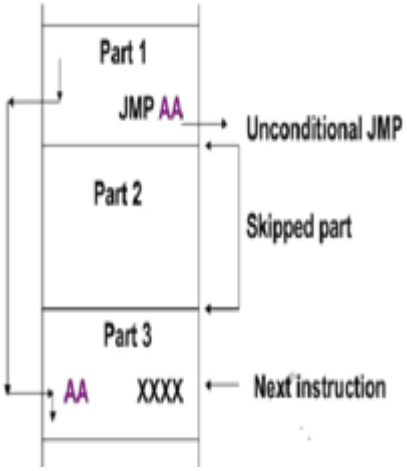
The **WAIT instruction** when executed holds the operation of processor with the current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further execution.

Program Flow Control Instructions: The control transfer instructions are used to transfer the control from one memory location to another memory location. In 8086 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions.

Unconditional Jumps: The jmp (jump) instruction unconditionally transfers control to another point in the program. Intra segment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment.

JMP Address





Unconditional jump **Conditional jump**

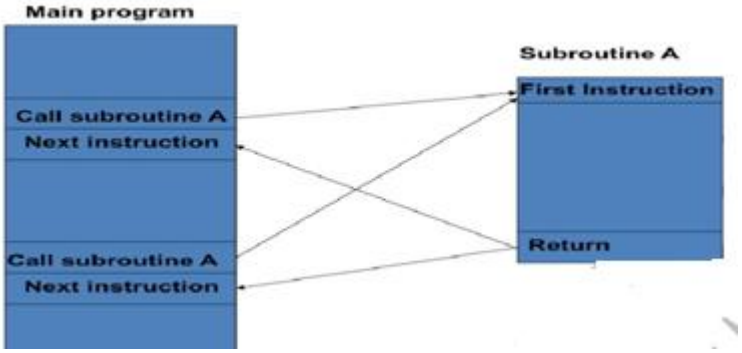
Conditional Jump: The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the if…..then statement. The conditional jumps test one or more bits in the status register to see if they match some particular pattern. If the pattern matches, control transfers to the target location. If the condition fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some instructions, for example, test the conditions of the sign, carry, overflow and zero flags.

Definition	Description	Condition
	Jump Based on Unsigned Data	
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JA / JNBE	Jump above or jump not below/ equal	C=0 & Z=0
JAE / JNB	Jump above/ equal or jump not below	C=0
JB / JNAE	Jump below or jump not above/ equal	C=1
JBE / JNA	Jump below/ equal or jump not above	C=1 or Z=1
	Jump Based on Signed Data	
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JG / JNLE	Jump greater or jump not less/ equal	N=0 & Z=0
JGE / JNL	Jump greater/ equal or jump not less	N=0
JL / JNGE	Jump less or jump not greater/ equal	N=1
JLE / JNG	Jump less/ equal or jump not greater	N=1 or Z=1
	Arithmetic Jump	
JS	Jump sign set	N=1
JNS	Jump no sign set	N=0
JC	Jump carry set	C=1
JNC	Jump no carry set	C=0
JO	Jump overflow set	O=1
JNO	Jump not overflow set	O=0
JP / JPE	Jump parity even	P=1
JNP / JPO	Jump parity odd	P=0

Loop Instruction:

- These instructions are used to repeat a set of instructions several times.
- Format: LOOP Short-Label
- Operation: (CX) (CX)-1
- Jump is initialized to location defined by short label if CX≠0. Otherwise, execute next sequential instruction.
- Instruction LOOP works with respect to contents of CX. CX must be preloaded with a count that represents the number of times the loop is to be repeat.
- Whenever the loop is executed, contents at CX are first decremented then checked to determine if they are equal to zero.
- If CX=0, loop is complete and the instruction following loop is executed.
- If CX ≠ 0, content return to the instruction at the label specified in the loop instruction.
- **LOOP AGAIN is <u>almost same</u> as: DEC CX, JNZ AGAIN**

SUBROUTINE & SUBROUTINE HANDILING INSTRUCTIONS: CALL, RET



- A subroutine is a special segment of program that can be called for execution from any point in a

program.

- An assembly language subroutine is also referred to as a "procedure".
- Whenever we need the subroutine, a single instruction is inserted in to the main body of the program to call subroutine.
- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL saves a return address on the
  - stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.
- To branch a subroutine the value in the IP or CS and IP must be modified.
- After execution, we want to return the control to the instruction that immediately follows the one called the subroutine i.e., the original value of IP or CS and IP must be preserved.
- Execution of the instruction causes the contents of IP to be saved on the stack. (this time (SP) (SP) -2)
- A new 16-bit (near-proc, mem16, reg16 i.e., Intra Segment) value which is specified by the instructions operand is loaded into IP.
- Examples: CALL 1234H

  CALL BX

  CALL [BX]

Return Instruction: RET instruction removes an address from the stack so the program returns to the instruction following the CALL

- Every subroutine must end by executing an instruction that returns control to the main program. This is the return (RET) instruction.
- By execution the value of IP or IP and CS that were saved in the stack to be returned back to their corresponding registers. (this time (SP) (SP)+2)

MACROS: The macro directive allows the programmer to write a named block of source statements, then use that name in the source file to represent the group of statements. During the assembly phase, the assembler automatically replaces each occurrence of the macro name with the statements in the macro definition.

Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if used repeatable. Procedures or subroutines take up less space, but the increased overhead of saving and restoring addresses and parameters can make them slower. In summary, the advantages and disadvantages of macros are,

Advantages

- Repeated small groups of instructions replaced by one macro
- Errors in macros are fixed only once, in the definition
- Duplication of effort is reduced
- In effect, new higher level instructions can be created
- Programming is made easier, less error prone
- Generally quicker in execution than subroutines

Disadvantages

In large programs, produce greater code size than procedures

When to use Macros

- To replace small groups of instructions not worthy of subroutines
- To create a higher instruction set for specific applications
- To create compatibility with other computers
- To replace code portions which are repeated often throughout the program

Eg:-Write a program for the addition of two 8 – bit numbers.

```
ASSUME CS: CODE, DS:DATA
DATA SEGMENT
N1  DB  00H
N2  DB  00H
RES    DB  00H
DATA ENDS
CODE
SEGMENT
            START: MOV AX,
                   DATA MOV
                   DS, AX MOV
                   AL, N1 MOV
                   BL, N2 ADD
                   AL, BL MOV
                   RES, AL INT 3H
                  CODE ENDS
                  END START
```

Arithmetic operation – Multi byte Addition and Subtraction, Multiplication and Division – Signed and unsigned Arithmetic operation, ASCII – arithmetic operation.

Program No: 2(a) **8-Bit Addition**

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
NUM1  DB  00H
NUM2  DB  00H
RES       DB  00H
DATA ENDS
CODE SEGMENT
            START: MOV AX, DATA
                        MOV DS, AX
                        MOV AL, NUM1
                        MOV BL, NUM2
                        ADD AL, BL
                        MOV RES, AL
                        INT 3H
             CODE ENDS
             END START
```

Input: 0000: 08h
0001: 02h
Output: 0002: 0Ah

```
RES       DB    00H
DATA ENDS
CODE SEGMENT
            START: MOV AX, DATA
                        MOV DS, AX
                        MOV AL, NUM1
                        MOV BL, NUM2
                        MUL BL
                        MOV RES, AL
                        INT 3H
             CODE ENDS
             END START
```

Program No: 2(d)

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
NUM1    DB    00H
NUM2    DB    00H
RES       DB    00H
DATA ENDS
CODE SEGMENT
            START: MOV AX, DATA
                        MOV DS, AX
                        MOV AL, NUM1
                        MOV BL, NUM2
                        DIV BL
```

```
                MOV RES, AL          Input:  0000:08h
                INT 3H                       0001:02h
            CODE ENDS                Output: 0002:04h
            END START
```

Program No: 2(e) **16-Bit Addition**

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
NUM1  DW  00H
NUM2  DW  00H
RES     DW  00H
DATA ENDS
CODE SEGMENT
          START: MOV AX, DATA
                MOV DS, AX
                MOV AX, NUM1
                MOV BX, NUM2
                ADD AX, BX
                MOV RES, AX          Input:  0000:08h
                INT 3H                       0002:02h
            CODE ENDS                Output: 0004:04h
            END START
```

Program No: 2(f) **16-Bit Subtraction**

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
NUM1  DW  00H
NUM2  DW  00H
RES     DW  00H
DATA ENDS
CODE SEGMENT
          START: MOV AX, DATA
                MOV DS, AX
                MOV AX, NUM1
                MOV BX, NUM2
                SUB AX, BX
                MOV RES, AX          Input:  0000:08h
                INT 3H                       0002:02h
            CODE ENDS                Output: 0004:04h
            END START
```

Program No: 2(g) **16-Bit Multiplication**

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
NUM1   DW   00H
NUM2   DW   00H
RES1   DW   00H
RES2   DW   00H
DATA ENDS
CODE SEGMENT
        START: MOV AX, DATA
               MOV DS, AX
               MOV AX, NUM1
               MOV BX, NUM2
               MUL BX
               MOV RES1, AX          Input:
               MOV RES2, DX
               INT 3H
        CODE ENDS                    Output:
        END START
```

Program No: 2(h) **16-Bit Division**

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
NUM1   DW   00H
NUM2   DW   00H
RES1   DW   00H
RES2   DW   00H
DATA ENDS
CODE SEGMENT
        START: MOV AX, DATA
               MOV DS, AX
               MOV AX, NUM1
               MOV BX, NUM2
               DIV BX
               MOV RES1, AX          Input:
               MOV RES2, DX
               INT 3H
        CODE ENDS                    Output: 0004:04h
        END START
```

Program No: 2(i) **Multi byte Addition**

```
Assume cs: code, ds: data
 Data segment
```

```
    Ip1 dd 12233449h
    Ip2 dd 56677889h
    Res dd 00000000h
 Data ends
Code segment
   Start: mov ax, data
          mov ds,ax
          mov si,offset ip1
          mov di,offset ip2
          mov bx,offset res
          mov cx,03
          sub ax,ax
          mov al,[si]
          mov dl,[di]
          add al,dl
          mov [bx],al
  back:  inc si
          inc di
          inc bx
          mov al,[si]
          mov dl,[di]
          adc al,dl
          mov [bx],al
          loop back
          nop
          int 03h
 code ends
        end start
```

INPUT:

IP1: 11223344H
IP2: 55667788H

OUTPUT:

RES: 6688AACCH

Program No: 2(j) **Multi byte Subtraction**

```
Assume cs: code, ds: data
 data segment
   ip1 dd 55667788h
   ip2 dd 11223344h
   res dd 00000000h
```

```
data ends
code segment
start: mov ax,data
```

```
        mov ds,ax
        mov si,offset ip1
        mov di,offset ip2
        mov bx,offset res
        mov cx,03
        sub ax,ax
        mov al,[si]
        mov dl,[di]
        sub al,dl
        mov [bx],al
  back:inc  si
        inc  di
        inc bx
        mov al,[si]
        mov dl,[di]
        sbb al,dl
        mov [bx],al
        loop back
        nop
        int 03h
   code ends
         end start
```

****

INPUT:

IP1: 55667788H
IP2: 11223344H

OUTPUT:

RES: 44444444H

ASCII ARTHMETICAL OPERATIONS

Program No: 2(k) **ASCII Addition**

```
Assume cs: code, ds: data
data segment
       asc1 db 09H
       asc2 db 06H
        res dw 00h
 data ends
code segment
```

```
start: mov ax,
data mov ds,ax
xor ax,ax
mov al,asc1
mov bl,asc2
add al,bl aaa

      or ax,3030h
      mov  res, ax
      int 3H
  code ends
  end start
```

INPUT: ASC1: 09H
ASC2: 06H

OUTPUT:
RES: 3135H

Program No: 2(l) **ASCII Subtraction**

```
Assume cs: code, ds: data
data segment
      asc1 db 09H
      asc2 db 06H
      res dw 00h
data ends
code segment
start: mov ax,data
mov ds,ax
 xor ax,ax
mov al,asc1
 mov bl,asc2
sub al,bl
 aas

      or ax,3030h
      mov res,ax
       int 3
```

```
  code ends
  end start
```

INPUT: ASC1: 09H ASC2: 06H

OUTPUT:

RES: 3033H

Program No: 2(m) ASCII Multiplication

```
Assume cs: code, ds: data
data segment
      asc1 db 06H
      asc2 db 02H
      res dw 00h
data ends
code segment
  start: mov  ax, data
  mov ds, ax
       xor ax,ax
       mov al,asc1
        mov bl,asc2
        mul bl
         aam
         or ax,3030h
          mov res,ax
         int 3H
code ends
        end start
```

INPUT: ASC1: 06H ASC2: 02H

OUTPUT:

RES: 3132H

program No: 2 (n) **ASCII Divisions**

```
Assume cs: code, ds: data
Data segment

       asc1 db 09H
        asc2 db 03H
        res dw 00h
Data ends
code segment
  start: mov ax,data
  mov ds,ax
  xor ax,ax
  mov al,asc1
  mov bl,asc2
  aad
         div bl
         or ax,3030h
          mov res,ax
         int 3H
   code ends
   end start
```

INPUT: ASC1: 09H
ASC2: 03H

OUTPUT:
RES: 3033H

Logic Operations - Shift and Rotate –Converting packed BCD to unpacked BCD, BCD to ASCII conversion.

Program No: 3(a) **Logical AND operation**

```
Assume cs: code, ds: data
```

```
 Data segment
        op1 dw 00h
         op2 dw 00h
        res dw 00h
Data ends
Code segment
 start: mov ax,data
  mov ds,ax
  mov ax,op1
mov bx,op2
and ax,bx
 mov res,ax
 int 3h
code ends
 end start
```

INPUT: OP1:01h
OP2:01h

OUTPUT:
RES:01h

Program No: 3(b) **Logical OR operation**

```
assume cs: code, ds: data
 data segment
        op1 dw 00h
        op2 dw 00h
         res dw 00h
data ends
code segment
 start: mov ax,data
 mov ds,ax
  mov ax,op1
  mov bx,op2
 or ax, bx
 mov res,ax
  int 3h
code ends
 end start
```

INPUT: OP1:01h
OP2:00h

OUTPUT:

RES:01h

Program No: 3(c) **Logical NOT operation**

```
Assume cs: code, ds: data
 data segment
       op1 dw 00h
       res dw 00h
data ends
code segment
 start: mov ax, data
  mov  ds, ax
 mov ax,op1

        not ax
        mov res, ax
        int 3h
code ends
end start
```

INPUT: OP1:04h

OUTPUT:

RES: FBh

Program No: 3(d) **Logical XOR operation**

```
Assume cs: code, ds: data
data segment
       op1 dw 00h
       op2 dw 00h
       res dw 00h
data ends
code segment
 start: mov ax ,data
 mov  ds, ax
 mov ax,op1

        mov bx,op2
         xor ax,bx
         mov res,ax
         int 3h
code ends
```

```
end start
```

INPUT: OP1:01h OP2:00h

OUTPUT:

RES:01h

Program No: 3(e) **RCL**

```
Assume cs: code, ds: data
 data segment
       op1 db 0h
       cnt db 00h
       res db 00h
data ends
code segment
 start: mov ax, data
 mov  ds ,ax
 sub ax,
 ax
 mov al,op1
 mov  cl,
 cnt
 rcl al ,cl
 mov res,al
  int 3h
code ends
 end start
```

INPUT: OP1:25h cnt:03h

OUTPUT:

RES:28h

Program No: 3(f) **RCR**

```
Assume cs: code, ds: data
 data segment
       op1 db 0h
       cnt db 00h
       res db 00h
data ends
```

```
code segment
 start: mov  ax, data
  mov  ds, ax
 sub ax, ax
  mov al,op1
 mov  cl, cnt
  rcr al, cl
  mov res, al
  int 3h
code ends end start
```

INPUT: OP1:25h cnt:03h
OUTPUT: RES:44h

Program No: 3(g) **ROL**

```
Assume cs: code, ds: data
data segment
        op1 db 0h
        cnt db 00h
        res db 00h
data ends code segment
 start: mov ax, data
  mov  ds, ax
 sub ax, ax
  mov al,op1

        mov  cl, cnt
         rol al, cl
        mov res, al
        int 3h
code ends
 end start
```

INPUT: OP1:25h
cnt:03h

OUTPUT:
RES:29h

Program No: 3(h) **ROR**

```
Assume cs: code, ds: data
data segment
       op1 db 0h
        cnt db 00h
        res db 00h
data ends
code segment
 start: mov  ax ,data
 mov  ds, ax
 sub ax, ax
  mov al,op1
 mov  cl, cnt
  ror al, cl
 mov  res, al
 int 3h

code ends
 end start
```

INPUT: OP1:25h
cnt:03h
OUTPUT: RES:A4h

Program No: 3(i) **SAR**

```
Assume cs: code,ds: data
 data segment
       op1 db 0h
       cnt db 00h
       res db 00h
data ends
code segment
 start: mov  ax, data
 mov  ds, ax
 sub ax, ax
  mov al,op1
```

```
 mov cl, cnt
  sar  al, cl
 mov  res, al
 int 3h

code ends
 end start
```

INPUT: OP1:25h
cnt:03h
OUTPUT:
RES:04h

Program No: 3(j) **SHL**

```
Assume cs: code, ds: data
 data segment
       op1 db 00h
       cnt db 00h
       res db 00h
data ends
code segment

 start: mov  ax, data
 mov  ds, ax
 sub ax, ax
  mov al,op1
 mov cl, cnt
 shl al, cl
 mov res, al
  int 3h
code ends
end start
```

INPUT: OP1:25h
cnt:03h

OUTPUT:

RES:28h

Program No: 3(k) **SHR**

```
Assume cs: code, ds: data
data segment
       op1 db 0h
        cnt db 00h
       res db 00h
data ends
code segment
start: mov  ax, data
 mov ds, ax
sub ax, ax
 mov al,op1
mov cl, cnt
 shr al, cl
 mov res ,al
 int 3h
code ends
 end start
```

INPUT: OP1:25h
cnt:03h

OUTPUT:
RES:04h

Program No: 3(l) **PACKED TO ASCII**

```
Assume cs: code, ds: data
data segment
    ip1 db 56h
    res dw 00h
data ends
code segment
 start: mov ax, data
```

```
    mov ds, ax
    xor ax, ax
    mov al,ip1
    mov dl, al
    and al,0f0h
    mov cl,4
    ror al, cl
    mov bh, al
    and dl,0fh
    mov bl, dl
    add bx,3030h
    mov res, bx
    int 03h


code ends
      end start
```

INPUT: OP1:56h

OUTPUT:
RES:3536h

Program No: 3(m) **PAC`KED TO UNPACKED**

```
assume cs: code, ds: data
data segment
   ip1 db 56h
   res dw 00h
data ends
code segment
 start: mov ax, data
  mov ds, ax
 xor ax, ax
 mov al,ip1
  mov dl, al
 and al,0f0h
  mov cl,4
```

```
      ror al, cl
      mov bh, al
      and dl,0fh
      mov bl, dl
      mov res,bl
int 03h
code ends
       end start
```

INPUT: OP1:56h

OUTPUT:

RES:0506h

4) By Using String Operation and Instruction Prefix: Move Block, Reverse String, Sorting, Inserting, Deleting, Length of the String, String Comparison.

Program No: 4(a) **ASCENDING ORDER**

```
 Assume cs: code, ds: data
  data segment
      list db 76h,34h,23h,22h,02h,49h,15h,00h,00h
      cnt db 08h
data ends
 code segment
   start: mov ax, data
    mov ds, ax
    mov cl, cnt
      up: mov dl,cnt
           mov si, offset list
            xor ax, ax
    again: mov al,[si]
                  cmp al,[si+1]
                  jl next
                   xchg al,[si+1]
                   mov [si],al
          next: inc si
          dec dl
           jnz again
           dec cl
```

```
    jnz up
     int 3h

      code ends
       end start
```

Input: ds: 76 34 23 22 02 49 15 00 00
Output: ds: 0000:00 00 02 15 22 23 34 49 76 08.

Program No: 4(b) **BLOCK TRANSFER**

```
Assume cs: code, ds: data, es: extra data segment
    ipstr db 'empty vessels make much noise',24h data ends
extra segment
   opstr db 100h dup(00) extra ends
code segment start: mov ax,data
        mov ds,ax mov ax,extra mov es,ax
        mov si,offset ipstr mov di,offset opstr cld

        mov cx,29
         rep movsb
        nop
        int 03h
        code ends
     end start
```

OUTPUT: es: 0000 : empty vessels make much noise.

Program No: 4(c) **STRING LENGTH**

```
Assume cs: code, ds: data data segment
      string1 db 'empty vessels make more noise$' strlen equ($-string1)
     org 0100h
        res     db 00h
       cort    db 'strlength found correct$'
      incort db 'strlength found incorrect$' data ends
code segment
start: mov ax, data
         mov ds, ax
         sub cl, cl
          mov bl, strlen
         mov si, offset string1
         back: lodsb
         inc cl
         cmp al,'$'
         jnz back
         mov res, cl
         int 3h
code ends
end start
```

INPUT: 'empty vessels make more noise'

OUTPUT: depending on string

Program No: 4(d) **STRING COMPARISION**

```
Assume cs: code, ds: data, es: data data segment
    string1 db 'empty1$' strlen equ($-string1)
    op1 db 'strings are unequal$' op2 db 'strings are equal$'
    org 100h
    res db 00h data ends extra segment
      string2 db 'empty$' extra ends
code segment start: mov ax,data
        mov ds,ax mov ax,extra mov es,ax
        mov si,offset string1 movdi,offset string2 cld
        mov cx, strlen
         repe cmpsb
         jz next
        mov ah,09h
        mov dx,'u'

        int 21h jmp exitp
  next: mov ah, 09h
  mov  dx, 'e'
   int 21h
 exitp: nop
        mov  res, dl
        mov ah,4ch
        int 21h
  code ends
  end start
```

Program No: 4(e)

STRING REVERSE

```
Assume cs: code, ds: data, es: data
 data segment
     string1 db 'Empty$' strlen equ($-string1)
      string2 db 5h dup(0)
data ends
code segment
   start: mov ax, data
   mov ds, ax mov es, ax
          mov  bx, offset string1
          mov si, bx
          mov di, offset string2
          add di,5
          cld
          mov cx, strlen
           sl: mov al,[si]
          mov es:[di],al
          inc si
          dec di
          loop sl
          rep
          mov sb,nop
           int 03h
           code ends
          end start
```

INPUT: string1: empty

OUTPUT: string2: ytpme

DOS / BIOS Programming: Reading Keyboard (buffered with and without echo) - Display Characters & Strings.

a) Read a character(s) from keyboard and echo using DOS function calls

```
Assume cs: code, ds: data data
segment
   msg db 'enter characters from keyboard (# to end):','$' data
```

```
ends
code segment
start: mov ax, data
        mov ds, ax mov
        ah,09h
        mov dx,offset msg int
        21h
 next:mov ah,08h int 21h
        mov ah,02h mov
        dl,al int 21h cmp
        al,'#' jne next mov
        ah,4ch mov al,00h

           int 21h code
    ends
        end start
```

OUTPUT: enter the character from keyboard

b) Reads a character(s) from keyboard without echo using DOS function calls.

```
assume cs:code, ds:data data
 segment
      msg db 'enter characters from keyboard (# to end):','$' data
ends
code segment
start: mov ax, data
           mov ds, ax mov
           ah,09h
           mov dx,offset msg int
           21h
   next: mov ah,08h int
           21h
    cmp al,'#'
    jne next
   mov ah,4ch
   mov al,00h
    int 21h
code ends
          end start
```

OUTPUT: enter characters from keyboard: #

UNIT -III

UNIT-III

MSP 430 ARCHITECTURE:

Introduction:

The types of devices such as microprocessor, microcontroller, processor, digital signal processor (DSP), amongst others, in a certain manner, are related to the same device – the ASIC (Application Specific Integrated Circuit). Each processing device executes instructions, following a determined program applied to the inputs and shares architectural characteristics developed from the first microprocessors created in 1971. In the three decades after the development of the first microprocessor, huge developments and innovations have been made in this engineering field.

The programmable SoC (system-on-chip) concept started in 1972 with the 4-bit TMS1000 microcomputer developed by Texas Instruments (TI), and in those days it was ideal for applications such as calculators and ovens. This term was changed to Microcontroller Unit (MCU), which was more descriptive of a typical application. Nowadays, MCUs are at the heart of many physical systems, with higher levels of integration and processing power at lower power consumption.

The following list presents several qualities that define a microcontroller:

1. Cost: Usually, the microcontrollers are high-volume, low cost devices;
2. Clock frequency: Compared with other devices (microprocessors and DSPs), microcontrollers use a low clock frequency.
3. Microcontrollers today can run up to 100 MHz/ 100 Million Instructions Per Second (MIPS)
4. Power consumption: orders of magnitude lower than their DSP and MPU cousins;
5. Bits: 4 bits (older devices) to 32 bits devices;
6. Memory: Limited available memory, usually less than 1 MByte;
7. Input/Output (I/O): Low to high (8-150) pin-out count.

Main characteristics of a MSP430 microcontroller

Although there are variants in devices in the family, a MSP430 microcontroller can be characterized by:

Low power consumption:

- A for RAM data retention;
- 0.8 A for real time clock mode operation;
- 250 A/MIPS at active operation.

Low operation voltage (from 1.8 V to 3.6 V).

- < 1 s clock start-up.
- < 50 nA port leakage.
- Zero-power Brown-Out Reset (BOR).

On-chip analogue devices:

- 10/12/16-bit Analogue-to-Digital Converter (ADC);
- 12-bit dual Digital-to-Analogue Converter (DAC);
- Comparator-gated timers;
- Operational Amplifiers (OP Amps);
- Supply Voltage Supervisor (SVS).

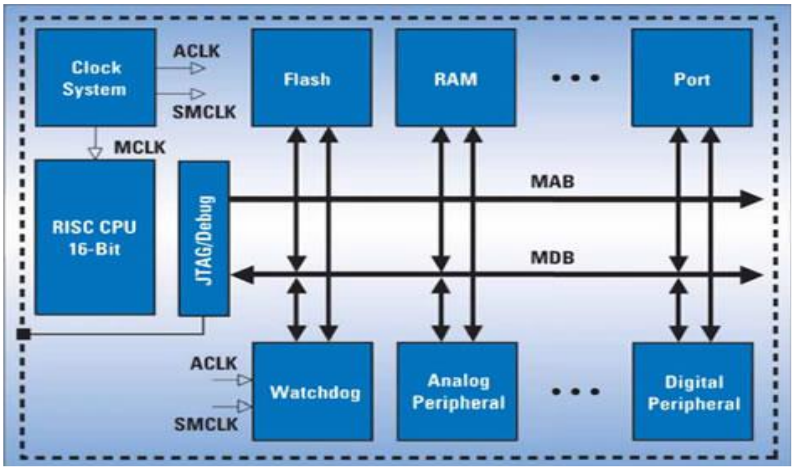
16 bit RISC CPU:

- Instructions processing on either bits, bytes or words;
- Compact core design reduces power consumption and cost;
- Compiler efficient;
- 27 core instructions;
- 7 addressing modes;
- Extensive vectored-interrupt capability.

Flexibility:

- Up to 256 kB In-System Programmable (ISP) Flash;
- Up to 100 pin options;
- USART, I2C, Timers;
- LCD driver;Embedded emulation.

MSP430 architecture:



Address space**:**

All memory, including RAM, Flash/ROM, information memory, special function registers (SFRs), and peripheral registers are mapped into a single, contiguous address space.

The MSP430 is available with either Flash or ROM memory types.

The memory type is identified by the letter immediately following "MSP430" in the part numbers.

Flash devices: Identified by the letter "F" in the part numbers, having the advantage that the code space can be erased and reprogrammed.ROM devices: Identified by the letter "C" in the part numbers. They have the advantage of being very inexpensive because they are shipped pre programmed, which is the best solution for high-volume

. Memory Map.

Memory Address		Description	Access
End:	0FFFFh	**Interrupt Vector Table**	Word/Byte
Start:	0FFE0h		
End:	0FFDFh	**Flash/ROM**	Word/Byte
Start *:	0F800h		
	01100h		
End *:	010FFh	**Information Memory (Flash devices only)**	Word/Byte
	0107Fh		
Start:	01000h		
End:	0FFFh	**Boot Memory (Flash devices only)**	Word/Byte
Start:	0C00h		
End *:	09FFh	**RAM**	Word/Byte
	027Fh		
Start:	0200h		
End:	01FFh	**16-bit Peripheral modules**	Word
Start:	0100h		
End:	00FFh	**8-bit Peripheral modules**	Byte
Start:	0010h		
End:	000Fh	**Special Function Registers**	Byte
Start:	0000h		

* Depending on device family.

For all devices, each memory location is formed by 1 data byte. The CPU is capable of addressing data values either as bytes (8 bits) or words (16 bits). Words are always addressed at an even address, which contain the least significant byte, followed by the next odd address, which contains the most significant byte. For 8-bit operations, the data can be accessed from either odd or even addresses, but for 16-bit operations, the data values can only beaccessed from even addresses.

Flash/ROM:

The start address of Flash/ROM depends on the amount of Flash/ROM present on the device. The start address varies between 01100h (60k devices) to 0F800h (2k devices) and always runs to the end of the address space at location 0FFFFh. Flash can be used for both code and data. Word or byte tables can also be stored and read by the program from Flash/ROM. All code, tables, and hard-coded constants reside in this memory space.

Information memory (*Flash devices only*):

The MSP430 flash devices contain an address space for information memory. It is like an onboard EEPROM, where variables needed for the next power up can be stored during power down. It can also be used as code memory. Flash memory may be written one byte or word at a time, but must be erased in segments. The information memory is divided into two 128-byte segments. The first of these segments is located at addresses 01000h through to 0107Fh (Segment B), and the second is at address 01080h through to 010FFh (Segment A). This is the case in 4xx devices. It is 256 bytes (4 segments of 64 bytes each) in 2xx devices.

Boot memory (*Flash devices only*):

The MSP430 flash devices contain an address space for boot memory, located between addresses 0C00h through to 0FFFh. The "bootstrap loader" is located in this memory space, which is an external interface that can be used to program the flash memory in addition to the JTAG. This memory region is not accessible by other applications, so it cannot be overwritten accidentally. The bootstrap loader performs some of the same functions as the JTAG interface (excepting the security fuse programming), using the TI data structure protocol for UART communication at a fixed data rate of 9600 baud.

RAM:

RAM always starts at address 0200h. The end address of RAM depends on the amount of RAM present on the device. RAM is used for both code and data.

Peripheral Modules:

Peripheral modules consist of all on-chip peripheral registers that are mapped into the address space. These modules can be accessed with byte or word instructions, depending if the peripheral module is 8-bit or 16-bit respectively. The 16-bit peripheral modules are located in the address space from addresses 0100 through to 01FFh and the 8-bit peripheral modules are mapped into memory from addresses 0010h through to 00FFh.0

Special Function Registers (SFRs):

Some peripheral functions are mapped into memory with special dedicated functions. The Special Function Registers (SFRs) are located at memory addresses from 0000h to 000Fh, and are the specific registers for:

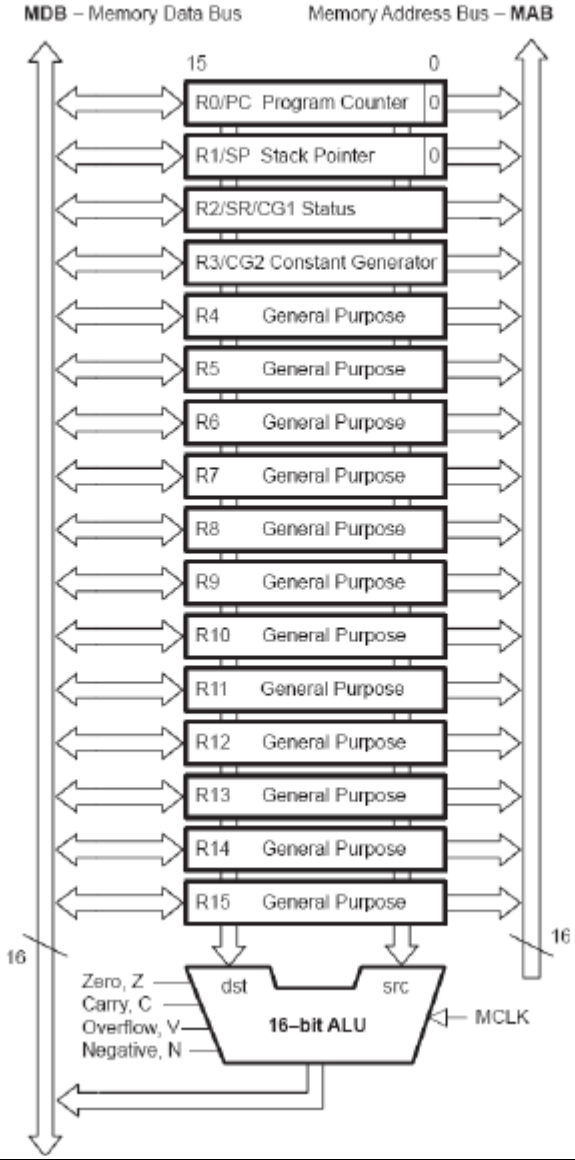
- Interrupt enables (locations 0000h and 0001h);
- Interrupt flags (locations 0002h and 0003h);
- Enable flags (locations 0004h and 0005h);

SFRs must be accessed using byte instructions only. See the device specific data sheets for the applicable SFR bits.

Central Processing Unit (MSP430 CPU):

The RISC type architecture of the CPU is based on a short instruction set (27 instructions), interconnected by a 3-stage instruction pipeline for instruction decoding. The CPU has a 16-bit ALU, four dedicated registers and twelve working registers, which makes the MSP430 a high performance microcontroller suitable for low power applications. The addition of twelve working general purpose registers saves CPU cycles by allowing the storage of frequently used values and variables instead of using RAM. The orthogonal instruction set allows the use of any addressing mode for any instruction, which makes programming clear and consistent, with few exceptions, increasing the compiler efficiency for high-level languages such as C.

. MSP430 CPU block diagram.



Arithmetic Logic Unit (ALU)

The MSP430 CPU includes an arithmetic logic unit (ALU) that handles addition, subtraction, comparison and logical (AND, OR, XOR) operations. ALU operations can affect the overflow, zero, negative, and carry flags in the status register.

MSP430 CPU registers

The CPU incorporates sixteen 16-bit registers:

- Four registers (R0, R1, R2 and R3) have dedicated functions;
- There are 12 working registers (R4 to R15) for general use.

R0: Program Counter (PC)

The 16-bit Program Counter (PC/R0) points to the next instruction to be read from memory and executed by the CPU. The Program counter is implemented by the number of bytes used by the instruction (2, 4, or 6 bytes, always even). It is important to remember that the PC is aligned at even addresses, because the instructions are 16 bits, even though the individual memory addresses contain 8-bit values.

R1: Stack Pointer (SP)

The Stack Pointer (SP/R1) is located in R1. 1st: stack can be used by user to store data for later use (instructions: store by PUSH, retrieve by POP); 2nd: stack can be used by user or by compiler for subroutine parameters (PUSH, POP in calling routine; addressed via offset calculation on stack pointer (SP) in called subroutine); 3rd: used by subroutine calls to store the program counter value for return at subroutine's end (RET); 4th: used by interrupt - system stores the actual PC value first, then the actual status register content (on top of stack) on return from interrupt (RETI) the system get the same status as just before the interrupt happened (as long as none has changed the value on TOS) and the same program counter value from stack.

R2: Status Register (SR)

The Status Register (SR/R2) stores the state and control bits. The system flags are changed automatically by the CPU depending on the result of an operation in a register. The reserved bits of the SR are used to support the constants generator. See the device-specific data sheets for more details.

R2/R3: Constant Generator Registers (CG1/CG2)

Depending of the source-register addressing modes (As) value, six commonly used constants can be generated without a code word or code memory access to retrieve them. This is a very powerful feature, which allows the implementation of emulated instructions, for example, instead of implementing a core instruction for an increment, the constant generator is used.

Table 4-2. Values of the constant generator registers.

Register	As	Constant	Remarks
R2	00	-	Register mode
R2	01	(0)	Absolute mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	-1, word processing

Main features of the MSP430X CPU architecture:

The MSP430X CPU extends the addressing capabilities of the MSP430 family beyond 64 kB to 1 MB. To achieve this, there are some changes to the addressing modes and two new types of instructions. One type of new instructions allows access to the entire address space, and the other is designed for address calculations.

The MSP430X CPU address bus is 20 bits, but the data bus is still 16 bits. The CPU supports 8-bit, 16-bit and 20-bit memory accesses. Despite these changes, the MSP430X CPU remains compatible with the MSP430 CPU, having a similar number of registers

Addressing modes:

The MSP430 supports seven addressing modes for the source operand and four addressing modes for the destination operand .The following sections describe each of the addressing modes, with a brief description, an example and the number of CPU clock cycles required for an instruction, depending on the instruction format and the addressing modes used.

Register Mode:

Register mode operations work directly on the processor registers, R4 through R15, or on special function registers, such as the program counter or status register. They are very efficient in terms of both instruction speed and code space. *Description:* Register contents are operands. *Source mode bits:* As = 00 (source register defined in the opcode). *Destination mode bit:* Ad=0 (destination register defined in the opcode).*Syntax:* Rn. *Length:* One or two words.
Comment: Valid for source and destination.
Example 1: Move (copy) the contents of source (register R4) to destination (register R5). Register R4 is not affected. Before operation: R4=A002h R5=F50Ah PC = PCpos
Operation: MOV R4, R5
After operation: R4=A002h R5=A002h PC = PCpos + 2

The first operand is in register mode and depending on the second operand mode, the cycles required to complete an instruction will differ. *Table 4-7* shows the cycles required to complete an instruction, depending on the second operand mode.

Table 4-7. Cycles required to perform the instruction in the first operand, in register mode.

Operands	2nd operand mode	Operator	Cycles	Length (words)
2	Register	Any	1*	1
2	Indexed, Symbolic or Absolute	Any	4	2
1	N/A	RRA, RRC, SWPB or SXT	1	1
1	N/A	PUSH	3	1
1	N/A	CALL	4	1

*Register mode instructions where the destination is the Program Counter (PC) require 2 cycles instead of 1.

Indexed mode:

The Indexed mode commands are formatted as X(Rn), where X is a constant and Rn is one of the CPU registers. The absolute memory location X+Rn is addressed. Indexed mode addressing is useful for applications such as lookup tables.

Description: (Rn + X) points to the operand. X is stored in the next word.

Source mode bits: As = 01 (memory location is defined by the word immediately following the opcode). *Destination mode bit:* Ad=1 (memory location is defined by the word immediately following the opcode).

Syntax: X(Rn).

Length: Two or three words.

Comment: Valid for source and destination.

Example 2: Move (copy) the contents at source address (F000h +R5) to destination (register R4).Before operation: R4=A002h R5=050Ah Loc:0xF50A=0123h

Operation: MOV F000h(R5), R4

After operation: R4=0123h R5=050Ah Loc:0xF50A=0123h

Table 4-8. Cycles required to perform an instruction contained in the first operand, using indexed mode.

Operands	2nd operand mode	Operator	Cycles	Length (words)
2	Register	Any	3	2
2	Indexed, Symbolic or Absolute	Any	6	3
1	N/A	RRA, RRC, SWPB or SXT	4	2
1	N/A	CALL or PUSH	5	2

Symbolic mode:

Symbolic mode allows the assignment of labels to fixed memory locations, so that those locations can be addressed. This is useful for the development of embedded programs. *Description:* (PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC)

is used. *Source mode bits:* As = 01 (memory location is defined by the word immediately following the opcode). *Destination mode bit:* Ad=1 (memory location is defined by the word immediately following the opcode).
Syntax: ADDR.
Length: Two or three words.
Comment: Valid for source and destination.
Example 3: Move the content of source address XPT (x pointer) to the destination address YPT (y pointer).
Before operation: XPT=A002h Location YPT=050Ah
Operation: MOV XPT, YPT
After operation: XPT= A002h Location YPT=A002h

Table 4-9. Cycles required to perform an instruction contained in the first operand, in symbolic mode.

Operands	2nd operand mode	Operator	Cycles	Length (words)
2	Register	Any	3	2
2	Indexed, Symbolic or Absolute	Any	6	3
1	N/A	RRA, RRC, SWPB or SXT	4	2
1	N/A	CALL or PUSH	5	2

Absolute mode:

Similar to Symbolic mode, with the difference that the label is preceded by "&".
Description: The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used. *Source mode bits:* As = 01 (memory location is defined by the word immediately following the opcode). *Destination mode bit:* Ad=1 (memory location is defined by the word immediately following the opcode).
Syntax: &ADDR.
Length: Two or three words.
Comment: Valid for source and destination.
Example 4: Move the content of source address XPT to the destination address YPT.
Before operation: Location XPT=A002h Location YPT=050Ah
Operation: MOV &XPT, &YPT
After operation: Location XPT= A002h Location YPT=A002h

Table 4-10. Cycles required to perform an instruction contained in the first operand in absolute mode.

Operands	2nd operand mode	Operator	Cycles	Length (words)
2	Register	Any	3	2
2	Indexed, Symbolic or Absolute	Any	6	3
1	N/A	RRA, RRC, SWPB or SXT	4	2
1	N/A	CALL or PUSH	5	2

Indirect register mode:

The data word addressed is located in the memory location pointed to by Rn. Indirect mode is not valid for destination operands, but can be emulated with the indexed mode format 0(Rn). *Description:* Rn is used as a pointer to the operand.

Source mode bits: As = 10.

Syntax: @Rn.

Length: One or two words.

Comment: Valid only for source operand. The substitute for destination operand is 0(Rn).

Example 5: Move the contents of the source address (contents of R4) to the destination (register R5). Register R4 is not modified.

Before operation: R4=A002h R5=050Ah Loc:0xA002=0123h

Operation: MOV @(R4), R5

After operation: R4= A002h R5=0123h Loc:0xA002=0123h

Table 4-11. Cycles required to perform an instruction contained in the first operand, in indirect register mode.

Operands	2nd operand mode	Operator	Cycles	Length (words)
2	Register	Any	2*	1
2	Indexed, Symbolic or Absolute	Any	5	2
1	N/A	RRA, RRC, SWPB or SXT	3	1
1	N/A	CALL or PUSH	4	1

*Indirect register mode instructions where destination is Program Counter (PC) require 3 cycles.

ndirect auto increment mode:

Similar to indirect register mode, but with indirect auto increment mode, the operand is incremented as part of the instruction. The format for operands is @Rn+. This is useful for working on blocks of data.

Description: Rn is used as a pointer to the operand. Rn is

incremented afterwards by 1 for byte instructions and by 2 for word instructions.

Source mode bits: As = 11.

Syntax: @Rn+.

Length: One or two words.

Comment: Valid only for source operand. The substitute for destination operand is 0(Rn) plus second instruction INCD Rn.

Example 6: Move the contents of the source address (contents of R4) to the destination (register R5), then increment the value in register R4 to point to the next word.

Before operation: R4=A002h R5=050Ah Loc:0xA002=0123h

Operation: MOV @R4+, R5

After operation: R4= A004h R5=0123h Loc:0xA002=0123h

Table 4-12. Cycles required to perform an instruction contained in the first operand, in indirect auto increment mode.

Operands	2nd operand mode	Operator	Cycles	Length (words)
2	Register	Any	2*	1
2	Indexed, Symbolic or Absolute	Any	5	2
1	N/A	RRA, RRC, SWPB or SXT	3	1
1	N/A	PUSH	4	1
1	N/A	CALL	5	1

*Indirect autoincrement mode instructions where destination is Program Counter (PC) require 3 cycles.

Immediate mode:

Immediate mode is used to assign constant values to registers or memory locations.

Description: The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

Source mode bits: As = 11.

Syntax: #N.

Length: Two or three words. It is one word less if a constant in CG1 or CG2 can be used.

Comment: Valid only for source operand.

Example 7: Move the immediate constant E2h to the destination (register R5).

Before operation: R4=A002h R5=050Ah

Operation: MOV #E2h, R5

After operation: R4= A002h R5=00E2h

Table 4-13. Cycles required to perform an instruction contained in the first operand, in immediate mode.

