LECTURE NOTES

ON

MICROPROCESSORS & MICROCONTROLLERS

(15A04607)

III B. Tech II Semester

Prepared by
Mr. N. KIRAN KUMAR, Asst. Professor
DEPT. OF ECE

Syllabus

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR
B. Tech III-II Sem. (ECE)
15A04607
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

UNIT-I

Introduction

Definition

The microprocessor is a multipurpose, programmable device(IC) that reads binary instructions from the memory and process the data according to the instructions and then produces the output.

Features of Microprocessor

Some of the features of any microprocessor are-

Cost effective: The microprocessor chips are available at low prices and results its low cost.

Size: The microprocessor is of small size chip, hence it is portable.

Low power consumption: Microprocessors are manufactured by using metal oxide semiconductor technology, which has low power consumption.

Versatility: The microprocessors are versatile as the same chips in number of applications can be used by configuring the software program.

Reality: The failure rate of an IC in microprocessors is very low, hence it is reliable.

Microprocessors Characteristics

These are the most important defining characteristics of a microprocessor −

- Clock speed
- Instruction set
- Word size

Clock Speed

Every microprocessor has an **internal clock** that regulates the speed at which it executes instructions and also synchronizes it with other components. The speed at which the microprocessor executes instructions is called **clock speed**. It determines the number of operations per second the processor can perform. Clock speeds are measured in MHz or GHz where 1 MHz means 1 million cycles per second whereas 1 GHz equals to 1 billion cycles per second. Here cycle refers to single electric signal cycle.

Currently microprocessors have clock speed in the range of 3 GHz, which is maximum that current technology can attain. Speeds more than this generate enough heat to damage the chip itself. To overcome this, manufacturers are using multiple processors working in parallel on a chip.

Instruction Set

Each microprocessor is designed to execute a specific group of operations. This group of operations is called an instruction set. This instruction set defines what the microprocessor can and cannot do. The programs can be written for microprocessor based on its instruction set.

Word Size

Number of bits that can be processed by a processor in a single instruction is called its **word size**. Word size determines the amount of RAM that can be accessed at one go and total number of pins on the microprocessor. Total number of input and output pins in turn determines the architecture of the microprocessor. It depends upon the width of internal data bus, registers, ALU, etc.

An 8-bit microprocessor can process 8-bit data at a time. The word size or word length ranges from 4 bits to 64 bits depending upon the type of the microcomputer.

8086 Microprocessor Features:

- The 8086 microprocessor is a 16 bit microprocessor.
- The term “16 bit” means that its arithmetic logic unit, internal registers and most of its instructions are designed to work with 16 bit binary words.
- The 8086 microprocessor has a 16 bit data bus.
- It can read data from or write data to memory or ports either 16 bits or 8 bits at a time.
- The 8086 microprocessor has a 20 bit address bus, so it can directly access 2^20 or 1,048,576 (1 MB) memory locations.
- The 8086 microprocessor can generate 16 bit I/O address; hence it can access 2^16 or 65536 ports.
- It performs the arithmetic and logical operations on bit, byte, word and decimal numbers including multiplication and division.
- The 8086 microprocessor is designed to operate in two modes, the minimum mode and the maximum mode.
- When only one CPU is used in the system, the 8086 microprocessor operates in the minimum mode.
- In multiprocessor system, 8086 microprocessor operates in the maximum mode.
- The 8086 microprocessor supports multiprogramming ability.

Architecture of the 8086 microprocessor

The 8086 microprocessor is divided into two functional units, the Bus Interface Unit (BIU) and the Execution Unit (EU). These two units can work simultaneously to increase system throughput. The throughput is a measure of number of instructions executed per unit time.

Execution Unit (EU)

The EU contains ALU, Control Unit and registers. This feature enables the EU to execute instructions and perform arithmetic and logical operations.