Operands	2nd operand mode	Operator	Cycles	Length (words)
2	Register	Any	2*	2
2	Indexed, Symbolic or Absolute	Any	5	3
1	N/A	RRA, RRC, SWPB or SXT	N/A	N/A
1	N/A	PUSH	4	2
1	N/A	CALL	5	2

*Immediate mode instructions where destination is Program Counter (PC) require 3 cycles.

Instruction set:

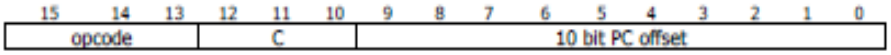
The MSP430 instruction set consists of 27 core instructions. Additionally, it supports 24 emulated instructions. The core instructions have unique op-codes decoded by the CPU, while the emulated ones need assemblers and compilers to generate their mnemonics.

There are three core-instruction formats:

- Double operand;
- Single operand;
- Program flow control - Jump.

Byte, word and address instructions are accessed using the .B, .W or .A extensions. If the extension is omitted, the instruction is interpreted as a word instruction.

The jump instruction is formatted as follows:



Bit		Description
15-13	opcode	
12-10	C	
9-0	PC offset	PCnew = PCold + 2 + PCoffset × 2

Table 4-18 shows the program flow control (jump) instructions that are not emulated.

Table 4-18. Program flow control (jump) instructions.

Mnemonic	Description
Program flow control instructions	
JEQ/JZ label	Jump to label if zero flag is set
JNE/JNZ label	Jump to label if zero flag is reset
JC label	Jump to label if carry flag is set
JNC label	Jump to label if carry flag is reset
JN label	Jump to label if negative flag is set
JGE label	Jump to label if greater than or equal
JL label	Jump to label if less than
JMP label	Jump to label unconditionally

MSP430 Family:

Introduction:

The MSP430 is a 16-bit microcontroller that has a number of special features not commonly available with other microcontrollers:

- Complete system on-a-chip — includes LCD control, ADC, I/O ports, ROM, RAM, basic timer, watchdog timer, UART, etc.
- Extremely low power consumption — only 4.2 nW per instruction, typical High speed — 300 ns per instruction @ 3.3 MHz clock, in register and register addressing mode
- RISC structure — 27 core instructions
- Orthogonal architecture (any instruction with any addressing mode)
- Seven addressing modes for the source operand
- Four addressing modes for the destination operand
- Constant generator for the most often used constants (–1, 0, 1, 2, 4, 8)
- Only one external crystal required — a frequency locked loop (FLL) oscillator
- derives all internal clocks
- Full real-time capability — stable, nominal system clock frequency is available after only six clocks when the MSP430 is restored from low-power mode (LPM) 3; — no waiting for the main crystal to begin oscillation and stabilize.

MSP430 Family:

The MSP430 family currently consists of three subfamilies:

1. MSP430C31x
2. MSP430C32x
3. MSP430C33x

MSP430 Sub-Families Hardware Features

Hardware Item	MSP430C31x	MSP430C32x	MSP430C33x
14-bit ADC	No	Yes	No
16-bit timer_A	No	No	Yes
Basic timer	Yes	Yes	Yes
FLL oscillator	Yes	Yes	Yes
HW/SW UART	Yes	Yes	Yes
HW-multiplier	No	No	Yes
I/O ports with interrupt	8	8	24
I/O ports without interrupt	0	0	16
LCD segment lines	23	21	30
Package	56 SSOP	64 QFP	100 QFP
Universal timer/port module	Yes	Yes	Yes
USART (SCI or SPI)	No	No	Yes
Watchdog timer	Yes	Yes	Yes

MSP430C31x:

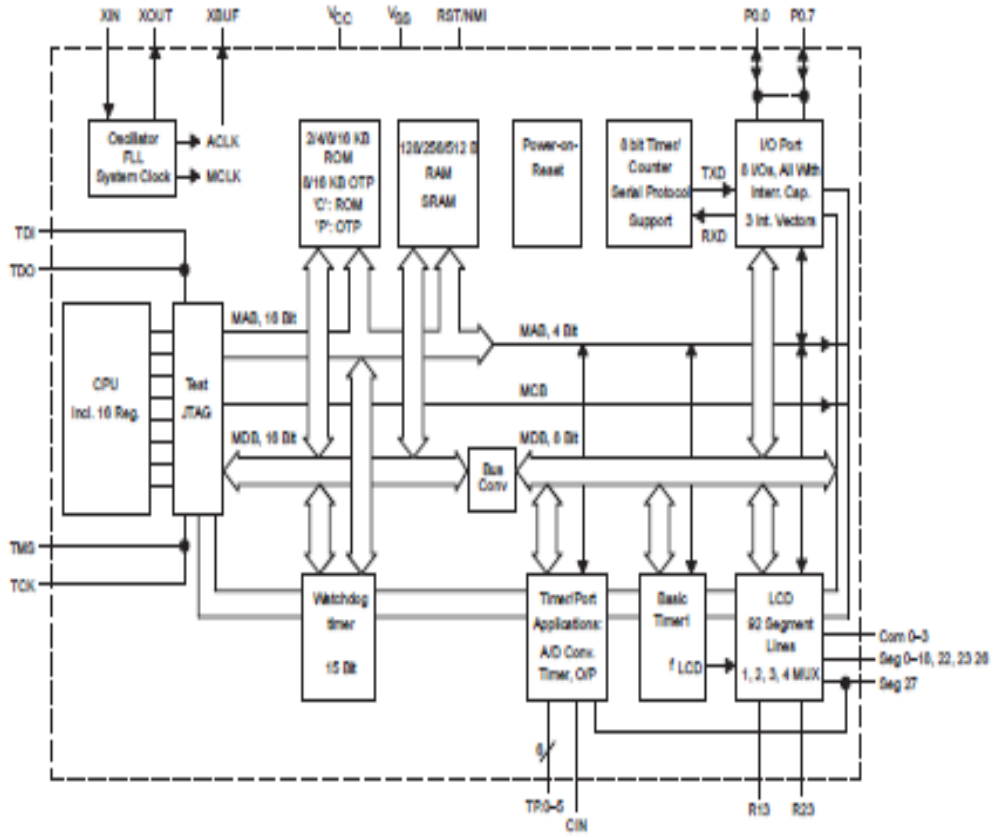


Figure 1–1. MSP430C31x Block Diagram

MSP430C32x:

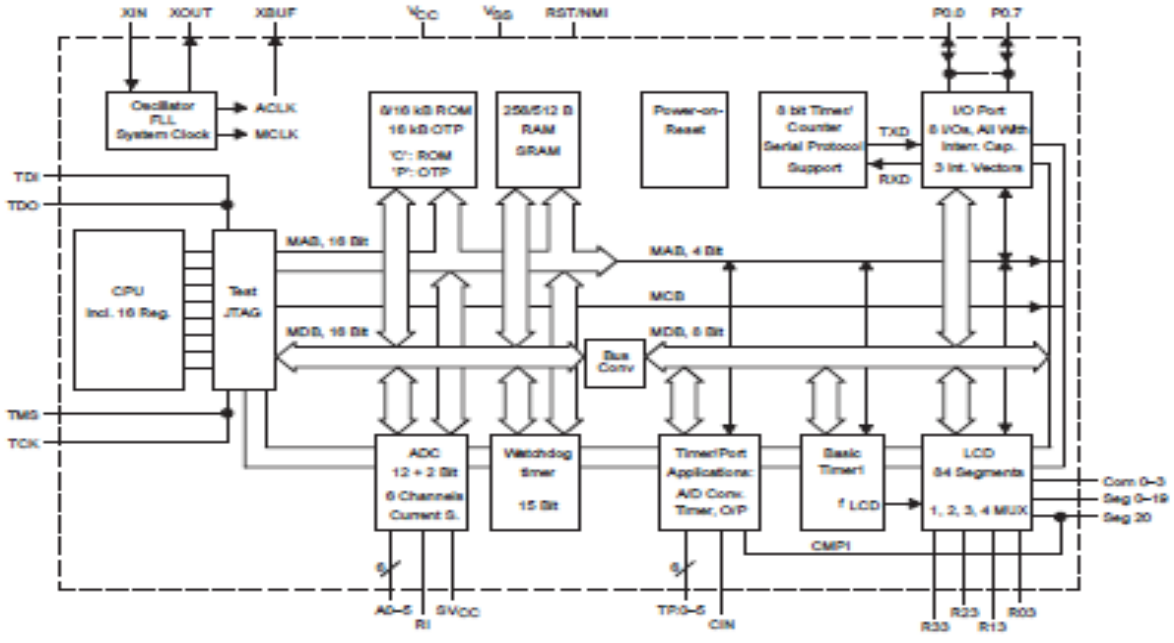


Figure 1–2. MSP430C32x Block Diagram

Advantages of the MSP430 Concept:

The MSP430 concept differs considerably from other microcontrollers and offers some significant advantages over more traditional designs.

RISC Architecture Without RISC Disadvantages

Typical RISC architectures show their highest performance in calculation- intensive applications in which several registers are loaded with input data, all calculations are made within the registers, and the results are stored back into RAM. Memory accesses (using addressing modes) are necessary only for the LOAD instructions at the beginning and the STORE instructions at the end of the calculations. The MSP430 can be programmed for such operation, for example, performing a pure calculation task in the floating point without any I/O accesses.

Pure RISC architectures have some disadvantages when running real-time applications that require frequent I/O accesses, however. Time is lost whenever an operand is fetched and loaded from RAM, modified, and then stored back into RAM.

The MSP430 architecture was designed to include the best of both worlds, taking advantage of RISC features for fast and efficient calculations, and addressing modes for real-time requirements:

1. The RISC architecture provides a limited number of powerful instructions, numerous registers, and single-cycle execution times.
2. The more traditional microcomputer features provide addressing modes for all instructions. This functionality is further enhanced with 100% orthogonality, allowing any instruction to be used with any addressing mode.

MSP430C33x:

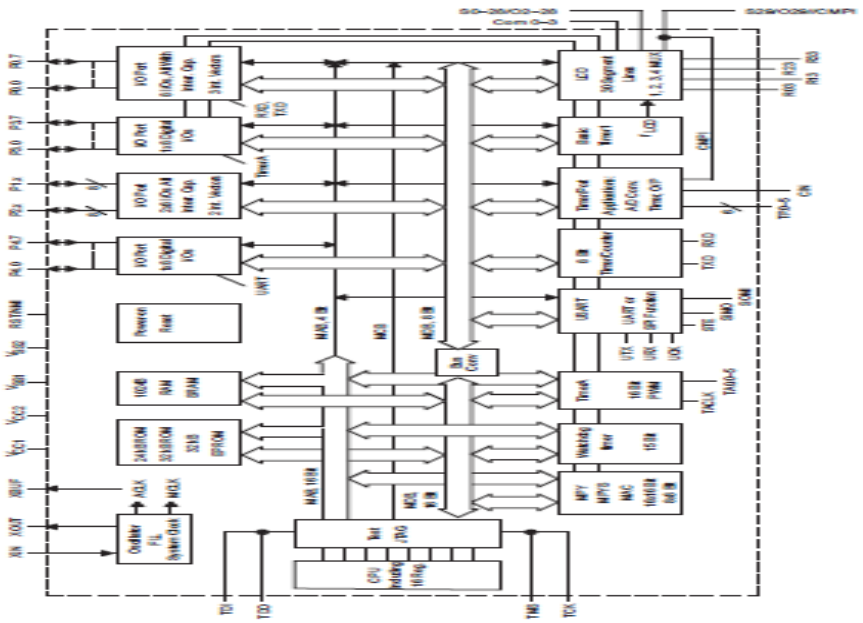


Figure 1–3. MSP430C33x Block Diagram

Real-Time Capability With Ultra-Low Power Consumption

The design of the MSP430 was driven by the need to provide full real-time capability while still exhibiting extremely low power consumption. Average power consumption is reduced to the minimum by running the CPU and certain other functions of the MSP430 only when it is necessary. The rest of the time (the majority of the time), power is conserved by keeping only selected low-power peripheral functions active. But to have a true real-time capability, the device must be able to shift from a low-power mode with the CPU off to a fully active mode with the CPU and all other device functions operating nominally in a very short time. This was accomplished primarily with the design of the system clock:

1. No second high frequency crystal is used — inherent delays can range from 20 ms to 200 ms until oscillator stability is reached
2. Instead, a sophisticated FLL system clock generator is used — generator output frequency (MCLK) reaches the nominal frequency within 8 cycles after activation from low power mode 3 (LPM3) or sleep mode
3. This design provides real-time capability almost immediately after the device comes out of a LPM — as if the CPU is always active. Only two additional MCLK cycles (2 □s @ fC = 1 MHz) are necessary to get the device from LPM3 to the first instruction of the interrupt handler.

Digitally Controlled Oscillator Stability:

The digitally controlled oscillator (DCO) is voltage and temperature dependent, which does not mean that its frequency is not stable. During the active mode, the integral error is corrected to approximately zero every 30.5 □s . This is accomplished by switching between two different DCO frequencies. One frequency is higher than the programmed MCLK frequency and the other is lower, causing the errors to essentially cancel-out. The two DCO frequencies are

interlaced as much as possible to provide the smallest possible error at any given time. See System Clock Generator for more information.

Stack Processing Capability:

The MSP430 is a true stack processor, with most of the seven addressing modes implemented for the stack pointer (SP) as well as the other CPU registers (PC and R4 through R15). The capabilities of the stack include:

1. Free access to all items on the stack — not only to the top of the stack (TOS)
2. Ability to modify subroutine and interrupt return addresses located on the stack
3. Ability to modify the stored status register of interrupt returns located on the stack
4. No special stack instructions — all of the implemented instructions may be used for the stack and the stack pointer
5. Byte and word capability for the stack
6. Free mix of subroutine and interrupt handling — as long as no stack modification (PUSH, POP, etc.) is made, no errors can occur

UNIT- IV

UNIT – IV

Flash Memory:

Introduction:

Flash memory is the most popular type of non-volatile storage, and it has been adopted by most micrcontroller manufacturers. The ability to read/write hunder of thousands of times, even while the processor is running, allows a degree of flexibility not previously available with the older ROM and PROM technologies.

The flash memory in the MSP430 is used primarily to hold code. Because of the flexibility in its design, it is possible to write and read the flash at run time with the CPU. This allows you, for example, to reprogram the microcontroller with a new firmware image. It also enables you to store information that will survive a restart, making it an easy and inexpensive alternative to external

EEPROM devices:

Flash and Memory Organization:

The Flash memory in the MSP430 is usually divided into two sections, the main flash and the Information flash.Main Flash - Represents the bulk of the MSP430's flash and used for program code and constant data. Itself divided into several flash banks, with each bank further divided into segments. When a microcontroller is denoted to have "32kB" of Flash, it is referring to the Main flash:

Information Flash - Several extra segments of flash separate from the Main flash. These are primarily intended for information such as calibration constants, but are much smaller than the main banks. The location and amount of each of these sections depends on the specific MSP430 you're using, and the details are contained in the Memory Map included in the datasheet of the particual MSP430 you're using. Here is an example of the memory organization of several MSP430 devices, taken from their datasheets. Only Flash sections are included. This information was taken directly from the datasheet of the devices. Please note that some details such as the breakdown of the flash segments were obtained from the linker file since the datasheet lacked these specifics. The use of the linker will be discussed later.

We can see that for the FG4618 device, we have a total main flash size of:

01FFFFh - 003100h = 1CEFFh = 118527 bytes

This calculation is easily done use the Windows Calculator in hex mode (the h after each number denotes it is in hex). After the subtraction, convert 1CEFF to decimal.

		MSP430F5438A	MSP430F2274	MSP430FG4618
Main Flash		256kB 045BFFh–005C00h	32kB 0FFFFh-08000h	116KB 01FFFFh - 003100h
Main Code Memory	Bank 3	64 KB 03FFFFh–030000h	N/A	N/A
	Bank 2	64 KB 02FFFFh–020000h		
	Bank 1	64 KB 01FFFFh–010000h		
	Bank 0	64 KB 0045BFFh–0040000h 00FFFFh–005C00h		
Info Memory	Info A	128B 0019FFh-001980h	64 Byte 10C0-10FF	64 Byte 1080-10FF
	Info B	128B 00197Fh-001900h	64 Byte 1080-10BF	64 Byte 1000-107F
	Info C	128B 0018FFh-001890h	64 Byte 1040-107F	None
	Info D	128B 00187Fh-001800h	64 Byte 1000-103F	None

Interrupts:

Interrupts represent an extremely important concept often not covered in programming since they are not readily used in a PC by most programmers. However, it is critical for anyone who programs microcontrollers to understand them and use them because of they advantages they bring.

Imagine a microcontroller waiting for a packet to be received by a transceiver connected to it. In this case, the microcontroller must constantly poll (check) the transceiver to see whether any packet was received. This means that the microcontroller will draw a lot of current since the CPU must act and use the communications module to do the polling (and the transceiver must do the same). Worse, the CPU is occupied taking care of the polling while it could be doing something more important (such as preparing to send a packet itself or processing other information). Clearly,

If the CPU and the system are constantly polling the transceiver, both energy and CPU load are wasted. Interrupts represent an elegant solution to this problem. In the application above, a bettern solution would be to set things up so that the microcontroller would be informed (interrupted) when a packet has been received. The interrupt set in this case would cause the CPU to run a specific routing (code) to handle the received packet, such as sending it to the PC. In the meantime and while no interrupt has occurred, the CPU is left to do its business or be in low power mode and consume little current. This solution is much more efficient and, as we will later cover, allows for significant power savings. Interrupts can occur for many reasons, and are very flexible. Most microcontrollers have them and the MSP430 is no exception.

There are in general three types of interrupts:

1. System Reset
2. Non-maskable Interrupts (NMI)
3. Maskable Interrupts

The interrupts are presented in decreasing order of system importance. The first two types are related to the microcontroller operating as it is supposed two. The last case is where all the

modules allow interrupt capability and the user can take advantage of for the application. System Reset interrupts simply occur because of any of the conditions that resets the microcontroller (a reset switch, startup, etc). These reset the microcontroller completely and are considered the most critical of interrupts. usually you don't configure anything that they do because they simply restart the microcontroller. You have no or little control of these interrupts.

The second type of interrupts is the non maskable ones. Mask ability refers to the fact that these interrupts cannot be disabled collectively and must be disabled and enabled individually. These are interrupts in the category where the error can possible be handled without a reset. Just like a normal PC, a microcontroller is a machine that has to be well oiled and taken care of. The supply voltage has to be satisfied; the clocks have to be right, etc. These interrupts occur for the following reasons:

- An edge on the RST/NMI pin when configured in NMI mode
- An oscillator fault has occurred because of failure
- An access violation to the flash memory occurred

These interrupts do not usually occur but they should be handled. By handled I mean code needs to be written to do something to deal with the problem. If an oscillator has failed, a smart thing to do would be to switch to another oscillator. The User's Guide for the MSP430F2274 provides more information about these type of interrupts. The last type of interrupts is the most common one and the one you have a large control over. These are the interrupts that are produced by the different modules of the MSP430 such as the I/O, the ADC, the Timers, etc. To use the interrupt, we need the following procedure:

1. Setup the interrupt you wish and its conditions
2. Enable the interrupt/s
3. Write the interrupt handler

When the interrupt happens, the CPU stops executing anything it is currently executing. It then stores information about what it was executing before the interrupt so it can return to it when the interrupt handler is done (unless the interrupt handler changes things). The interrupt handler is then called and the code in it is executed. Whenever the interrupt ends, the system goes back to its original condition executing the original code (unless we changed something). Another possibility is that the system was in a Low Power Mode, which means the CPU was off and not executing any instructions. The procedure is similar to the one detailed above except that once the interrupt handler has finished executing the MSP430 will return to the Low Power Mode. Note that the CPU is always sourced from the DCO (or an external high speed crystal) and this source must be on for the interrupt handler to be processed. The CPU will activate to run the interrupt handler.

The process of going from CPU execution (or wakeup) to interrupt handler takes some time. This is especially true whenever the system is originally in sleep mode and must wakeup the DCO to source the CPU. Normally, it is nor critical but some applications might have issues with it taken longer than desired. We will now discuss a useful example: Using the switch of the EZ430-RF2500 to turn on and off the LED. This is best done by example. Two different listings

are provided, the traditional way and an interrupt driver. You'll realize the immense benefits of using the interrupt solution quickly

Listing 9.1: EZ430-RF2500 Toggle LED using Switch - Polling

```
#include "msp430x22x4.h"

void configureClocks();
volatile unsigned int i;       // volatile to prevent optimization

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;     // Stop watchdog timer
  configureClocks();

  // Configure LED on P1.0
  P1DIR = BIT0;                 // P1.0 output
  P1OUT &= ~BIT0;               // P1.0 output LOW, LED Off

  // Configure Switch on P1.2
  P1REN |= BIT2;                // P1.2 Enable Pullup/Pulldown
  P1OUT = BIT2;                 // P1.2 pullup

  while(1)
  {
    if(P1IN & ~BIT2)            // P1.2 is Low?
    {
      P1OUT ^= BIT0;            // Toggle LED at P1.0
    }
  }
}

void configureClocks()
{
  // Set system DCO to 8MHz
  BCSCTL1 = CALBC1_8MHZ;
  DCOCTL = CALDCO_8MHZ;

  // Set LFXT1 to the VLO @ 12kHz
  BCSCTL3 |= LFXT1S_2;
}
```

Low Power Modes:

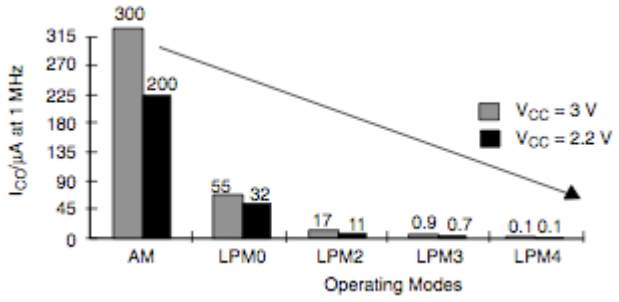
Low Power Modes (LPMs) represents the ability to scale the microcontroller's power usage by shutting off parts of the MSP430. The CPU and several other modules such as clocks are not always needed. Many applications which wait until a sensor detects a certain event can benefit from turning off unused (but still running) parts of the microcontroller to save energy, turning them back on quickly when needed.

Each subsequent LPM in the MSP430 turns off more and more modules or parts of the microcontroller, the CPU, clocks and . T covers the LPM modes available in the MSP430F1611, which are similar to the ones available in most MSP430 derivatives. For the most accurate information, refer to the User's Guide and Datasheet of your particular derivative. Not mentioned here are LPM5 and some other specialized LPMs. These are available only in select MSP430 devices (in particular the newer F5xx series). However, the basic idea behind LPMs is the same, i.e. gradual shut down of the microcontroller segments to achieve power reduction. Active - Nothing is turned off (except maybe individual peripheral modules). No power savings

1. LPM0 - CPU and MCLK are disabled while SMCLK and ACLK remain active
2. LPM1 - CPU and MCLK are disabled, and DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active
3. LPM2- CPU, MCLK, SMCLK, DCO are disabled DC generator remains enabled. ACLK is active
4. LPM3 - CPU, MCLK, SMCLK, DCO are disabled, DC generator is disabled, ACLK is active
5. LPM4 - CPU and all clocks disabled

It is important to note that once parts of the microcontroller are shut off, they willIt is important to note that once parts of the microcontroller are shut off, they will not operate until specifically turned on again. However, we can exit a low power mode (and turn these parts back on), and interrupts are extremely useful in this respect. Another Low Power Mode, LPM5, is available in some derivatives of the MSP430

As an example of the savings that can be achieved by incorporating Low Power Modes into the software design, I present Figure 9.1, which is shown in the MSP430F2274 User's Guide: We usually enter a low power mode with the interrupts enabled. If we don't, we will not be able to wake up the system.



Low Power Mode Savings

The x represents LPMs from 0 to 4 in the MSP430F2274. Another important thing to know is that the low power modes don't power down any modules such as the ADC or timers and these must be turned off individually. This is because we might want to combine the low power mode with them.

A simple example is going in a Low Power Mode while the ADC is sampling and then waking up from the ADC interrupt when the sample has been obtained.

Analog to Digital Converters – ADCs:

Introduction:

The ADC is arguably one of the most important parts of a Microcontroller. Why is this? The real world is analog and to be able to manipulate it digitally some sort of conversion is required. Although many techniques are possible, using an Analog to Digital Converter (ADC or A2D) is the most common when good signal fidelity is necessary. A more simple interface could

be some threshold circuit that would allow you to know whether a signal passed a certain threshold.

The resolution is obviously very poor and would not be used to sample real signals unless it is required. The ADC is also probably the most difficult module to master. It is so because despite the simplicity that is assumed during courses that cover it (in which we just assume the ADC samples at a particular frequency and it does so perfectly to the available resolution.).

We will cover the ADC present in the MSP430F2274, a 10-bit SAR ADC. Most of the information will easily carry over to other ADCs, even if these ADCs have different features. What is an ADC? It is a device which converts the analog waveform at the input to the equivalent representation in the digital domain. It does so by a variety of techniques which depend on the architecture of the ADC itself. SAR stands for Successive Approximation and this name indicates the general idea behind the conversion.

There are many architectures out there. The ADC samples and quantizes the input waveform. Sampling means that it measures the voltage of the waveform at a particular moment in time. In reality, the ADC cannot use an impulse but rather a larger time frame to sample so that the value is stable. When we talk about the sampling frequency, we refer to the fact that the ADC will be continuously sample. This is the most common mode discussed in DSP courses. However, an ADC can sample one or in many other ways, depending on the application. Quantization means it assigns a particular value, usually binary, to represent the analog voltage present. Both of these important concepts will be discussed next.

ADC Parameters:

ADC Resolution:

Probably the most important and most cited parameter of the ADC is its resolution, usually expressed in bits. The MSP430F2274 in our case has 10-bits of resolution. The resolution is an extremely important parameter because it dictates how the lowest difference we can differentiate. The higher the resolution, the more we can distinguish between minute differences in the voltage. The resolution is specified in usually bits. This tells us how many different voltage levels we can represent.

Quantization Levels = 2bits

For a 10-bit ADC we have a maximum of

210 = 1024 possible levels of voltages we can represent. That's why higher bit ADCs such as 16 and even 24 are used: you can readily distinguish between smaller changes in the signal and your digital representation better approximates the original. For 16-bits we have 216 = 65536 and for 24-bit we have more than 16 million different levels, a huge increase due to the exponential nature. Note however that noise is a prevalent issue in ADCs and becomes the major constraint at a certain point

The question now comes to how the number of levels is related to the input voltage? This is simple. The number of levels possible is divided equally for the voltage range. If we have a 1V

on the ADC's input pin and assuming the ADC is operating with a voltage range of 3V, each successive level from 0V to 3V is: 3V 1024 □ 1 = 0:00293V = 3mV

If the signal changes by less than 3mV we can't distinguish the difference, but every 3mV change in signal level translates into an addition of 1 to the binary number. If we have 0V at the input (without any noise), then we have 10 zeros (since it's a 10-bit ADC and no signal at the input). However, if we have 3mV at the input then we have 9 zeros followed by a one.

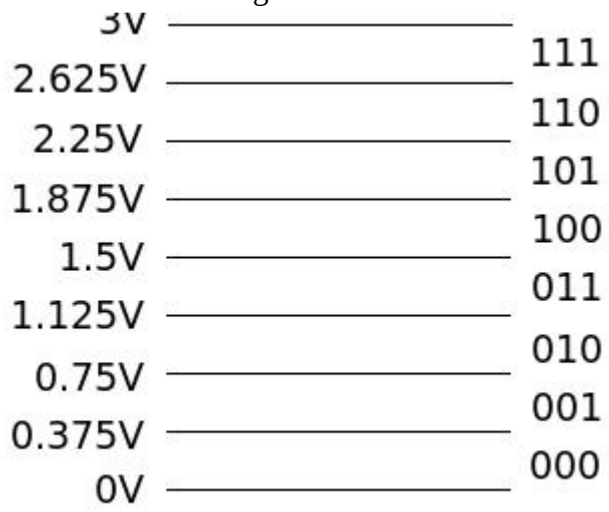
0V! 0000000000 = 0x00

3mV! 0000000001 = 0x01

You can think about the ADC as a staircase. For each range of voltages we assign an equivalent binary number usually referred to as the "code". The binary representation is usually replaced by hex since it is easier to read, as discussed in previous chapter. Note however that this is the ideal case and no offset or errors are taken into account.

The following shows a simple 3-bit ADC. Such ADC does not exist but serves to show how each new level is represented in binary:

Unfortunately, we cannot cover all the important details of ADCs. The interested reader is referred to the following excellent sources of information:



The general equation of the MSP430F2274 (which can be assumed for most other ADCs given a few changes) is:

NADC = 1023 _ Vin, VR VR+ VR

A simple example follows: Lets assume that the input voltage is 1V. VR+ and VR□ represent the upper and lower limits of the voltage range. In our case we will have VR+ = VCC and VR+ = VCC = VSS = 0V.

Using the equation above we have:

NADC = 1023 _1V □ 0V 3V □ 0V = 1023 _ 0:333 = 341 = 0x155

The binary representation is not shown because it is large and hexadecimal represents it nicely. Notice that 1V is exactly a third of the voltage range and therefore the output code will be a third of all the maximum number, 1023 in this case.

ADC Sampling Frequency:

The Sampling Frequency is another critical parameter of the systems. Usually the ADC manufacturer specifies this in SPS (samples per second). The waveform at the input of the ADC might not stay constant. It might remain constant for long periods of time or change continuously (as with a voice at the input). When we sample the data, if it doesn't change, we can sample once. However, it is more likely that we will setup the ADC to sample regularly. Different signals at the input of the ADC will require different sampling speeds. It is possible to oversample by sampling at a very high speed and therefore be able to capture different signals.

Realize that if not sampling fast enough, very quick events can not be detected. Speech signals for example are generally considered to have important information for the region of 300Hz to 4000Hz. Using Nyquist's theorem we realize that we must therefore sample at a frequency twice the highest frequency (This is for low pass signals, i.e. signals where the lowest frequency is relatively close to 0Hz).

I/O Ports:

The most straightforward form of input and output is through the digital input/output ports using binary values (low or high, corresponding to 0 or 1).We already used these for driving LEDs and reading switches. In this section we look at their wider capabilities. There are 10–80 input/output pins on different devices in the current portfolio of MSP430s; the F20xx has one complete 8-pin port and 2 pins on a second port, while the largest devices have ten full ports. Almost all pins can be used either for digital input/output or for other functions and their operation must be configured when the devices starts up.

This can be tricky. For example, pin P1.0 on the F2013 can be a digital input, digital output, input TACLK, output ACLK, or analog input A0+. This is a choice of five functions and therefore needs at least 3 bits for selection. It was hard to puzzle this out for older devices but newer data sheets have an admirably clear table for each pin in the section *Application Information*. There is also a schematic drawing of the circuit associated with the pin. For example, the function of P1.0 depends on P1DIR, P1SEL, and the analog enable register SD16AE. This pin is a digital input by default after reset, which is true for most pins but not all.

A convenient feature of all peripherals in the MSP430 is that they are implemented in much the same way in all devices and families. For example, ports P1 and P2 have interrupts in all cases, from the 14-pin F20xx to the 100-pin FG4618. Up to eight registers are associated with the digital input/output functions for each pin. Here are the registers for port P1 on a MSP430F2xx, which has the maximum number. Each pin can be configured and controlled individually; thus some pins can be digital inputs, some outputs, some used for analog functions, and so on.

Port P1 input, P1IN: reading returns the logical values on the inputs if they are configured for digital input/output. This register is read-only and volatile. It does not need to be initialized because its contents are determined by the external signals.

Port P1 output, P1OUT**:** writing sends the value to be driven to each pin if it is configured as a digital output. If the pin is not currently an output, the value is stored in

a buffer and appears on the pin if it is later switched to be an output. This register is not initialized and you should therefore write to P1OUT *before* configuring the pin for output.

Port P1 direction, P1DIR**:** clearing a bit to 0 configures a pin as an input, which is the default in most cases.Writing a 1 switches the pin to become an output. This is for digital input and output; the register works differently if other functions are selected using P1SEL.
Port P1 resistor enable, P1REN**:** setting a bit to 1 activates a pull-up or pull-down resistor on a pin. Pull-ups are often used to connect a switch to an input as in the section "Read Input from a Switch" on page 80. The resistors are inactive by default (0). When the resistor is enabled (1), the corresponding bit of the P1OUT register selects whether the resistor pulls the input up to *V*CC (1) or down to *V*SS (0).

Port P1 selection, P1SEL**:** selects either digital input/output (0, default) or an alternative function (1). Further registers may be needed to choose the particular function.
Port P1 interrupt enable, P1IE**:** enables interrupts when the value on an input pin changes. This feature is activated by setting appropriate bits of P1IE to 1. Interrupts are off (0) by default. The whole port shares a single interrupt vector although pins can be enabled individually.

Port P1 interrupt edge select, P1IES**:** can generate interrupts either on a positive edge (0), when the input goes from low to high, or on a negative edge from high to low (1). It is not possible to select interrupts on both edges simultaneously but this is not a problem because the direction can be reversed after each transition. Care is needed if the direction is changed while interrupts are enabled because a spurious interrupt may be generated. This register is not initialized and should therefore be set up before interrupts are enabled.

Port P1 interrupt flag, P1IFG**:** a bit is set when the selected transition has been detected on the input. In addition, an interrupt is requested if it has been enabled. These bits can also be set by software, which provides a mechanism for generating a software interrupt (SWI).

In some cases the configuration of a pin selected by these registers can be overruled by another function. For example, P1.0 in the F2013 can also be used as input A0+ to the analog-to-digital converter (SD16_A). This module includes an analog input enable register SD16AE. Selecting channel 0 with this register connects P1.0 to the SD16_A, regardless of the settings of P1SEL and P1DIR. This is made clear in the *Application Information* but needs careful reading.
Other ports are similar, although most have fewer registers: Ports other than P1 and P2 have only the four registers PnIN, PnOUT, PnDIR, and PnSEL in the MSP430x1xx and MSP430x4xx families. Here are a few points to watch about the input/output ports:
Interrupts are available only on ports P1 and P2 so the PnIE, PnIES, and PnIFG registers are provided for only these two.

1. Pull resistors and the PnREN register are provided only in the MSP430F2xx family and newer MSP430x4xx devices. *Do not activate pull/resistors unless you are using a pin for*

digital input. Selecting a pull-up or pull-down on an output pin removes the full output drive and gives only a feeble current through the pull-up to resistor instead.

2. Pins P2.6 and P2.7 on many devices in the MSP430F2xx family are exceptions to the general rule and are *not* digital inputs by default. These pins can also be used for a crystal, which is their default configuration. You should reconfigure these pins if the internal VLO is used instead of a crystal.
3. There is no port 0 (P0) on modern devices. It was present in some members of MSP430x3xx family and differed from the other ports in several ways, notably the handling of interrupts.

Example

Write code to configure the pins of a F2013 as follows:

P1.0 and P1.1 are inputs A0+ and A0− to the SD16_A analog-to-digital converter (ADC).

1. P1.2 is input CCI1A to Timer_A.
2. P1.3 is connected to the voltage reference VREF of SD16_A.
3. P1.4 is a digital output, initially driven low.
4. P1.5 is output TA0 from Timer_A.
5. P1.6 and P1.7 are not used; leave them unconfigured for now.
6. P2.6 and P2.7 are digital inputs with pull-up resistors; ACLK should be derived

The registers associated with input/output ports on many other microcontrollers are a little peculiar because they are not simple memories. Many designs, including the Microchip PIC16 and the Freescale HCS08, use the *same* register for input and output through a port.

In this case, reading the register usually returns the values on the pins, while writing to it drives the values onto pins if they are configured as outputs. This has the curious effect that writing a value to a port and immediately reading it back does not return the same value for bits that are configured as input: The write operation puts values into the output buffer but the read operation gives the values on the input pins due to the surrounding circuitry. Great care must be used when handling these schizophrenic registers. This is not an issue with the MSP430 because it has separate input and output registers. You would expect P1IN.x=P1OUT.x when pin x is an output, which should be true provided that the output is not overloaded.

The digital input/output ports are sometimes called *parallel ports* because all eight pins can be used simultaneously to read or write a complete byte. However, they are nothing like the parallel port found on the back of old PCs. One byte is the largest unit that can be handled in most MSP430 devices because of the way in which the addresses of the registers are arranged. The exception is the FG6461X, where the registers for ports P7–P10 are laid out in the memory map so that they may be accessed individually as bytes or in pairs as words. Thus the bytewide ports P7 and P8 can also be handled as the wordwide port PA. For example, the (byte) register P7IN has address 0x0038 and P8IN is at 0x039, so the 2 bytes can be treated as the word at address 0x0038, which is the (word) register PAIN. Similarly, port PB is equivalent to ports P9 and P10.

Circuit of an Input/Output Pin

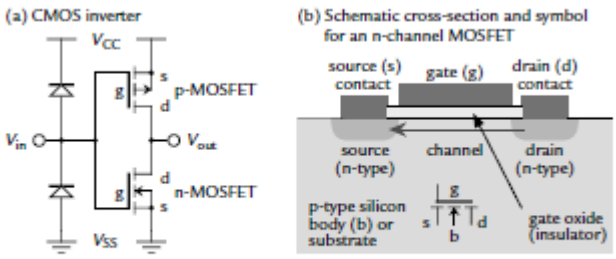


Figure 7.1: (a) Circuit of a simple CMOS inverter including the input protection diodes. (b) Schematic cross-section of an n-channel MOSFET.

often symmetric. More precisely, the channel exists when the voltage between the gate and source, *V*gs, exceeds a critical value called the *threshold voltage*, *Vt* . In most MOSFETs *Vt* > 0 so current cannot flow between the source and drain when there is no difference in voltage between the gate and source. This, called an *enhancement-mode device*, is indicated by the broken line in the symbol for the devices in the figure.

We normally think of a MOSFET as a three-terminal device but in practice there is a fourth connection to the *body* or substrate. The reason is that the source–body and drain–body junctions act like n–p diodes, which must be kept reverse biased for correct operation of the device. The body is typically connected to *V*SS in an integrated circuit, which is the most negative point. In a discrete device, the body is connected to the source, as shown in the symbol. A diode remains between the source and drain, which comes into play with inductive loads as we see in the section "Driving Heavier Loads" on page 247.

The signs of the voltages and current are reversed in a p-channel MOSFET. The drain is negative with respect to the source and the gate–source voltage must be made more negative than the threshold voltage to turn on the channel. One of the most significant features of a MOSFET is that the gate is separated from the channel by a thin layer of silicon dioxide, which is an insulator. Thus there is no direct electrical connection between the source and channel. Instead, the gate, oxide, and channel form a capacitor. This is reflected in the symbol by a gap between the gate and channel. No current flows into a capacitor when the voltage across it is constant, which is a key factor in the low power consumption of CMOS circuits. Thus the MSP430 can retain the contents of its registers in LPM4 while drawing less than 1_A. On the other hand, current must flow to charge and discharge the gate–channel capacitance when the transistors change state and these accounts for most of the supply current in CMOS.

This gate oxide is only a few nanometers (10−9 m) thick in a modern transistor. Silicon Dioxide is an excellent insulator but even this breaks down if the electric field across it becomes too large. It is easy for a person to acquire large voltages through static electricity—walking across a nylon carpet in dry weather generates enough charge to cause a spark when you next touch a grounded conducting object. CMOS devices must therefore be handled only at workstations that are protected against static electricity by grounding and an appropriate choice of materials. Integrated circuits themselves are also protected by connecting diodes between

inputs and the supply rails, as shown in Figure 7.1(a). Recall that current flows only in the direction shown by the arrow symbol for the diode. These are reverse-biased in normal operation, where *V*SS <*V*in <*V*CC. They turn on to protect the circuit when the input strays by more than about 0.3V outside the supply rails, *V*in <*V*SS−0.3V or *V*in >*V*CC+0.3V. The magnitude of the current through these diodes should not exceed 2 mA.

The input protection diodes can cause a puzzling side effect. Suppose that a logical high input is applied to a circuit whose power supply is not connected. Current flows through the protection diode from the input to *V*CC, from where it supplies the rest of the circuit.

Thus the circuit works almost normally, despite having no apparent source of power.

After that diversion, we can explain the operation of a CMOS inverter with the aid of Figure 7.2. A model inverter can be made with a pair of controlled switches, one between the output and *V*SS to pull the output down to logic 0 and the other between the output and *V*CC to provide a logic 1. Ideally one switch should always be closed and one open.

If the input is a logical 1, the output should be a logical 0, which needs the switch to *V*SS closed and that to *V*CC open, as in Figure 7.2(a). The lower switch should therefore close when the input is relatively positive (near *V*CC) and the upper switch should be open under the same conditions, passing no current. Everything is reversed when the input is a logical 0, near *V*SS.

This can be achieved by using an n-channel MOSFET (n-MOSFET for short) for the lower switch and a p-MOSFET for the upper switch, as shown in Figure 7.2(b). A channel is created in the n-MOSFET, allowing conduction in the same way as a closed switch, when its gate is driven positive by a high input or logical 1. When the input falls to a logic 0, so *V*in = *V*SS, there is no difference in potential between the gate and source of the MOSFET.

Thus *V*gs = 0, the channel vanishes, and no current flows. This is just like an open sw

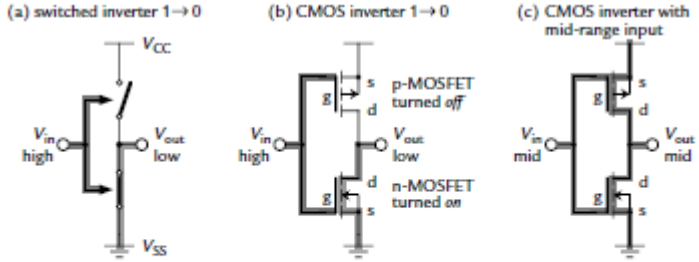


Figure 7.2: Operation of a CMOS inverter. (a) Model inverter using switches with a logical input of 1 and output of 0. (b) Corresponding operation of MOSFETs. (c) Operation when the input lies near the middle of the supply voltages V_{SS} and V_{CC} rather than close to either extreme. Both MOSFETs conduct and a large current flows from V_{CC} to V_{SS}.

Configuration of Unused Pins

Not all of the input/output pins are used in most applications. *Unused pins must never be left unconnected in their default state as inputs.* This follows a general rule that inputs to CMOS must never be left unconnected or "floating." A surprising number of problems can be caused by floating inputs. The most trivial is that the input circuit draws an excessive current from the power supply. This is because the input is likely to float to the midpoint of *V*SS and *V*CC, turning on both MOSFETs and leading to the situation shown in Figure 7.2(c). The shoot-through current may exceed 40_A, a huge waste by the standards of the MSP430.

Old CMOS circuits, such as the 74HC family, are amazingly sensitive to floating inputs. They may oscillate or refuse to work at all if an input is floating, even if the input belongs to an unused gate or flip-flop. I have seen this happen many times when students have not taken heed of the rule about floating inputs. Missing decoupling (bypass) capacitors can cause similar problems. Floating inputs are also susceptible to noise and to static electricity if the product is handled, which may lead to permanent damage.

There are three ways of avoiding these problems:

1. Wire the unused pins externally to a well-defined voltage, either *V*SS or *V*CC, and configure them as inputs. The danger with this is that you might damage the MCU if the pins are accidentally configured as outputs.
2. Leave the pins unconnected externally but connect them internally to either *V*SS or *V*CC by using the pull-down or pull-up resistors. They are again configured as inputs. I prefer this approach but it is restricted to the MSP430F2xx family because the others lack internal pull resistors.
3. Leave the pins unconnected and configure them as outputs. The value in the output register does not matter. This is perhaps the most robust solution and is recommended for MSP430 devices that lack internal pull resistors. I am less keen on this approach for experimental systems because it is easy to short-circuit pins with a test probe.

There is a helpful list of recommended connections for unused pins at the end of th chapter on *System Resets, Interrupts, and Operating Modes* in the family user's guides.

Digital Inputs:

Digital inputs to the MSP430 are typically connected to digital outputs from other circuits or to components such as switches.We already used the digital inputs many times for straightforward connections to a push button using the standard circuit shown in Figure 4.4. The programs in Chapter 4 used polling but this is wasteful for inputs that change on a human timescale—very slowly by electronic standards. Interrupts may be more efficient. A different approach is also needed when a large number of inputs must be read.

Interrupts on Digital Inputs:

Ports P1 and P2 can request an interrupt when the value on an input changes. This is one of the few interrupts that remains active in LPM4 and is therefore useful to wake the CPU in portable equipment that lies idle for a long time. Interrupts for port P1 are controlled by the registers P1IE and P1IES, mentioned previously, and similarly for port P2. There is a single vector for each port, so the user must check P1IFG to determine the bit that caused the interrupt. This bit must be cleared explicitly; it does not happen automatically as with interrupts that have a single source.

The direction of the transition that causes the interrupt can be changed in P1IES at anyn time by the program. This is useful if both edges of a pulse need to be detected, for example, when a button is pressed and released. There is a danger that spurious interrupts may be generated, so it is a good idea to disable this interrupt, change P1IES, and clear any spurious

flags in P1IFG before reenabling the interrupt. In fact, to practice this should not be a problem if the direction is changed in the most obvious way. For instance, the port may wait for a low-to-high transition while the input is low. An interrupt is requested when the input goes high. The sensitivity is then changed to high-to-low to detect the next edge.

The use of interrupts is illustrated in Listing 7.1, which is perhaps the ultimate development of the programs to light an LED when a button is pressed. The device spends most of its time in LPM4, waiting for an interrupt on pin P2.1. Both the LED and the direction of the transition for an interrupt are toggled in the ISR. Any pending requests for an interrupt are cleared by a loop before returning to LPM4. I included this code as an example of how to avoid spurious interrupts. Unfortunately it is not a particularly good idea here because it loses the second edge of short pulses.

Multiplexed Inputs: Scanning a Matrix Keypad:

Many products require numerical input and provide a keypad for the user. These often have 12 keys, like a telephone, or more. An individual connection for each switch would use an exorbitant number of pins so they are usually arranged as a matrix instead. Only seven pins are needed for a 12-key pad, as shown in Figure 7.3, or eight pins for 16 keys. As usual this economy comes at a price. The matrix must be scanned, which is more complicated than reading individual inputs. Moreover, the reading may become ambiguous if more than one key is depressed.

There are, as usual, many ways of dealing with a keypad. Here is a straightforward approach, although it needs refinement in practice. I do not worry about debouncing at this stage and assume that no more than one switch is closed.

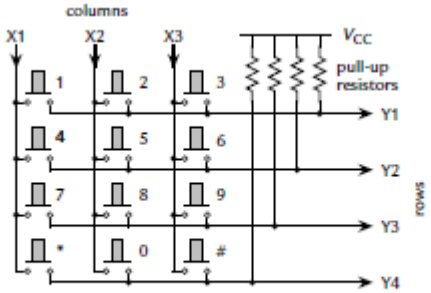


Figure 7.3: Connection of a 12-button telephone keypad as a 3 × 4 matrix.

Connect the rows Y1–Y4 as inputs to the microcontroller while the columns X1–X3 are driven by outputs. (It could equally well be the other way around.) Pull-up resistors are required on the inputs. These could be internal for the MSP430F2xx family but must be provided externally for other devices.

1. Drive X1 low and the other columns X2 and X3 high. This makes the switches in column X1 active and the corresponding Y input goes low if a button is pressed.

Thus we can detect the state of switches 1, 4, 7, or *. The switches in the other columns have no effect because both of their terminals are at *V*CC.

2. Drive X2 low and the other columns high to read the switches in column X2.

3. Repeat this for column X3.

This process can be repeated as often as required.

A problem with this simple method arises if two buttons, such as 1 and 2, are pressed, which short-circuits the column drives X1 and X2. This damages the output of the microcontroller if they are connected directly. Resistors should therefore be connected between the pins of the microcontroller and the columns of the keypad. Diodes could be used instead. Another possibility for the MSP430F2xx family is to use the internal pull-ups instead of full-strength outputs for columns that are not being addressed. Please remember to disable the pull-up before trying to use the pin for "real" output. Usually only one key should be pressed at a time on the pad. In fact there should be no problem identifying two keys held down simultaneously. Difficulties arise when three buttons on the corners of a rectangle are pressed because it appears that the button on the fourth corner is also down. An error must be noted for a standard keypad in this case.

In other applications it is necessary to be able to read all the switches independently and the solution is to put a diode in series with each switch. It is a waste of energy to scan the keypad when no button is being pressed. In this case it is more efficient to drive *all* columns low and wait for an interrupt generated by a falling edge on any of the row inputs. The keypad can then be scanned to determine which key has been pressed. A complete program is given in the application note *Implementing an Ultralow-Power Keypad Interface with the MSP430* (slaa139). Section 5.5.5 of

Application Reports (slaa024) shows how the ideas can be extended to scan different types of input

Analog Aspects of "Digital" Inputs:

It is usually safe to assume that signals inside the microcontroller are straightforward logical zeros and ones although there are a few exceptions to this comfortable situation, such as the output of a comparator (see the section "Comparator_A" on page 371). Outside the microcontroller there is no escape from the fact that the real world is analog. This raises many issues, of which the most basic is the question, What analog voltages correspond to the digital values 0 and 1? Take *V*SS = 0 for clarity. The precise input voltages *V*in that correspond to logical 0 and 1 depend on the technology but typical values for CMOS are

1. Inputs of 0 to 0.3*V*CC give logic 0.
2. Inputs of 0.7*V*CC to *V*CC give logic 1.

These are symmetric. The output voltage *V*out for CMOS is typically below 0.1*V*CC for a logical 0 and above 0.9*V*CC for a logical 1. This is again symmetric and the large gap between the ranges for logical 0 and 1 means that CMOS is relatively insensitive to noise.

The logical value is undefined for an input that lies in the transition region between the two ranges, which is typically 0.3*V*CC to 0.7*V*CC for CMOS. Inputs in this range also cause an excessive current to flow through the input buffer, as shown in Figure 7.2(c). The apparent logical value as seen from inside the microcontroller may also oscillate wildly between 0 and 1.

The voltage on inputs should therefore pass rapidly through the transition zone. This is particularly important for inputs that generate interrupts or provide clocks
(to a timer, for instance). An excellent feature of the MSP430 is that its inputs are provided with *Schmitt triggers*, which eliminate many of these problems. This includes the standard port inputs, which also protects their interrupts. Other digital inputs, such as the external clock to the timer, also pass through the Schmitt triggers. The details are shown in *Port pin schematics* in the data sheets. Figure 7.4 shows the output voltage as a function of the input voltage for a conventional buffer and a Schmitt trigger. The major difference is that a Schmitt trigger displays
hysteresis. This is most easily explained by looking at the response to a slow triangular, wave on the input in Figure 7.4(c).The input and output are initially at 0 voltage and the input voltage rises gradually. The output remains safely in the range for logical 0 until the input passes through the upper

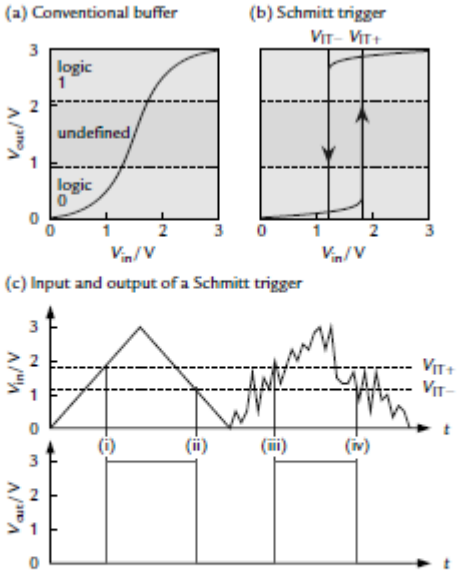


Figure 7.4: Transfer characteristic (output voltage as a function of input voltage) for (a) conventional buffer and (b) Schmitt trigger. The Schmitt trigger shows hysteresis and never gives an output with an undefined logic value. (c) Input to and output from a Schmitt trigger as a function of time. The trigger turns a slowly varying input into sharp transitions and eliminates noise.

Positive-going threshold voltage *V*IT+ at (i). At this point the output jumps abruptly to a value that is well inside the range for a logical 1. It stays here while the input continues to rise and after it has started to fall again. The output remains at a logical 1 even after the input has fallen through *V*IT+ and does not change until the input crosses the lower threshold voltage *V*IT− at (ii). At this point the output jumps to a logical 0 and remains here as the input falls further.

The second half of the plot shows the effect of (rather fanciful) noise on the input. The output jumps from 0 to 1 when the input first goes above *V*IT+ at (iii). The noise pulls the input back down below this threshold but the output remains cleanly at 1. Similarly, on thedownward half of the wave, the output falls from 1 to 0 at point (iv) when the input first falls below *V*IT− and is not affected by fluctuations that take the input back above this threshold. Thus a Schmitt trigger has two desirable effects:

1. It turns slowly varying inputs, which might cause problems while they pass slowly through the undefined range of input voltages, into abrupt, clean logical transitions. It eliminates the effect of noise on the input, provided that it is not large enough to span the gap between the upward and downward thresholds.
2. Schmitt triggers have many other applications. A simple oscillator can be made by adding a resistor and capacitor, for instance.

Another analog aspect of the inputs is that a small current flows into or out of the pin. Ideally this would be 0 because the gates of MOSFETs act like capacitors but in reality the capacitors leak slightly, as do the input protection diodes. Of course the circuit associated with each pin is much more complicated than a simple inverter too. Having said all that, the pins of the MPS430 have a low input leakage current, below ±50nA. This is roughly equivalent to a resistance of 50M_ and means that leakage should rarely cause a serious drop in voltage across a resistor in series with the input. The leakage current would not drop more than ±0.1V across a 2M_ resistor, for example. This allows large pull-up resistors to be used to save current, although the input may then become sensitive to noise.

The low leakage current is also important if the input is used to detect the voltage on a small capacitor (a few pF). This arises in touch sensors, which are also mentioned in the section "Capacitative Touch Sensing with Comparator_A" on page 391. Finally, low leakage contributes to a long battery life, particularly in devices with over 100 pins.

Before leaving this topic, let us look more closely at the way in which thresholds and other parameters are specified in the data sheet. Table 7.1 shows a small extract for the F20xx. There is a minimum and maximum for each of the two threshold voltages. When is each important? Suppose that the input is initially at *V*SS:

1. The system should not respond to noise on the input, so any fluctuations in voltage must be kept below the threshold *V*IT+. In this case we should choose the *minimum* value of 1.35V to ensure that the input is never triggered.
2. On the other hand, we want to guarantee that the microcontroller responds when the desired signal appears on the input.We must ensure that the input goes above the *maximum* value of 2.25V to be certain that the Schmitt input is triggered.

Table 7.1: Selected electrical characteristics of Schmitt trigger inputs with $V_{CC} = 3$ V adapted from the data sheet for the F20xx.

	Parameter	Minimum	Typical	Maximum	Unit
V_{IT+}	Positive-going input threshold voltage	1.35		2.25	V
V_{IT-}	Negative-going input threshold voltage	0.75		1.65	V
V_{hys}	Input voltage hysteresis, $V_{IT+} - V_{IT-}$	0.3		1.0	V
R_{Pull}	Pull-up/pull-down resistor	20	35	50	kΩ
C_I	Input capacitance		5		pF
$T_{(int)}$	External interrupt timing	20			ns
I_{lkg}	High-impedance leakage current			±50	nA

Digital Outputs:

The standard circuits for connecting an LED to a pin of a microcontroller are shown in Figure 4.3 and are repeated in Figure 7.9, which includes the transistors inside the MSP430. Always include current-limiting resistors in series with the LEDs. Remember also that LED

stands for light-emitting *diode* and that a diode passes current in only one direction, shown by the arrow in the symbol. This refers to conventional current, which flows from positive to negative. No light is produced if the LED is connected backward.

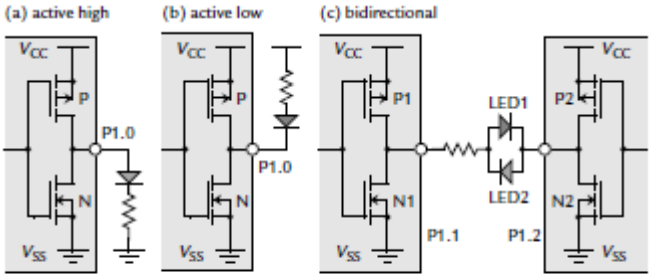


Figure 7.9: Standard connection of an LED to pin P1.0 in (a) active high and (b) active low configurations. (c) Connection of a bidirectional pair of LEDs between a pair of pins, P1.1 and P1.2.

In the active high circuit (a) the LED lights when the p-MOSFET is switched on and the n-MOSFET is off. Current flows from *V*CC through the p-MOSFET, out of the pin andn through the LED to *V*SS. The pin therefore acts as a *source* of current. The opposite is true in the active low circuit (b). This time current flows from *V*CC through the LED, into the pin and through the n-MOSFET to *V*SS. The pin is said to be a current *sink*. In general n-MOSFETs have better performance than p-MOSFETs and this is why many older ICs were better at sinking current than sourcing current. LEDs were therefore usually connected active low. Most modern microcontrollers are designed so that the performance of the output is more or less symmetric.

An important parameter is the maximum recommended current in or out of the port pins. The data sheets are surprisingly reticent: No limiting currents are specified at the time of writing.\ The section on *Electrical Characteristics* includes plots of the output voltage as a function of current that go up to ±40 mA. This would be a startlingly high current for any microcontroller, let alone a low-power device. I presume that it is measured for very short pulses to avoid destructive overheating. In contrast, the product information center recommends that the current should be limited to 4 or 5mA per pin and 25mA per port.

Consult them if your application approaches these bounds or see the section “Driving Heavier Loads” on page 247 for circuits that allow the MSP430 to switch heavier loads.

There is usually no problem with connecting a few inputs to a single output. On the other hand, you should *never* connect two ordinary outputs together because this causes contention if they attempt to produce different outputs, which may damage them. Special circuits are used where outputs must be connected together, such as on a bus. Three-state outputs are one type. These have the two usual high or low states when they are driving the bus. The pin has a high impedance in the third state so that it does not affect the voltage on the bus, which is released for other outputs. This can be done in the MSP430 by switching the pin from output to input. Some sort of control is needed to ensure that only one output attempts to drive the bus at a time. A simpler approach is to use open-drain or open-collector outputs, which can pull the output down to *V*SS but not up to *V*CC, for which a pull-up resistor is provided. More details will be given in the section “Hardware for I²C” on page 535.

Figure 7.9(c) shows how to connect a bicolor LED. This has two LEDs in a common package, connected so that one color lights when the current flows in a particular direction and the other color lights for current in the opposite direction. The package must be connected between a pair of pins, which act as a simple H-bridge (see the section "Driving Heavier Loads" on page 247). Suppose that pin P1.1 is driven high and P1.2 low. Current flows from *V*CC through P1, LED1, and N2 to *V*SS. Similarly, LED2 is lit by driving P1.1 low and P1.2 high. Neither LED is active if both pins are driven high or both low. The same technique can be used for other loads that need both directions of current.

Multiplexed Displays

The idea behind the bidirectional output can be carried further as a way of multiplexing LEDs, as shown in Figure 7.10. In general, $N(N-1)$ LEDs can be driven from N pins by extending this circuit. It relies on the one-way characteristic of a diode and its nonlinear relation between current and voltage. A single LED is selected by driving one pin high,

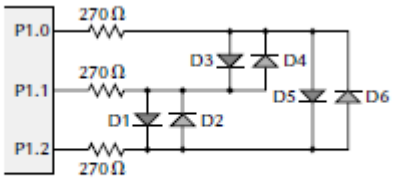


Figure 7.10: Circuit to select one LED by multiplexing. Up to $N(N-1)$ LEDs can be driven from N pins of the microcontroller.

one low, and configuring all other pins as inputs. For example, suppose that P1.0 is high, P1.2 is low, and P1.1 is an input and therefore effectively disconnected. Current flows through D5 and two of the series resistors, which limit the current in the usual way.

A parallel path lies through D3 and D1 but this has two LEDs in series. Each receives only half the voltage, so very little current flows. The remaining LEDs are reverse biased.
We could light D6 instead by driving P1.0 low and P1.2 high, and similarly for other LEDs

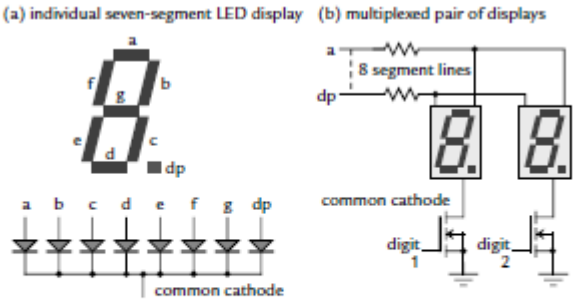


Figure 7.11: (a) Physical layout and circuit of a single, common-cathode, seven-segment LED display. The segments are labeled in a standard way and *dp* stands for decimal point. (b) Multiplexed connection of two digits. The eight segment lines and the gates of the n-MOSFETs, one per display, are connected to pins of the microcontroller.

Only one LED can be addressed at a time, which may seem a serious disadvantage. This is resolved by repeatedly addressing each LED in turn. The eye does not detect that the LEDs are flashing rather than continuously illuminated, provided the frequency is above 100 Hz or so. In

fact LEDs are often more efficient when operated in this way, in the sense that they need a smaller average current to produce the same apparent brightness. As usual this feature comes at a price: The current must be higher during each pulse, and this may exceed the limit of the MSP430's pins.

A more conventional form of multiplexing is often used with seven-segment LED displays. These tend to consume a large current, which clashes with the low-power ethos of the MSP430, but are simpler than liquid crystal displays. The layout and circuit of a single digit are shown in Figure 7.11(a). Note that there are usually eight segments, despite the name; the eighth segment is a decimal point. In the circuit shown, the cathodes of all the LEDs are connected to give a *common cathode* display. (The cathode is the negative terminal when the diode is forward biased, shown by the bar on the symbol.) Common anode displays are equally common. The usual resistors should be connected in series with each segment to limit the current. A higher value may be needed for the decimal point because it draws a lower current. Sometimes the segments comprise two or more LEDs in *series*, but this requires a higher voltage than an MSP430 can provide.

Suppose that two digits are needed. This would require 16 pins of the microcontroller if the displays were connected separately, which is excessive. They are usually multiplexed instead, as shown in Figure 7.11(b). These are again common-cathode devices. The corresponding segment pins of the displays are connected in parallel and the common cathodes are used to select a particular digit. In principle each cathode could be connected to a pin of the microcontroller but the current would exceed the rating of the MSP430 so I use an n-MOSFET as a switch instead; this is described fully in the section "Driving Heavier Loads" on page 247.

The two digits are driven alternately. To select digit 1, the gate of its FET is driven to *V*CC, which turns it on, while the gate of the FET for digit 2 is driven to *V*SS to turn it off. Individual segments of digit 1 can then be lit by bringing the corresponding pins high or turned off by pulling them low. Both displays feel the voltages on the segment lines but only digit 1 is able to respond to them. The voltages on the FETs are then reversed to make digit 2 active and the segment lines are changed to give the desired pattern. The details of the software, such as using lookup tables to get the correct patterns on the segment linesA large, seven-segment display needs more current than the MSP430 can provide itself.

This is not a problem because plenty of special ICs are available to drive LEDs. The LEDs are usually driven from constant-current sources to give better control of illumination than a simple resistor. Many have serial interfaces such as SPI or I²C (see Chapter 10), which save pins on the microcontroller.

Watchdog Timer:

The main purpose of the watchdog timer is to protect the system against failure of the software, such as the program becoming trapped in an unintended, infinite loop. Left to itself, the watchdog counts up and resets the MSP430 when it reaches its limit. The code must therefore keep clearing the counter before the limit is reached to prevent a reset.

The operation of the watchdog is controlled by the 16-bit register WDTCTL. It is guarded against accidental writes by requiring the password WDTPW = 0x5A in the upper byte. A reset will occur if a value with an incorrect password is written to WDTCTL. This can be done deliberately if you need to reset the chip from software. Reading WDTCTL returns 0x69 in the upper byte, so reading WDTCTL and writing the value back violates the password and causes a reset. The lower byte of WDTCTL contains the bits that control the operation of the watchdog timer, shown in Figure 8.1. The RST/NMI pin is also configured using this register, which you must not forget when servicing the watchdog—we see why shortly. This pin is described in the section "Nonmaskable Interrupts" on page 195. Most bits are reset to 0 after a power-on reset (POR) but are unaffected by a power-up clear (PUC). This distinction is important in handling resets caused by the watchdog. The exception is the

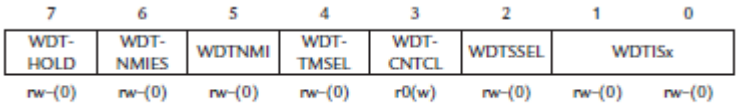


Figure 8.1: The lower byte of the watchdog timer control register WDTCTL.

WDTCNTCL bit, labeled r0(w). This means that the bit always reads as 0 but a 1 can be written to stimulate some action, clearing the counter in this case. The watchdog counter is a 16-bit register WDTCNT, which is not visible to the user. It is clocked from either SMCLK (default) or ACLK, according to the WDTSSEL bit. The reset output can be selected from bits 6, 9, 13, or 15 of the counter. Thus the period is 26 = 64, 512, 8192, or 32,768 (default) times the period of the clock. This is controlled by the WDTISx bits in WDTCTL. The intervals are roughly 2, 16, 250, and 1000 ms if the watchdog runs from ACLK at 32 KHz.

The watchdog is always active after the MSP430 has been reset. By default the clock is SMCLK, which is in turn derived from the DCO at about 1 MHz. The default period of the watchdog is the maximum value of 32,768 counts, which is therefore around 32 ms. You must clear, stop, or reconfigure the watchdog before this time has elapsed. In almost all programs in this book, I take the simplest approach of stopping the watchdog, which means setting the WDTHOLD bit. This goes back to the first program to light LEDs,

If the watchdog is left running, the counter must be repeatedly cleared to prevent it counting up as far as its limit. This is done by setting the WDTCNTCL bit in WDTCTL.
The task is often called *petting*, *feeding*, or *kicking the dog*, depending on your attitude toward canines. The bit automatically clears again after WDTCNT has been reset.

The MSP430 is reset if the watchdog counter reaches its limit. Recall from the section "Resets" on page 157 that there are two levels of reset. The watchdog causes a power-up clear, which is the less drastic form. Most registers are reset to default values but some retain their contents, which is vital so that the source of the reset can be determined. The watchdog timer sets the WDTIFG flag in the special function register IFG1. This is cleared by a power-on reset but its value is preserved during a PUC. Thus a program can check this bit to find out whether a reset arose from the watchdog. shows a trivial program to demonstrate the watchdog. I selected the clock from ACLK (WDTSSEL = 1) and the longest period (WDTISx = 00), which gives 1s

with a 32 KHz crystal for ACLK. It is wise to restart any timer whenever its configuration is changed so I also cleared the counter by setting the WDTCNTCL bit. LED1 shows the state of button B1 and LED2 shows WDTIFG. The watchdog is serviced by rewriting the configuration value in a loop while button B1 is held down. If the button is left up for more than 1s the watchdog times out, raises the flag WDTIFG, and resets the device with a PUC.
This is shown by LED2 lighting.

Suppose that the program became stuck in one of the tasks of the paced loop. Interrupts would still be generated periodically and the watchdog would continue to be serviced correctly if this were done in the ISR. On the other hand, the watchdog would expire and cause a reset with the structure in Listing 8.2.

It is possible that the watchdog may time out during the initialization of a program, which is carried out by the startup code before the user's main() function is called. This would happen if it took longer than 32 ms to initialize the RAM, which is possible if a large number of global or static variables are used. In EW430 you can supply a function _low_level_init(), which is called before the RAM is initialized. The watchdog can be stopped or reconfigured here.

Watchdogs vary considerably from one type of microcontroller to another. Some have a set of passwords that must be used in a prescribed order, rather than a single value; a reset occurs if a password is used out of sequence.Windowed watchdogs must be serviced only during a particular part of their period, such as the last quarter; clearing the watchdog earlier than this causes a reset. Some have their own built-in oscillator, which protects them from failure of the main clocks. Many watchdogs are controlled by write-once registers, which means that their configuration cannot be changed after an initial value has been written. This would be a problem in the MSP430, where the watchdog may need to be reconfigured for low-power modes.

The reset caused by the watchdog can be a nuisance during development because a PUC destroys much of the evidence that could help you to detect a problem that caused the watchdog to time out. A solution might be to generate an interrupt rather than a reset by using the watchdog as an interval timer, which is described shortly. The interrupt service routine could copy critical data to somewhere safe, signal a problem by lighting an LED, or simply cause execution to stop on a breakpoint. Nagy [4] has further suggestions.

Failsafe Clock Source for Watchdog Timer:

Newer devices, including the MSP430F2xx family and recent members of the MSP430x4xx, have the enhanced watchdog timer+ (WDT+). This includes fail-safe logic to preserve the watchdog's clock. Suppose that the watchdog is configured to use ACLK and the program enters low-power mode 4 to wait for an external interrupt, as in Listing 7.1. The old watchdog (WDT) stops during LPM4 and resumes counting when the device is awakened. In contrast, WDT+ does not let the device enter LPM4 because that would disable its clock. Therefore it is not possible to use LPM4 with WDT+ active; thewatchdog must first be stopped by setting WDTHOLD. Similarly, it is not possible to use LPM3 if WDT+ is active and gets its clock from SMCLK. If its clock fails, WDT+ switches from ACLK or SMCLK to MCLK and

takes this from the DCO if an external crystal fails. The watchdog interval may change dramatically but there must be serious problems elsewhere if this happens.

Watchdog as an Interval Timer:

The watchdog can be used as an interval timer if its protective function is not desired. Set the WDTTMSEL bit in WDTCTL for interval timer mode. The periods are the same as before and again WDTIFG is set when the timer reaches its limit, but no reset occurs. The counter rolls over and restarts from 0. An interrupt is requested if the WDTIE bit in the special function register IE1 is set. This interrupt is maskable and as usual takes effect only if GIE is also set. The watchdog timer has its own interrupt vector, which is fairly high in priority but not at the top. It is not the same as the reset vector, which is taken if the counter times out in watchdog mode. The WDTIFG flag is automatically cleared when the interrupt is serviced. It can be polled if interrupts are not used. Many applications need a periodic "tick," for which the watchdog timer could be used in interval mode. The disadvantage is the limited selection of periods, but 1s is convenient for a clock. Some of the previous examples that used Timer_A could be rewritten for the watchdog instead and its use is illustrated in the standard sets of code examples from TI

Clock System:

All microcontrollers contain a clock module to drive the CPU and peripherals. The conflicting requirements for clocks in high-performance, low-power microcontrollers were described in the section "Clock Generator" on page 33. Figure 5.8 shows a simplified diagram of the Basic Clock Module+ (BCM+) for the MSP430F2xx family. I concentrate on this because it is the most recent design; I mention the extra features of the MSP430x4xx later. The details vary between devices, even in the same family, and the second crystal oscillator XT2 is often omitted. Recall that the clock module provides three outputs:

1. **Master clock, MCLK** is used by the CPU and a few peripherals.
2. **Sub-system master clock, SMCLK** is distributed to peripherals.
3. **Auxiliary clock, ACLK** is also distributed to peripherals.

Most peripherals can choose either SMCLK, which is often the same as MCLK and in the megahertz range, or ACLK, which is typically much slower and usually 32 KHz. A few peripherals, such as analog-to-digital converters, can also use MCLK and some, such as

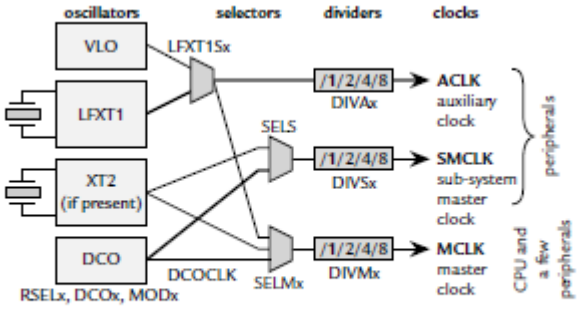


Figure 5.8: Simplified block diagram of the clock module of the MSP430F2xx family, showing some of the more important bits in the peripheral registers that control its operation. Heavy lines indicate the default configuration.

Timers, have their own clock inputs. The frequencies of all three clocks can be divided in the BCM+ as shown in Figure 5.8. For example, you might wish to run the CPU at 8MHz for rapid execution of code and therefore choose *f*MCLK = *f*DCOCLK = 8MHz. On the other hand, it may be more convenient if the peripherals run from a slower clock, in which case you might configure the divider for SMCLK with DIVSx to give *f*SMCLK = *f*DCOCLK/8 = 1MHz. Most peripherals have their own dividers for their clock sources, which gives yet more control. Up to four sources are available for the clock, depending on the family and variant:

Low- or high-frequency crystal oscillator, LFXT1**:** Available in all devices. It is usually used with a low-frequency watch crystal (32 KHz) but can also run with a high-frequency crystal (typically a few MHz) in most devices. An external clock signal can be used instead of a crystal if it is important to synchronize the MSP430 with other devices in the system.

High-frequency crystal oscillator, XT2**:** Similar to LFXT1 except that it is restricted to high frequencies. It is available in only a few devices and LFXT1 (or VLO) is used instead if XT2 is missing.

Internal very low-power, low-frequency oscillator, VLO**:** Available in only the more recent MSP430F2xx devices. It provides an alternative to LFXT1 when the accuracy of a crystal is not needed. Digitally controlled oscillator, DCO**:** Available in all devices and one of the highlights of the MSP430. It is basically a highly controllable *RC* oscillator that starts in less than 1 _s in newer devices. It is not quite true that any of the clocks can come from any of the sources, but the selection can seem bewildering. Fortunately the default configuration described in the section

"Clock Generator" on page 33 is a good starting point. As a reminder, this is as follows:

1. ACLK comes from a low-frequency crystal oscillator at 32 KHz. There is no choice in almost all devices, the exceptions being those with a VLO.
2. Both MCLK and SMCLK are supplied by the DCO with a frequency of around

1 MHz. This is stabilized by the FLL where present. You may wish to raise this frequency provided that *V*CC is high enough to support it.

Most applications do not need MCLK to be highly accurate, so there is rarely a need for a high-frequency crystal. A possible exception is those that use fast, asynchronous value.

We look at the characteristics of the sources in more detail. The Basic Clock Module+ is controlled by four registers, DCOCTL and BCSCTL1–3. In addition there are bits in the special function registers IFG1 and IE2 for reporting faults with the oscillators. Some of the clock signals can be brought out to pins if needed to supply external components or for testing (I use this later for Figure 5.10). This typically needs the pin to be configured for output using P1DIR and the signal switched to the clock from the usual digital port with P1SEL.

Crystal Oscillators, LFXT1 and XT2:

Crystals are used when an accurate, stable frequency is needed:

1. *Accurate* means that the frequency is close to what it says on the package, typically within 1 part in 105.
2. *Stable* means that does not change significantly with time or temperature.

Crystals are cut from carefully grown, high-quality quartz with specific orientations to give them high stability. Traditional crystals oscillated at frequencies of a few MHz but most small microcontrollers use low-frequency watch crystals with a frequency of 32 KHz. These are machined into complicated tuning fork shapes to give the low frequency. Adisadvantage is that their frequency is more sensitive to temperature than high-frequency crystals, but they are designed to be most stable near 25◦C. A change of 10◦C in the temperature causes the frequency to fall by about 4 parts per million (ppm). Detailed specifications are given by the manufacturers, such as Micro Crystal [58].

Stray capacitance of the PCB tracks to the crystal, which must be kept as short as possible. External capacitors are needed with many microcontrollers but they are integrated into the MSP430 for low-frequency crystals. The value is selected with the XCAPx bits in the BCSCTL3 register of the F20xx. The older Basic Clock Module (BCM, without the +) in the MSP430x1xx has a fixed capacitance of 12 pF per pin. For a highly accurate frequency, the drive current of the LFXT1 should also match the specification of the crystal. This sounds rather specialized but systems with the MSP430 may run for 10 years on a single battery and extreme accuracy and stability are needed if the clock is not to drift over this lifetime. The application note *MSP430 LFXT1 Oscillator Accuracy* (slaa225) describes some of the issues.

The oscillator is designed to run at low power and this renders it susceptible to electromagnetic interference. This means that the printed circuit board must be laid out carefully. There is detailed advice in the application note *MSP430 32 KHz Crystal Oscillators* (slaa322). Having said that, the crystal on the Olimex 1121STK is perched up in the air over the MCU itself, which seems to violate the rules but nevertheless works reliably.

It is also possible to use high-frequency crystals (above 400 KHz) with LFXT1 in most devices and some have a second oscillator XT2, which works only at high frequencies.
External capacitors must be used with high-frequency crystals and the module must be configured for a suitable range of frequencies. The module also accepts an external clock signal on XIN.

The stability of crystals is reflected electrically in their high *Q* factor. This means that they oscillate for a long time after being excited, like the ringing tone from a wine glass after it has been tapped. The disadvantage of this is that the oscillator takes an equally long time to reach a stable state, typically around 105 cycles. An oscillator based on a watch crystal therefore takes nearly a second to start and is almost always left running continuously. It might seem better that a 10MHz crystal starts in “only” 10 ms but the

CPU could have used around 105 cycles from the DCO in this time, which is enough to complete many tasks and return to a low-power mode. A software delay loop can be included if it is important to wait for the crystal to stabilize. This can be similar to those in the section “Automatic Control Flashing Light by Software Delay” onThe auxiliary clock ACLK can be derived only from LFXT1 in most devices so ACLK will not be available if there is no crystal. Beware if you are using a TI development kit
Internal Low-Power, Low-Frequency Oscillator, VLO:

The VLO is an internal *RC* oscillator that runs at around 12 KHz and can be used instead of LFXT1 in some newer devices. It saves the cost and space required for a crystal and reduces the current drawn. The data sheet for the F2013 shows that LFXT1 draws about 0.8_A, which falls to 0.5_A with the VLO. (Both are impressively small currents.) Of course this comes at a cost: accuracy and stability. The same data sheet quotes a range of frequency for *f*VLO from 4 to 20 KHz. This looks terrible at first sight but closer reading shows that it covers the full operating range of the device in voltage and temperature.

A variation of 10◦C changes the frequency by about 5% instead of 5 ppm for a crystal. Clearly you would not use the VLO for serious timing. On the other hand, its purpose is often to wake the device periodically to check whether any inputs have changed, and accuracy is not important. ACLK is taken from LFXT1 by default even where the VLO is present. This means that current is wasted in LFXT1 and that pins P2.6 and P2.7 in the F20xx are configured for a crystal. It is usually a good idea to reconfigure the BCM+ to use the VLO and redirect port

Digitally Controlled Oscillator, DCO:

One of the aims of the original design of the MSP430 was that it should be able to start rapidly at full speed from a low-power mode, without waiting a long time for the clock to settle. Early versions of the DCO started in 6_s, which has been reduced to 1–2_s in the MSP430F2xx family. There are no erratic pulses: The output from the DCO starts cleanly after this delay. The stability and accuracy also improved significantly since the early days of the MSP430 and calibration values are now stored for a set of frequencies, giving an accuracy of 1–2%.

The frequency can be controlled through sets of bits in the module's registers at three levels. The numbers are taken from the data sheet for the F2013. The first two levels set the DCO to a constant frequency:

RSELx**:** Selects one of 16 coarse ranges of frequency. The frequencies in each range are larger than those in the one below by a factor of 1.3–1.4. The overall range is about 0.09–20 MHz.
DCOx**:** Selects one of eight steps within each range. Each step increases the frequency by about 8%, giving a factor of 1.7 from bottom to top of the range. Thus the ranges overlap slightly. Figure 5.9 shows the frequency of the DCO in the low part of its range as a function of RSELx and DCOx. There is roughly a constant *ratio* between values so a logarithmic scale would be needed to show the full range clearly. TI does not provide much information about the innards of the DCO. It appears to be based on a type of *RC* oscillator whose frequency is programmed by a current, which is selected by RSEL. This feeds a ring counter whose period is adjusted with DCO. An external resistor *R*osc can be connected in some devices to regulate the current instead of RSEL.

This could improve the stability of the frequency in older devices but is less useful in newer ones. It could also be used to control the frequency by an external analog signal.

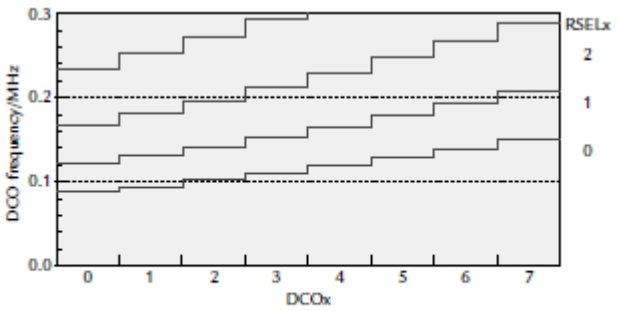


Figure 5.9: Frequency of digitally controlled oscillator in the F20xx. There are eight steps selected with DCOx in the lowest few ranges, which are chosen with RSELx.

Finer control of the *average* frequency is obtained by modulating the frequency of the oscillator between the selected value of DCO and the next step up *(*DCO+1*)*. This needs DCO < 7 of course. Each period of 32 clock cycles contains MOD cycles with the higher frequency given by *(*DCO+1*)* and *(*32−MOD*)* cycles with the lower frequency given by DCO. The average period over these 32 cycles is therefore

$$T_{\text{average}} = \frac{\text{MOD} \times T_{\text{DCO+1}} + (32 - \text{MOD}) \times T_{\text{DCO}}}{32}.$$

The frequencies are given by $f = 1/T$ so the average frequency is

$$f_{\text{average}} = \frac{32 f_{\text{DCO}} f_{\text{DCO+1}}}{\text{MOD} \times f_{\text{DCO}} + (32 - \text{MOD}) \times f_{\text{DCO+1}}}.$$

The DCO does not simply produce MOD pulses of one frequency followed by *(*32−MOD*)* of the other, but mixes them thoroughly. For example, setting MOD = 16 gives an equal number of pulses of the two frequencies and they alternate: Each period given by DCO is followed by one given by *(*DCO+1*)*. This is shown on the oscilloscope in Figure 5.10. The top trace shows the clean square wave seen in a single sweep. There is a longer period *(*DCO = 0*)* in the center with shorter periods *(*DCO = 1*)* on either side.

The lower trace shows repeated sweeps with persistence, as would be seen on an analog oscilloscope. The clock now appears to have jitter on the falling edges because of the two different periods, but the positive edges, which are used for triggering, all overlap. A truly periodic clock is found over the full period of 32 pulses (or fewer in special cases, such as that shown here). The values of RSEL, DCO, and MOD can be changed at any time to alter the frequency. Modulation can be turned off by setting MOD = 0 if a constant period is more important than an accurate, average frequency. The modulator serves little purpose if the DCO runs freely without calibration. There is nearly a factor of 2 between minimum and maximum frequencies given in the data sheet for given values of RSEL and DCO, although this covers the full range of temperature and supply voltage. Therefore it is pointless to specify the frequency better than can be done with RSEL and DCO alone. A possible advantage of modulation is that there is less electromagnetic interference (EMI) from the clock, because the energy is spread over a wider range of frequencies, but this is not a large effect with periods that are only 8% apart. A method to reduce EMI further is described in the application note *Spread-Spectrum*

Clock Source Using an MSP430 (slaa291). Modulation is also used to generate accurate baud rates for asynchronous communication as we shall see in the

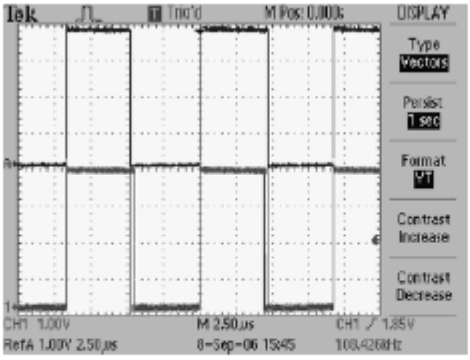


Figure 5.10: Oscilloscope traces of the clock from the DCO in a F2013 to show the effect of modulation. The basic settings are RSEL = 0 and DCO = 0, which give the lowest frequency of about 105 KHz. The modulator is set to MOD = 16, which gives alternating cycles of DCO = 1 with DCO = 0.

The write to BCSCTL1 affects the whole byte, not just the RSELx bits, and clears the other bits. It may therefore be necessary to reconfigure the other functions controlled by this register (XT2OFF, XTS, and DIVAx) but the defaults are usually satisfactory. The calibration is typically accurate within about 2% over the full range of operating conditions and to 0.2% at room temperature with a 3V supply. This is still not as good as a crystal but is impressive compared with a "plain vanilla" *RC* oscillator, whose frequency might vary by ±25%.

The F20xx has calibrated frequencies of 1, 8, 12, and 16 MHz. The values are stored in segment A of the information memory, which is locked by default against programming and erasing. You should not be able to overwrite these values by accident when downloading your program unless you override the settings in the debugger

Control of the Clock Module through the Status Register

The clock module is controlled by 4 bits in the status register as well as its own peripheral registers. This is because of the intimate connection between clocks and low-power modes, which is discussed fully in the section "Low-Power Modes of Operation" on page 198. It is rarely necessary to alter these bits directly because there are intrinsic functions or predefined constants for each low-power mode. All bits are clear in the full-power, active mode. This is the main effect of setting each bit in the MSP430F2xx:

1. **CPUOFF** disables MCLK, which stops the CPU and any peripherals that use MCLK.
2. **SCG1** disables SMCLK and peripherals that use it.
3. **SCG0** disables the DC generator for the DCO (disables the FLL in the MSP430x4xx family).
4. **OSCOFF** disables VLO and LFXT1.

It is not as straightforward as this because the effects of the different bits interact with other. For example, setting only SCG0 and SCG1 does not stop the DCO if it supplies MCLK, because that would paralyze the processor. The DCO stops only if the source of MCLK is also switched to LFXT1 or VLO. This is illustrated by the code example msp430x20x3_1_vlo. Here are the

relevant two lines. You might worry that the first line would stop the CPU but it does not; the DCO remains active while MCLK

Oscillator Faults:

A failure of a clock, MCLK in particular, is crippling. The clock module therefore detects and recovers from the most likely fault, a failure of an oscillator that relies on a vulnerable external crystal. Each oscillator has a flag that is raised to indicate a fault, which also sets the OFIFG bit in the interrupt flag register IFG1. This in turn requests a nonmaskable interrupt if it has been enabled. It also switches MCLK to the DCO if it was not already being used, which ensures that the CPU remains active. The user's software can then take appropriate action.

The flag LFXT1OF in BCSCTL3 is set if a fault is detected in LFXT1. A device with a VLO can switch to this instead of LFXT1 but otherwise there is no other source for ACLK It might be possible to reconfigure peripherals to use SMCLK from the DCO rather than ACLK. Alternatively, the CPU can repeatedly poll LFXT1OF, which is cleared by the hardware if the oscillator recovers. The OFIFG flag is not cleared automatically; this must be done in software. The high-frequency oscillator XT2 has similar protection but a weakness of the MSP430x1xx family is that it cannot detect a failure of LFXT1 with a low-frequency crystal.

OFIFG is also set by a power-on reset, which ensures that MCLK is initially taken from the DCO. Again the flag must be cleared by the user. This should be done repeatedly in a loop because the flag will be set again immediately until the crystal oscillator has reached a stable state. This also means that unused oscillators must not be enabled, or they will appear to have a permanent fault and OFIFG will never clear.

TI code examples demonstrate various configurations of the clocks. For example, msp430x20x3_lpm3_vlo shows how to use the VLO instead of the default LFXT1 in the F20xx and msp430x20x3_LFxtal_nmi shows how to handle an oscillator fault. There are similar examples for other devices.

Frequency-Locked Loop, FLL:

The MSP430x4xx family has the more sophisticated FLL+ clock module. Much of this, such as LFXT1 and XT2, is similar to the MSP430F2xx but the registers and bits have different names. For example, the load capacitance for a low-frequency crystal is controlled by the XCAPxPF bits in the FLL_CTL0 register. There are no dividers for the internal clocks but the external signal from ACLK can be divided.

The main difference is of course the *frequency-locked loop*. This is hardware that aims to lock the frequency of the DCO to that of LFXT1. The DCO in the FLL+ has only five ranges but each covers a factor of about 10 in frequency and is divided into 29 taps. The modulator works in the same way as that in the BCM+. The name makes the FLL sound complicated but its basic mode of operation is simple. It relies on a feedback loop shown in Figure 5.11:

1. The range of the DCO is set with the FN_x bits and modulation may be suppressed by setting SCFQ_M. Its output is at a frequency *f*DCO.

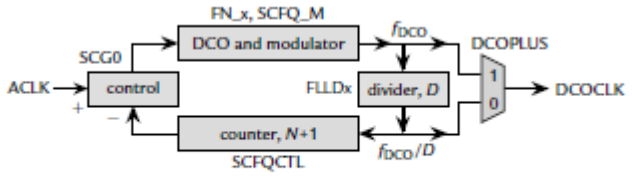


Figure 5.11: Simplified block diagram of the frequency-locked loop in the FLL+ module and the principal bits that configure it.

2. This is divided by a factor *D* specified with the FLLDx bits. This gives a frequency of *f*DCO/*D*.
3. The divided signal is fed into a counter with a period of *(N* +1*)*, where *N* is stored in the lower 7 bits of SCFQCTL.
4. The counter overflows at a frequency of *f*DCO/[*D(N* +1*)*], which is compared with the frequency of ACLK.

The controller adjusts *f*DCO one step up or down with the aim of bringing these frequencies together. Thus the frequency of the DCO itself is given by *f*DCO = *D(N* +1*)f*ACLK when the loop has locked but this is not necessarily the frequency of the output. DCOCLK can be taken either before or after the divider according to the setting of the DCOPLUS bit. When this bit is clear, the divided output is taken and

$$f_{DCOCLK} = (N+1)\,f_{ACLK} \qquad (\text{DCOPLUS} = 0).$$

This does not depend on *D*. When DCOPLUS is set,

$$f_{DCOCLK} = D(N+1)\,f_{ACLK} \qquad (\text{DCOPLUS} = 1).$$

The nomenclature is a little confusing because the divider apparently increases the frequency of the clock, which is the opposite of the usual case. Do not forget the *(N* +1*)* in the multiplier—it is not just *N*. There are 7 bits for *N* in SCFQCTL so the maximum value of *(N* +1*)* is 128 and the maximum value of *f*DCOCLK is 4MHz (binary megahertz to be precise) if *f*ACLK = 32KHz and DCOPLUS = 0. This is well below the maximum frequency at which the CPU can run, which is 16MHz in newer devices. DCOPLUS must be set for higher frequencies. Itshows an example where DCOPLUS is used to raise the frequency of DCOCLK.

After a PUC, the FLL+ is configured for its lowest range of 0.65–6.1 MHz, DCOPLUS is clear, *D* = 2, and *N* = 31. This gives *f*DCOCLK = 32*f*ACLK, which is a binary megahertz (220 Hz) if *f*ACLK = 32 KHz. The DCO itself runs at twice the frequency of DCOCLK because of *D*.

It takes some time for the FLL to lock and a software delay loop can be used to wait for this (and for the crystal to stabilize). The FLL starts at the bottom of its range after a PUC. It may have to reach the top of the range for the desired frequency, which requires up to 32×28 ≈ 900 steps. This is the number of modulator steps times the number of usable taps—the highest tap is not useful because it cannot be modulated. Each step takes one cycle of ACLK, which corresponds to *(N* +1*)* or *D(N* +1*)* cycles of MCLK. This tells us the length of the delay loop needed. A simple loop takes three cycles of MCLK per iteration, as we found in Listing 4.12, so the stabilization loop needs about 300*(N* +1*)* or 300*D(N* +1*)* iterations. This is about 10,000≈

0x2700 iterations using the default settings. You can therefore use the few lines of C that follow to check that the FLL has locked to the default frequency. I also configured the capacitors to suit the TI MSP430FG4618/F2013 Experimenter's Board.

The program attempts to clear the oscillator fault flag OFIFG after the delay and checks that this was successful. If not, the delay loop is repeated. This is simple but the program will never leave the loop if OFIFG cannot be cleared. This could arise if ACLK fails or the
FLL has been incorrectly configured so that the desired frequency lies outside the range of the DCO. The DCO error flag DCOF is set if the DCO's frequency tap is in either its bottom or its top position and this in turn sets OFIFG. DCOF clears when the tap is moved from its extremes.
An FLL is much simpler than its analog equivalent, the phase-locked loop (PLL), and is far quicker to come into lock. Having said that, there appears to be no way of telling whether the FLL has locked—only if it fails so badly that it moves to its top or bottom tap, which sets DCOF.

UNIT- V

UNIT V

Use of Continuous Mode

The continuous mode can be used to generate independent time intervals and output frequencies. Each time an interval is completed, an interrupt is generated. The next time interval is added to the TAxCCRn register in the interrupt service routine. Figure 17-6 shows two separate time intervals, t0 and t1, being added to the capture/compare registers. In this usage, the time interval is controlled by hardware, not software, without impact from interrupt latency. Up to n (where n = 0 to 6), independent time intervals or output frequencies can be generated using capture/compare registers.

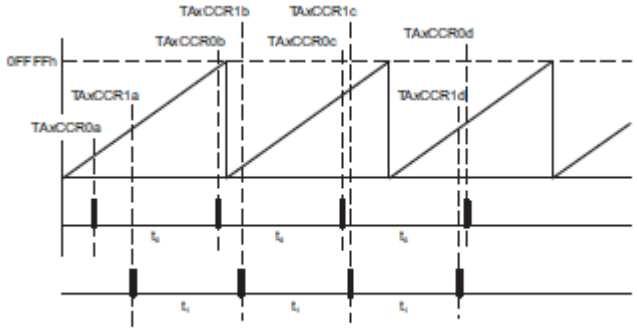


Figure 17-6. Continuous Mode Time Intervals

Time intervals can be produced with other modes as well, where TAxCCR0 is used as the period register. Their handling is more complex since the sum of the old TAxCCRn data and the new period can be higher than the TAxCCR0 value. When the previous TAxCCRn value plus tx is greater than the TAxCCR0 data, the TAxCCR0 value must be subtracted to obtain the correct time interval.

Up/Down Mode

The up/down mode is used if the timer period must be different from 0FFFFh counts, and if symmetrical pulse generation is needed. The timer repeatedly counts up to the value of compare register TAxCCR0 and back down to zero (see Figure 17-7). The period is twice the value in TAxCCR0.

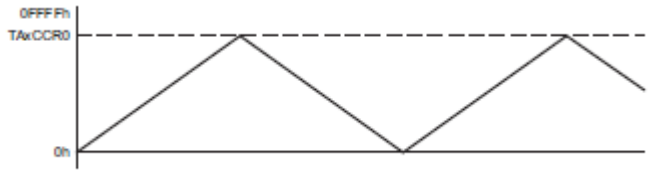


Figure 17-7. Up/Down Mode

The count direction is latched. This allows the timer to be stopped and then restarted in the same direction it was counting before it was stopped. If this is not desired, the TACLR bit must be set to clear the direction. Setting TACLR also clears the TAR value and the clock divider counter logic (the divider setting remains unchanged).

In up/down mode, the TAxCCR0 CCIFG interrupt flag and the TAIFG interrupt flag are set only once during a period, separated by one-half the timer period. The TAxCCR0 CCIFG interrupt flag is set when the timer *counts* from TAxCCR0-1 to TAxCCR0, and TAIFG is set

when the timer completes *counting* down from 0001h to 0000h. Figure 17-8 shows the flag set cycle.

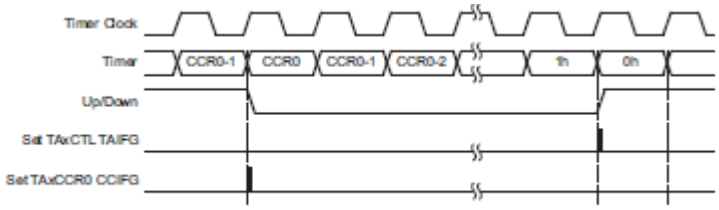


Figure 17-8. Up/Down Mode Flag Setting

Changing Period Register TAxCCR0

When changing TAxCCR0 while the timer is running and counting in the down direction, the timer continues its descent until it reaches zero. The new period takes effect after the counter counts down to zero. When the timer is counting in the up direction, and the new period is greater than or equal to the old period or greater than the current count value, the timer counts up to the new period before counting down. When the timer is counting in the up direction and the new period is less than the current count value, the timer begins counting down. However, one additional count may occur before the counter begins counting down.

Use of Up/Down Mode

The up/down mode supports applications that require dead times between output signals (see section *Timer_A Output Unit*). For example, to avoid overload conditions, two outputs driving an H-bridge must never be in a high state simultaneously. In the example shown in Figure 17-9, the tdead is:

tdead = ttimer × (TAxCCR1 – TAxCCR2)

Where:

tdead = Time during which both outputs need to be inactive

ttimer = Cycle time of the timer clock

TAxCCRn = Content of capture/compare register n

The TAxCCRn registers are not buffered. They update immediately when written to. Therefore, any required dead time is not maintained automatically.

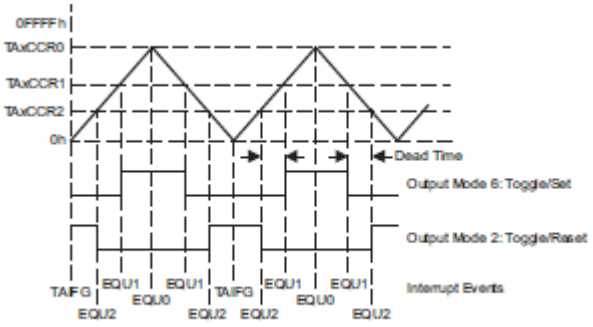


Figure 17-9. Output Unit in Up/Down Mode

Capture/Compare Blocks

Up to seven identical capture/compare blocks, TAxCCRn (where n = 0 to 7), are present in Timer_A. Any of the blocks may be used to capture the timer data or to generate time intervals.

Capture Mode

The capture mode is selected when CAP = 1. Capture mode is used to record time events. It can be used for speed computations or time measurements. The capture inputs CCIxA and CCIxB are connected to external pins or internal signals and are selected with the CCIS bits. The CM bits select the capture edge of the input signal as rising, falling, or both. A capture occurs on the selected edge of the input signal. If a capture occurs:

- The timer value is copied into the TAxCCRn register.
- The interrupt flag CCIFG is set.

The input signal level can be read at any time via the CCI bit. Devices may have different signals connected to CCIxA and CCIxB. See the device-specific data sheet for the connections of these signals.

The capture signal can be asynchronous to the timer clock and cause a race condition. Setting the SCS bit synchronizes the capture with the next timer clock. Setting the SCS bit to synchronize the capture signal with the timer clock is recommended (see Figure 17-10).

Overflow logic is provided in each capture/compare register to indicate if a second capture was performed before the value from the first capture was read. Bit COV is set when this occurs .COV must be reset with software.

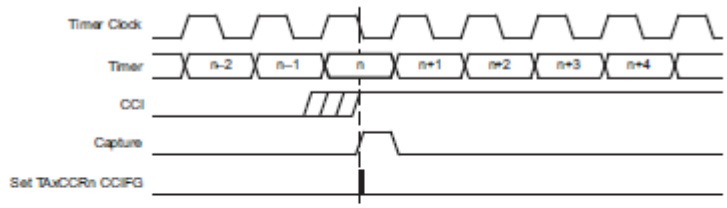


Figure 17-10. Capture Signal (SCS = 1)

NOTE: Changing Capture Inputs

Changing capture inputs while in capture mode may cause unintended capture events. To avoid this scenario, capture inputs should only be changed when capture mode is disabled (CM = {0} or CAP = 0).

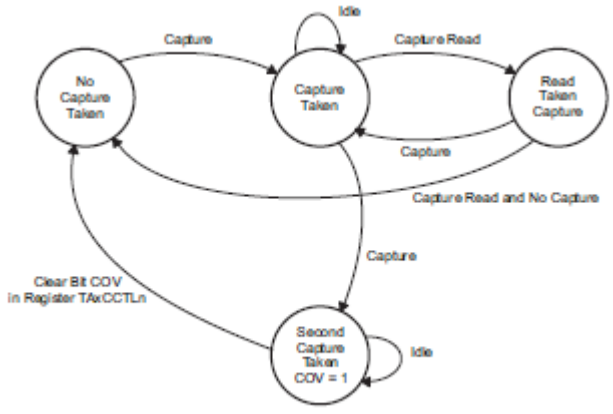


Figure 17-11. Capture Cycle

Capture Initiated by Software

Captures can be initiated by software. The CMx bits can be set for capture on both edges. Software then sets CCIS1 = 1 and toggles bit CCIS0 to switch the capture signal between VCC and GND, initiating a capture each time CCIS0 changes state:

```
MOV #CAP+SCS+CCIS1+CM_3,&TA0CCTL1 ; Setup TA0CCTL1, synch. capture mode
; Event trigger on both edges of capture input.
XOR #CCIS0,&TA0CCTL1 ; TA0CCR1 = TA0R
```

NOTE: Capture Initiated by Software

In general, changing capture inputs while in capture mode may cause unintended capture events. For this scenario, switching the capture input between VCC and GND, disabling the capture mode is not required.

Compare Mode

The compare mode is selected when CAP = 0. The compare mode is used to generate PWM output signals or interrupts at specific time intervals. When TAxR *counts* to the value in a TAxCCRn, where n represents the specific capture/compare register.

- Interrupt flag CCIFG is set.
- Internal signal EQUn = 1.
- EQUn affects the output according to the output mode.
- The input signal CCI is latched into SCCI.

Output Unit

Each capture/compare block contains an output unit. The output unit is used to generate output signals, such as PWM signals. Each output unit has eight operating modes that generate signals based on the EQU0 and EQUn signals.

Output Modes

The output modes are defined by the OUTMOD bits and are described in Table 17-2. The OUTn signal is changed with the rising edge of the timer clock for all modes except mode 0. Output modes 2, 3, 6, and 7 are not useful for output unit 0 because EQUn = EQU0.

Output Example—Timer in Up Mode

The OUTn signal is changed when the timer *counts* up to the TAxCCRn value and rolls from TAxCCR0 to zero, depending on the output mode. An example is shown in Figure 17-12 using TAxCCR0 and TAxCCR1.

Output Example – Timer in Continuous Mode

The OUTn signal is changed when the timer reaches the TAxCCRn and TAxCCR0 values, depending on the output mode. An example is shown in Figure 17-13 using TAxCCR0 and TAxCCR1.

Output Example – Timer in Up/Down Mode

The OUTn signal changes when the timer equals TAxCCRn in both count direction and when the timer equals TAxCCR0, depending on the output mode. An example is shown in Figure 17-14 using TAxCCR0 and TAxCCR2.

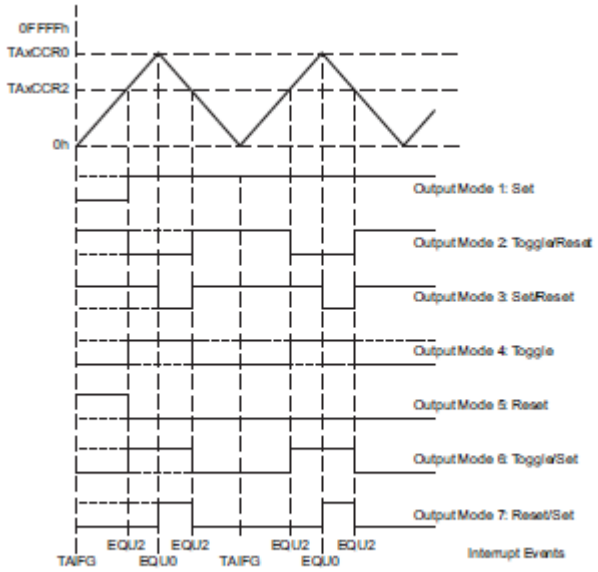


Figure 17-14. Output Example – Timer in Up/Down Mode

Real Time Clock:

RTC Overview:

RTC_A Introduction

The RTC_A module provides a real-time clock and calendar function that can also be configured as a general-purpose counter.

RTC_A features include:

• Configurable for real-time clock with calendar function or general-purpose counter

• Provides seconds, minutes, hours, day of week, day of month, month, and year in real-time clock with calendar function

• Interrupt capability

• Selectable BCD or binary format in real-time clock mode

• Programmable alarms in real-time clock mode

• Calibration logic for time offset correction in real-time clock mode

RTC_A Operation

The RTC_A module can be configured as a real-time clock with calendar function (calendar mode) or as a 32-bit general purpose counter (counter mode) with the RTCMODE bit.

Counter Mode

Counter mode is selected when RTCMODE is reset. In this mode, a 32-bit counter is provided that is directly accessible by software. Switching from calendar mode to counter mode resets the count value (RTCNT1, RTCNT2, RTCNT3, RTCNT4), as well as the prescale counters (RT0PS, RT1PS).

The clock to increment the counter can be sourced from ACLK, SMCLK, or prescaled versions of ACLK or SMCLK. Prescaled versions of ACLK or SMCLK are sourced from the prescale dividers (RT0PS and RT1PS). RT0PS and RT1PS output /2, /4, /8, 16, /32, /64, /128, and /256 versions of ACLK and SMCLK, respectively. The output of RT0PS can be cascaded with RT1PS. The cascaded output can be used as a clock source input to the 32-bit counter.

Four individual 8-bit counters are cascaded to provide the 32-bit counter. This provides 8-bit, 16-bit, 24-bit, or 32-bit overflow intervals of the counter clock. The RTCTEV bits select the respective trigger event. An RTCTEV event can trigger an interrupt by setting the RTCTEVIE bit. Each counter, RTCNT1 through RTCNT4, is individually accessible and may be written to. RT0PS and RT1PS can be configured as two 8-bit counters or cascaded into a single 16-bit counter.

RT0PS and RT1PS can be halted on an individual basis by setting their respective RT0PSHOLD and RT1PSHOLD bits. When RT0PS is cascaded with RT1PS, setting RT0PSHOLD causes both RT0PS and RT1PS to be halted. The 32-bit counter can be halted several ways depending on the configuration. If the 32-bit counter is sourced directly from ACLK or SMCLK, it can be halted by setting RTCHOLD. If it is sourced from the output of RT1PS, it can be halted by setting RT1PSHOLD or RTCHOLD. Finally, if it is sourced from the cascaded outputs of RT0PS and RT1PS, it can be halted by setting RT0PSHOLD, RT1PSHOLD, or RTCHOLD.

Calendar Mode

Calendar mode is selected when RTCMODE is set. In calendar mode, the RTC_A module provides seconds, minutes, hours, day of week, day of month, month, and year in selectable BCD or hexadecimal format. The calendar includes a leap-year algorithm that considers all years evenly divisible by four as leap years. This algorithm is accurate from the year 1901 through 2099.

Real-Time Clock and Prescale Dividers

The prescale dividers, RT0PS and RT1PS, are automatically configured to provide a 1-s clock interval for the RTC_A. RT0PS is sourced from ACLK. ACLK must be set to 32768 Hz (nominal) for proper RTC_A calendar operation. RT1PS is cascaded with the output ACLK/256 of RT0PS. The RTC_A is sourced with the /128 output of RT1PS, thereby providing the required 1-s interval. Switching from counter to calendar mode clears the seconds, minutes, hours, day-of-

week, and year counts and sets day-of-month and month counts to 1. In addition, RT0PS and RT1PS are cleared.

When RTCBCD = 1, BCD format is selected for the calendar registers. The format must be selected before the time is set. Changing the state of RTCBCD clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1. In addition, RT0PS and RT1PS are cleared.

In calendar mode, the RT0SSEL, RT1SSEL, RT0PSDIV, RT1PSDIV, RT0PSHOLD, RT1PSHOLD, and RTCSSEL bits are don't care. Setting RTCHOLD halts the real-time counters and prescale counters, RT0PS and RT1PS.

Real-Time Clock Alarm Function

The RTC_A module provides for a flexible alarm system. There is a single user-programmable alarm that can be programmed based on the settings contained in the alarm registers for minutes, hours, day of week, and day of month. The user-programmable alarm function is only available in the calendar mode of operation.

Each alarm register contains an alarm enable (AE) bit that can be used to enable the respective alarm register. By setting AE bits of the various alarm registers, a variety of alarm events can be generated.

- Example 1: A user wishes to set an alarm every hour at 15 minutes past the hour; that is, at 00:15:00, 01:15:00, 02:15:00, and so on. This is possible by setting RTCAMIN to 15. By setting the AE bit of the RTCAMIN and clearing all other AE bits of the alarm registers, the alarm is enabled. When enabled, the AF is set when the count transitions from 00:14:59 to 00:15:00, 01:14:59 to 01:15:00, 02:14:59 to 02:15:00, etc.
- Example 2: A user wishes to set an alarm every day at 04:00:00. This is possible by setting RTCAHOUR to 4. By setting the AE bit of the RTCHOUR and clearing all other AE bits of the alarm registers, the alarm is enabled. When enabled, the AF is set when the count transitions from 03:59:59 to 04:00:00.
- Example 3: A user wishes to set an alarm for 06:30:00. RTCAHOUR would be set to 6 and RTCAMIN would be set to 30. By setting the AE bits of RTCAHOUR and RTCAMIN, the alarm is enabled. Once enabled, the AF is set when the the time count transitions from 06:29:59 to 06:30:00. In this case, the alarm event occurs every day at 06:30:00.
- Example 4: A user wishes to set an alarm every Tuesday at 06:30:00. RTCADOW would be set to 2, RTCAHOUR would be set to 6 and RTCAMIN would be set to 30. By setting the AE bits of

  RTCADOW, RTCAHOUR and RTCAMIN, the alarm is enabled. Once enabled, the AF is set when the the time count transitions from 06:29:59 to 06:30:00 and the RTCDOW transitions from 1 to 2.
- Example 5: A user wishes to set an alarm the fifth day of each month at 06:30:00. RTCADAY would be set to 5, RTCAHOUR would be set to 6 and RTCAMIN would be set to 30. By setting the AE bits of RTCADAY, RTCAHOUR and RTCAMIN, the alarm

is enabled. Once enabled, the AF is set when the the time count transitions from 06:29:59 to 06:30:00 and the RTCDAY equals 5.

Reading or Writing Real-Time Clock Registers in Calendar Mode

Because the system clock may be asynchronous to the RTC_A clock source, special care must be taken when accessing the real-time clock registers.

In calendar mode, the real-time clock registers are updated once per second. To prevent reading any realtime clock register at the time of an update, which could result in an invalid time being read, a keepout window is provided. The keepout window is centered approximately -128/32768 s around the update transition. The read-only RTCRDY bit is reset during the keepout window period and set outside the keepout the window period. Any read of the clock registers while RTCRDY is reset is considered to be potentially invalid, and the time read should be ignored.

An easy way to safely read the real-time clock registers is to use the RTCRDYIFG interrupt flag. Setting RTCRDYIE enables the RTCRDYIFG interrupt. Once enabled, an interrupt is generated based on the rising edge of the RTCRDY bit, causing the RTCRDYIFG to be set. At this point, the application has nearly a complete second to safely read any or all of the real-time clock registers. This synchronization process prevents reading the time value during transition. The RTCRDYIFG flag is reset automatically when the interrupt is serviced, or can be reset with software. In counter mode, the RTCRDY bit remains reset. RTCRDYIE is a don't care and RTCRDYIFG remains reset.

Real-Time Clock Interrupts

The RTC_A module has five interrupt sources available, each with independent enables and flags.

Real-Time Clock Interrupts in Calendar Mode

In calendar mode, five sources for interrupts are available, namely RT0PSIFG, RT1PSIFG, RTCRDYIFG, RTCTEVIFG, and RTCAIFG. These flags are prioritized and combined to source a single interrupt vector.

The interrupt vector register (RTCIV) is used to determine which flag requested an interrupt. The highest-priority enabled interrupt generates a number in the RTCIV register (see register description).

This number can be evaluated or added to the program counter (PC) to automatically enter the appropriate software routine. Disabled RTC interrupts do not affect the RTCIV value.

Any access, read or write, of the RTCIV register automatically resets the highest-pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt.

In addition, all flags can be cleared via software.

The user-programmable alarm event sources the real-time clock interrupt, RTCAIFG. Setting RTCAIE enables the interrupt. In addition to the user-programmable alarm, the RTC_A module provides for an interval alarm that sources real-time clock interrupt, RTCTEVIFG. The interval alarm can be selected to cause an alarm event when RTCMIN changed or RTCHOUR

changed, every day at midnight (00:00:00) or every day at noon (12:00:00). The event is selectable with the RTCTEV bits. Setting the RTCTEVIE bit enables the interrupt.

The RTCRDY bit sources the real-time clock interrupt, RTCRDYIFG, and is useful in synchronizing the read of time registers with the system clock. Setting the RTCRDYIE bit enables the interrupt. RT0PSIFG can be used to generate interrupt intervals selectable by the RT0IP bits. In calendar mode, RT0PS is sourced with ACLK at 32768 Hz, so intervals of 16384 Hz, 8192 Hz, 4096 Hz, 2048 Hz,
1024 Hz, 512 Hz, 256 Hz, or 128 Hz are possible. Setting the RT0PSIE bit enables the interrupt.
RT1PSIFG can generate interrupt intervals selectable by the RT1IP bits. In calendar mode, RT1PS is sourced with the output of RT0PS, which is 128 Hz (32768/256 Hz). Therefore, intervals of 64 Hz, 32 Hz,
16 Hz, 8 Hz, 4 Hz, 2 Hz, 1 Hz, or 0.5 Hz are possible. Setting the RT1PSIE bit enables the interrupt

Real-Time Clock Interrupts in Counter Mode

In counter mode, three interrupt sources are available: RT0PSIFG, RT1PSIFG, and RTCTEVIFG. RTCAIFG and RTCRDYIFG are cleared. RTCRDYIE and RTCAIE are don't care. RT0PSIFG can be used to generate interrupt intervals selectable by the RT0IP bits. In counter mode,

RT0PS is sourced with ACLK or SMCLK, so divide ratios of /2, /4, /8, /16, /32, /64, /128, and /256 of the respective clock source are possible. Setting the RT0PSIE bit enables the interrupt. RT1PSIFG can be used to generate interrupt intervals selectable by the RT1IP bits. In counter mode, RT1PS is sourced with ACLK, SMCLK, or the output of RT0PS, so divide ratios of /2, /4, /8, /16, /32, /64, /128, and /256 of the respective clock source are possible. Setting the RT1PSIE bit enables the interrupt.

The RTC_A module provides for an interval timer that sources real-time clock interrupt, RTCTEVIFG. The interval timer can be selected to cause an interrupt event when an 8-bit, 16-bit, 24-bit, or 32-bit overflow occurs within the 32-bit counter. The event is selectable with the RTCTEV bits. Setting the RTCTEVIE bit enables the interrupt.

Real-Time Clock Calibration

The RTC_A module has calibration logic that allows for adjusting the crystal frequency in approximately +4-ppm or –2-ppm steps, allowing for higher time keeping accuracy from standard crystals.The RTCCAL bits are used to adjust the frequency. When RTCCALS is set, each RTCCAL LSB causes a ≈ +4-ppm adjustment. When RTCCALS is cleared, each RTCCAL LSB causes a ≈ –2-ppm adjustment. Calibration is available only in calendar mode. In counter mode (RTCMODE = 0), the calibration logic is disabled.

Calibration is accomplished by periodically adjusting the RT1PS counter based on the RTCCALS and RTCCALx settings. In calendar mode, the RT0PS divides the nominial 37268-Hz low-frequency (LF) crystal clock input by 256. A 64-minute period has 32768 cycles/sec × 60 sec/min × 64 min = 125829120 cycles. Therefore a –2-ppm reduction in frequency (down calibration) approximately equates to adding an additional 256 cycles every 125829120 cycles

(256/125829120 = 2.035 ppm). This is accomplished by holding the RT1PS counter for one additional clock of the RT0PS output within a 64-minute period.

Similary, a +4-ppm increase in frequency (up calibration) approximately equates to removing 512 cycles every 125829120 cycle (512/125829120 = 4.069 ppm). This is accomplished by incrementing the RT1PS counter for two additional clocks of the RT0PS output within a 64-minute period. Each RTCCALx calibration bit causes either 256 LF crystal clock cycles to be added every 64 minutes or 512 LF crystal clock cycles to be subtracted every 64 minutes, giving a frequency adjustment of approximately –2 ppm or +4 ppm, respectively.

To calibrate the frequency, the RTCCLK output signal is available at a pin. The RTCCALF bits can be used to select the frequency rate of the RTCCLK output signal, either no signal, 512 Hz, 256 Hz, or 1 Hz.

The basic flow to calibrate the frequency is as follows:

1. Configure the RTCCLK pin.
2. Measure the RTCCLK output signal with an appropriate resolution frequency counter; that is, within the resolution required.
3. Compute the absolute error in ppm: Absolute Error (ppm) = |106 × (fMEASURED fRTCCLK) / fRTCCLK|, where

fRTCCLK is the expected frequency of 512 Hz, 256 Hz, or 1 Hz.

4. Adjust the frequency, by performing the following:

(a) If the frequency is too low, set RTCALS = 1 and apply the appropriate RTCCALx bits, where

RTCCALx = (Absolute Error) / 4.069, rounded to the nearest integer.

(b) If the frequency is too high, clear RTCALS = 0 and apply the appropriate RTCCALx bits, where RTCCALx = (Absolute Error) / 2.035, rounded to the nearest integer.

For example, assume that RTCCLK is output at a frequency of 512 Hz. The measured RTCCLK is 511.9658 Hz. The frequency error is approximately 66.8 ppm low. To increase the frequency by 66.8 ppm,

RTCCALS would be set, and RTCCAL would be set to 16 (66.8/4.069). Similarly, assume that the measured RTCCLK is 512.0125 Hz. The frequency error is approximately 24.4 ppm high. To decrease the frequency by 24.4 ppm, RTCCALS would be cleared, and RTCCAL would be set to 12 (24.4 / 2.035).

The calibration corrects only initial offsets and does not adjust for temperature and aging effects. This can be handled by periodically measuring temperature and using the crystal's charateristic curve to adjust the ppm based on temperature as required. In counter mode (RTCMODE = 0), the calibration logic is disabled.

RTC_A Registers

The RTC_A module registers are listed in and Table 22-1. The base register for the RTC_A module registers can be found in the device-specific data sheet.

PWM (Pulse Width Modulation control):

Introduction

Pulse-width modulation (PWM) is a method by which digital circuit elements can output analog values using only high and low voltage signals. This is achieved by alternating between high and low at the correct intervals to achieve a signal with an equivalent DC voltage to the desired analog value. The fraction of the period in which the signal is high is known as the duty cycle. PWM signals have a variety of uses. This application note will discuss two possible uses, both implemented on the Texas Instruments (TI) DRV8412 motor driver card: 1) driving a brushed DC motor, and 2) communicating an analog value to a test point.

The information presented in this application note is an implementation of PWM on the TI MSP430G2231 microcontroller in C. This is one of the microcontrollers featured in TI's recently released MSP430 LaunchPad (MSP-EXP430G2). More specifically, this application note will cover how to code PWM on the MSP430 LaunchPad, as well as possible uses for the DRV8412 motor driver card Coding PWM through Software

Pulse-width modulation is done on the MSP430 through the timer. A timer on the MSP430 increases a register by one every clock cycle on the MSP430. There are four options for the MSP430 timer, pictured in Table 1 below. MC_0 disables the timer. MC_1 counts from 0x0000 to the value stored in the CCR0 register, resets to 0x0000, and repeats. MC_2 counts from 0x0000 to 0xFFFF, resets to 0x0000, and repeats. MC_3 counts from 0x0000 to the value stored in the CCR0 register, counts down from this value to 0x0000, and repeats. Graphical representations of MC_1 through MC_3 are attached as Figure 1 through Figure 3, which are taken from TI's MSP430x2xx User's Guide.

MC_0	Stop
MC_1	Up to CCR0
MC_2	Continuous Up
MC_3	Up/Down

Table 1: Timer modes

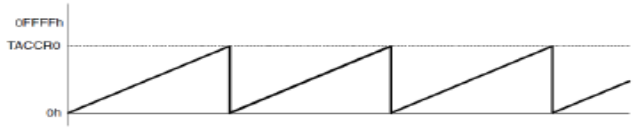


Figure 1: Up to CCR0

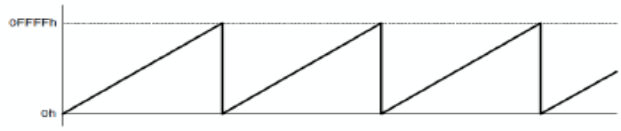


Figure 2: Continuous Up

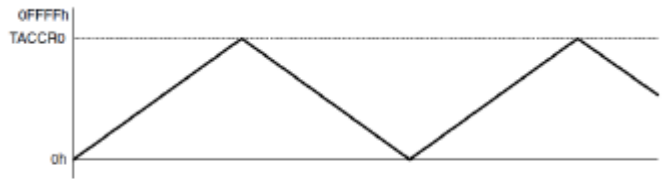


Figure 3: Up/Down Mode

To program the timer in a certain mode, we have to use the control register on the MSP430. For example, if we wish to program Timer A to count 1000 values, we use the TACTL register. TACTL stands for "Timer A Control". Also, since the count begins from 0x0000, to count 1000 values, we would set CCR0 to 999. The code would be as follows:

```
CCR0 = 1000 – 1;
TACTL = MC_1;
```

Now, we can set which pin to output the PWM signal. For example, from pg. 6 of the MSP430G2231 datasheet, pin 4 is "P1.2/TA0.1/A2". We want to use this pin as "TA0.1". To do this, we have to set second bit of both the P1SEL and P1DIR registers accordingly. Generally, to set the Px.y pin, we must set the PxSEL and PxDIR registers accordingly at bit y. A 0 in PxDIR is input; a 1 is output. A 0 in PxSEL means general purpose input/output, while a 1 in PxSEL reflects a special purpose based on the pin. For example, to use the timer, we would set the appropriate bits in PxSEL and PxDIR to 1. To use the ADC converter (A2), we would set PxSEL to 1 and PxDIR to 0. The code to set the pin as a timer is as follows:

```
P1DIR |= BIT2;
P1SEL |= BIT2;
```

We use the |= operator so the bits we do not wish to change remain the same. Next, we must set the PWM mode through the CCTLx register, where x is the port number of the pin. For example, to use "P1.2/TA0.1/A2", we would use CCTL1. The modes are pictured below in Table 2.

OUTMOD_0	PWM Disabled
OUTMOD_1	Set
OUTMOD_2	PWM Toggle/Reset
OUTMOD_3	PWM Set/Reset
OUTMOD_4	Toggle
OUTMOD_5	Reset
OUTMOD_6	PWM Toggle/Reset
OUTMOD_7	PWM Reset/Set

Table 2: PWM modes

If the mode used has only one description, e.g. "Set", then the pin performs this action when it reaches CCR1. So, if the PWM is in OUTMOD_1, the register will go high when the timer reaches CCR1 and stay high until it is reset manually. If the mode used has two descriptions, the first description is performed when CCR1 is reached, and the second is performed when CCR0 is reached. For example, if we are in OUTMOD_3, the register will set at

CCR1 and then reset at CCR0. To use the PWM to achieve a duty cycle of 20%, we would use the following code:

```
CCTL1 = OUTMOD_7;
CCR1 = 200 - 1;
```

Since we are in MC_2, which counts up to CCR0, the pin is set high just before 0x0000, when the timer register equals CCR0. Then, the pin outputs high until CCR1 is reached, when it resets. The timer, in our example code, will count from 0 to 999. From 0 to 199, the register will be high, and from 200 to 999, the register will be low. This achieves a 20% duty cycle.

Results

PWM has many possible applications. TI's DRV8412 motor driver card provides two examples of how PWM can be used. First, the PWM signal can be used to drive a brushed DC motor. In this case, increasing the duty cycle from 0% to 100% increases the speed and torque of the motor. Sensing current feedback through the DRV8412 can also allow for advanced PI control of the brushed DC motor.

Also, the DRV8412 has special PWM DACs. This allows a microcontroller to communicate analog values through digital signals sent to the DRV8412. At first, it may seem that DACs should merely provide an analog version of the pulse-width modulated signal. However, one must remember that a DAC is essentially a low-pass filter, and if the frequency of the pulses is significantly higher than the cut-off frequency of the DAC, the DAC will just output an analog value equal to the average DC voltage. The output of this DAC will be: duty cycle * Vcc. Beyond the DRV8412, there are many other uses for PWM signals. PWM allows us to digitally create analog voltage levels for control functions and power supplies. Also, PWM can create analog signals for arbitrary waveforms, including music and speech. With the ability to code PWM on the MSP430, a broad set of options for analog signal generation through digital sources are available.

ADC:

Introduction

The analog-to-digital converter (ADC) of the MSP430 family can work in two modes: the 12-bit mode or the 14-bit mode. Hardware registers allow easy adaptation to different ADC tasks. The following paragraphs describe the modes and hardware registers

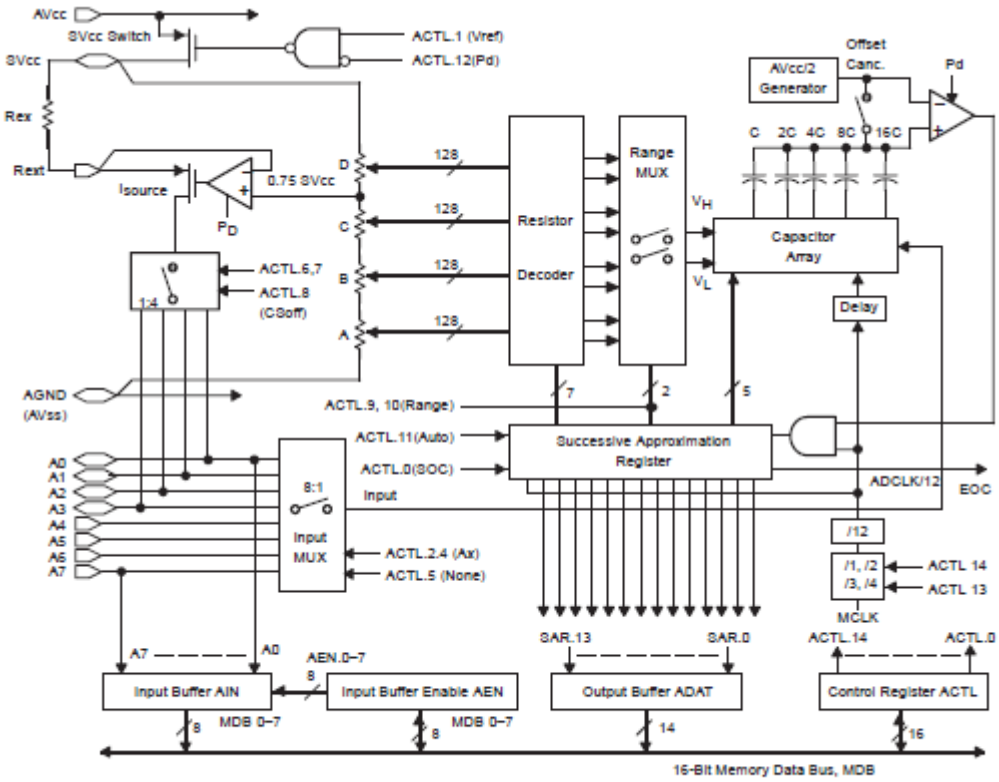


Figure 1. Hardware of the 14-Bit ADC

Characteristics of the 14-Bit ADC

1. Monotonic over the complete ADC range
2. Eight analog inputs; may be switched individually to digital input mode
3. Programmable current source on four analog inputs. Independent of the selected conversion input: current source output and ADC input pins may be different
4. Relative (ratiometric) or absolute measurement possible
5. Sample and hold function with defined sampling time
6. End-of-conversion flag usable with interrupt or polling
7. Last conversion result is stored until start of next conversion
8. Low power consumption and possibility to power down the peripheral
9. Interrupt mode without CPU processing possible
10. Programmable 12-bit or 14-bit resolution
11. Four programmable ranges (one quarter of SVcc each)
12. Fast conversion time
13. Four clock adaptations possible (MCLK, MCLK/2, MCLK/3, MCLK/4)
14. Internal and external reference supply possible
15. Large supply voltage range

ADC Function and Modes

The MSP430 14-bit ADC has two range modes and two measurement modes.

The two range modes are:

1.14-bit mode: The ADC converts the input range from AVss to SVcc. The ADC automatically searches for one of the four ADC ranges (A, B, C, or D) that is appropriate for the input voltage to be measured.

2.12-bit mode: The ADC uses only one of the four ranges (A, B, C, or D). The range is fixed by software. Each range covers a quarter of the voltage at the SVcc terminal. This conversion mode is used if the voltage range of the input signal is known.

The two measurement modes are:

1. Ratiometric mode: A value is measured as a ratio to other values, independent of the actual SVcc voltage.
2. Absolute mode: A value is measured as an absolute value.

Figure 2 shows different methods to connect analog signals to the MSP430 ADC.

The methods shown are valid for the 12-bit and 14-bit conversion modes:

1. Current supply for resistive sensors Rsens1 at analog input A0
2. Voltage supply for resistive sensors Rsens2 at analog input A1
3. Direct connection of input signals Vin at analog input A2
4. Four-wire circuitry with current supply Rsens3 at output A3 and inputs A4 and A5
5. Reference diode with voltage supply Dr1 at analog input A6
6. Reference diode with current supply Dr2 at analog input A7

The calculation formulas for all connection methods shown in Figure 2 are explained in the application report, Application Basics for the MSP430 14-Bit ADC

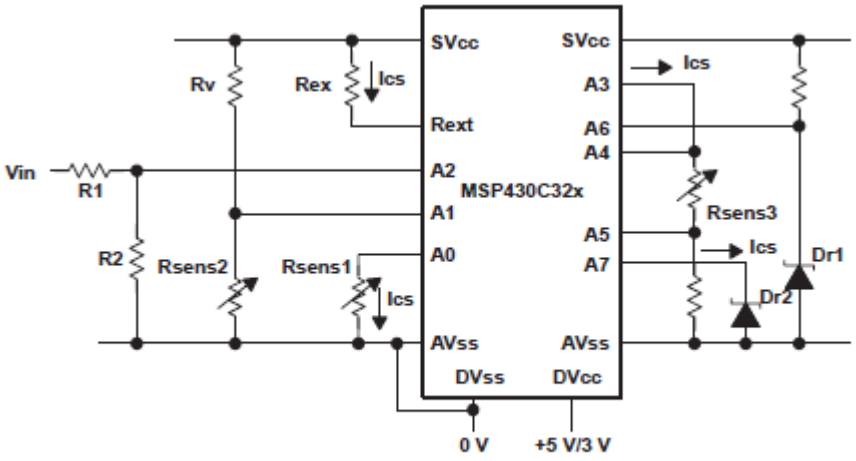


Figure 2. Possible Connections to the Analog-to-Digital Converter

Function of the ADC

See Figures 1, 9, and 12 for this explanation. The full range of the ADC is made by 4128 equal resistors connected between the SVcc pin and the AVss (AGND) pin. Setting the conversion-start (SOC) bit in the ACTL control register activates the ADC clock for a new conversion to begin.

The normal ADC sequence starts with the definition of the next conversion; this is done by setting the bits in the ACTL control register with a single instruction. The power-down (PD)

bit is set to zero; the SOC bit is not changed by this instruction. After a minimum 6-□s delay to allow the ADC hardware to settle, the SOC bit may be set. The ADC clock starts after the SOC bit is set, and a new conversion starts.

□ If the 12-bit mode is selected (RNGAUTO = 0) then a 12-bit conversion starts in a fixed range (A, B, C or D) selected by the bits ACTL.9 to ACTL.10. If the 14-bit mode is selected (RNGAUTO = 1), a sample is taken from the selected input Ax that is used only for the range decision. The found range is fixed afterwards – it delivers the two MSBs of the result – and the conversion continues like the 12-bit conversion. This first decision is made by the block range MUX. This first step fixes the range and therefore the 2 MSBs. Each range contains a block of 128 resistors. To obtain the 12 LSBs, a sample is taken from the selected input Ax and is used for the conversion. The 12-bit conversion consists of two steps: □The seven MSBs are found by a successive approximation using the blockn resistor decode. The sampled input voltage is compared to the voltages generated by the fixed 27 (128) equally weighted resistors connected in series. The resistor whose leg voltages are closest to the sampled input voltage-which means between the two leg voltages—is connected to the capacitor array (see Figure 1).

The five LSBs are found by a successive approximation process using the block capacitor array. The voltage across the selected resistor (the sampled voltage lies between the voltages at the two legs of the resistor) is divided into 25 (32) steps and compared to the sampled voltage. After these three sequences, a 14-bit respective 12-bit result is available in the register ADAT. Figure 3 shows where the result bits of an analog-to-digital conversion come from:

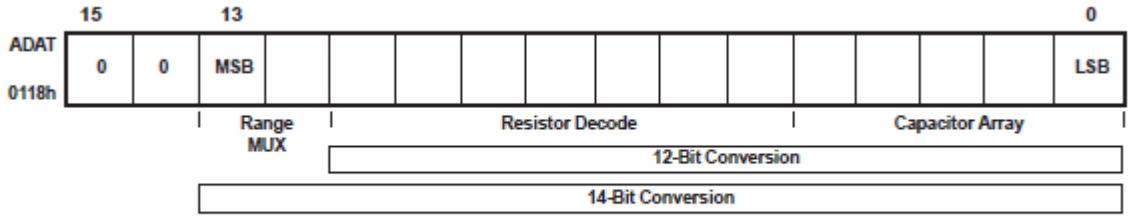


Figure 3. Sources of the Conversion Result

ADC Timing Restrictions

To get the full accuracy for the ADC measurements, some timing restrictions need to be considered: □If the ADCLK frequency is chosen too high, an accurate 14- or 12-bit conversion cannot be assured. This is due to the internal time constants of the sampling analog input and conversion network. The ADC is still functional, but the conversion results show a higher noise level (larger bandwidth of results for the same input signal) with higher conversion frequencies. □If the ADCLK frequency is chosen too low, then an accurate 14- or 12-bit conversion cannot be assured due to charge losses within the capacitor array of the ADC. This remains true even if the input signal is constant during the sampling time.

After the ADC module has been activated by resetting the power-down bit, at least 6 s (power-up time in Figure 9) must elapse before a conversion is started. This is necessary to allow the internal biases to settle. This power-up time is automatically ensured for MCLK frequencies up to 2.5 MHz if the measurement is started the usual way: by separation of the definition and the start of the measurement inside of the subroutine:

MOV #xxx,&ACTL ; Define ADC measurement
CALL #MEASR ; Start measurement with SOC=1, ADC result in ADAT

If higher MCLK frequencies are used, then a delay needs to be inserted between the definition and the start of the measurement. See the source of the MEASR subroutine in section 2.2.2. The number n of additional delay cycles (MCLK cycles) needed is:n □□(6 □s □□MCLK) –15

□If the input voltage changes very fast, then the range sample and the conversion sample may be captured in different ranges. See section 2.2.1 if this cannot be tolerated. For applications like an electricity meter, this doesn't matter: the error occurs as often for the increasing voltage as for the decreasing voltage so the resulting error is zero.

After the start of a conversion, no modification of the ACTL register is allowed until the conversion is complete. Otherwise the ADC result will be invalid. The previously described timing errors lead to spikes in the ADC characteristic: the ADC seems to get caught at certain steps of the ADC. This is not an ADC error; the reasons are violations of the ADC timing restrictions. See Figure 4. The x-axis shows the range A from step 0 to step 4096, the y-axis shows the ADC error (steps).

The ADC always runs at a clock rate set to one twelfth of the selected ADCLK.
The frequency of the ADCLK should be chosen to meet the conversion time defined in the electrical characteristics (see data sheet). The correct frequency for the ADCLK can be selected by two bits (ADCLK) in the control register ACTL. The MCLK clock signal is then divided by a factor of 1, 2, 3, or 4. See Section 3.5.

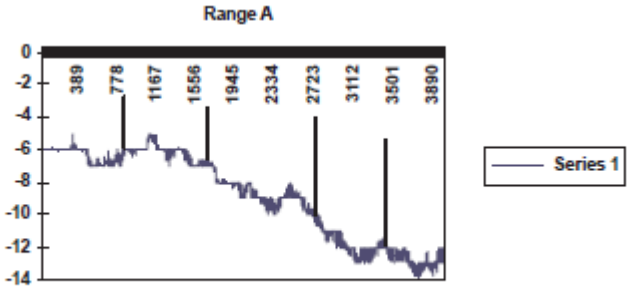


Figure 4. ADC Spikes Due to Violated Timing Restrictions

Sample and Hold

The sampling of the ADC input takes 12 ADCLK cycles; this means the sampling gate is open during this time (12 □s at1 MHz). The sampling time is identical for the range decision sample and the data conversion sample. The input circuitry of an ADC input pin, Ax, can be seen simplified as an RC low pass filter during the sampling period (12/ADCLK): 2 k□□in series with 42 pF. The 42-pF capacitor (the sample-and-hold capacitor) must be charged during the 12 ADCLK cycles to (nearly) the final voltage value to be measured, or to within 2–14 of this value.

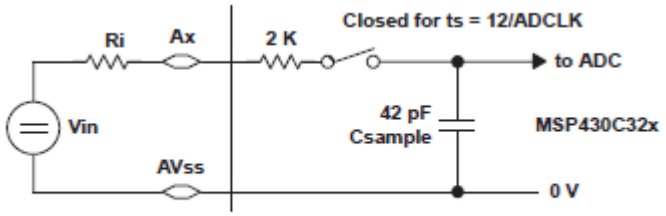


Figure 5. Simplified Input Circuitry for Signal Sampling

The sample time limits the internal resistance, Ri, of the source to be measured:

((Ri _2),42 pF _12

ln_214_ADCLK

Solved for Ri with ADCLK = 1 MHz this results in:

Ri _ 27.4 k

This means, for the full resolution of the ADC, the internal resistance of the input signal must be lower than 27.4 k. If a resolution of n bits is sufficient, then the internal resistance of the ADC input source can be higher:

Absolute and Relative Measurements

The 14-bit ADC hardware allows absolute and relative modes of measurement.

Relative Measurements

As Figure 6 shows, relative measurements use resistances (sensors) that are independent of the supply voltage. This is the typical way to use the ADC. The advantage is independence from the supply voltage; it does not matter if the battery is new (Vcc = 3.6 V) or if it has reached the end of life (Vcc = 2.5 V).

Absolute Measurements

As Figure 7 shows, absolute measurements measure voltages and currents. The reference used for the conversion is the voltage applied to the SVcc terminal, regardless of whether an external reference is used or if SVcc is connected to AVcc internally. An external reference is necessary if the supply voltage AVcc (the normal reference) cannot be used for reference purposes, for example a battery supply.

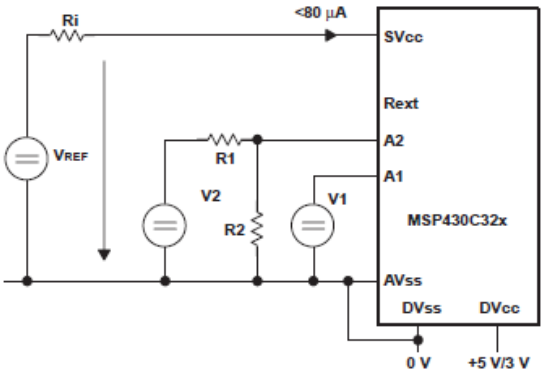


Figure 7. Absolute Measurements Using External Reference Voltage

Using the ADC in 14-Bit Mode

The 14-bit mode is used if the range of the input voltage exceeds one ADC range. The total input signal range is from analog ground (AVss) to the voltage at SVcc (external reference voltage or AVcc)

The dashed boxes at the AVss and SVcc voltage levels indicate the saturation areas of the ADC; the measured results are 0h at AVss and 3FFFh at SVcc. The saturation areas are smaller than 10 ADC steps. The nominal ADC formula for the 14-bit conversion is:

N _VAx

VREF 214_ VAx _N VREF214

Where:

N = 14-bit result of the ADC conversion

VAx = Input voltage at the selected analog input Ax [V]

VREF = Voltage at pin SVcc (external reference or internal AVcc) [V]

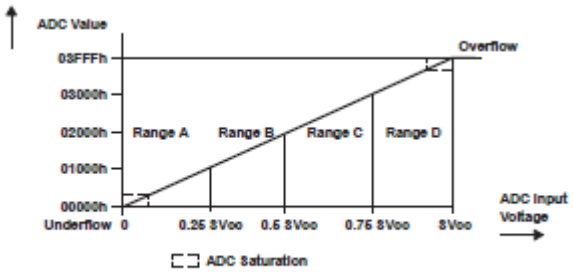


Figure 8. Complete 14-Bit ADC Range

Timing

The two ADCLK bits (ACTL.13 and ACTL.14) in the ACTL control register are used to select the ADCLK frequency best suited for the ADC. The MCLK clock signal can be divided by a factor 1, 2, 3, or 4 to get the best suited ADCLK. Using the autorange mode (RNGAUTO/ACTL.11 = 1) executes a 14-bit conversion. The selected analog input signal at input Ax is sampled twice. The range decision is made after the first sampling of the input signal; the 12-bit conversion is made after the second sampling. Both samplings are 12 ADCLK cycles in length. Altogether the 14-bit conversion takes 132 ADCLK cycles. See Figure 9 for timing details.

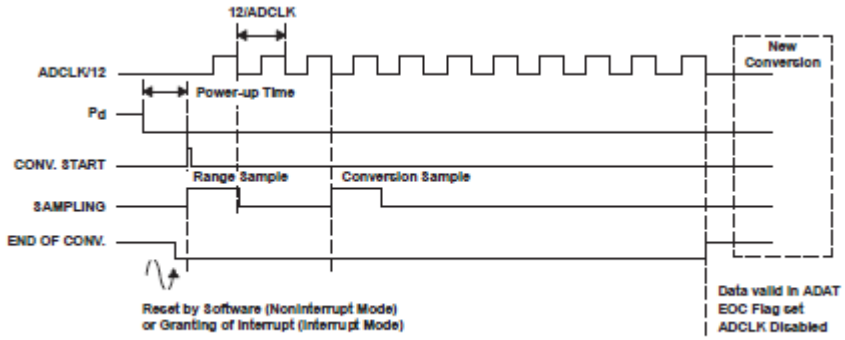


Figure 9. Timing for the 14-bit Analog-to-Digital Conversion

The input signal must be valid and steady during this sampling period to obtain an accurate conversion. It is also recommended that no activity occur during the conversion at analog inputs that are switched to the digital mode. If the input voltage to the ADC changes during the measurement, it is possible for the range decision sample to be taken in a different ADC range than the conversion sample. The result of these conditions is saturated values:

1. Increasing input voltage: nFFFh with range n = 0...2
2. Decreasing input voltage:n000h with range n = 1...3

The saturated result is the best possible result under this circumstance: an analog input that changes from 2FF0h to 3020h during the sampling period delivers the saturated result 2FFFh and not 2000h.

The following software sequence can be used to check the result of an A/D conversion if the two samples (range and conversion) were taken in different ranges. If this is the case, the measurement is repeated.

Software Example

The often-used measurement subroutine MEASR is shown below. It contains all necessary instructions for a measurement that uses polling for the completion check. The subroutine assumes a preset ACTL register; all bits except the SOC bit must be defined before the setting of the SOC bit. The subroutine may be used for 12-bit and 14-bit conversions. Up to an MCLK frequency of 2.5 MHz no additional delays are necessary to ensure the power-up time. ADC measurement subroutine.

```
Call: MOV #xxx,&ACTL ; Define ADC measurement. Pd=0
CALL #MEASR; Measure with ADC
BIS #PD,&ACTL ; Power down the ADC ; ... ; ADC result in ADAT
MEASR BIC.B #ADIFG,&IFG2 ; Clear EOC flag ; Insert delays here (NOPs)
BIS #SOC,&ACTL ; Start measurement
M0 BIT.B #ADIFG,&IFG2 ; Conversion completed?
JZ M0 ; No
RET ; Result in ADAT
```

Using the ADC in 12-Bit Mode

The following mode is used if the range of the input voltage is known. If, for example, a temperature sensor is used whose signal range always fits into one range (for example range B), then the 12-bit mode is the right selection. The measurement time with MCLK = 1 MHz is only 96 s compared with 132 if the auto range mode is used. Figure 10 shows the four ranges compared to the voltage at SVcc. The possible ways to connect sensors to the MSP430 are the same as shown for the 14-bit ADC in Figure 2. This mode should be used only if the signal range is known and the saved 36

ADCLK cycles are a real advantage.

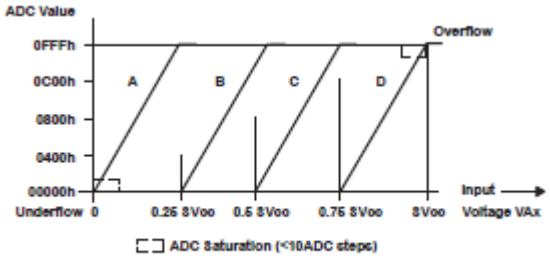


Figure 10. The Four 12-Bit ADC Ranges A to D

NOTE: The ADC results 0000h and 0FFFh mean underflow and
overflow: the voltage at the measured analog input is below or above the limits of the programmed range.

All of the formulas given for the 12-bit mode assume a faultless conversion result N:

$0 < N < 0FFFh$

If underflow or overflows are not checked, erroneous calculation results occur.

Figure 11 shows how any of the four ADC ranges appears to the software:

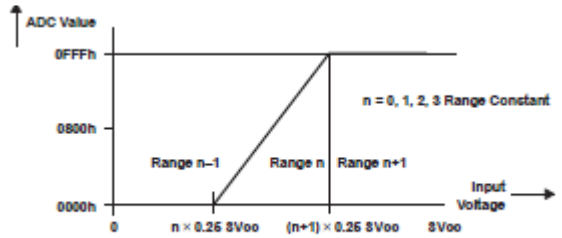


Figure 11. Single 12-Bit ADC Range

Comparator

The comparator compares the analog voltages at the + and – input terminals. If the + terminal is more positive than the – terminal, the comparator output CBOUT is high. The comparator can be switched on or off using control bit CBON. The comparator should be switched off when not in use to reduce current consumption. When the comparator is switched off, CBOUT is always low. The bias current of the comparator is programmable.

Analog Input Switches

The analog input switches connect or disconnect the two comparator input terminals to associated port pins using the CBIPSELx and CBIMSELx bits. The comparator terminal inputs can be controlled individually. The CBIPSELx/CBIMSELx bits allow:

- Application of an external signal to the + and – terminals of the comparator
- Application of an external current source (for example, a resistor) to the + or – terminal of the comparator
- The mapping of both terminals of the internal multiplexer to the outside

Internally, the input switch is constructed as a T-switch to suppress distortion in the signal path The CBEX bit controls the input multiplexer, permuting the input signals of the comparator's + and – terminals. Additionally, when the comparator terminals are permuted, the output signal from the comparator is inverted too. This allows the user to determine or compensate for the comparator input offset voltage.

Port Logic

The Px.y pins associated with a comparator channel are enabled by the CBIPSELx or CBIMSELx bits to disable its digital components while used as comparator input. Only one of the comparator input pins is selected as input to the comparator by the input multiplexer at a time.

Input Short Switch

The CBSHORT bit shorts the Comp_B inputs. This can be used to build a simple sample-and-hold for the Comparator

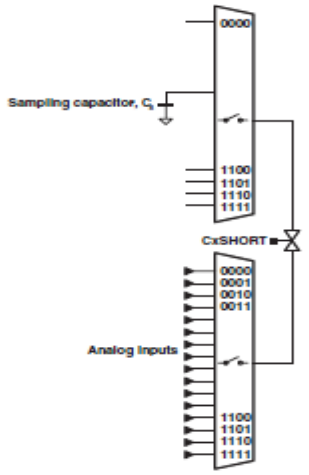


Figure 32-2. Comp_B Sample-And-Hold

input switches in series with the short switch (Ri), and the resistance of the external source (RS). The total internal resistance (RI) is typically in the range of 1 kΩ. The sampling capacitor CS should be greater thanN 100 pF. The time constant, Tau, to charge the sampling capacitor CS can be calculated with the following equation:

Tau = (RI + RS) × CS

Depending on the required accuracy, 3 to 10 Tau should be used as a sampling time. With 3 Tau the sampling capacitor is charged to approximately 95% of the input signals voltage level, with 5 Tau it is charged to more than 99%, and with 10 Tau the sampled voltage is sufficient for 12-bit accuracy.

Output Filter

The output of the comparator can be used with or without internal filtering. When control bit CBF is set, the output is filtered with an on-chip RC filter. The delay of the filter can be adjusted in four different steps. All comparator outputs are oscillating if the voltage difference across the input terminals is small. Internal and external parasitic effects and cross coupling on and between signal lines, power supply lines, and other parts of the system are responsible for this behavior as shown in Figure 32-3. The comparator output oscillation reduces the accuracy and resolution of the comparison result. Selecting the output filter can reduce errors associated with comparator oscillation.

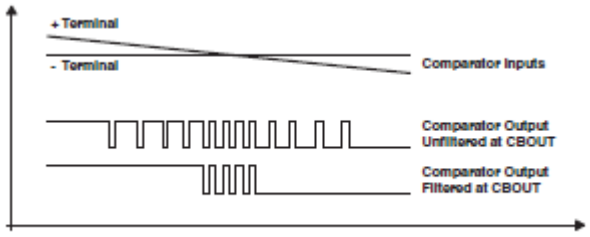
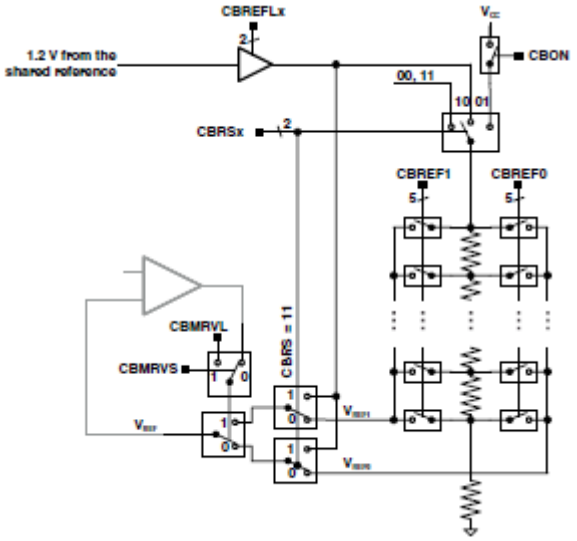


Figure 32-3. RC-Filter Response at the Output of the Comparator

Reference Voltage Generator

The Comp_B reference block diagram is shown



The voltage reference generator is used to generate VREF, which can be applied to either comparator input terminal. The CBREF1x (VREF1) and CBREF0x (VREF0) bits control the output of the voltage generator. The CBRSEL bit selects the comparator terminal to which VREF is applied. If external signals are applied to both comparator input terminals, the internal reference generator should be turned off to reduce current consumption. The voltage reference generator can generate a fraction of the device's VCC or of the voltage reference of the integrated precision voltage reference source. Vref1 is used while CBOUT is 1 and Vref0 is used while CBOUT is 0. This allows the generation of a hysteresis without using external components.

Comp_B, Port Disable Register CBCTL3

The comparator input and output functions are multiplexed with the associated I/O port pins, which are digital CMOS gates. When analog signals are applied to digital CMOS gates, parasitic current can flow from VCC to GND. This parasitic current occurs if the input voltage is near the transition level of the gate. Disabling the port pin buffer eliminates the parasitic current flow and therefore reduces overall current consumption.

The CBPDx bits in the CBCTL3 register, when set, disable the corresponding Px.y input buffer as shown in Figure 32-5. When current consumption is critical, any Px.y pin connected to analog signals should be disabled with their associated CBPDx bits.Selecting an input pin to the comparator multiplexer with the CBIPSEL or CBIMSEL bits automatically disables the input buffer for that pin, regardless of the state of the associated CBPDx bit.

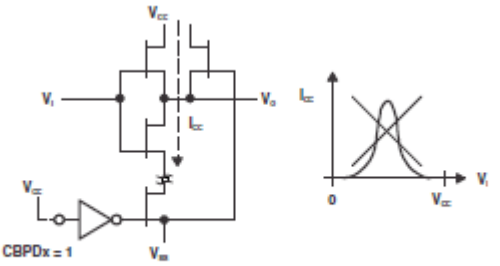


Figure 32-5. Transfer Characteristic and Power Dissipation in a CMOS Inverter/Buffer

Comp_B Interrupts

One interrupt flag and one interrupt vector is associated with the Comp_B. The interrupt flag CBIFG is set on either the rising or falling edge of the comparator output, selected by the CBIES bit. If both the CBIE and the GIE bits are set, then the CBIFG interrupt flag generates an interrupt request.

NOTE: Changing the value of the CBIES bit might set the comparator interrupt flag CBIFG. This can happen even when the comparator is disabled (CBON = 0). It is recommended to clear CBIFG after configuring the comparator for proper interrupt behavior during operation.

Comp_B Used to Measure Resistive Elements

The Comp_B can be optimized to precisely measure resistive elements using single slope analog-todigital conversion. For example, temperature can be converted into digital data using a thermistor, by comparing the thermistor's capacitor discharge time to that of a reference resistor as shown in Figure 32-A reference resister Rref is compared to Rmeas.

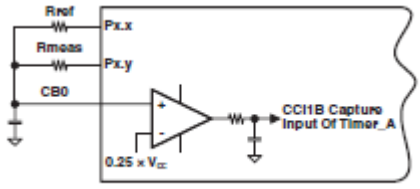


Figure 32-6. Temperature Measurement System

The resources used to calculate the temperature sensed by Rmeas are:

- Two digital I/O pins charge and discharge the capacitor.
- I/O is set to output high (VCC) to charge capacitor, reset to discharge.
- I/O is switched to high-impedance input with CBPDx set when not in use.
- One output charges and discharges the capacitor through Rref.
- One output discharges capacitor through Rmeas.
- The + terminal is connected to the positive terminal of the capacitor.
- The – terminal is connected to a reference level, for example 0.25 × VCC.
- The output filter should be used to minimize switching noise.
- CBOUT is used to gate Timer_A CCI1B, capturing capacitor discharge time.

More than one resistive element can be measured. Additional elements are connected to CB0 with available I/O pins and switched to high impedance when not being measured. The

thermistor measurement is based on a ratiometric conversion principle. The ratio of two capacitor discharge times is calculated as shown in Figure 32-7.

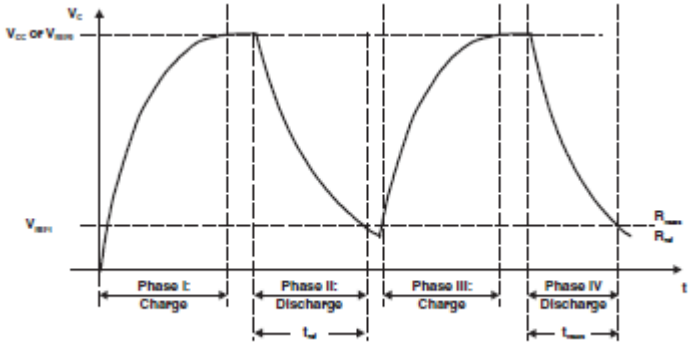


Figure 32-7. Timing for Temperature Measurement Systems

The VCC voltage and the capacitor value should remain constant during the conversion, but are not critical since they cancel in the ratio

$$\frac{N_{rtest}}{N_{ref}} = \frac{-R_{rtest} \times C \times \ln\frac{V_{refi}}{V_{CC}}}{-R_{ref} \times C \times \ln\frac{V_{refi}}{V_{CC}}}$$

$$\frac{N_{rtest}}{N_{ref}} = \frac{R_{rtest}}{R_{ref}}$$

$$R_{rtest} = R_{ref} \times \frac{N_{rtest}}{N_{ref}}$$

Direct Memory Access:

Introduction

The DMA controller transfers data from one address to another, without CPU intervention, across the entire address range. For example, the DMA controller can move data from the ADC conversion memory to RAM. Devices that contain a DMA controller may have up to eight DMA channels available. Therefore, depending on the number of DMA channels available, some features described in this chapter are not applicable to all devices. See the device-specific data sheet for number of channels supported.

Using the DMA controller can increase the throughput of peripheral modules. It can also reduce system power consumption by allowing the CPU to remain in a low-power mode, without having to awaken to move data to or from a peripheral.

DMA controller features include:

- Up to eight independent transfer channels
- Configurable DMA channel priorities
- Requires only two MCLK clock cycles per transfer
- Byte or word and mixed byte and word transfer capability
- Block sizes up to 65535 bytes or words
- Configurable transfer trigger selections
- Selectable-edge or level-triggered transfer
- Four addressing modes
- Single, block, or burst-block transfer modes

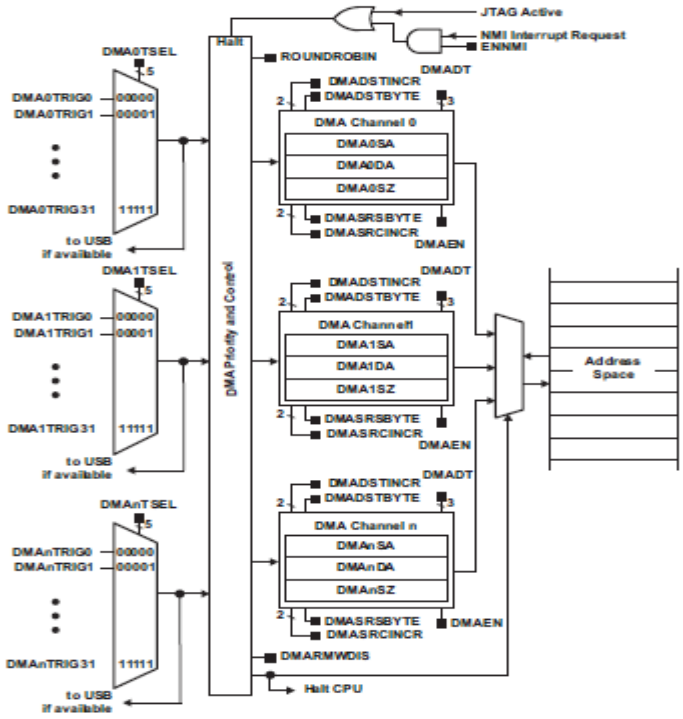


Figure 11-1. DMA Controller Block Diagram

DMA Operation

The DMA controller is configured with user software. The setup and operation of the DMA is discussed in the following sections.

DMA Addressing Modes

The DMA controller has four addressing modes. The addressing mode for each DMA channel is independently configurable. For example, channel 0 may transfer between two fixed addresses, while channel 1 transfers between two blocks of addresses. The addressing modes are shown in Figure 11-2.

The addressing modes are:

- Fixed address to fixed address
- Fixed address to block of addresses
- Block of addresses to fixed address
- Block of addresses to block of addresses

The addressing modes are configured with the DMASRCINCR and DMADSTINCR control bits. The DMASRCINCR bits select if the source address is incremented, decremented, or unchanged after each transfer. The DMADSTINCR bits select if the destination address is incremented, decremented, or unchanged after each transfer.

Transfers may be byte to byte, word to word, byte to word, or word to byte. When transferring word to byte, only the lower byte of the source-word transfers. When transferring byte to word, the upper byte of the destination-word is cleared when the transfer occurs.

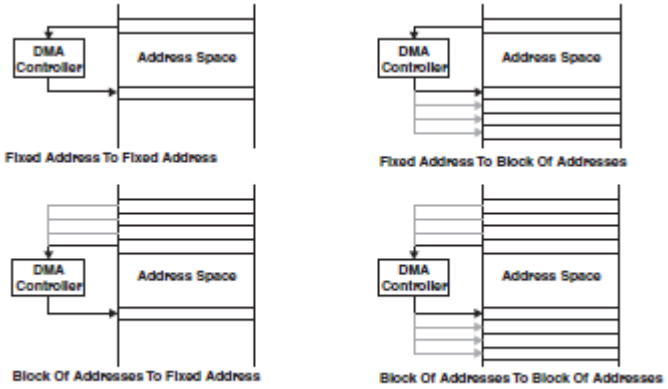


Figure 11-2. DMA Addressing Modes

DMA Transfer Modes

The DMA controller has six transfer modes selected by the DMADT bits as listed in Table 11-1. Each channel is individually configurable for its transfer mode. For example, channel 0 may be configured in single transfer mode, while channel 1 is configured for burst-block transfer mode, and channel 2 operates in repeated block mode. The transfer mode is configured independently from the addressing mode. Any addressing mode can be used with any transfer mode.

Two types of data can be transferred selectable by the DMAxCTL DSTBYTE and SRCBYTE fields. The source and/or destination location can be either byte or word data. It is also possible to transfer byte to byte, word to word, or any combination.

Table 11-1. DMA Transfer Modes

DMADT	Transfer Mode	Description
000	Single transfer	Each transfer requires a trigger. DMAEN is automatically cleared when DMAxSZ transfers have been made.
001	Block transfer	A complete block is transferred with one trigger. DMAEN is automatically cleared at the end of the block transfer.
010, 011	Burst-block transfer	CPU activity is interleaved with a block transfer. DMAEN is automatically cleared at the end of the burst-block transfer.
100	Repeated single transfer	Each transfer requires a trigger. DMAEN remains enabled.
101	Repeated block transfer	A complete block is transferred with one trigger. DMAEN remains enabled.
110, 111	Repeated burst-block transfer	CPU activity is interleaved with a block transfer. DMAEN remains enabled.

Single Transfer

In single transfer mode, each byte/word transfer requires a separate trigger. The single transfer state diagram is shown in Figure 11-3.

The DMAxSZ register is used to define the number of transfers to be made. The DMADSTINCR and DMASRCINCR bits select if the destination address and the source address are incremented or decremented after each transfer. If DMAxSZ = 0, no transfers occur.

The DMAxSA, DMAxDA, and DMAxSZ registers are copied into temporary registers. The temporary values of DMAxSA and DMAxDA are incremented or decremented after each transfer. The DMAxSZ register is decremented after each transfer. When the DMAxSZ register decrements to zero, it is reloaded from its temporary register and the corresponding DMAIFG flag is set. When DMADT = {0}, the DMAEN bit is cleared automatically when DMAxSZ decrements to zero and must be set again for another transfer to occur.

In repeated single transfer mode, the DMA controller remains enabled with DMAEN = 1, and a transfer occurs every time a trigger occurs.

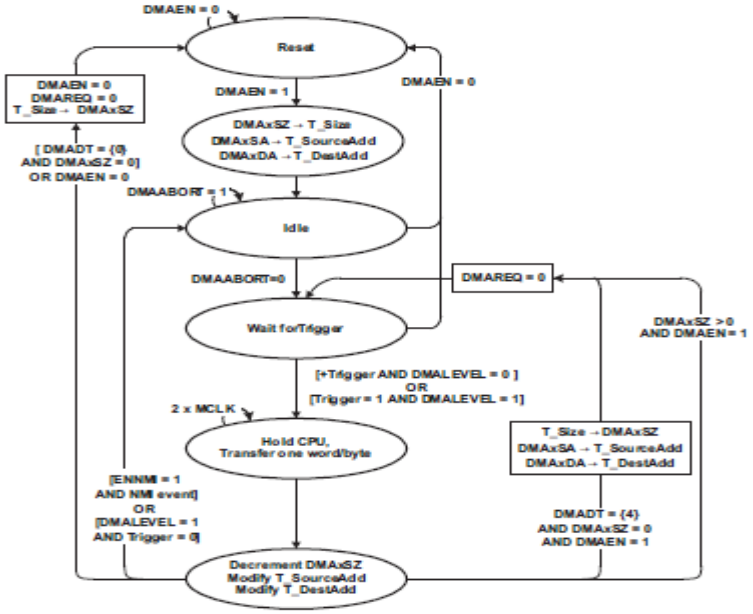


Figure 11-3. DMA Single Transfer State Diagram

Block Transfer

In block transfer mode, a transfer of a complete block of data occurs after one trigger. When DMADT = {1} ,the DMAEN bit is cleared after the completion of the block transfer and must be set again before another block transfer can be triggered. After a block transfer has been triggered, further trigger signals occurring during the block transfer are ignored. The block transfer state diagram is shown in Figure 11-4.

The DMAxSZ register is used to define the size of the block, and the DMADSTINCR and DMASRCINCR bits select if the destination address and the source address are incremented or decremented after each transfer of the block. If DMAxSZ = 0, no transfers occur. The DMAxSA, DMAxDA, and DMAxSZ registers are copied into temporary registers. The temporary values of DMAxSA and DMAxDA are incremented or decremented after each transfer in the block. The DMAxSZ register is decremented after each transfer of the block and shows the number of transfers remaining in the block. When the DMAxSZ register decrements to zero, it is reloaded from its temporary register and the corresponding DMAIFG flag is set.

During a block transfer, the CPU is halted until the complete block has been transferred. The block transfer takes 2 × MCLK × DMAxSZ clock cycles to complete. CPU execution resumes with its previous state after the block transfer is complete.

In repeated block transfer mode, the DMAEN bit remains set after completion of the block transfer. The next trigger after the completion of a repeated block transfer triggers another block transfer

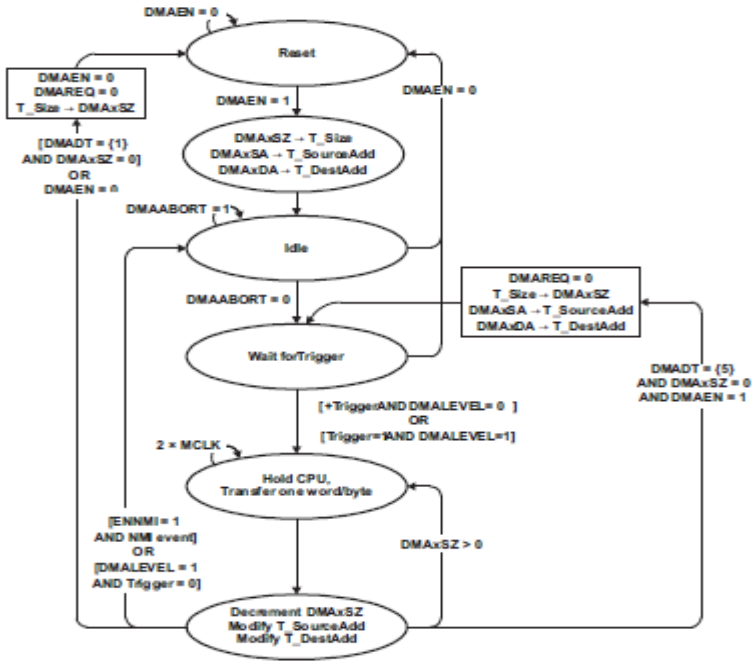


Figure 11-4. DMA Block Transfer State Diagram

Burst-Block Transfer

In burst-block mode, transfers are block transfers with CPU activity interleaved. The CPU executes two MCLK cycles after every four byte/word transfers of the block, resulting in 20% CPU execution capacity. After the burst-block, CPU execution resumes at 100% capacity and the DMAEN bit is cleared. DMAEN must be set again before another burst-block transfer can be triggered. After a burst-block transfer has been triggered, further trigger signals occurring during the burst-block transfer are ignored. The burst-block transfer state diagram is shown in Figure 11-5.

The DMAxSZ register is used to define the size of the block, and the DMADSTINCR and DMASRCINCR bits select if the destination address and the source address are incremented or decremented after each transfer of the block. If DMAxSZ = 0, no transfers occur.

The DMAxSA, DMAxDA, and DMAxSZ registers are copied into temporary registers. The temporary values of DMAxSA and DMAxDA are incremented or decremented after each transfer in the block. The DMAxSZ register is decremented after each transfer of the block and shows the number of transfers remaining in the block. When the DMAxSZ register decrements to zero, it is reloaded from its temporary register and the corresponding DMAIFG flag is set.

In repeated burst-block mode, the DMAEN bit remains set after completion of the burst-block transfer and no further trigger signals are required to initiate another burst-block transfer. Another burst-block transfer begins immediately after completion of a burst-block transfer. In this case, the transfers must be stopped by clearing the DMAEN bit, or by an (non)maskable interrupt (NMI) when ENNMI is set. In repeated burstblock mode the CPU executes at 20% capacity continuously until the repeated burst-block transfer is stopped.

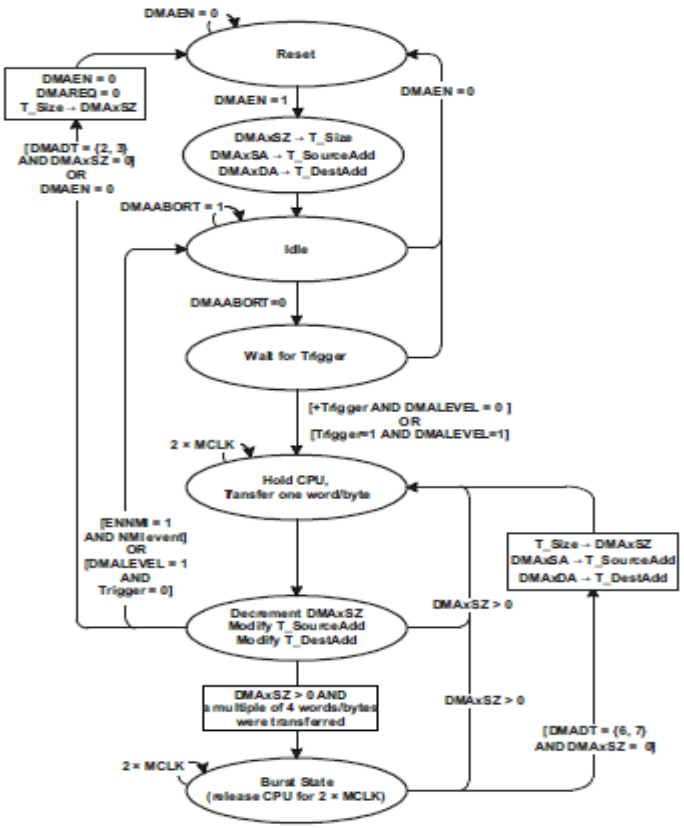


Figure 11-5. DMA Burst-Block Transfer State Diagram

Initiating DMA Transfers

Each DMA channel is independently configured for its trigger source with the DMAxTSEL. The DMAxTSEL bits should be modified only when the DMACTLx DMAEN bit is 0. Otherwise, unpredictable

DMA triggers may occur.Table 11-2 describes the trigger operation for each type of module. See the device-specific data sheet for the list of triggers available, along with their respective DMAxTSEL values. When selecting the trigger, the trigger must not have already occurred, or the transfer does not take place.

NOTE: DMA trigger selection and USB

On devices that contain a USB module, the triggers selection from DMA channels 0, 1, or 2

can be used for the USB time stamp event selection (see the USB module description for further details).

Edge-Sensitive Triggers

When DMALEVEL = 0, edge-sensitive triggers are used, and the rising edge of the trigger signal initiates the transfer. In single-transfer mode, each transfer requires its own trigger. When using block or burstblock modes, only one trigger is required to initiate the block or burst-block transfer.

Level-Sensitive Triggers

When DMALEVEL = 1, level-sensitive triggers are used. For proper operation, level-sensitive triggers can only be used when external trigger DMAE0 is selected as the trigger. DMA transfers are triggered as long as the trigger signal is high and the DMAEN bit remains set.

The trigger signal must remain high for a block or burst-block transfer to complete. If the trigger signal goes low during a block or burst-block transfer, the DMA controller is held in its current state until the trigger goes back high or until the DMA registers are modified by software. If the DMA registers are not modified by software, when the trigger signal goes high again, the transfer resumes from where it was when the trigger signal went low. When DMALEVEL = 1, transfer modes selected when DMADT = {0, 1, 2, 3} are recommended because the DMAEN bit is automatically reset after the configured transfer.

Halting Executing Instructions for DMA Transfers

The DMARMWDIS bit controls when the CPU is halted for DMA transfers. When DMARMWDIS = 0, the CPU is halted immediately and the transfer begins when a trigger is received. In this case, it is possible that CPU read-modify-write operations can be interrupted by a DMA transfer. When DMARMWDIS = 1, the CPU finishes the currently executing read-modify-write operation before the DMA controller halts the CPU and the transfer begins

Stopping DMA Transfers

There are two ways to stop DMA transfers in progress:

• A single, block, or burst-block transfer may be stopped with an NMI, if the ENNMI bit is set in register

DMACTL1.

• A burst-block transfer may be stopped by clearing the DMAEN bit.

DMA Channel Priorities

The default DMA channel priorities are DMA0 through DMA7. If two or three triggers happen simultaneously or are pending, the channel with the highest priority completes its transfer (single, block, or burst-block transfer) first, then the second priority channel, then the third priority channel. Transfers in progress are not halted if a higher-priority channel is triggered. The higher-priority channel waits until the transfer in progress completes before starting.

The DMA channel priorities are configurable with the ROUNDROBIN bit. When the ROUNDROBIN bit is set, the channel that completes a transfer becomes the lowest priority. The *order* of the priority of the channels always stays the same, DMA0-DMA1-DMA2, for example, for three channels. When the ROUNDROBIN bit is cleared, the channel priority returns to the default priority.

DMA Priority	Transfer Occurs	New DMA Priority
DMA0-DMA1-DMA2	DMA1	DMA2-DMA0-DMA1
DMA2-DMA0-DMA1	DMA2	DMA0-DMA1-DMA2
DMA0-DMA1-DMA2	DMA0	DMA1-DMA2-DMA0

DMA Transfer Cycle Time

The DMA controller requires one or two MCLK clock cycles to synchronize before each single transfer or complete block or burst-block transfer. Each byte/word transfer requires two MCLK cycles after synchronization, and one cycle of wait time after the transfer. Because the

DMA controller uses MCLK, the DMA cycle time is dependent on the MSP430 operating mode and clock system setup.

If the MCLK source is active but the CPU is off, the DMA controller uses the MCLK source for each transfer, without reenabling the CPU. If the MCLK source is off, the DMA controller temporarily restarts MCLK, sourced with DCOCLK, for the single transfer or complete block or burst-block transfer. The CPU remains off and after the transfer completes, MCLK is turned off. The maximum DMA cycle time for all operating modes is shown in Table 11-3.

Table 11-3. Maximum Single-Transfer DMA Cycle Time

CPU Operating Mode Clock Source	Maximum DMA Cycle Time
Active mode MCLK = DCOCLK	4 MCLK cycles
Active mode MCLK = LFXT1CLK	4 MCLK cycles
Low-power mode LPM0/1 MCLK = DCOCLK	5 MCLK cycles
Low-power mode LPM3/4 MCLK = DCOCLK	5 MCLK cycles + 5 µs[1]
Low-power mode LPM0/1 MCLK = LFXT1CLK	5 MCLK cycles
Low-power mode LPM3 MCLK = LFXT1CLK	5 MCLK cycles
Low-power mode LPM4 MCLK = LFXT1CLK	5 MCLK cycles + 5 µs[1]

(1) The additional 5 µs are needed to start the DCOCLK. It is the t_{LPM0} parameter in the data sheet.

Using DMA with System Interrupts

DMA transfers are not interruptible by system interrupts. System interrupts remain pending until the completion of the transfer. NMIs can interrupt the DMA controller if the ENNMI bit is set. System interrupt service routines are interrupted by DMA transfers. If an interrupt service routine or other routine must execute with no interruptions, the DMA controller should be disabled prior to executing the routine.

DMA Controller Interrupts

Each DMA channel has its own DMAIFG flag. Each DMAIFG flag is set in any mode when the corresponding DMAxSZ register counts to zero. If the corresponding DMAIE and GIE bits are set, an interrupt request is generated.

All DMAIFG flags are prioritized, with DMA0IFG being the highest, and combined to source a single interrupt vector. The highest-priority enabled interrupt generates a number in the DMAIV register. This number can be evaluated or added to the program counter (PC) to automatically enter the appropriate software routine. Disabled DMA interrupts do not affect the DMAIV value. Any access, read or write, of the DMAIV register automatically resets the highest pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt.

For example, assume that DMA0 has the highest priority. If the DMA0IFG and DMA2IFG flags are set when the interrupt service routine accesses the DMAIV register, DMA0IFG is reset automatically. After the RETI instruction of the interrupt service routine is executed, the DMA2IFG generates another interrupt

Remote Controller of Air Conditioner using MSP430:

Design Features

1. Ultra Low Power with FRAM Technology

2. Infrared Code Sending with Optimized Timer
3. Matrix Key Scan for 14 Buttons
4. Segment LCD

UNIT-V

UNIT V

Universal Serial Communication Interface (USCI) Overview

The USCI modules support multiple serial communication modes. Different USCI modules support different modes. Each different USCI module is named with a different letter. For example, USCI_A is different from USCI_B, etc. If more than one identical USCI module is implemented on one device, those modules are named with incrementing numbers. For example, if one device has two USCI_A modules, they are named USCI_A0 and USCI_A1. See the device-specific data sheet to determine which USCI modules, if any, are implemented on which devices.

USCI_Ax modules support:

- UART mode
- Pulse shaping for IrDA communications
- Automatic baud-rate detection for LIN communications
- SPI mode

USCI_Bx modules support:

- I2C mode
- SPI mode

USCI Introduction – UART Mode

In asynchronous mode, the USCI_Ax modules connect the device to an external system via two external pins, UCAxRXD and UCAxTXD. UART mode is selected when the UCSYNC bit is cleared. UART mode features include:

- 7- or 8-bit data with odd, even, or non-parity
- Independent transmit and receive shift registers
- Separate transmit and receive buffer registers
- LSB-first or MSB-first data transmit and receive
- Built-in idle-line and address-bit communication protocols for multiprocessor systems
- Receiver start-edge detection for auto wake up from LPMx modes (wake up from LPMx.5 is not supported)
- Programmable baud rate with modulation for fractional baud-rate support
- Status flags for error detection and suppression
- Status flags for address detection

USCI Operation – UART Mode

In UART mode, the USCI transmits and receives characters at a bit rate asynchronous to another device. Timing for each character is based on the selected baud rate of the USCI. The transmit and receive functions use the same baud-rate frequency.

USCI Initialization and Reset

- Independent interrupt capability for receive and transmit

Figure 36-1 shows the USCI_Ax when configured for UART mode.

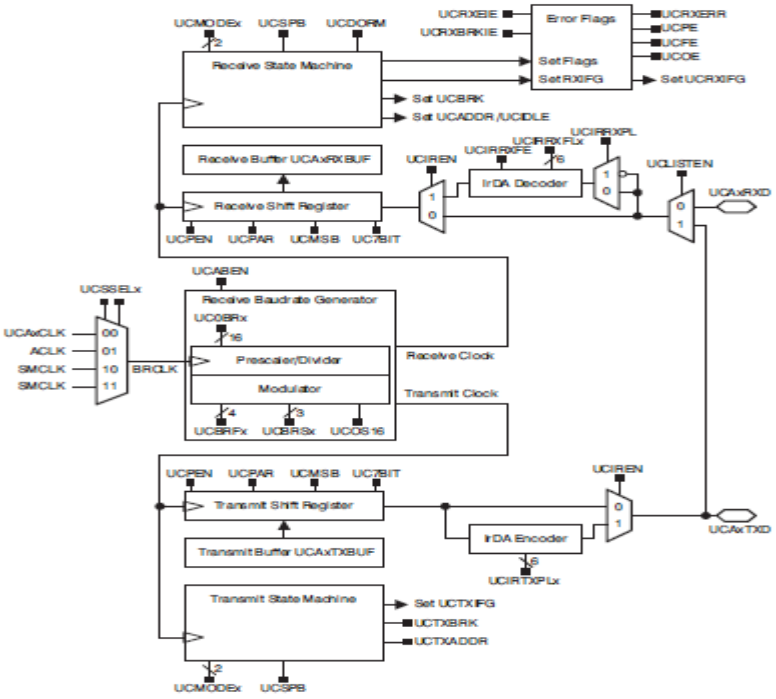


Figure 36-1. USCI_Ax Block Diagram – UART Mode (UCSYNC = 0)

The USCI is reset by a PUC or by setting the UCSWRST bit. After a PUC, the UCSWRST bit is automatically set, keeping the USCI in a reset condition. When set, the UCSWRST bit resets the UCRXIE, UCTXIE, UCRXIFG, UCRXERR, UCBRK, UCPE, UCOE, UCFE, UCSTOE, and UCBTOE bits, and sets the UCTXIFG bit. Clearing UCSWRST releases the USCI for operation. To avoid unpredictable behavior, configure or reconfigure the USCI_A module only when UCSWRST is set.

Character Format

The UART character format (see Figure 36-2) consists of a start bit, seven or eight data bits, an even/odd/no parity bit, an address bit (address-bit mode), and one or two stop bits. The UCMSB bit controls the direction of the transfer and selects LSB or MSB first. LSB first is typically required for UART communication.

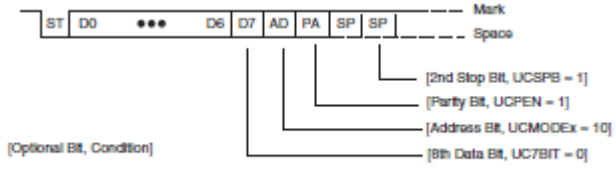


Figure 36-2. Character Format

Asynchronous Communication Format

When two devices communicate asynchronously, no multiprocessor format is required for the protocol. When three or more devices communicate, the USCI supports the idle-line and address-bit multiprocessor communication formats.

Idle-Line Multiprocessor Format

When UCMODEx = 01, the idle-line multiprocessor format is selected. Blocks of data are separated by an idle time on the transmit or receive lines (see Figure 36-3). An idle receive line is detected when ten or more continuous ones (marks) are received after the one or two stop bits of a character. The baud-rate generator is switched off after reception of an idle line until the next start edge is detected. When an idle line is detected, the UCIDLE bit is set.

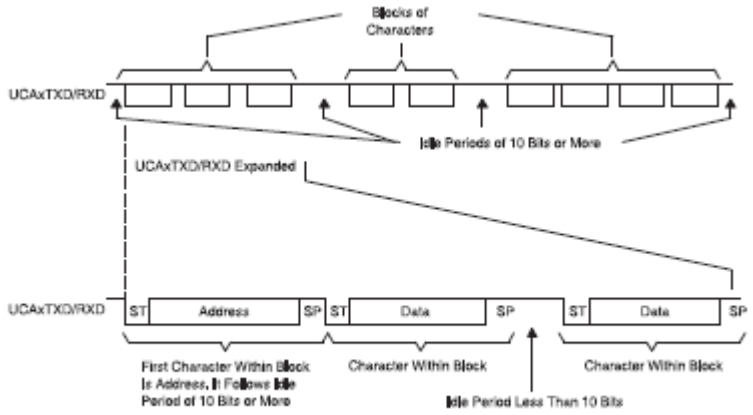


Figure 36-3. Idle-Line Format

The first character received after an idle period is an address character. The UCIDLE bit is used as an address tag for each block of characters. In idle-line multiprocessor format, this bit is set when a received character is an address.

The UCDORM bit is used to control data reception in the idle-line multiprocessor format. When

UCDORM = 1, all non-address characters are assembled but not transferred into the UCAxRXBUF, and interrupts are not generated. When an address character is received, the character is transferred into

UCAxRXBUF, UCRXIFG is set, and any applicable error flag is set when UCRXEIE = 1. When UCRXEIE = 0 and an address character is received but has a framing error or parity error, the character is not transferred into UCAxRXBUF and UCRXIFG is not set.

If an address is received, user software can validate the address and must reset UCDORM to continue receiving data. If UCDORM remains set, only address characters are received. When UCDORM is cleared during the reception of a character, the receive interrupt flag is set after the reception completed. The UCDORM bit is not modified by the USCI hardware automatically.

For address transmission in idle-line multiprocessor format, a precise idle period can be generated by the USCI to generate address character identifiers on UCAxTXD. The double-buffered UCTXADDR flag indicates if the next character loaded into UCAxTXBUF is preceded by an idle line of 11 bits. UCTXADDR is automatically cleared when the start bit is generated.

Transmitting an Idle Frame

The following procedure sends out an idle frame to indicate an address character followed by associated data:

1. Set UCTXADDR, then write the address character to UCAxTXBUF. UCAxTXBUF must be ready for
new data (UCTXIFG = 1). This generates an idle period of exactly 11 bits followed by the address character. UCTXADDR is reset automatically when the address character is transferred from UCAxTXBUF into the shift register.
2. Write desired data characters to UCAxTXBUF. UCAxTXBUF must be ready for new data (UCTXIFG =
1). The data written to UCAxTXBUF is transferred to the shift register and transmitted as soon as the shift register is ready for new data. The idle-line time must not be exceeded between address and data transmission or between data transmissions. Otherwise, the transmitted data is misinterpreted as an address

Address-Bit Multiprocessor Format

When UCMODEx = 10, the address-bit multiprocessor format is selected. Each processed character contains an extra bit used as an address indicator (see Figure 36-4). The first character in a block of characters carries a set address bit that indicates that the character is an address. The USCI UCADDR bit is set when a received character has its address bit set and is transferred to UCAxRXBUF.

The UCDORM bit is used to control data reception in the address-bit multiprocessor format. When UCDORM is set, data characters with address bit = 0 are assembled by the receiver but are not transferred to UCAxRXBUF and no interrupts are generated. When a character containing a set address bit is received, the character is transferred into UCAxRXBUF, UCRXIFG is set, and any applicable error flag is set when UCRXEIE = 1. When UCRXEIE = 0 and a character containing a set address bit is received but has a framing error or parity error, the character is not transferred into UCAxRXBUF and
UCRXIFG is not set.

If an address is received, user software can validate the address and must reset UCDORM to continue receiving data. If UCDORM remains set, only address characters with address bit = 1 are received. The UCDORM bit is not modified by the USCI hardware automatically.

When UCDORM = 0, all received characters set the receive interrupt flag UCRXIFG. If UCDORM is cleared during the reception of a character, the receive interrupt flag is set after the reception is completed.

For address transmission in address-bit multiprocessor mode, the address bit of a character is controlled by the UCTXADDR bit. The value of the UCTXADDR bit is loaded into the address bit of the character transferred from UCAxTXBUF to the transmit shift register. UCTXADDR is automatically cleared when the start bit is generated.

Break Reception and Generation

When UCMODEx = 00, 01, or 10, the receiver detects a break when all data, parity, and stop bits are low, regardless of the parity, address mode, or other character settings. When a break is detected, the UCBRK bit is set. If the break interrupt enable bit (UCBRKIE) is set, the receive interrupt flag UCRXIFG is also set.

In this case, the value in UCAxRXBUF is 0h, because all data bits were zero. To transmit a break, set the UCTXBRK bit, then write 0h to UCAxTXBUF. UCAxTXBUF must be ready for new data (UCTXIFG = 1). This generates a break with all bits low. UCTXBRK is automatically cleared when the start bit is generated.

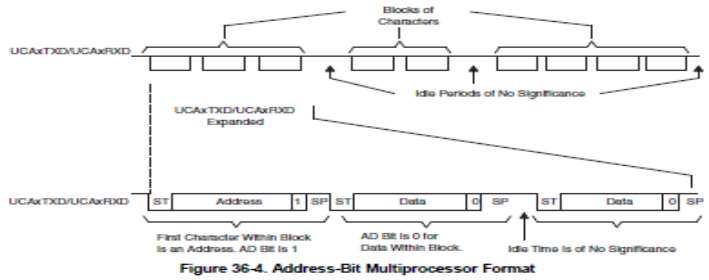


Figure 36-4. Address-Bit Multiprocessor Format

Automatic Baud-Rate Detection

When UCMODEx = 11, UART mode with automatic baud-rate detection is selected. For automatic baudrate detection, a data frame is preceded by a synchronization sequence that consists of a break and a synch field. A break is detected when 11 or more continuous zeros (spaces) are received. If the length of the break exceeds 21 bit times the break timeout error flag UCBTOE is set. The USCI can not transmit data while receiving the break/sync field. The synch field follows the break as shown in Figure 36-5.

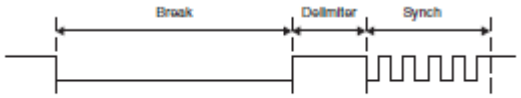


Figure 36-5. Auto Baud-Rate Detection – Break/Synch Sequence

For LIN conformance, the character format should be set to eight data bits, LSB first, no parity, and one stop bit. No address bit is available.

The synch field consists of the data 055h inside a byte field (see Figure 36-6). The synchronization is based on the time measurement between the first falling edge and the last falling edge of the pattern. The transmit baud-rate generator is used for the measurement if automatic baud-rate detection is enabled by setting UCABDEN. Otherwise, the pattern is received but not measured. The result of the measurement is transferred into the baud-rate control registers (UCAxBR0, UCAxBR1, and UCAxMCTL). If the length of the synch field exceeds the measurable time, the synch timeout error flag UCSTOE is set.

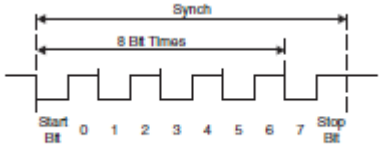


Figure 36-6. Auto Baud-Rate Detection – Synch Field

The UCDORM bit is used to control data reception in this mode. When UCDORM is set, all characters are received but not transferred into the UCAxRXBUF, and interrupts are not generated. When a break/synch field is detected, the UCBRK flag is set. The character following the break/synch field is transferred into UCAxRXBUF and the UCRXIFG interrupt flag is set. Any applicable error flag is also set. If the UCBRKIE bit is set, reception of the break/synch sets the UCRXIFG. The UCBRK bit is reset by user software or by reading the receive buffer UCAxRXBUF.

When a break/synch field is received, user software must reset UCDORM to continue receiving data. If UCDORM remains set, only the character after the next reception of a break/synch field is received. The UCDORM bit is not modified by the USCI hardware automatically. When UCDORM = 0, all received characters set the receive interrupt flag UCRXIFG. If UCDORM is cleared during the reception of a character, the receive interrupt flag is set after the reception is complete.

The counter used to detect the baud rate is limited to 07FFFh (32767) counts. This means the minimum baud rate detectable is 488 baud in oversampling mode and 30 baud in low-frequency mode.

The automatic baud-rate detection mode can be used in a full-duplex communication system with some restrictions. The USCI can not transmit data while receiving the break/sync field and, if a 0h byte with framing error is received, any data transmitted during this time gets corrupted. The latter case can be discovered by checking the received data and the UCFE bit.

Transmitting a Break/Synch Field

The following procedure transmits a break/synch field:

1. Set UCTXBRK with UMODEx = 11.
2. Write 055h to UCAxTXBUF. UCAxTXBUF must be ready for new data (UCTXIFG = 1).

This generates a break field of 13 bits followed by a break delimiter and the synch character. The length of the break delimiter is controlled with the UCDELIMx bits. UCTXBRK is reset automatically when the synch character is transferred from UCAxTXBUF into the shift register.

3. Write desired data characters to UCAxTXBUF. UCAxTXBUF must be ready for new data

(UCTXIFG = 1).

The data written to UCAxTXBUF is transferred to the shift register and transmitted as soon as the shift register is ready for new data.

IrDA Encoding and Decoding

When UCIREN is set, the IrDA encoder and decoder are enabled and provide hardware bit shaping for IrDA communication.

IrDA Encoding The encoder sends a pulse for every zero bit in the transmit bitstream coming from the UART (see Figure 36-7). The pulse duration is defined by UCIRTXPLx bits specifying the number of one-half clock periods of the clock selected by UCIRTXCLK.

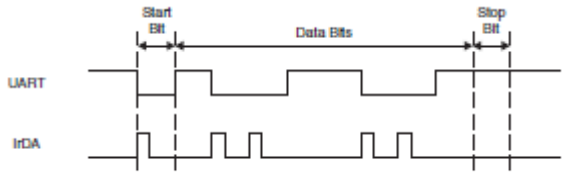


Figure 36-7. UART vs IrDA Data Format

To set the pulse time of 3/16 bit period required by the IrDA standard, the BITCLK16 clock is selected with UCIRTXCLK = 1 ,and the pulse length is set to six one-half clock cycles with UCIRTXPLx = 6 – 1 = 5. When UCIRTXCLK = 0, the pulse length tPULSE is based on BRCLK and is calculated as:

UCIRTXPLx = tPULSE × 2 × fBRCLK – 1 When UCIRTXCLK = 0 ,the prescaler UCBRx must to be set to a value greater or equal to 5.

IrDA Decoding

The decoder detects high pulses when UCIRRXPL = 0. Otherwise, it detects low pulses. In addition to the analog deglitch filter, an additional programmable digital filter stage can be enabled by setting UCIRRXFE. When UCIRRXFE is set, only pulses longer than the programmed filter length are passed. Shorter pulses are discarded. The equation to program the filter length UCIRRXFLx is:

UCIRRXFLx = (tPULSE − tWAKE) × 2 × fBRCLK – 4

Where:

tPULSE = Minimum receive pulse width

tWAKE = Wake time from any low-power mode. Zero when the device is in active mode.

Automatic Error Detection

Glitch suppression prevents the USCI from being accidentally started. Any pulse on UCAxRXD shorter than the deglitch time tt (approximately 150 ns) is ignored (see the device-specific data sheet for parameters). When a low period on UCAxRXD exceeds tt, a majority vote is taken for the start bit. If the majority vote fails to detect a valid start bit, the USCI halts character reception and waits for the next low period on UCAxRXD. The majority vote is also used for each bit in a character to prevent bit errors.

The USCI module automatically detects framing errors, parity errors, overrun errors, and break conditions when receiving characters. The bits UCFE, UCPE, UCOE, and UCBRK are set when their respective condition is detected. When the error flags UCFE, UCPE, or UCOE are set, UCRXERR is also set. The error conditions are described in Table 36-1.

Table 36-1. Receive Error Conditions

Error Condition	Error Flag	Description
Framing error	UCFE	A framing error occurs when a low stop bit is detected. When two stop bits are used, both stop bits are checked for framing error. When a framing error is detected, the UCFE bit is set.
Parity error	UCPE	A parity error is a mismatch between the number of 1s in a character and the value of the parity bit. When an address bit is included in the character, it is included in the parity calculation. When a parity error is detected, the UCPE bit is set.
Receive overrun	UCOE	An overrun error occurs when a character is loaded into UCAxRXBUF before the prior character has been read. When an overrun occurs, the UCOE bit is set.
Break condition	UCBRK	When not using automatic baud-rate detection, a break is detected when all data, parity, and stop bits are low. When a break condition is detected, the UCBRK bit is set. A break condition can also set the interrupt flag UCRXIFG if the break interrupt enable UCBRKIE bit is set.

When UCRXEIE = 0 and a framing error or parity error is detected, no character is received into
UCAxRXBUF. When UCRXEIE = 1, characters are received into UCAxRXBUF and any applicable error bit is set. When any of the UCFE, UCPE, UCOE, UCBRK, or UCRXERR bit is set, the bit remains set until user software resets it or UCAxRXBUF is read. UCOE must be reset by reading UCAxRXBUF. Otherwise, it does not function properly. To detect overflows reliably the following flow is recommended. After a character was received and UCAxRXIFG is set, first read UCAxSTAT to check the error flags including the overflow flag UCOE. Read UCAxRXBUF next. This clears all error flags except UCOE, if UCAxRXBUF was overwritten between the read access to UCAxSTAT and to UCAxRXBUF. Therefore, the UCOE flag should be checked after reading UCAxRXBUF to detect this condition. Note that, in this case, the UCRXERR flag is not set.

USCI Receive Enable

The USCI module is enabled by clearing the UCSWRST bit and the receiver is ready and in an idle state. The receive baud rate generator is in a ready state but is not clocked nor producing any clocks.

The falling edge of the start bit enables the baud rate generator and the UART state machine checks for a valid start bit. If no valid start bit is detected the UART state machine returns to its idle state and the baud rate generator is turned off again. If a valid start bit is detected, a character is received.

When the idle-line multiprocessor mode is selected with UCMODEx = 01 the UART state machine checks for an idle line after receiving a character. If a start bit is detected another character is received. Otherwise the UCIDLE flag is set after 10 ones are received and the UART state machine returns to its idle state and the baud rate generator is turned off.

Receive Data Glitch Suppression

Glitch suppression prevents the USCI from being accidentally started. Any glitch on UCAxRXD shorter than the deglitch time tt (approximately 150 ns) is ignored by the USCI, and further action is initiated as shown in Figure 36-8 (see the device-specific data sheet for parameters).

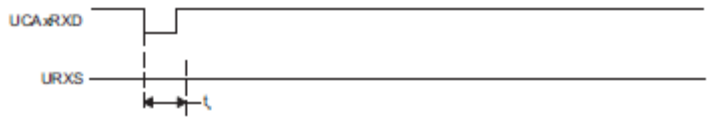


Figure 36-8. Glitch Suppression, USCI Receive Not Started

When a glitch is longer than tt, or a valid start bit occurs on UCAxRXD, the USCI receive operation is started and a majority vote is taken (see Figure 36-9). If the majority vote fails to detect a start bit, the USCI halts character reception.

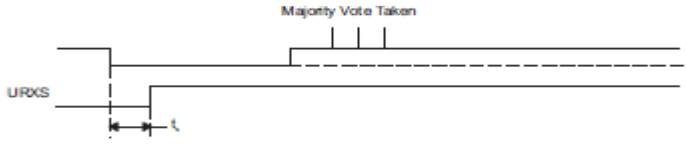


Figure 36-9. Glitch Suppression, USCI Activated

USCI Transmit Enable

The USCI module is enabled by clearing the UCSWRST bit and the transmitter is ready and in an idle state. The transmit baud-rate generator is ready but is not clocked nor producing any clocks. A transmission is initiated by writing data to UCAxTXBUF. When this occurs, the baud-rate generator is enabled, and the data in UCAxTXBUF is moved to the transmit shift register on the next BITCLK after the transmit shift register is empty. UCTXIFG is set when new data can be written into UCAxTXBUF. Transmission continues as long as new data is available in UCAxTXBUF at the end of the previous byte transmission. If new data is not in UCAxTXBUF when the previous byte has transmitted, the transmitter returns to its idle state and the baud-rate generator is turned off.

UART Baud-Rate Generation

The USCI baud-rate generator is capable of producing standard baud rates from nonstandard source frequencies. It provides two modes of operation selected by the UCOS16 bit. The baud-rate is generate using the BRCLK that can be sourced by the external clock UCAxCLK, or the internal clocks ACLK or SMCLK depending on the UCSSELx settings.

Low-Frequency Baud-Rate Generation

The low-frequency mode is selected when UCOS16 = 0. This mode allows generation of baud rates from low frequency clock sources (for example, 9600 baud from a 32768-Hz crystal). By using a lower input frequency, the power consumption of the module is reduced. Using this mode with higher frequencies and higher prescaler settings causes the majority votes to be taken in an increasingly smaller window and, thus, decrease the benefit of the majority vote.

In low-frequency mode, the baud-rate generator uses one prescaler and one modulator to generate bit clock timing. This combination supports fractional divisors for baud-rate generation. In this mode, the maximum USCI baud rate is one-third the UART source clock frequency BRCLK. Timing for each bit is shown in Figure 36-10. For each bit received, a majority vote is taken to determine the bit value. These samples occur at the N/2 – 1/2, N/2, and N/2 + 1/2 BRCLK periods, where N is the number of BRCLKs per BITCLK.

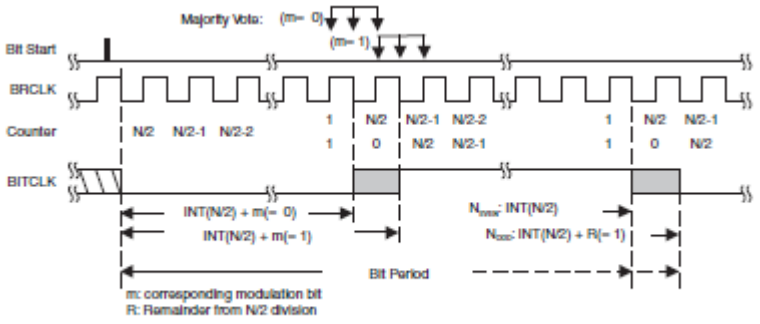


Figure 36-10. BITCLK Baud-Rate Timing With UCOS16 = 0

Modulation is based on the UCBRSx setting (see Table 36-2). A 1 in the table indicates that m = 1 and the corresponding BITCLK period is one BRCLK period longer than a BITCLK riod with m = 0. The modulation wraps around after eight bits but restarts with each new start bit.

Table 36-2. BITCLK Modulation Pattern

UCBRSx	Bit 0 (Start Bit)	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0
3	0	1	0	1	0	1	0	0
4	0	1	0	1	0	1	0	1
5	0	1	1	1	0	1	0	1
6	0	1	1	1	0	1	1	1
7	0	1	1	1	1	1	1	1

Oversampling Baud-Rate Generation

The oversampling mode is selected when UCOS16 = 1. This mode supports sampling a UART bitstream with higher input clock frequencies. This results in majority votes that are always 1/16 of a bit clock period apart. This mode also easily supports IrDA pulses with a 3/16 bit time when the IrDA encoder and decoder are enabled.

This mode uses one prescaler and one modulator to generate the BITCLK16 clock that is 16 times faster than the BITCLK. An additional divider and modulator stage generates BITCLK from BITCLK16. This combination supports fractional divisions of both BITCLK16 and BITCLK for baud-rate generation. In this mode, the maximum USCI baud rate is 1/16 the UART source clock frequency BRCLK. When UCBRx is set to 0 or 1, the first prescaler and modulator stage is bypassed and BRCLK is equal to BITCLK16 – in this case, no modulation for the BITCLK16 is possible and, thus, the UCBRFx bits are ignored. Modulation for BITCLK16 is based on the UCBRFx setting (see Table 36-3). A 1 in the table indicates that the corresponding BITCLK16 period is one BRCLK period longer than the periods m = 0. The modulation restarts with each new bit timing. Modulation for BITCLK is based on the UCBRSx setting (see Table 36-2) as previously described.

Table 36-3. BITCLK16 Modulation Pattern

UCBRFx	Number of BITCLK16 Clocks After Last Falling BITCLK Edge															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
01h	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
02h	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
03h	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1
04h	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1
05h	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1
06h	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1
07h	0	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1
08h	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1
09h	0	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1
0Ah	0	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1
0Bh	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1
0Ch	0	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1
0Dh	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1
0Eh	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
0Fh	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Setting a Baud Rate

For a given BRCLK clock source, the baud rate used determines the required division factor N:

N = fBRCLK/Baudrate The division factor N is often a noninteger value, thus, at least one divider and one modulator stage is used to meet the factor as closely as possible. If N is equal or greater than 16, the oversampling baud-rate generation mode can be chosen by setting UCOS16.

Low-Frequency Baud-Rate Mode Setting

In low-frequency mode, the integer portion of the divisor is realized by the prescaler: UCBRx = INT(N) and the fractional portion is realized by the modulator with the following nominal formula: UCBRSx = round[(N – INT(N)) × 8] Incrementing or decrementing the UCBRSx setting by one count may give a lower maximum bit error for any given bit. To determine if this is the case, a detailed error calculation must be performed for each bit for each UCBRSx setting.

Oversampling Baud-Rate Mode Setting

In the oversampling mode, the prescaler is set to: UCBRx = INT(N/16) and the first stage modulator is set to: UCBRFx = round([(N/16) – INT(N/16)] × 16)

When greater accuracy is required, the UCBRSx modulator can also be implemented with values from 0 to 7. To find the setting that gives the lowest maximum bit error rate for any given bit, a detailed error calculation must be performed for all settings of UCBRSx from 0 to 7 with the initial UCBRFx setting, and with the UCBRFx setting incremented and decremented by one.

Transmit Bit Timing

The timing for each character is the sum of the individual bit timings. Using the modulation features of the baud-rate generator reduces the cumulative bit error. The individual bit error can be calculated using the following steps.

36.3.11.1 Low-Frequency Baud-Rate Mode Bit Timing

In low-frequency mode, calculate the length of bit i $T_{bit,TX}[i]$ based on the UCBRx and UCBRSx settings:

$$T_{bit,TX}[i] = (1/f_{BRCLK})(UCBRx + m_{UCBRSx}[i])$$

Where:

$m_{UCBRSx}[i]$ = Modulation of bit i from Table 36-2

36.3.11.2 Oversampling Baud-Rate Mode Bit Timing

In oversampling baud-rate mode, calculate the length of bit i $T_{bit,TX}[i]$ based on the baud-rate generator UCBRx, UCBRFx and UCBRSx settings:

$$T_{bit,TX}[i] = \frac{1}{f_{BRCLK}}\left((16 + m_{UCBRSx}[i]) \times UCBRx + \sum_{j=0}^{15} m_{UCBRFx}[j]\right)$$

Where:

$\sum_{j=0}^{15} m_{UCBRFx}[j]$ = Sum of ones from the corresponding row in Table 36-3

$m_{UCBRSx}[i]$ = Modulation of bit i from Table 36-2

This results in an end-of-bit time $t_{bit,TX}[i]$ equal to the sum of all previous and the current bit times:

$$T_{bit,TX}[i] = \sum_{j=0}^{i} T_{bit,TX}[j]$$

To calculate bit error, this time is compared to the ideal bit time $t_{bit,ideal,TX}[i]$:

$$t_{bit,ideal,TX}[i] = (1/Baudrate)(i + 1)$$

This results in an error normalized to one ideal bit time (1/baudrate):

$$Error_{TX}[i] = (t_{bit,TX}[i] – t_{bit,ideal,TX}[i]) \times Baudrate \times 100\%$$

Receive Bit Timing

Receive timing error consists of two error sources. The first is the bit-to-bit timing error similar to the transmit bit timing error. The second is the error between a start edge occurring and the start edge being accepted by the USCI module. Figure 36-11 shows the asynchronous timing errors between data on the UCAxRXD pin and the internal baud-rate clock. This results in an additional synchronization error. The synchronization error tSYNC is between –0.5 BRCLKs and +0.5 RCLKs, independent of the selected baudrate generation mode.

Using the USCI Module in UART Mode With Low-Power Modes

The USCI module provides automatic clock activation for use with low-power modes. When the USCI clock source is inactive because the device is in a low-power mode, the USCI module automatically activates it when needed, regardless of the control-bit settings for the clock source. The clock remains active until the USCI module returns to its idle condition. After the USCI module returns to the idle condition, control of the clock source reverts to the settings of its control bits.

USCI Interrupts in UART Mode

The USCI has only one interrupt vector that is shared for transmission and for reception. USCI_Ax and USC_Bx do not share the same interrupt vector.

UART Transmit Interrupt Operation

The UCTXIFG interrupt flag is set by the transmitter to indicate that UCAxTXBUF is ready to accept another character. An interrupt request is generated if UCTXIE and GIE are also set. UCTXIFG is automatically reset if a character is written to UCAxTXBUF. UCTXIFG is set after a PUC or when UCSWRST = 1. UCTXIE is reset after a PUC or when UCSWRST = 1.

UART Receive Interrupt Operation

The UCRXIFG interrupt flag is set each time a character is received and loaded into UCAxRXBUF. An interrupt request is generated if UCRXIE and GIE are also set. UCRXIFG and UCRXIE are reset by a system reset PUC signal or when UCSWRST = 1. UCRXIFG is automatically reset when UCAxRXBUF is read.

Additional interrupt control features include:

- When UCAxRXEIE = 0, erroneous characters do not set UCRXIFG.
- When UCDORM = 1, nonaddress characters do not set UCRXIFG in multiprocessor modes. In plain

  UART mode, no characters are set UCRXIFG.
- When UCBRKIE = 1, a break condition sets the UCBRK bit and the UCRXIFG flag.

UCAxIV, Interrupt Vector Generator

The USCI interrupt flags are prioritized and combined to source a single interrupt vector. The interrupt vector register UCAxIV is used to determine which flag requested an interrupt. The highest-priority enabled interrupt generates a number in the UCAxIV register that can be evaluated or added to the program counter to automatically enter the appropriate software

routine. Disabled interrupts do not affect the UCAxIV value. Any access, read or write, of the UCAxIV register automatically resets the highest-pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt.

USB Introduction

The features of the USB module include:

• Fully compliant with the USB 2.0 full-speed specification
  – Full-speed device (12 Mbps) with integrated USB transceiver (PHY)
  – Up to eight input and eight output endpoints
  – Supports control, interrupt, and bulk transfers
  – Supports USB suspend, resume, and remote wakeup
• A power supply system independent from the PMM system
  – Integrated 3.3-V LDO regulator with sufficient output to power entire MSP430 and system circuitry
  from 5-V VBUS
  – Integrated 1.8-V LDO regulator for PHY and PLL
  – Easily used in either bus-powered or self-powered operation
  – Current-limiting capability on 3.3-V LDO output
  – Autonomous power-up of device on arrival of USB power possible (low or no battery condition)
• Internal 48-MHz USB clock
  – Integrated programmable PLL
  – Highly-flexible input clock frequencies for use with lowest-cost crystals
• 1904 bytes of dedicated USB buffer space for endpoints, with fully configurable size to a granularity of
eight bytes
• Timestamp generator with 62.5-ns resolution
• When USB is disabled
  – Buffer space is mapped into general RAM, providing additional 2KB to the system
  – USB interface pins become high-current general purpose I/O pins

NOTE: Use of the word *device*

The word *device* is used throughout the chapter. This word can mean one of two things, depending on the context. In a USB context, it means what the USB specification refers to as a device, function, or peripheral; that is, a piece of equipment that can be attached to a USB host or hub. In a semiconductor context, it refers to an integrated circuit such as then MSP430. To avoid confusion, the term *USB device* in this document refers to the USB-context meaning of the word. The word *device* by itself refers to silicon devices such as the MSP430.

Figure 42-1 shows a block diagram of the USB module.

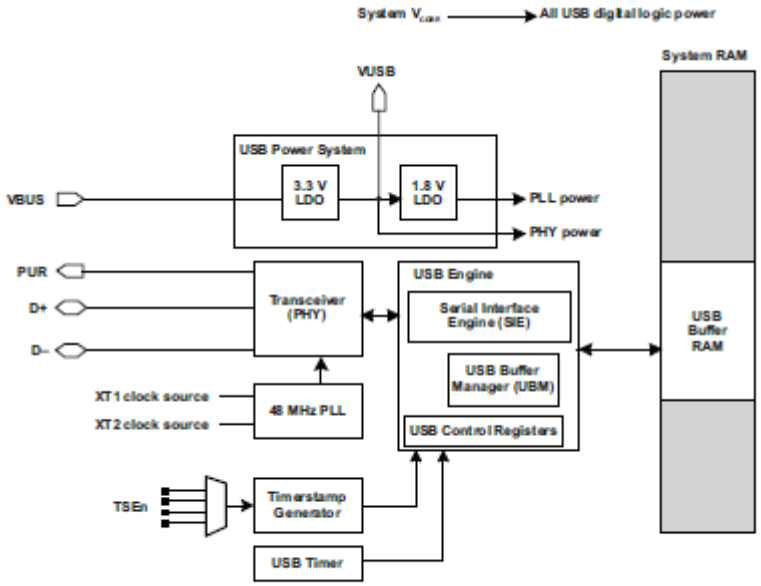


Figure 42-1. USB Block Diagram

USB Operation

The USB module is a comprehensive full-speed USB device compliant with the USB 2.0 specification. The USB engine coordinates all USB-related traffic. It consists of the USB SIE (serial interface engine) and USB Buffer Manager (UBM). All traffic received on the USB receive path is de-serialized and placed into receive buffers in the USB buffer RAM. Data in the buffer RAM marked 'ready to be sent' are serialized into packets and sent to the USB host.

The USB engine requires an accurate 48-MHz clock to sample the incoming data stream. This is generated by a PLL that is fed from one of the system oscillators (XT1 or XT2). A crystal of 4 MHz or greater is required. In addition to crystal operation, the crystal bypass mode can also be used to supply the clock required by the PLL. The PLL is very flexible and can adapt to a wide range of crystal and input frequencies, allowing for cost-effective clock designs.

NOTE: The reference clock to the PLL depends on the device configuration. On devices that contain the optional XT2, the reference clock to the PLL is XT2CLK, regardless of whether or not

XT1 is available. If the device has only XT1, then the reference is XT1CLK. See the devicespecific data sheet for clock sources available.

NOTE: The USB module only supports active operation during power modes AM through LPM1.

The USB buffer memory is where data is exchanged between the USB interface and the application software. It is also where the usage of endpoints 1 to 7 are defined. This buffer memory is implemented such that it can be easily accessed like RAM by the CPU or DMA while USB module is not in suspend condition.

USB Transceiver (PHY)

The physical layer interface (USB transceiver) is a differential line driver directly powered from VUSB (3.3 V). The line driver is connected to the DP and DM pins, which form the signaling mechanism of the USB interface. When the PUSEL bit is set, DP and DM are configured to function as USB drivers controlled by the USB core logic. When the bit is cleared, these two pins become "Port U", which is a pair of high-current general purpose I/O pins. In this case, the pins are controlled by the Port U control registers. Port U is powered from the VUSB rail, separate from the main device DVCC. If these pins are to be used, whether for USB or general purpose use, it is necessary that VUSB be properly powered from either the internal regulators or an external source.

D+ Pullup Via PUR Pin

When a full-speed USB device is attached to a USB host, it must pull up the D+ line (DP pin) for the host to recognize its presence. The MSP430 USB module implements this with a software-controlled pin that activates a pullup resistor. The bit that controls this function is PUR_EN. If software control is not desired, the pullup can be connected directly to VUSB.

Shorts on Damaged Cables and Clamping

USB devices must tolerate connection to a cable that is damaged, such that it has developed shorts on either ground or VBUS. The device should not become damaged by this event, either electrically or physically. To this end, the MSP430 USB power system features a current limitation mechanism that limits the available transceiver current in the event of a short to ground. The transceiver interface itself therefore does not need a current limiting function. Note that if VUSB is to be powered from a source other than the integrated regulator, the absence of current-limiting in the transceiver means that the external power source must itself be tolerant of this same shorting event, through its own means of current limiting.

Port U Control

When PUSEL is cleared, the Port U pins (PU.0 and PU.1) function as general-purpose, high-current I/O pins. These pins can only be configured together as either both inputs or both outputs. Port U is supplied by the VUSB rail. If the 3.3-V LDO is not being used in the system (disabled), the VUSB pin can be supplied externally.

PUOPE controls the enable of both outputs residing on the Port U pins. Setting PUIPE =1 causes both input buffers to be enabled. When Port U outputs are enabled (PUOPE = 1), the PUIN0 and PUIN1 pins mirror what is present on the outputs assuming PUIPE = 1. To use the Port U pins as inputs, the outputs should be disabled by setting PUOPE = 0, and enabling the input buffers by setting PUIPE = 1. Once configured as inputs (PUIPE = 1), the PUIN0 and PUIN1 bits can be read to determine the respective input values.

When PUOPE is set, both Port U pins function as outputs, controlled by PUOUT0 and PUOUT1. When driven high, they use the VUSB rail, and they are capable of a drive current higher than other I/O pins on the device. See the device-specific datasheet for parameters. By

default, PUOPE and PUIPE are cleared. PU.0 and PU.1 are high-impedance (input buffers are disabled and outputs are disabled).

USB Power System

The USB power system incorporates dual LDO regulators (3.3 V and 1.8 V) that allow the entire MSP430 device to be powered from 5-V VBUS when it is made available from the USB host. Alternatively, the power system can supply power only to the USB module, or it can be unused altogether, as in a fully selfpowered device. The block diagram is shown in Figure 42-2

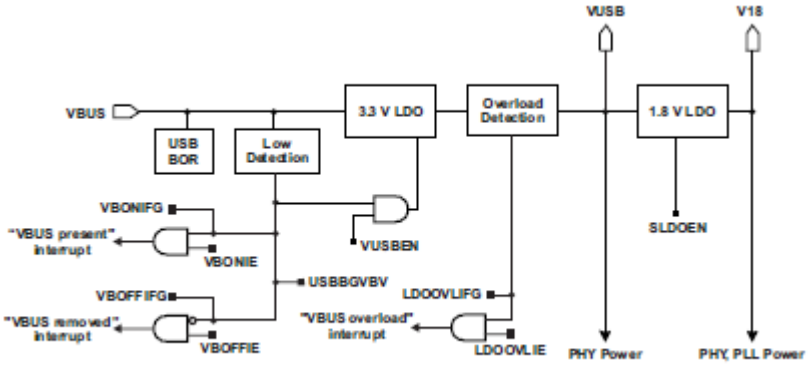


Figure 42-2. USB Power System

The 3.3-V LDO receives 5 V from VBUS and provides power to the transceiver, as well as the VUSB pin. Using this setup prevents the relatively high load of the transceiver and PLL from loading a local system power supply, if used. Thus it is very useful in battery-powered devices.

The 1.8-V LDO receives power from the VUSB pin – which is to be sourced either from the internal 3.3-V LDO or externally – and provides power to the USB PLL and transceiver. The 1.8-V LDO in the USB module is not related to the LDO that resides in the MSP430 Power Management Module (PMM). The inputs and outputs of the LDOs are shown in Figure 42-2. VBUS, VUSB, and V18 need to be connected to external capacitors. The V18 pin is not intended to source other components in the system, rather it exists solely for the attachment of a load capacitor.

Enabling and Disabling

The 3.3-V LDO is enabled or disabled by setting or clearing VUSBEN, respectively. Even if enabled, if the voltage on VBUS is detected to be low or nonexistent, the LDO is suspended. No additional current is consumed while the LDO is suspended. When VBUS rises above the USB power brownout level, the LDO reference and low voltage detection become enabled. When VBUS rises further above the launch voltage VLAUNCH, the LDO module becomes enabled (see Figure 42-3). See device-specific data sheet for value of VLAUNCH.

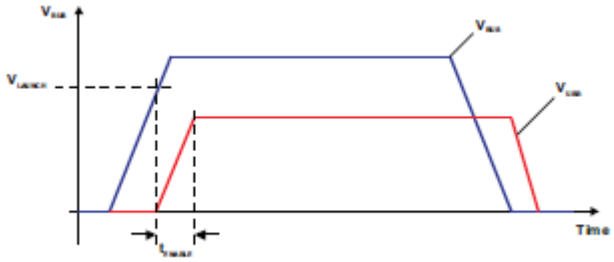


Figure 42-3. USB Power Up and Down Profile

The 1.8-V LDO can be enabled or disabled by setting SLDOEN accordingly. By default, the 1.8-V LDO is controlled automatically according to whether power is available on VBUS. This auto-enable feature is controlled by SLDOAON. In this case, that the SLDOEN bit does not reflect the state of the 1.8-V LDO. If the user wishes to know the state while using the auto-enable feature, the USBBGVBV bit in
USBPWRCTL can be read. In addition, to disable the 1.8-V LDO, SLDOAON must be cleared along with SLDOEN. If providing VUSB from an external source, rather than through the integrated 3.3-V LDO, keep in mind that if 5 V is not present on VBUS, the 1.8-V LDO is not automatically enabled. In this situation, either VBUS much be attached to USB bus power, or the SLDOAON bit must be cleared and SLDOEN set.

It is required that power from the USB cable's VBUS be directed through a Schottky diode prior to entering the VBUS terminal. This prevents current from draining into the cable's VBUS from the LDO input, allowing the MSP430 to tolerate a suspended or unpowered USB cable that remains electrically connected.

The VBONIFG flag can be used to indicate that the voltage on VBUS has risen above the launch voltage. In addition to the VBONIFG being set, an interrupt is also generated when VBONIE = 1. Similarly, the VBOFFIFG flag can be used to indicate that the voltage on VBUS has fallen below the launch voltage. In addition to the VBOFFIFG being set, an interrupt is also generated when VBOFFIE = 1. The USBBGVBV bit can also be polled to indicate the level of VBUS; that is, above or below the launch voltage.

Powering the Rest of the MSP430 From USB Bus Power Via VUSB

The output of the 3.3-V LDO can be used to power the entire MSP430 device, sourcing the DVCC rail. If this is desired, the VUSB and DVCC should be connected externally. Power from the 3.3-V LDO is sourced into DVCC (see Figure 42-4).

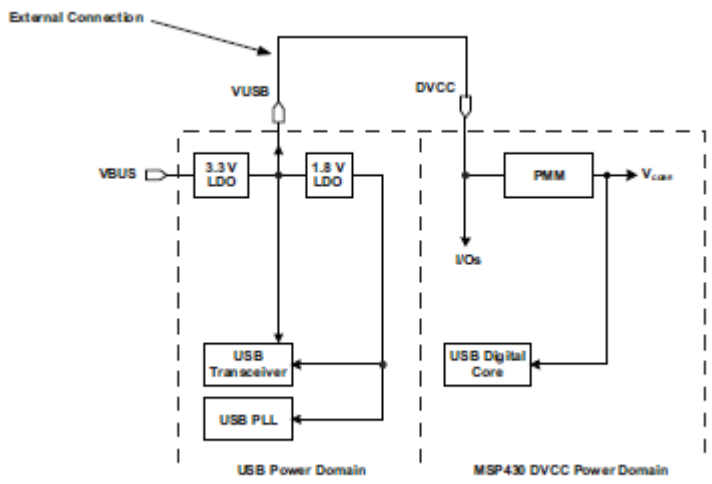


Figure 42-4. Powering Entire MSP430 From VBUS

With this connection made, the MSP430 allows for autonomous power up of the device when VBUS rises above VLAUNCH. If no voltage is present on VCORE – meaning the device is unpowered (or, in LPMx.5 mode) – then both the 3.3-V and 1.8-V LDOs automatically turn on when VBUS rises above VLAUNCH. Note that if DVCC is being driven from VUSB in this manner, and if power is available from VUSB, attempting to place the device into LPMx.5 results in the device immediately re-powering. This is because it re-creates the conditions of the autonomous feature described above (no VCORE but power available on VBUS). The resulting drop of VCORE would cause the system to immediately power up again. When DVCC is being powered from VUSB, it is up to the user to ensure that the total current being drawn from VBUS stays below IDET.

Powering Other Components in the System from VUSB

There is sufficient current capacity available from the 3.3-V LDO to power not only the entire MSP430 but also other components in the system, via the VUSB pin.

If the device is to always be connected to USB, then perhaps no other power system is needed. If it only occasionally connects to USB and is battery-powered otherwise, then sourcing system power via the 3.3-V LDO takes power burden away from the battery. Alternatively, if the battery is rechargeable, the recharging can be driven from VUSB.

Self-Powered Devices

Some applications may be self-powered, in that the VUSB power is supplied externally. In these cases, the 3.3-V LDO would be disabled (VUSBEN = 0). For proper USB operation, the voltage on VBUS can still be detected, even while the 3.3-V LDO disabled, by setting USBDETEN = 1. When VBUS rises above the USB power brownout level, low voltage detection becomes enabled. When VBUS rises further above the launch voltage VLAUNCH, the voltage on VBUS is detected.

Current Limitation and Overload Protection

The 3.3-V LDO features current limitation to protect the transceiver during shorted-cable conditions. A short or overload condition – that is, when the output of the LDO becomes current-

limited to IDET. This is reported to software via the VUOVLIFG flag. See device-specific data sheet for value of IDET.

If this event occurs, it means USB operation may become unreliable, due to insufficient power supply. As a result, software may wish to cease USB operation. If the OVLAOFF bit is set, USB operation is automatically terminated by clearing VUSBEN.

During overload conditions, VUSB and V18 drop below their nominal output voltage. In power scenarios where DVCC is exclusively supplied from VUSB, repetitive system restarts may be triggered as long the short or overload condition exists. For this reason, firmware should avoid re-enabling USB after detection of an overload on the previous power session, until the cause of failure can be identified. Ultimately, it is the user's responsibility to ensure that the current drawn from VBUS does not exceed IDET.

The VUOVLIFG flag can be used to indicate an overcurrent condition on the 3.3-V LDO. When an

overcurrent condition is detected, VUOVLIFG = 1. In addition to the VUOVLIFG being set, an interrupt is also generated when VUOVLIE = 1. The USB power system brownout circuit is supplied from VBUS or DVCC, whichever carries the higher voltage.

USB Phase-Locked Loop (PLL)

The PLL provides the low-jitter high-accuracy clock needed for USB operation (see Figure

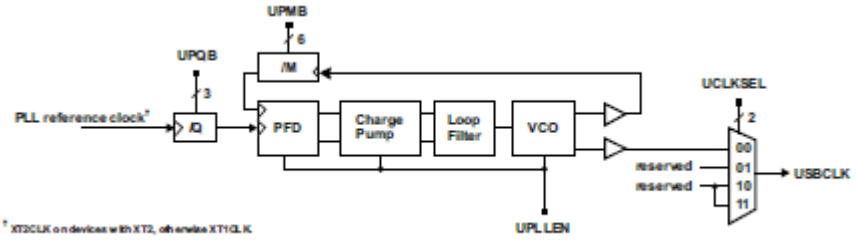


Figure 42-5. USB-PLL Analog Block Diagram

The reference clock to the PLL depends on the device configuration. On devices that contain the optional XT2, the reference clock to the PLL is XT2CLK, regardless if XT1 is available. If the device has only XT1, then the reference is XT1CLK. A four-bit prescale counter controlled by the UPQB bits allows division of the reference to generate the PLL update clock. The UPMB bits control the divider in the feedback path and define the multiplication rate of the PLL (see Equation 20).

Where CLKSEL is the PLL reference clock frequency DIVQ is derived from Table 42-1

DIVM represents the value of UPMB field Table 42-2 lists some common clock input frequencies for CLKSEL, along with the appropriate register settings for generating the nominal 48-MHz clock required by the USB serial engine. For crystal operation, a 4 MHz or higher crystal is required. For crystal bypass mode of operation, 1.5 MHz is the lowest external clock input possible for CLKSEL.

If USB operation is used in a bus-powered configuration, disabling the PLL is necessary to pass the USB requirement of not consuming more than 500 µA. The UPLLEN bit enables or

disables the PLL. The PFDEN bit must be set to enable the phase and frequency discriminator. Out-of-lock, loss-of-signal, and out-of-range are indicated and flagged in the interrupt flags OOLIFG, LOSIFG, OORIFG, respectively.

Modifying the Divider Values

Updating the values of UPQB (DIVQ) and UPMB (DIVM) to select the desired PLL frequency must occur simultaneously to avoid spurious frequency artifacts. The values of UPQB and UPMB can be calculated and written to their buffer registers; the final update of UPQB and UPMB occurs when the upper byte of UPLLDIVB (UPQB) is written.

PLL Error Indicators

The PLL can detect three kinds of errors. Out-of-lock (OOL) is indicated if a frequency correction is performed in the same direction (that is, up or down) for four consecutive update periods. Loss-of-signal (LOS) is indicated if a frequency correction is performed in the same direction (that is, up or down) for 16 consecutive update periods. Out-of-range (OOR) is indicated if PLL was unable to lock for more than 32 update periods. OOL, LOS, and OOR trigger their respective interrupt flags (USBOOLIFG, USBLOSIFG, USBOORIFG) if errors occur, and interrupts are generated if enabled by their enable bits (USBOOLIE, USBLOSIE, USBOORIE).

Table 42-2. Register Settings to Generate 48 MHz Using Common Clock Input Frequencies (continued)

CLK_{REF} (MHz)	UPQB	UPMB	DIVQ	DIVM	CLKLOOP (MHz)	UPLLCLK (MHz)	ACCURACY (ppm)
9.6	011	010011	4	20	2.4	48	0
10.66 ≈ (32/3)	011	010001	4	18	2.6667	48	0
12[70]	011	001111	4	16	3	48	0
12.8	101	011101	8	30	1.6	48	0
14.4	100	010011	6	20	2.4	48	0
16	100	010001	6	18	2.6667	48	0
16.9344	100	010000	6	17	2.8224	47.98	-400
16.94118	100	010000	6	17	2.8235	48	0
18	100	001111	6	16	3	48	0
19.2	101	010011	8	20	2.4	48	0
24[70]	101	001111	8	16	3	48	0
25.6	111	011101	16	30	1.6	48	0
26.0	110	010111	13	24	2	48	0
32	111	010111	16	24	2.6667	48	0

PLL Startup Sequence

To achieve the fastest startup of the PLL, the following sequence is recommended.

1. Enable VUSB and V18.
2. Wait 2 ms for external capacitors to charge, so that proper VUSB is in place. (During this time, the

USB registers and buffers can be initialized.)

3. Activate the PLL, using the required divider values.
4. Wait 2 ms and check PLL. If it stays locked, it is ready to be used.

USB Controller Engine

The USB controller engine transfers data packets arriving from the USB host into the USB buffers, and also transmits valid data from the buffers to the USB host. The controller engine has dedicated, fixed buffer space for input endpoint 0 and output endpoint 0, which are the default USB endpoints for control transfers.

The 14 remaining endpoints (seven input and seven output) may have one or more USB buffers assigned to them. All the buffers are located in the USB buffer memory. This memory is implemented as "multiport" memory, in that it can be accessed both by the USB buffer manager and also by the CPU and DMA. Each endpoint has a dedicated set of descriptor registers that describe the use of that endpoint (see Figure 42-6). Configuration of each endpoint is performed by setting its descriptor registers. These data structures are located in the USB buffer memory and contain address pointers to the next memory buffer for receive or transmit.

Assigning one or two data buffers to an endpoint, of up to 64 bytes, requires no further software involvement after configuration. If more than two buffers per endpoint are desired, however, software must change the address pointers on the fly during a receive or transmit process. Synchronization of empty and full buffers is done using validation flags. All events are indicated by flags and fire a vector interrupt when enabled. Transfer event indication can be enabled separately.

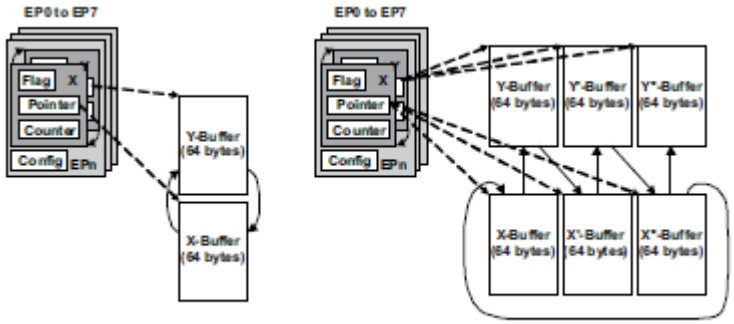


Figure 42-6. Data Buffers and Descriptors

USB Serial Interface Engine (SIE)

The SIE logic manages the USB packet protocol requirements for the packets being received and transmitted on the bus. For packets being received, the SIE decodes the packet identifier field (packet ID) to determine the type of packet being received and to ensure the packet ID is valid. For token and data packets being received, the SIE calculates the packet cycle redundancy check (CRC) and compares the value to the CRC contained in the packet to verify that the packet was not corrupted during transmission.

For token and data packets being transmitted, the SIE generates the CRC that is transmitted with the packet. For packets being transmitted, the SIE also generates the synchronization field (SYNC), which is an eight-bit field at the beginning of each packet. In addition, the SIE generates the correct packet ID for all packets being transmitted. Another major function of the SIE is the overall serial-to-parallel conversion of the data packets being received or transmitted.

USB Buffer Manager (UBM)

The USB buffer manager provides the control logic that interfaces the SIE to the USB endpoint buffers. One of the major functions of the UBM is to decode the USB device address to determine if the USB host is addressing this particular USB device. In addition, the endpoint address field and direction signal are decoded to determine which particular USB endpoint is being addressed. Based on the direction of the USB transaction and the endpoint number, the UBM either writes or reads the data packet to or from the appropriate USB endpoint data buffer.

The TOGGLE bit for each output endpoint configuration register is used by the UBM to track successful output data transactions. If a valid data packet is received and the data packet ID matches the expected packet ID, the TOGGLE bit is toggled. Similarly, the TOGGLE bit for each input endpoint configuration is used by the UBM to track successful input data transactions. If a valid data packet is transmitted, the TOGGLE bit is toggled. If the TOGGLE bit is cleared, a DATA0 packet ID is transmitted in the data packet to the host. If the TOGGLE bit is set, a DATA1 packet ID is transmitted in the data packet to the host. See Section 42.3 regarding details of USB transfers.

USB Buffer Memory

The USB buffer memory contains the data buffers for all endpoints and for SETUP packets. In that the buffers for endpoints 1 to 7 are flexible, there are USB buffer configuration registers that define them, and these too are in the USB buffer memory. (Endpoint 0 is defined with a set of registers in the USB control register space.) Storing these in open memory allows for efficient, flexible use, which is advantageous because use of these endpoints is very application-specific.

This memory is implemented as "multiport" memory, in that it can be accessed both by the USB buffer manager and also by the CPU and DMA. The SIE allows CPU or DMA access, but reserves priority. As a result, CPU or DMA access is delayed using wait states if a conflict arises with an SIE access. When the USB module is disabled (USBEN = 0), the buffer memory behaves like regular RAM. When changing the state of the USBEN bit (enabling or disabling the USB module), the USB buffer memory should not be accessed within four clocks before and eight clocks after changing this bit, as doing so reconfigures the access method to the USB memory.

Accessing of the USB buffer memory by CPU or DMA is only possible if the USB PLL is active. When a host requests suspend condition the application software (for example, USB stack) of client has to switch off the PLL within 10 ms. Note that the MSP430 USB suspend interrupt occurs around 5 ms after the host request.

Each endpoint is defined by a block of six configuration "registers" (based in RAM, they are not true registers in the strict sense of the word). These registers specify the endpoint type, buffer address, buffer size and data packet byte count. They define an endpoint buffer space that is 1904 bytes in size. An additional 24 bytes are allotted to three remaining blocks – the EP0_IN buffer, the EP0_OUT buffer, and the SETUP packet buffer (see Table 42-3).

Table 42-3. USB Buffer Memory Map

Memory	Short Form	Access Type	Address Offset
Start of buffer space	STABUFF	Read/Write	0000h
1904 bytes of configurable buffer space	⋮	Read/Write	⋮
End of buffer space	TOPBUFF	Read/Write	076Fh
Output endpoint_0 buffer	USBOEP0BUF	Read/Write	0770h
		Read/Write	⋮
		Read/Write	0777h
Input endpoint_0 buffer	USBIEP0BUF	Read/Write	0778h
		Read/Write	⋮
		Read/Write	077Fh
Setup Packet Block	USBSUBLK	Read/Write	0780h
		Read/Write	⋮
		Read/Write	0787h

Software can configure each buffer according to the total number of endpoints needed. Single or double buffering of each endpoint is possible. Unlike the descriptor registers for endpoints 1 to 7, which are defined as memory entries in USB RAM, endpoint 0 is described by a set of four registers (two for output and two for input) in the USB control register set. Endpoint 0 has no base-address register, since these addresses are hardwired. The bit positions have been preserved to provide consistency with endpoint_n (n = 1 to 7).

USB Fine Timestamp

The USB module is capable of saving a timestamp associated with particular USB events (see Figure 42- 7). This can be useful in compensating for delays in software response. The timestamp values are based on the USB module's internal timer, driven by USBCLK.

Up to four events can be selected to generate the timestamp, selected with the TSESEL bits. When they occur, the value of the USB timer is transferred to the timestamp register USBTSREG, and thus the exact moment of the event is recorded. The trigger options include one of three DMA channels, or a softwaredriven event. The USB timer cannot be directly accessed by reading.

Furthermore, the value of the USB timer can be used to generate periodic interrupts. Since the USBCLK can have a frequency different from the other system clocks, this gives another option for periodic system interrupts. The UTSEL bits select the divider from the USB clock. UTIE must be set for an interrupt vector to get triggered. The timestamp register is set to zero on a frame-number-receive event and pseudo-start-of-frame. TSGEN enables or disables the time stamp generator.

Suspend and Resume Logic

The USB suspend and resume logic detects suspend and resume conditions on the USB bus. These events are flagged in SUSRIFG and RESRIFG, respectively, and they fire dedicated interrupts, if the interrupts are enabled (SUSRIE and RESRIE). The remote wakeup mechanism, in which a USB device can cause the USB host to awaken and resume the device, is triggered by setting the RWUP bit of the USBCTL register. See Section 42.2.6 for more information.

Reset Logic

A PUC resets the USB module logic. When FRSTE = 1, the logic is also reset when a USB reset event occurs on the bus, triggered from the USB host. (A USB reset also sets the RSTRIFG flag.) USB buffermemory is not reset by a USB reset.

USB Vector Interrupts

The USB module uses a single interrupt vector generator register to handle multiple USB interrupts. All USB-related interrupt sources trigger the USBVECINT (also called USBIV) vector, which then contains a 6-bit vector value that identifies the interrupt source. Each of the interrupt sources results in a different offset value read. The interrupt vector returns zero when no interrupt is pending.

Reading the interrupt vector register clears the corresponding interrupt flag and updates its value. The interrupt with highest priority returns the value 0002h; the interrupt with lowest priority returns the value 003Eh when reading the interrupt vector register. Writing to this register clears all interrupt flags.

For each input and output endpoints resides an USB transaction interrupt indication enable. Software may set this bit to define if interrupts are to be flagged in general. To generate an interrupt the corresponding interrupt enable and flag must be set.

Power Consumption

USB functionality consumes more power than is typically drawn in the MSP430. Since most MSP430applications are power sensitive, the MSP430 USB module has been designed to protect the battery by ensuring that significant power load only occurs when attached to the bus, allowing power to be drawn from VBUS.

The two components of the USB module that draw the most current are the transceiver and the PLL. The transceiver can consume large amounts of power while transmitting, but in its quiescent state – that is, when not transmitting data – the transceiver actually consumes very little power. This is the amount specified as IIDLE. This amount is so little that the transceiver can be kept active during suspend mode without presenting a problem for bus-powered applications. Fortunately the transceiver always has access to VBUS power when drawing the level of current required for transmitting.

The PLL consumes a larger amount of current. However, it need only be active while connected to the host, and the host can supply the power. When the PLL is disabled (for example, during USB suspend), USBCLK automatically is sourced from the VLO.

Suspend and Resume

All USB devices must support the ability to be suspended into a no-activity state, and later resumed. When suspended, a device is not allowed to consume more than 500uA from the USB's VBUS power rail, if the device is drawing any power from that source. A suspended device must also monitor for a resume event on the bus.

The host initiates a suspend condition by creating a constant idle state on the bus for more than 3.0 ms. It is the responsibility of the software to ensure the device enters its low power suspend state within 10 ms of the suspend condition. The USB specification requires that a suspended bus-powered USB device not draw in excess of 500 μA from the bus.

Entering Suspend

When the host suspends the USB device, a suspend interrupt is generated (SUSRIFG). From this point, the software has 10 ms to ensure that no more than 500uA is being drawn from the host via VBUS.

For most applications, the integrated 3.3-V LDO is being used. In this case, the following actions should be taken:

• Disable the PLL by clearing UPLLEN (UPLLEN = 0)

• Limit all current sourced from VBUS that causes the total current sourced from VBUS equal to 500 μA minus the suspend current, ISUSPEND (see the device-specific data sheet).

Disabling the PLL eliminates the largest on-chip draw of power from VBUS. During suspend, the USBCLK is automatically sourced by the VLO (VLOCLK), allowing the USB module to detect resume when it occurs. It is a good idea to also then ensure that the RESRIE bit is also set, so that an interrupt is generated when the host resumes the device. If desired, the high frequency crystal can also be disabled to save additional system power, however it does not contribute to the power from VBUS since it draws power from the DVCC supply.

Entering Resume Mode

When the USB device is in a suspended condition, any non-idle signaling, including reset signaling, on the host side is detected by the suspend and resume logic and device operation is resumed. RESRIFG is set, causing an USB interrupt. The interrupt service routine can be used to resume USB operation.

USB Transfers

The USB module supports control, bulk, and interrupt data transfer types. In accordance with the USB specification, endpoint 0 is reserved for the control endpoint and is bidirectional. In addition to the control endpoint, the USB module is capable of supporting up to 7 input endpoints and 7 output endpoints. These additional endpoints can be configured either as bulk or interrupt endpoints. The software handles all control, bulk, and interrupt endpoint transactions.

Control Transfers

Control transfers are used for configuration, command, and status communication between the host and the USB device. Control transfers to the USB device use input endpoint 0 and output endpoint 0. The three types of control transfers are control write, control write with no data stage, and control read. Note that the control endpoint must be initialized before connecting the USB device to the USB

Control Write Transfer

The host uses a control write transfer to write data to the USB device. A control write transfer consists of a setup stage transaction, at least one output data stage transaction, and an input status stage transaction. The stage transactions for a control write transfer are:

• Setup stage transaction:

1. Input endpoint 0 and output endpoint 0 are initialized by programming the appropriate USB endpoint configuration blocks. This entails enabling the endpoint interrupt (USBIIE = 1) and enabling the endpoint (UBME = 1). The NAK bit for both input endpoint 0 and output endpoint 0 must be cleared.

2. The host sends a setup token packet followed by the setup data packet addressed to output endpoint 0. If the data is received without an error, then the UBM writes the data

to the setup data packet buffer, sets the setup stage transaction bit (SETUPIFG = 1) in the USB Interrupt Flag register (USBIFG), returns an ACK handshake to the host, and asserts the setup stage transaction interrupt. Note that as long as SETUPIFG = 1, the UBM returns a NAK handshake for any data stage or status stage transactions regardless of the endpoint 0 NAK or STALL bit values.

3. The software services the interrupt, reads the setup data packet from the buffer, and then decodes the command. If the command is not supported or invalid, the software should set the STALL bit in the output endpoint 0 configuration register (USBOEPCNFG_0) and the input endpoint 0 configuration register (USBIEPCNFG_0) . This causes the device to return a STALL handshake for any data or status stage transaction. For control write transfers, the packet ID used by the host for the first data packet output is a DATA1 packet ID and the TOGGLE bit must match.

NOTE: When using USBIV, SETUPIFG is cleared upon reading USBIV. In addition, the NAK onb input endpoint 0 and output endpoint 0 are also cleared. In this case, the host may send or receive the next setup packet even if MSP430 did not perform the first setup packet. To prevent this, first read the SETUPIFG directly, perform the required setup, and then use the USBIV for further processing.

NOTE: The priority of input endpoint 0 is higher than the setup flag inside USBIV (SETUPIFG). Therefore, if both the USBIEPIFG.EP0 and SETUPIFG are pending, reading USBIV gives the higher priority interrupt (EP0) as opposed to SETUPIFG. Therefore, read SETUPIFG directly, process the pending setup packet, then proceed to read the USBIV.

• Data stage transaction:

1. The host sends an OUT token packet followed by a data packet addressed to output endpoint 0. If the data is received without an error, the UBM writes the data to the output endpoint buffer

(USBOEP0BUF), updates the data count value, toggles the TOGGLE bit, sets the NAK bit, returns an ACK handshake to the host , and asserts the output endpoint interrupt 0 (OEPIFG0).

2. The software services the interrupt and reads the data packet from the output endpoint buffer. To read the data packet, the software first needs to obtain the data count value inside the

USBOEPBCNT_0 register. After reading the data packet, the software should clear the NAK bit to allow the reception of the next data packet from the host.

3. If the NAK bit is set when the data packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the data packet is received, the UBM simply returns a STALL handshake to the host. If a CRC or bit stuff error occurs when the data packet is received, then no handshake is returned to the host.

• Status stage transaction:

1. For input endpoint 0, the software updates the data count value to zero, sets the TOGGLE bit, then clears the NAK bit to enable the data packet to be sent to the host. Note that for a status stage transaction, a null data packet with a DATA1 packet ID is sent to the host.

2. The host sends an IN token packet addressed to input endpoint 0. After receiving the IN token, the UBM transmits a null data packet to the host. If the data packet is received without errors by the host, then an ACK handshake is returned. The UBM then toggles the TOGGLE bit and sets the NAK bit.

3. If the NAK bit is set when the IN token packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the IN token packet is received, the UBM simply returns a STALL handshake to the host. If no handshake packet is received from the host, then the UBM prepares to retransmit the same data packet again.

Control Write Transfer with No Data Stage Transfer

The host uses a control write transfer to write data to the USB device. A control write with no data stage transfer consists of a setup stage transaction and an input status stage transaction. For this type of transfer, the data to be written to the USB device is contained in the two byte value field of the setup stage transaction data packet.

The stage transactions for a control write transfer with no data stage transfer are:

• Setup stage transaction:

1. Input endpoint 0 and output endpoint 0 are initialized by programming the appropriate USB endpoint configuration blocks. This entails programming the buffer size and buffer base address,

selecting the buffer mode, enabling the endpoint interrupt (USBIIE = 1), initializing the TOGGLE bit, enabling the endpoint (UBME = 1). The NAK bit for both input endpoint 0 and output endpoint 0 must be cleared.

2. The host sends a setup token packet followed by the setup data packet addressed to output endpoint 0. If the data is received without an error then the UBM writes the data to the setup data packet buffer, sets the setup stage transaction (SETUP) bit in the USB status register, returns an

ACK handshake to the host, and asserts the setup stage transaction interrupt. Note that as long as the setup transaction (SETUP) bit is set, the UBM returns a NAK handshake for any data stage or status stage transaction regardless of the endpoint 0 NAK or STALL bit values.

3. The software services the interrupt and reads the setup data packet from the buffer then decodes the command. If the command is not supported or invalid, the software should set the STALL bits in the output endpoint 0 and the input endpoint 0 configuration registers before clearing the setup stage transaction (SETUP) bit. This causes the device to return a STALL handshake for data or status stage transactions. After reading the data packet and decoding the command, the software should clear the interrupt, which automatically clears the setup stage transaction status bit.

NOTE: When using USBIV, the SETUPIFG is cleared upon reading USBIV. In addition, the NAK o input endpoint 0 and output endpoint 0 is also cleared. In this case, the host may send or receive the next setup packet even if MSP430 did not perform the first setup packet. To prevent this, first read the SETUPIFG directly, perform the required setup, and then use the USBIV for further processing.

NOTE: The priority of input endpoint 0 is higher than setup flag inside USBIV. Therefore, if both the USBIEPIFG.EP0 and SETUPIFG are pending, reading the USBIV gives the higher priority interrupt (EP0) as opposed to the SETUPIFG. Therefore, read SETUPIFG directly, process the pending setup packet, then proceed to read the USBIV.

• Status stage transaction:

1. For input endpoint 0, the software updates the data count value to zero, sets the TOGGLE bit then clears the NAK bit to enable the data packet to be sent to the host. Note that for a status stage transaction a null data packet with a DATA1 packet ID is sent to the host.

2. The host sends an IN token packet addressed to input endpoint 0. After receiving the IN token, the

UBM transmits a null data packet to the host. If the data packet is received without errors by the host, then an ACK handshake is returned. The UBM then toggles the TOGGLE bit, sets the NAK bit, and asserts the endpoint interrupt.

3. If the NAK bit is set when the IN token packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the IN token packet is received, the UBM simply returns a STALL handshake to the host. If no handshake packet is received from the host, then the UBM prepares to retransmit the same data packet again.

Control Read Transfer

The host uses a control read transfer to read data from the USB device. A control read transfer consists of a setup stage transaction, at least one input data stage transaction and an output status stage transaction. The stage transactions for a control read transfer are:

• Setup stage transaction:

1. Input endpoint 0 and output endpoint 0 are initialized by programming the appropriate USB endpoint configuration blocks. This entails enabling the endpoint interrupt (USBIIE = 1) and enabling the endpoint (UBME = 1). The NAK bit for both input endpoint 0 and output endpoint 0 must be cleared.

2. The host sends a setup token packet followed by the setup data packet addressed to output endpoint 0. If the data is received without an error, then the UBM writes the data to the setup buffer, sets the setup stage transaction (SETUP) bit in the USB status register, returns an ACK handshake to the host, and asserts the setup stage transaction interrupt. Note that as long as the setup transaction (SETUP) bit is set, the UBM returns a NAK handshake for any data stage or status stage transactions regardless of the endpoint 0 NAK or STALL bit values.

3. The software services the interrupt and reads the setup data packet from the buffer then decodes the command. If the command is not supported or invalid, the software should set the STALL bits in the output endpoint 0 and the input endpoint 0 configuration registers before clearing the setup stage transaction (SETUP) bit. This causes the device to return a STALL handshake for a data stage or status stage transactions. After reading the data packet and decoding the command, the software should clear the interrupt, which automatically clears the setup stage transaction status bit. The software should also set the TOGGLE bit in the input endpoint 0 configuration register. For control read transfers, the packet ID used by the host for the first input data packet is a DATA1 packet ID.

NOTE: When using USBIV, the SETUPIFG is cleared upon reading USBIV. In addition, it also the clears NAK on input endpoint 0 and output endpoint 0. In this case, the host may send or receive the next setup packet even if MSP430 did not perform the first setup packet. To prevent this, first read the SETUPIFG directly, perform the required setup, and then use the USBIV for further processing.

NOTE: The priority of input endpoint 0 is higher than the setup flag inside USBIV. Therefore, if both the USBIEPIFG.EP0 and SETUPIFG are pending, reading the USBIV gives the higher priority interrupt (EP0) as opposed to the SETUPIFG. Therefore, read SETUPIFG directly, process the pending setup packet, then proceed to read the USBIV.

• Data stage transaction:

1. The data packet to be sent to the host is written to the input endpoint 0 buffer by the software. The software also updates the data count value then clears the input endpoint 0 NAK bit to enable the data packet to be sent to the host.
2. The host sends an IN token packet addressed to input endpoint 0. After receiving the IN token, the UBM transmits the data packet to the host. If the data packet is received without errors by the host, then an ACK handshake is returned. The UBM sets the NAK bit and asserts the endpoint interrupt.
3. The software services the interrupt and prepares to send the next data packet to the host.
4. If the NAK bit is set when the IN token packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the IN token packet is received, the UBM simply returns a STALL handshake to the host. If no handshake packet is received from the host, then the UBM prepares to retransmit the same data packet again.
5. The software continues to send data packets until all data has been sent to the host.

• Status stage transaction:

1. For output endpoint 0, the software sets the TOGGLE bit, then clears the NAK bit to enable the data packet to be sent to the host. Note that for a status stage transaction a null data packet with a DATA1 packet ID is sent to the host.
2. The host sends an OUT token packet addressed to output endpoint 0. If the data packet is received *Transfers* without an error then the UBM updates the data count value, toggles the TOGGLE bit, sets the NAK bit, returns an ACK handshake to the host, and asserts the endpoint interrupt.
3. The software services the interrupt. If the status stage transaction completed successfully, then the software should clear the interrupt and clear the NAK bit.
4. If the NAK bit is set when the input data packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the in data packet is received, the UBM simply returns a STALL handshake to the host. If a CRC or bit stuff error occurs when the data packet is received, then no handshake is returned to the host.

Interrupt Transfers

The USB module supports interrupt data transfers both to and from the host. Devices that need to send or receive a small amount of data with a specified service period are best served by the interrupt transfer type. Input endpoints 1 through 7 and output endpoints 1 through 7 can be configured as interrupt endpoints.

Interrupt OUT Transfer

The steps for an interrupt OUT transfer are:

1. The software initializes one of the output endpoints as an output interrupt endpoint by programming the appropriate endpoint configuration block. This entails programming the buffer size and buffer base address, selecting the buffer mode, enabling the endpoint interrupt, initializing the toggle bit, enabling the endpoint, and clearing the NAK bit.
2. The host sends an OUT token packet followed by a data packet addressed to the output endpoint. If the data is received without an error then the UBM writes the data to the endpoint buffer, updates the data count value, toggles the toggle bit, sets the NAK bit, returns an ACK handshake to the host, and asserts the endpoint interrupt.
3. The software services the interrupt and reads the data packet from the buffer. To read the data packet, the software first needs to obtain the data count value. After reading the

data packet, the software should clear the interrupt and clear the NAK bit to allow the reception of the next data packet from the host.

4. If the NAK bit is set when the data packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the data packet is received, the UBM simply returns a STALL handshake to the host. If a CRC or bit stuff error occurs when the data packet is received, then no handshake is returned to the host device.

In double buffer mode, the UBM selects between the X and Y buffer based on the value of the toggle bit. If the toggle bit is a 0, the UBM writes the data packet to the X buffer. If the toggle bit is a 1, the UBM writes the data packet to the Y buffer. When a data packet is received, the software could determine which buffer contains the data packet by reading the toggle bit. However, when using double buffer mode, the possibility exists for data packets to be received and written to both the X and Y buffer before the software responds to the endpoint interrupt. In this case, simply using the toggle bit to determine which buffer contains the data packet would not work. Hence, in double buffer mode, the software should read the X buffer NAK bit, the Y buffer NAK bit, and the toggle bits to determine the status of the buffers.

Interrupt IN Transfer

The steps for an interrupt IN transfer are:

1. The software initializes one of the input endpoints as an input interrupt endpoint by programming the appropriate endpoint configuration block. This entails programming the buffer size and buffer base address, selecting the buffer mode, enabling the endpoint interrupt, initializing the toggle bit, enabling the endpoint, and setting the NAK bit.

2. The data packet to be sent to the host is written to the buffer by the software. The software also updates the data count value then clears the NAK bit to enable the data packet to be sent to the host.

3. The host sends an IN token packet addressed to the input endpoint. After receiving the IN token, the UBM transmits the data packet to the host. If the data packet is received without errors by the host, then an ACK handshake is returned. The UBM then toggles the toggle bit, sets the NAK bit, and asserts the endpoint interrupt.

4. The software services the interrupt and prepares to send the next data packet to the host.

5. If the NAK bit is set when the in token packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the IN token packet is received, the UBM simply returns a STALL handshake to the host. If no handshake packet is received from the host, then the UBM prepares to retransmit the same data packet again. In double buffer mode, the UBM selects between the X and Y buffer based on the value of the toggle bit. If the toggle bit is a 0, the UBM reads the data packet from the X buffer. If the toggle bit is a 1, the UBM reads the data packet from the Y buffer.

Bulk Transfers

The USB module supports bulk data transfers both to and from the host. Devices that need to send or receive a large amount of data without a suitable bandwidth are best served by the bulk transfer type. In endpoints 1 through 7 and out endpoints 1 through 7 can all be configured as bulk endpoints.

Bulk OUT Transfer

The steps for a bulk OUT transfer are:

1. The software initializes one of the output endpoints as an output bulk endpoint by programming the appropriate endpoint configuration block. This entails programming the buffer size and buffer base address, selecting the buffer mode, enabling the endpoint interrupt, initializing the toggle bit, enabling the endpoint, and clearing the NAK bit.
2. The host sends an out token packet followed by a data packet addressed to the output endpoint. If the data is received without an error then the UBM writes the data to the endpoint buffer, updates the data count value, toggles the toggle bit, sets the NAK bit, returns an ACK handshake to the host, an asserts the endpoint interrupt.
3. The software services the interrupt and reads the data packet from the buffer. To read the data packet, the software first needs to obtain the data count value. After reading the data packet, the software should clear the interrupt and clear the NAK bit to allow the reception of the next data packet from the host.
4. If the NAK bit is set when the data packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the data packet is received, the UBM simply returns a STALL handshake to the host. If a CRC or bit stuff error occurs when the data packet is received, then no handshake is returned to the host.

In double buffer mode, the UBM selects between the X and Y buffer based on the value of the toggle bit. If the toggle bit is a 0, the UBM writes the data packet to the X buffer. If the toggle bit is a 1, the UBM writes the data packet to the Y buffer. When a data packet is received, the software could determine which buffer contains the data packet by reading the toggle bit. However, when using double buffer mode, the possibility exists for data packets to be received and written to both the X and Y buffer before the software responds to the endpoint interrupt. In this case, simply using the toggle bit to determine which buffer contains the data packet would not work. Hence, in double buffer mode, the software should read the X buffer NAK bit, the Y buffer NAK bit, and the toggle bits to determine the status of the buffers.

Bulk IN Transfer

The steps for a bulk IN transfer are:

1. The software initializes one of the input endpoints as an input bulk endpoint by programming the appropriate endpoint configuration block. This entails programming the buffer size and buffer base address, selecting the buffer mode, enabling the endpoint interrupt, initializing the toggle bit, enabling the endpoint, and setting the NAK bit.
2. The data packet to be sent to the host is written to the buffer by the software. The software also updates the data count value then clears the NAK bit to enable the data packet to be sent to the host.
3. The host sends an IN token packet addressed to the input endpoint. After receiving the IN token, the UBM transmits the data packet to the host. If the data packet is received without errors by the host, then an ACK handshake is returned. The UBM then toggles the toggle bit, sets the NAK bit, and asserts the endpoint interrupt.
4. The software services the interrupt and prepares to send the next data packet to the host.
5. If the NAK bit is set when the in token packet is received, the UBM simply returns a NAK handshake to the host. If the STALL bit is set when the In token packet is received, the UBM simply returns a STALL handshake to the host. If no handshake packet is received from the host, then the UBM prepares to retransmit the same data packet again.

In double buffer mode , the UBM selects between the X and Y buffer based on the value

of the toggle bit. If the toggle bit is a 0, the UBM reads the data packet from the X buffer. If the toggle bit is a 1, the UBM reads the data packet from the Y buffer.

Universal Serial Communication Interface (USCI): SPI mode

The universal serial communication interface (USCI) modules support multiple serial communication modes. Different USCI modules support different modes. Each different USCI module is named with a different letter. For example, USCI_A is different from USCI_B. If more than one identical USCI module is implemented on one device, those modules are named with incrementing numbers. For example, if one device has two USCI_A modules, they are named USCI_A0 and USCI_A1. See the device-specific data sheet to determine which USCI modules, if any, are implemented on which devices.

USCI_Ax modules support:

- UART mode
- Pulse shaping for IrDA communications
- Automatic baud-rate detection for LIN communications
- SPI mode

USCI_Bx modules support:

- I2C mode
- SPI mode

USCI Introduction – SPI Mode

In synchronous mode, the USCI connects the device to an external system via three or four pins:

UCxSIMO, UCxSOMI, UCxCLK, and UCxSTE. SPI mode is selected when the UCSYNC bit is set, and

SPI mode (3-pin or 4-pin) is selected with the UCMODEx bits.

SPI mode features include:

- 7-bit or 8-bit data length
- LSB-first or MSB-first data transmit and receive
- 3-pin and 4-pin SPI operation
- Master or slave modes
- Independent transmit and receive shift registers
- Separate transmit and receive buffer registers
- Continuous transmit and receive operation
- Selectable clock polarity and phase control
- Programmable clock frequency in master mode
- Independent interrupt capability for receive and transmit
- Slave operation in LPM4

USCI Operation – SPI Mode

In SPI mode, serial data is transmitted and received by multiple devices using a shared clock provided by the master. An additional pin, UCxSTE, is provided to enable a device to receive and transmit data and is controlled by the master.

Three or four signals are used for SPI data exchange:

- UCxSIMO – slave in, master out

  Master mode: UCxSIMO is the data output line.

  Slave mode: UCxSIMO is the data input line.
- UCxSOMI – slave out, master in

Master mode: UCxSOMI is the data input line.
Slave mode: UCxSOMI is the data output line.
• UCxCLK – USCI SPI clock
Master mode: UCxCLK is an output.
Slave mode: UCxCLK is an input.
• UCxSTE – slave transmit enable
Used in 4-pin mode to allow multiple masters on a single bus. Not used in 3-pin mode.

USCI Initialization and Reset

The USCI is reset by a PUC or by the UCSWRST bit. After a PUC, the UCSWRST bit is automatically set, keeping the USCI in a reset condition. When set, the UCSWRST bit resets the UCRXIE, UCTXIE, UCRXIFG, UCOE, and UCFE bits, and sets the UCTXIFG flag. Clearing UCSWRST releases the USCI for operation.

Character Format

The USCI module in SPI mode supports 7-bit and 8-bit character lengths selected by the UC7BIT bit. In 7- bit data mode, UCxRXBUF is LSB justified and the MSB is always reset. The UCMSB bit controls the direction of the transfer and selects LSB or MSB first.

Master Mode

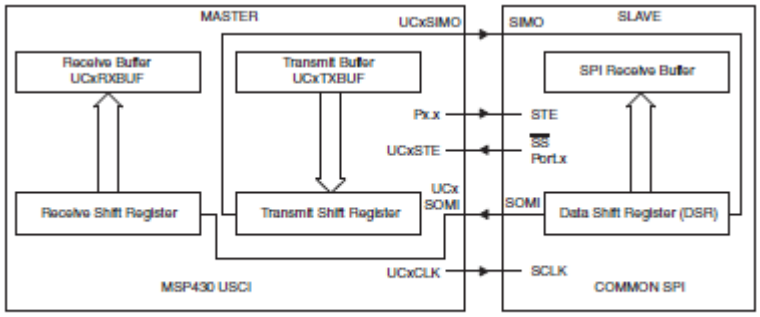


Figure 37-2. USCI Master and External Slave

shows the USCI as a master in both 3-pin and 4-pin configurations. The USCI initiates data transfer when data is moved to the transmit data buffer UCxTXBUF. The UCxTXBUF data is moved to the

transmit (TX) shift register when the TX shift register is empty, initiating data transfer on UCxSIMO starting with either the MSB or LSB, depending on the UCMSB setting. Data on UCxSOMI is shifted into the receive shift register on the opposite clock edge. When the character is received, the receive data is moved from the receive (RX) shift register to the received data buffer UCxRXBUF and the receive interrupt flag UCRXIFG is set, indicating the RX/TX operation is complete.

A set transmit interrupt flag, UCTXIFG, indicates that data has moved from UCxTXBUF to the TX shift register and UCxTXBUF is ready for new data. It does not indicate RX/TX completion. To receive data into the USCI in master mode, data must be written to UCxTXBUF, because receive and transmit operations operate concurrently.

Pin SPI Master Mode

In 4-pin master mode, UCxSTE is used to prevent conflicts with another master and controls the master as described in Table 37-1. When UCxSTE is in the master-inactive state:

• UCxSIMO and UCxCLK are set to inputs and no longer drive the bus.

• The error bit UCFE is set, indicating a communication integrity violation to be handled by the user.

• The internal state machines are reset and the shift operation is aborted. If data is written into UCxTXBUF while the master is held inactive by UCxSTE, it is transmit as soon as UCxSTE transitions to the master-active state. If an active transfer is aborted by UCxSTE transitioning to the master-inactive state, the data must be rewritten into UCxTXBUF to be transferred when UCxSTE transitions back to the master-active state. The UCxSTE input signal is not used in 3-pin master mode.

Slave Mode

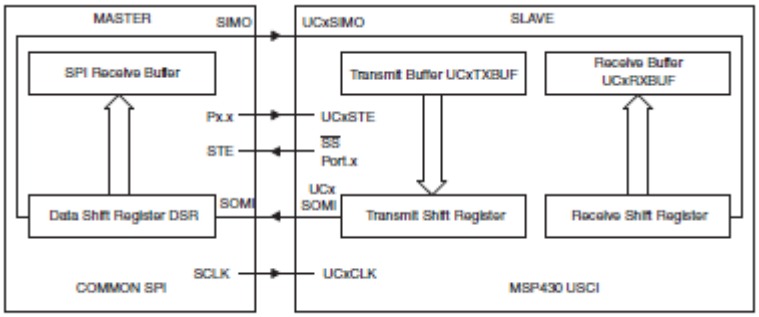


Figure 37-3. USCI Slave and External Master

shows the USCI as a slave in both 3-pin and 4-pin configurations. UCxCLK is used as the input for the SPI clock and must be supplied by the external master. The data-transfer rate is determined by this clock and not by the internal bit clock generator. Data written to UCxTXBUF and moved to the TX shift register before the start of UCxCLK is transmitted on UCxSOMI. Data on UCxSIMO is shifted into the receive shift register on the opposite edge of UCxCLK and moved to UCxRXBUF when the set number of bits are received. When data is moved from the RX shift register to UCxRXBUF, the UCRXIFG interrupt flag is set, indicating that data has been received. The overrun error bit UCOE is set when the previously received data is not read from UCxRXBUF before new data is moved to UCxRXBUF.

4-Pin SPI Slave Mode

In 4-pin slave mode, UCxSTE is used by the slave to enable the transmit and receive operations and is provided by the SPI master. When UCxSTE is in the slave-active state, the slave operates normally. When UCxSTE is in the slave- inactive state:

• Any receive operation in progress on UCxSIMO is halted.

• UCxSOMI is set to the input direction.

• The shift operation is halted until the UCxSTE line transitions into the slave transmit active state. The UCxSTE input signal is not used in 3-pin slave mode.

SPI Enable

When the USCI module is enabled by clearing the UCSWRST bit, it is ready to receive and transmit. In master mode, the bit clock generator is ready, but is not clocked nor producing any clocks. In slave mode, the bit clock generator is disabled and the clock is provided by the

master. A transmit or receive operation is indicated by UCBUSY = 1. A PUC or set UCSWRST bit disables the USCI immediately and any active transfer is terminated.

Transmit Enable

In master mode, writing to UCxTXBUF activates the bit clock generator, and the data begins to transmit. In slave mode, transmission begins when a master provides a clock and, in 4-pin mode, when the UCxSTE is in the slave-active state.

Receive Enable

The SPI receives data when a transmission is active. Receive and transmit operations operate concurrently.

Serial Clock Control

UCxCLK is provided by the master on the SPI bus. When UCMST = 1, the bit clock is provided by the USCI bit clock generator on the UCxCLK pin. The clock used to generate the bit clock is selected with the UCSSELx bits. When UCMST = 0, the USCI clock is provided on the UCxCLK pin by the master, the bit clock generator is not used, and the UCSSELx bits are don't care. The SPI receiver and transmitter operate in parallel and use the same clock source for data transfer.

The 16-bit value of UCBRx in the bit rate control registers (UCxxBR1 and UCxxBR0) is the division factor of the USCI clock source, BRCLK. The maximum bit clock that can be generated in master mode is BRCLK. Modulation is not used in SPI mode, and UCAxMCTL should be cleared when using SPI mode for USCI_A. The UCAxCLK/UCBxCLK frequency is given by:

fBitClock = fBRCLK / UCBRx

If UCBRx = 0, fBitClock = fBRCLK

Even UCBRx settings result in even divisions and, thus, generate a bit clock with a 50/50 duty cycle.

Odd UCBRx settings result in odd divisions. In this case, the high phase of the bit clock is one BRCLK cycle longer than the low phase. When UCBRx = 0, no division is applied to BRCLK, and the bit clock equals BRCLK.

Serial Clock Polarity and Phase

The polarity and phase of UCxCLK are independently configured via the UCCKPL and UCCKPH control bits of the USCI. Timing for each case is shown in Figure

Using the SPI Mode With Low-Power Modes

The USCI module provides automatic clock activation for use with low-power modes. When the USCI clock source is inactive because the device is in a low-power mode, the USCI module automatically activates it when needed, regardless of the control-bit settings for the clock source. The clock remains active until the USCI module returns to its idle condition. After the USCI module returns to the idle condition, control of the clock source reverts to the settings of its control bits. In SPI slave mode, no internal clock source is required because the clock is provided by the external master. It is possible to operate the USCI in SPI slave mode while the

device is in LPM4 and all clock sources are disabled. The receive or transmit interrupt can wake up the CPU from any low-power mode.

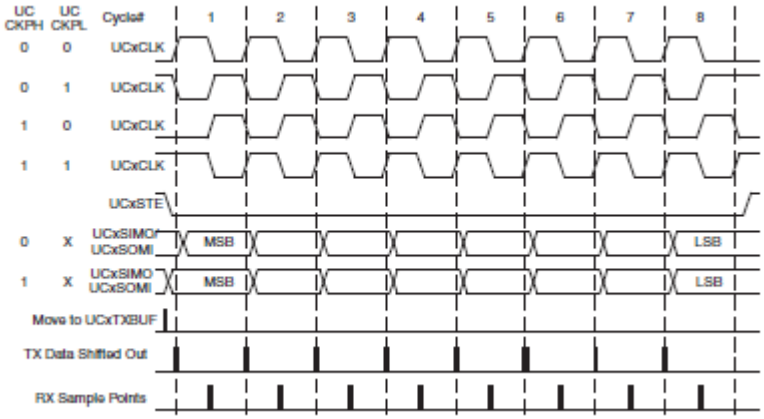


Figure 37-4. USCI SPI Timing With UCMSB = 1

USCI Interrupts in SPI Mode

The USCI has only one interrupt vector that is shared for transmission and for reception. USCI_Ax and USC_Bx do not share the same interrupt vector.

SPI Transmit Interrupt Operation

The UCTXIFG interrupt flag is set by the transmitter to indicate that UCxTXBUF is ready to accept another character. An interrupt request is generated if UCTXIE and GIE are also set. UCTXIFG is automatically reset if a character is written to UCxTXBUF. UCTXIFG is set after a PUC or when UCSWRST = 1. UCTXIE is reset after a PUC or when UCSWRST = 1.

SPI Receive Interrupt Operation

The UCRXIFG interrupt flag is set each time a character is received and loaded into UCxRXBUF. An interrupt request is generated if UCRXIE and GIE are also set. UCRXIFG and UCRXIE are reset by a system reset PUC signal or when UCSWRST = 1. UCRXIFG is automatically reset when UCxRXBUF is read.

UCxIV, Interrupt Vector Generator

The USCI interrupt flags are prioritized and combined to source a single interrupt vector. The interrupt vector register UCxIV is used to determine which flag requested an interrupt. The highest-priority enabled interrupt generates a number in the UCxIV register that can be evaluated or added to the program counter (PC) to automatically enter the appropriate software routine. Disabled interrupts do not affect the UCxIV value. Any access, read or write, of the UCxIV register automatically resets the highest-pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt.

Universal Serial Communication Interface (USCI) Overview: I2C Mode

The USCI modules support multiple serial communication modes. Different USCI modules support different modes. Each different USCI module is named with a different letter. For example, USCI_A is different from USCI_B, etc. If more than one identical USCI module is implemented on one device, those modules are named with incrementing numbers. For example, if one device has two USCI_A modules, they are named USCI_A0 and USCI_A1. See the device-specific data sheet to determine which USCI modules, if any, are implemented on each device.

USCI_Ax modules support:

- UART mode
- Pulse shaping for IrDA communications
- Automatic baud-rate detection for LIN communications
- SPI mode

USCI_Bx modules support:

- I2C mode
- SPI mode

USCI Introduction – I2C Mode

In I2C mode, the USCI module provides an interface between the device and I2C-compatible devices connected by the two-wire I2C serial bus. External components attached to the I2C bus serially transmit and/or receive serial data to/from the USCI module through the 2-wire I2C interface.

The I2C mode features include:

- Compliance to the Philips Semiconductor I2C specification v2.1
- 7-bit and 10-bit device addressing modes
- General call• START/RESTART/STOP
- Multi-master transmitter/receiver mode• Slave receiver/transmitter mode
- Standard mode up to 100 kbps and fast mode up to 400 kbps support
- Programmable UCxCLK frequency in master mode• Designed for low power
- Slave receiver START detection for auto wake up from LPMx modes (wake up from LPMx.5 is not supported)• Slave operation in LPM4

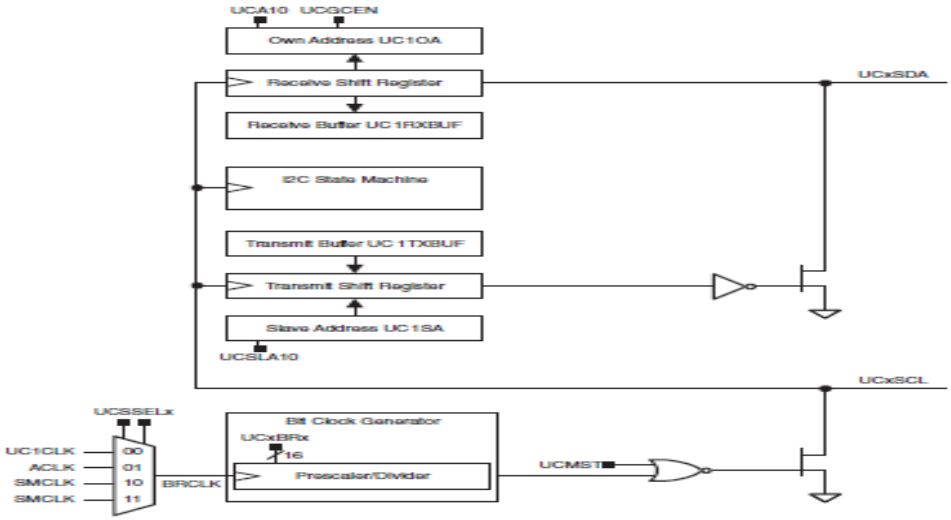


Figure 38-1. USCI Block Diagram – I²C Mode

USCI Operation – I2C Mode

The I2C mode supports any slave or master I2C-compatible device. Figure 38-2 shows an example of an I2C bus. Each I2C device is recognized by a unique address and can operate as either a transmitter or a receiver. A device connected to the I2C bus can be considered as the master or the slave when performing data transfers. A master initiates a data transfer and generates the clock signal SCL. Any device addressed by a master is considered a slave. I2C data

is communicated using the serial data (SDA) pin and the serial clock (SCL) pin. Both SDA and SCL are bidirectional and must be connected to a positive supply voltage using a pullup resistor.

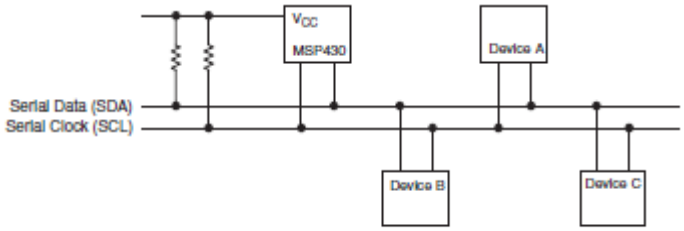


Figure 38-2. I²C Bus Connection Diagram

USCI Initialization and Reset

The USCI is reset by a PUC or by setting the UCSWRST bit. After a PUC, the UCSWRST bit is automatically set, keeping the USCI in a reset condition. To select I2C operation, the UCMODEx bits must be set to 11. After module initialization, it is ready for transmit or receive operation. Clearing UCSWRST releases the USCI for operation. To avoid unpredictable behavior, configure or reconfigure the USCI module only when UCSWRST is set. Setting UCSWRST in I2C mode has the following effects:

- I2C communication stops.
- SDA and SCL are high impedance.
- UCBxI2CSTAT, bits 6–0 are cleared.
- Registers UCBxIE and UCBxIFG are cleared.
- All other bits and register remain unchanged.

I2C Serial Data

One clock pulse is generated by the master device for each data bit transferred. The I2C mode operates with byte data. Data is transferred MSB first as shown in Figure 38-3. The first byte after a START condition consists of a 7-bit slave address and the R/W bit. When R/W = 0, the master transmits data to a slave. When R/W = 1, the master receives data from a slave. The ACK bit is sent from the receiver after each byte on the ninth SCL clock.

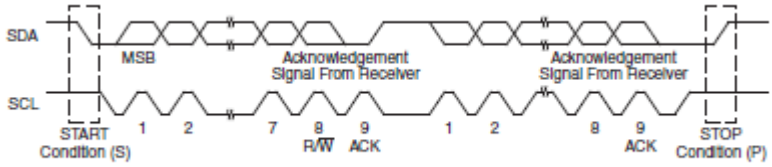


Figure 38-3. I²C Module Data Transfer

START and STOP conditions are generated by the master and are shown in Figure 38-3. A START condition is a high-to-low transition on the SDA line while SCL is high. A STOP condition is a low-to-high transition on the SDA line while SCL is high. The bus busy bit, UCBBUSY, is set after a START and cleared after a STOP. Data on SDA must be stable during the high period of SCL (see Figure 38-4). The high and low state of SDA can only change when SCL is low, otherwise START or STOP conditions are generated.

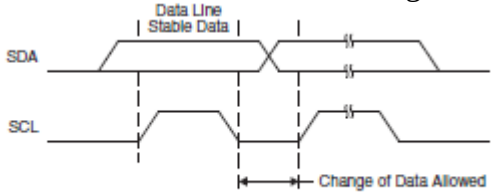


Figure 38-4. Bit Transfer on I²C Bus

I2C Addressing Modes

The I2C mode supports 7-bit and 10-bit addressing modes.

7-Bit Addressing

In the 7-bit addressing format (see Figure 38-5), the first byte is the 7-bit slave address and the R/W bit. The ACK bit is sent from the receiver after each byte.

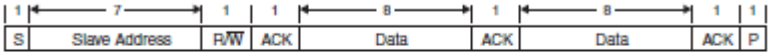


Figure 38-5. I²C Module 7-Bit Addressing Format

10-Bit Addressing

In the 10-bit addressing format (see Figure 38-6), the first byte is made up of 11110b plus the two MSBs of the 10-bit slave address and the R/W bit. The ACK bit is sent from the receiver after each byte. Them next byte is the remaining eight bits of the 10-bit slave address, followed by the ACK bit and the 8-bit data. See I2C Slave 10-bit Addressing Mode and I2C Master 10-bit Addressing Mode for details how to use the 10-bit addressing mode with the USCI module.

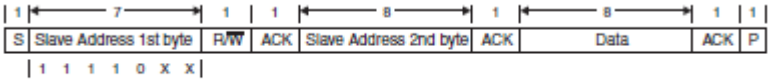


Figure 38-6. I²C Module 10-Bit Addressing Format

Repeated Start Conditions

The direction of data flow on SDA can be changed by the master, without first stopping a transfer, by issuing a repeated START condition. This is called a RESTART. After a RESTART is issued, the slave address is again sent out with the new data direction specified by the R/W bit. The RESTART condition is shown in Figure

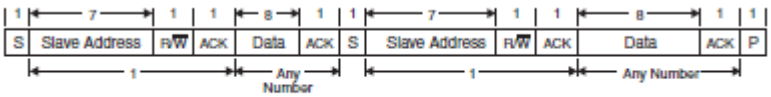


Figure 38-7. I²C Module Addressing Format With Repeated START Condition

I2C Module Operating Modes

In I2C mode, the USCI module can operate in master transmitter, master receiver, slave transmitter, or slave receiver mode. The modes are discussed in the following sections. Time lines are used to illustrate the modes.Figure 38-8 shows how to interpret the time-line figures. Data transmitted by the master is represented by grey rectangles; data transmitted by the slave is represented by white rectangles. Data transmitted by the USCI module, either as master or slave, is shown by rectangles that are taller than the others. Actions taken by the USCI module are shown in grey rectangles with an arrow indicating where in the data stream the action occurs. Actions that must be handled with software are indicated with white rectangles with an arrow pointing to where in the data stream the action must take place.

Slave Mode

The USCI module is configured as an I2C slave by selecting the I2C mode with UCMODEx = 11 and UCSYNC = 1 and clearing the UCMST bit. Initially, the USCI module must to be configured in receiver mode by clearing the UCTR bit to receive thenI2C address. Afterwards, transmit and receive operations are controlled automatically, depending on the R/W bit received together with the slave address.

The USCI slave address is programmed with the UCBxI2COA register. When UCA10 = 0, 7-bit addressing is selected. When UCA10 = 1, 10-bit addressing is selected. The UCGCEN bit selects if the slave responds to a general call. When a START condition is detected on the

bus, the USCI module receives the transmitted address and compare it against its own address stored in UCBxI2COA. The UCSTTIFG flag is set when address received matches the USCI slave address.

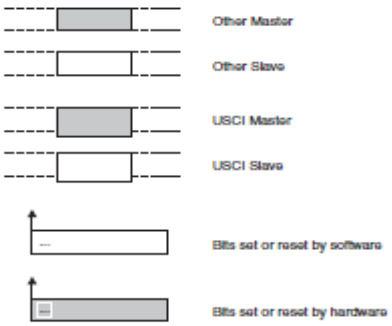


Figure 38-8. I²C Time-Line Legend

I2C Slave Transmitter Mode

Slave transmitter mode is entered when the slave address transmitted by the master is identical to its own address with a set R/W bit. The slave transmitter shifts the serial data out on SDA with the clock pulses that are generated by the master device. The slave device does not generate the clock, but it does hold SCL low while intervention of the CPU is required after a byte has been transmitted.

If the master requests data from the slave, the USCI module is automatically configured as a transmitter and UCTR and UCTXIFG become set. The SCL line is held low until the first data to be sent is written into the transmit buffer UCBxTXBUF. Then the address is acknowledged, the UCSTTIFG flag is cleared, and the data is transmitted. As soon as the data is transferred into the shift register, the UCTXIFG is set again. After the data is acknowledged by the master, the next data byte written into UCBxTXBUF is transmitted or, if the buffer is empty, the bus is stalled during the acknowledge cycle by holding SCL low until new data is written into UCBxTXBUF. If the master sends a NACK succeeded by a STOP condition, the UCSTPIFG flag is set. If the NACK is succeeded by a repeated START condition, the USCI I2C state machine returns to its address-reception state.

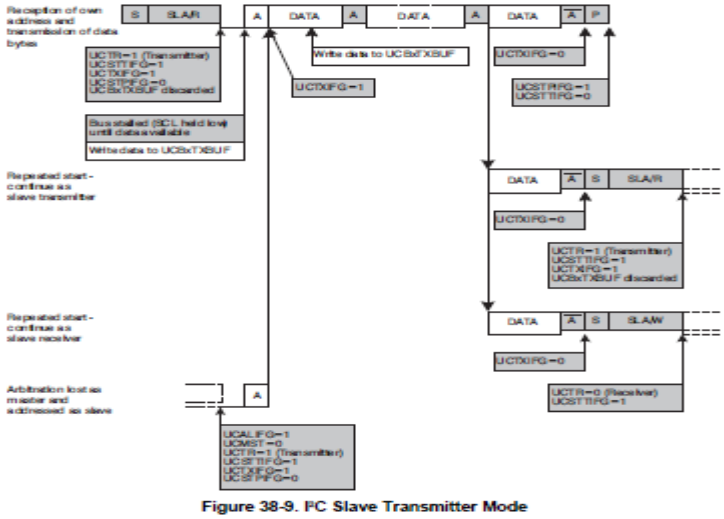


Figure 38-9. I²C Slave Transmitter Mode

I2C Slave Receiver Mode

Slave receiver mode is entered when the slave address transmitted by the master is identical to its own address and a cleared R/W bit is received. In slave receiver mode, serial data

bits received on SDA are shifted in with the clock pulses that are generated by the master device. The slave device does not generate the clock, but it can hold SCL low if intervention of the CPU is required after a byte has been received. If the slave should receive data from the master, the USCI module is automatically configured as a receiver and UCTR is cleared. After the first data byte is received, the receive interrupt flag UCRXIFG is set. The USCI module automatically acknowledges the received data and can receive the next data byte.

If the previous data was not read from the receive buffer UCBxRXBUF at the end of a reception, the bus is stalled by holding SCL low. As soon as UCBxRXBUF is read, the new data is transferred into UCBxRXBUF, an acknowledge is sent to the master, and the next data can be received. Setting the UCTXNACK bit causes a NACK to be transmitted to the master during the next acknowledgment cycle. A NACK is sent even if UCBxRXBUF is not ready to receive the latest data. If the UCTXNACK bit is set while SCL is held low, the bus is released, a NACK is transmitted immediately, and UCBxRXBUF is loaded with the last received data. Because the previous data was not read, that data is lost. To avoid loss of data, the UCBxRXBUF must be read before UCTXNACK is set**.** When the master generates a STOP condition, the UCSTPIFG flag is set. If the master generates a repeated START condition, the USCI I2C state machine returns to its address reception state.

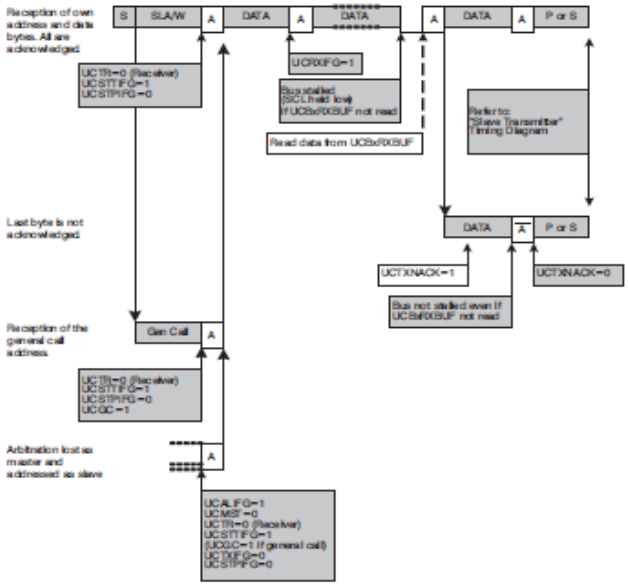


Figure 38-10. I²C Slave Receiver Mode

I2C Slave 10-Bit Addressing Mode

The 10-bit addressing mode is selected when UCA10 = 1 and is as shown in Figure 38-11. In 10-bit addressing mode, the slave is in receive mode after the full address is received. The USCI module indicates this by setting the UCSTTIFG flag while the UCTR bit is cleared. To switch the slave into transmitter mode, the master sends a repeated START condition together with the first byte of the address but with the R/W bit set. This sets the UCSTTIFG flag if it was previously cleared by software, and the USCI modules switches to transmitter mode with UCTR = 1.

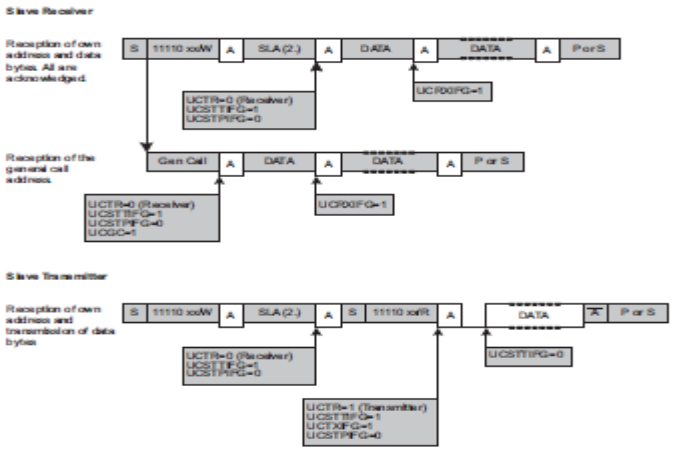


Figure 38-11. I²C Slave 10-Bit Addressing Mode

Master Mode

The USCI module is configured as an I2C master by selecting the I2C mode with UCMODEx = 11 and UCSYNC = 1 and setting the UCMST bit. When the master is part of a multi-master system, UCMM must be set and its own address must be programmed into the UCBxI2COA register. When UCA10 = 0, 7-bit addressing is selected. When UCA10 = 1, 10-bit addressing is selected. The UCGCEN bit selects if the USCI module responds to a general call.

I2C Master Transmitter Mode

After initialization, master transmitter mode is initiated by writing the desired slave address to the UCBxI2CSA register, selecting the size of the slave address with the UCSLA10 bit, setting UCTR for transmitter mode, and setting UCTXSTT to generate a START condition. The USCI module checks if the bus is available, generates the START condition, and transmits the slave address. The UCTXIFG bit is set when the START condition is generated and the first data to be transmitted can be written into UCBxTXBUF. As soon as the slave acknowledges the address, the UCTXSTT bit is cleared.

The data written into UCBxTXBUF is transmitted if arbitration is not lost during transmission of the slave address. UCTXIFG is set again as soon as the data is transferred from the buffer into the shift register. If there is no data loaded to UCBxTXBUF before the acknowledge cycle, the bus is held during the acknowledge cycle with SCL low until data is written into UCBxTXBUF. Data is transmitted or the bus is held, as long as the UCTXSTP bit or UCTXSTT bit is not set.

Setting UCTXSTP generates a STOP condition after the next acknowledge from the slave. If UCTXSTP is set during the transmission of the slave's address or while the USCI module waits for data to be written into UCBxTXBUF, a STOP condition is generated, even if no data was transmitted to the slave. When transmitting a single byte of data, the UCTXSTP bit must be set while the byte is being transmitted or anytime after transmission begins, without writing new data into UCBxTXBUF. Otherwise, only the address is transmitted. When the data is transferred from the buffer to the shift register, UCTXIFG is set, indicating data transmission has begun, and the UCTXSTP bit may be set.

Setting UCTXSTT generates a repeated START condition. In this case, UCTR may be set or cleared to configure transmitter or receiver, and a different slave address may be written into UCBxI2CSA if desired. If the slave does not acknowledge the transmitted data, the not-acknowledge interrupt flag UCNACKIFG is set. The master must react with either a STOP condition or a repeated START condition. If data was already written into UCBxTXBUF, it is

discarded. If this data should be transmitted after a repeated START, it must be written into UCBxTXBUF again. Any set UCTXSTT is also discarded. To trigger a repeated START, UCTXSTT must be set again.

I2C Master Receiver Mode

After initialization, master receiver mode is initiated by writing the desired slave address to the UCBxI2CSA register, selecting the size of the slave address with the UCSLA10 bit, clearing UCTR for receiver mode, and setting UCTXSTT to generate a START condition.

The USCI module checks if the bus is available, generates the START condition, and transmits the slave address. As soon as the slave acknowledges the address, the UCTXSTT bit is cleared. After the acknowledge of the address from the slave, the first data byte from the slave is received and acknowledged and the UCRXIFG flag is set. Data is received from the slave, as long as UCTXSTP or

UCTXSTT is not set. If UCBxRXBUF is not read, the master holds the bus during reception of the last data bit and until the UCBxRXBUF is read. If the slave does not acknowledge the transmitted address, the not-acknowledge interrupt flag

UCNACKIFG is set. The master must react with either a STOP condition or a repeated START condition. Setting the UCTXSTP bit generates a STOP condition. After setting UCTXSTP, a NACK followed by a STOP condition is generated after reception of the data from the slave, or immediately if the USCI module is currently waiting for UCBxRXBUF to be read. If a master wants to receive a single byte only, the UCTXSTP bit must be set while the byte is being received. For this case, the UCTXSTT may be polled to determine when it is cleared:

```
BIS.B #UCTXSTT, &UCB0CTL1 ;Transmit START cond.
POLL_STT BIT.B #UCTXSTT, &UCB0CTL1 ;Poll UCTXSTT bit
JC POLL_STT ;When cleared,
BIS.B #UCTXSTP, &UCB0CTL1 ;transmit STOP cond.
```

Setting UCTXSTT generates a repeated START condition. In this case, UCTR may be set or cleared to configure transmitter or receiver, and a different slave address may be written into UCBxI2CSA if desired.

Arbitration

If two or more master transmitters simultaneously start a transmission on the bus, an arbitration procedure is invoked. Figure 38-15 shows the arbitration procedure between two devices. The arbitration procedure

uses the data presented on SDA by the competing transmitters. The first master transmitter that generates a logic high is overruled by the opposing master generating a logic low. The arbitration procedure gives priority to the device that transmits the serial data stream with the lowest binary value. The master transmitter that lost arbitration switches to the slave receiver mode and sets the arbitration lost flag UCALIFG. If two or more devices send identical first bytes, arbitration continues on the subsequent bytes.

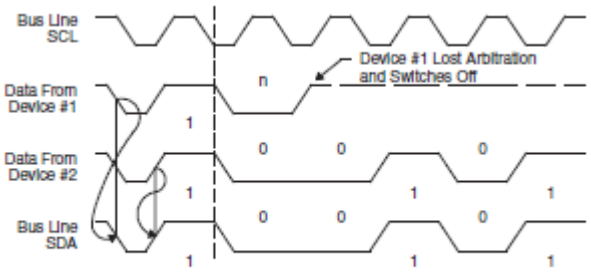


Figure 38-15. Arbitration Procedure Between Two Master Transmitters

If the arbitration procedure is in progress when a repeated START condition or STOP condition is transmitted on SDA, the master transmitters involved in arbitration must send the repeated START condition or STOP condition at the same position in the format frame. Arbitration is not allowed between:

- A repeated START condition and a data bit
- A STOP condition and a data bit
- A repeated START condition and a STOP condition

I2C Clock Generation and Synchronization

The I2C clock SCL is provided by the master on the I2C bus. When the USCI is in master mode, BITCLK is provided by the USCI bit clock generator and the clock source is selected with the UCSSELx bits. In slave mode, the bit clock generator is not used and the UCSSELx bits are don't care.n The 16-bit value of UCBRx in registers UCBxBR1 and UCBxBR0 is the division factor of the USCI clock source, BRCLK. The maximum bit clock that can be used in single master mode is fBRCLK/4. In multi-master mode, the maximum bit clock is fBRCLK/8. The BITCLK frequency is given by:

fBitClock = fBRCLK/UCBRx

The minimum high and low periods of the generated SCL are:

tLOW,MIN = tHIGH,MIN = (UCBRx/2)/fBRCLK when UCBRx is even

tLOW,MIN = tHIGH,MIN = ((UCBRx – 1)/2)/fBRCLK when UCBRx is odd

The USCI clock source frequency and the prescaler setting UCBRx must to be chosen such that themminimum low and high period times of the I2C specification are met.

During the arbitration procedure the clocks from the different masters must be synchronized. A device that first generates a low period on SCL overrules the other devices, forcing them to start their own low periods. SCL is then held low by the device with the longest low period. The other devices must wait for SCL to be released before starting their high periods. Figure 38-16 shows the clock synchronization. This allows a slow slave to slow down a fast master.

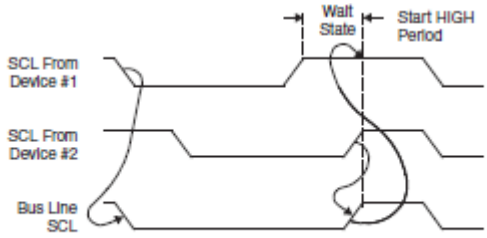


Figure 38-16. Synchronization of Two I²C Clock Generators During Arbitration

Clock Stretching

The USCI module supports clock stretching and also makes use of this feature as described in the Operation Mode sections. The UCSCLLOW bit can be used to observe if

another device pulls SCL low while the USCI module already released SCL due to the following conditions:

- USCI is acting as master and a connected slave drives SCL low.
- USCI is acting as master and another master drives SCL low during arbitration.

The UCSCLLOW bit is also active if the USCI holds SCL low because it is waiting as transmitter for data being written into UCBxTXBUF or as receiver for the data being read from UCBxRXBUF.

The UCSCLLOW bit might get set for a short time with each rising SCL edge because the logic observes the external SCL and compares it to the internally generated SCL.

Using the USCI Module in I2C Mode With Low-Power Modes

The USCI module provides automatic clock activation for use with low-power modes. When the USCI clock source is inactive because the device is in a low-power mode, the USCI module automatically activates it when needed, regardless of the control-bit settings for the clock source. The clock remains active until the USCI module returns to its idle condition. After the USCI module returns to the idle condition, control of the clock source reverts to the settings of its control bits. In I2C slave mode, no internal clock source is required because the clock is provided by the external master. It is possible to operate the USCI in I2C slave mode while the device is in LPM4 and all internal clock sources are disabled. The receive or transmit interrupts can wake up the CPU from any low-power mode.

USCI Interrupts in I2C Mode

The USCI has only one interrupt vector that is shared for transmission, reception, and the state change. USCI_Ax and USC_Bx do not share the same interrupt vector. Each interrupt flag has its own interrupt enable bit. When an interrupt is enabled and the GIE bit is set, the interrupt flag generates an interrupt request. DMA transfers are controlled by the UCTXIFG and UCRXIFG flags on devices with a DMA controller.

I2C Transmit Interrupt Operation

The UCTXIFG interrupt flag is set by the transmitter to indicate that UCBxTXBUF is ready to accept another character. An interrupt request is generated if UCTXIE and GIE are also set. UCTXIFG is automatically reset if a character is written to UCBxTXBUF or if a NACK is received. UCTXIFG is set when UCSWRST = 1 and the I2C mode is selected. UCTXIE is reset after a PUC or when UCSWRST = 1.

I2C Receive Interrupt Operation

The UCRXIFG interrupt flag is set when a character is received and loaded into UCBxRXBUF. An interrupt request is generated if UCRXIE and GIE are also set. UCRXIFG and UCRXIE are reset after a

PUC signal or when UCSWRST = 1. UCRXIFG is automatically reset when UCxRXBUF is read.

I2C State Change Interrupt Operation

Table 38-1. I²C State Change Interrupt Flags

Interrupt Flag	Interrupt Condition
UCALIFG	Arbitration-lost. Arbitration can be lost when two or more transmitters start a transmission simultaneously, or when the USCI operates as master but is addressed as a slave by another master in the system. The UCALIFG flag is set when arbitration is lost. When UCALIFG is set, the UCMST bit is cleared and the I²C controller becomes a slave.
UCNACKIFG	Not-acknowledge interrupt. This flag is set when an acknowledge is expected but is not received. UCNACKIFG is automatically cleared when a START condition is received.
UCSTTIFG	START condition detected interrupt. This flag is set when the I²C module detects a START condition together with its own address while in slave mode. UCSTTIFG is used in slave mode only and is automatically cleared when a STOP condition is received.
UCSTPIFG	STOP condition detected interrupt. This flag is set when the I²C module detects a STOP condition while in slave mode. UCSTPIFG is used in slave mode only and is automatically cleared when a START condition is received.

UCBxIV, Interrupt Vector Generator

The USCI interrupt flags are prioritized and combined to source a single interrupt vector. The interrupt vector register UCBxIV is used to determine which flag requested an interrupt. The highest-priority enabled interrupt generates a number in the UCBxIV register that can be evaluated or added to the PC to automatically enter the appropriate software routine. Disabled interrupts do not affect the UCBxIV value. Any access, read or write, of the UCBxIV register automatically resets the highest-pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt

A Low-Power Battery-Less Wireless Temperature and Humidity Sensor for the TI PaLFI Device:

Introduction

Several applications require hermetically sealed environments, where physical parameter measurements such as temperature, humidity, or pressure are measured and, for several reasons, a battery-less operation is required. In such applications, a wireless data and power transfer is necessary. This application report shows how to implement an easy-to-use low-power wireless humidity and temperature sensor comprising a SHT21 from Sensrion, a MSP430F2274 microcontroller, and a TMS37157 PaLFI (passive low-frequency interface). The complete power for the wireless sensor and the MSP430F2274 is provided by the RFID base station (ADR2) reader included in the eZ430-TMS37157 demo kit.

The application is divided in four steps:

- Charge phase: Generate an RF field of 134.2 kHz from the ADR2 reader to the wireless sensor module to charge the power capacitor.
- Downlink phase: Send command or instruction to wireless sensor to start measurement.
- Measurement and recharge phase: Trigger measurement of temperature, recharge the power capacitor on the sensor device, and trigger humidity measurement.
- Uplink phase: Send measurement results via RF interface (134.2 kHz) back to ADR2 reader.

Hardware Description

Device Specifications

MSP430F2274

The MSP430F2274 is a 16-bit microcontroller from the 2xx family of the ultra-low-power MSP430™ family of devices from Texas Instruments.[2] The supply voltage for this

microcontroller ranges from 1.8 V to 3.6 V. The MCU is capable of operating at frequencies up to 16 MHz. It also has an internal very-low-power low-frequency oscillator (VLO) that operates at 12 kHz at room temperature. It has two 16-bit timers (Timer_A and Timer_B), each with three capture/compare registers. An integrated 10-bit analog-to-digital converter (ADC10) supports conversion rates of up to 200 ksps. The current consumption of 0.7 mA during standby mode (LPM3) and 250 mA during active mode makes it an excellent choice for battery-powered applications.

TMS37157 PaLFI

The TMS37157 PaLFI is a dual interface passive RFID product from Texas Instruments. The device can communicate via the RF and the SPI (wired) interfaces. It offers 121 bytes of programmable EEPROM memory. The complete memory can be altered through the wireless interface, if the communication/read distances between the reader antenna and the PaLFI antenna are less than 10 cm to 30 cm (depending on the antenna geometry and reader power). For wireless memory access, a battery supply is not required. A microcontroller with a SPI interface has access to the entire memory through the 3-wire SPI interface of the TMS37157. In addition, the TMS37157 can pass through received data from the wireless interface to the microcontroller and send data from the microcontroller back over the wireless interface. If the TMS37157 is connected to a battery, it offers a battery charge function and a battery check function without waking the microcontroller. If connected to a battery, the TMS37157 has an ultralow power consumption of about 60 nA in standby mode and about 70 μA in active mode. The PaLFI can completely switch off the microcontroller, resulting in an ultralow power consumption of the complete system.

SHT21 Humidity and Temperature Sensor

The extremely small SHT21 digital humidity and temperature sensor integrates sensors, calibration memory, and digital interface on 3x3 mm footprint. This results in cost savings, because no additional components are need and no investments in calibration equipment or process are necessary. One-chip integration allows for lowest power consumption, thus enabling energy harvesting and passive RFID solutions. The complete over-molding of the sensor chip, with the exception of the humidity sensor area, protects the reflow solderable sensor against external impact and leads to an excellent long term stability.

About Sensirion

The Swiss sensor manufacturer Sensirion AG is a leading international supplier of CMOS-based sensor components and systems. Its range of high-quality products includes humidity and temperature sensors, mass flow meters and controllers, gas and liquid flow sensors, and differential pressure sensors. Sensirion supports its international OEM customers with tailor-made sensor system solutions for a wide variety of applications. Among others, they include analytical instruments, consumer goods, and applications in the medical technology, automotive and HVAC sectors. Sensirion products are distinguished by their use of patented CMOSens® technology. This enables customers to benefit from intelligent system integration,

Interfaces from MSP430F2274 to TMS37157 and SHT21

Interface Between MSP430F2274 and TMS37157 PaLFI

Figure 1 shows the interface between MSP430F2274 and TMS37157. The TMS37157 is connected to the MSP430F2274 through a 3-wire SPI interface. To simplify communication between the MSP430F2274 and TMS37157, the BUSY pin of the TMS37157 is connected to the MSP430. The BUSY pin indicates the readiness of the TMS37157 to receive the next data byte from the MSP430F2274. The PUSH pin is used to wake up the PaLFI from standby mode so that

the MSP430F2274 can access the EEPROM of the PaLFI. CLKAM is used for the antenna automatic tune feature of the PaLFI target board.

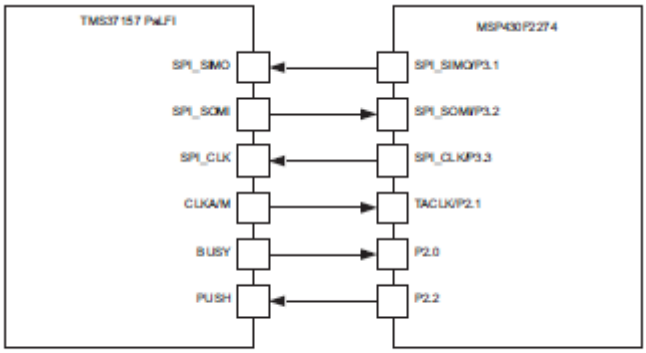


Figure 1. Block Diagram of Interface Between MSP430F2274 and TMS37157

Interface Between MSP430F2274 and SHT21

Figure 2 shows the interface between MSP430F2274 and SHT21. I2C is used to connect both devices. The MSP430F2274 contains two communication modules. One is used as UART connection to a host PC, the other one is used to communicate to the TMS37157. Therefore, the I2C interface has been implemented completely in software.

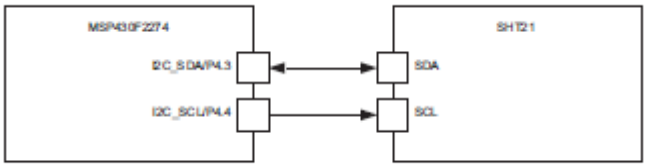


Figure 2. Block Diagram of Interface Between MSP430F2274 and SHT21

Hardware Changes to Original PaLFI Board

Several changes were made to the standard PaLFI board to implement the wireless sensor application. The most important change is to use an external DC/DC converter attached to VCL to generate a VBAT/VCC voltage out of the 134.2-kHz RF field. Figure 3 shows the basic principle of this circuit.

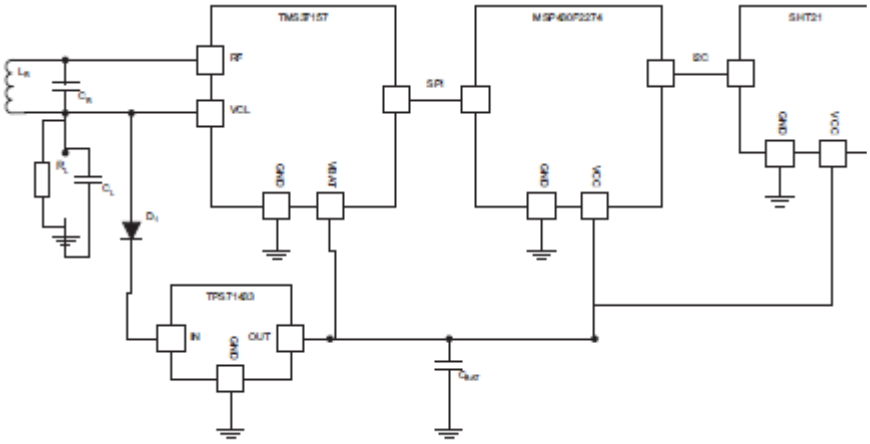


Figure 3. Principle Schematic of the Wireless Sensor

The input of the dc/dc converter TPS71433 is connected to VCL via diode D1. D1 prevents the resonance circuit (consisting of LR and CR) from any disturbances coming from the dc/dc converter. Capacitor CBAT stores the energy derived from the RF field.
Using an external dc/dc converter instead of the internal of the TMS37157 overcomes two issues. The first advantage of an external dc/dc converter is that it can provide higher output currents in comparison to the internal regulator (80 mA compared to 5 mA). The second advantage using an external regulator is the simpler flow for the application and the firmware (see Table 1).

Layout

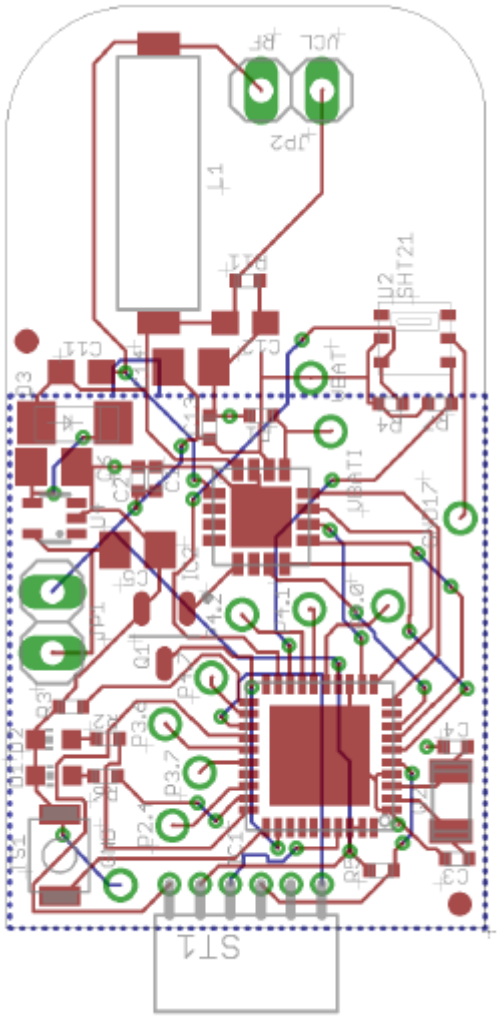


Figure 5. Eagle Layout of the Wireless Sensor

Software and Firmware Description

This section describes the Windows GUI and the MSP430F2274 firmware used for this application report. The Windows software used in this application report is based on the software that is supplied with the eZ430-TMS37157. An additional tab was inserted to control the

wireless sensor application. Prior to using this application report, make sure that you have the right firmware, corresponding to this application report, loaded onto your PaLFI Wireless Sensor target board.