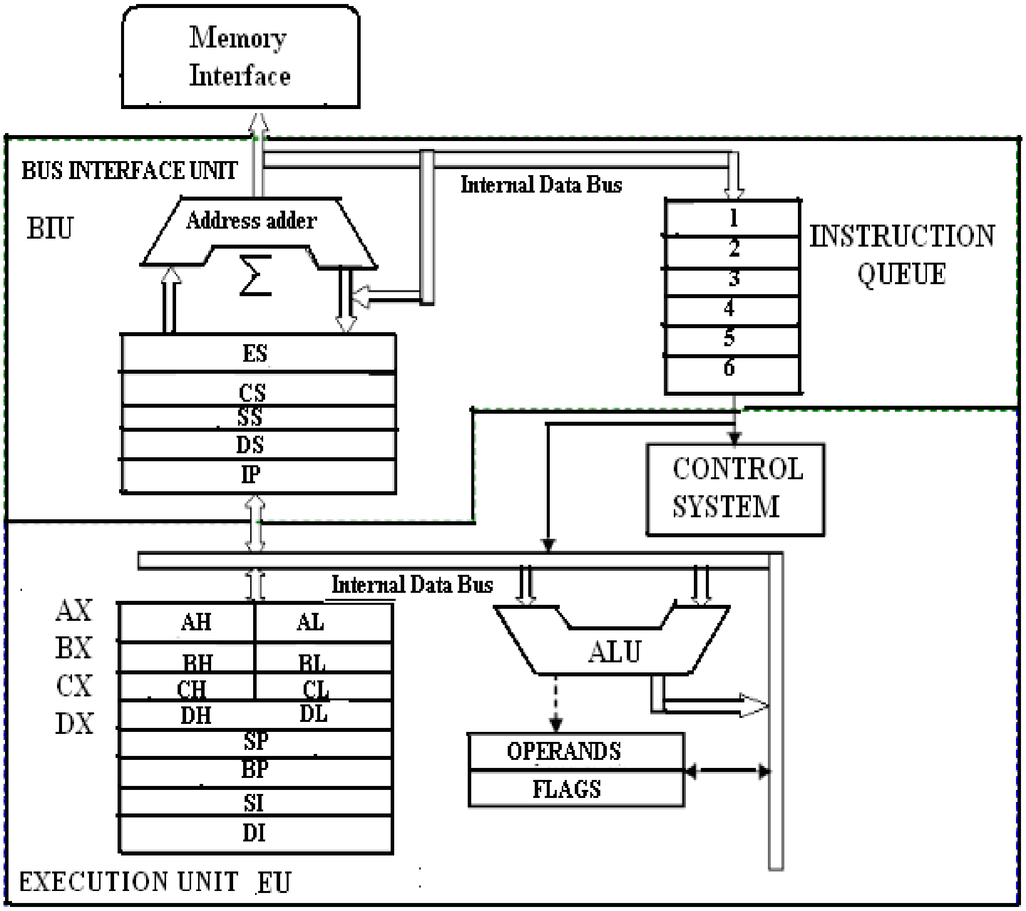
Functions of EU

- fetch the instructions
- decode the instructions
- execute the instructions

Components of EU

- General purpose registers
- Pointers and Index Registers
- Flag Registers

- Arithmetic Logic Unit (ALU)
- Instruction Decoder
- Control unit



General purpose registers:

Execution unit consists of four 16-bit registers. They are AX, BX, CX and DX registers. These 16 bit registers can be used as two 8-bit registers. The general purpose registers are either used for holding the data, variables and intermediate results temporarily or for other purpose like counter or for storing offset address for some particular addressing modes etc.

Pointer registers:

There are three pointer registers. They are BP, SP and IP. These registers are 16 bit registers. The main purpose of these registers is to store the offset address.

Index registers:

There are two index registers available in 8086. They are SI and DI. These registers are 16 bit registers. The main purpose of these registers is to store the source and destination addresses.

Arithmetic and logical unit

Arithmetic and logical unit performs all arithmetic and logical operations. The results of ALU are stored in the accumulator.

Instruction decoder

The instructions from pre fetch instruction queue are taken and decoded in execution unit and then the control signals are generated in the control circuit and then the instructions are executed.

Control unit

Execution unit contains control circuit for controlling the internal operations. The control signals are generated in the control unit.

Bus Interface Unit (BIU)

The BIU sends out the address, fetches the instructions from the memory; read data from ports, and memory and writes data to the ports and memory. It provides a 16 bit bidirectional data bus and 20 bit address bus. The BIU is responsible for performing all external bus operations for execution unit.

Functions of BIU

- Sends address to the Memory or I/O unit.
- Fetches instructions from Memory.
- Reads data from ports or Memory.
- Writes data on ports or Memory.
- Supports instruction queuing.
- Provides address relocation facility.

Components of BIU

- Instruction Queue.
- Segment registers.
- Instruction Pointers.
- Address summer.

Instruction queue

BIU contains an instruction queue. The size of instruction queue is 6 bytes and it follows FIFO order. When the EU executes instructions, the BIU gets up to 6 bytes of the next instruction and stores them in the instruction queue and this process is called instruction pre fetch. This is a process to speed up the processor. These instructions are used in pipelining.

Pipelining is a process of fetching the next instruction while current instruction is executing.

Segment Registers

In order to access the segments there are segment registers. The four segment register in 8086 are

- Code segment register (CS)
- Data segment register (DS)
- Extra segment register (ES)
- Stack segment register (SS)

Code segment register (CS):

This is used for addressing memory location in the code segment of the memory, where the executable program is stored. Code segment register contains the base address or starting address of code segment.

Data segment register (DS):

This register points to the data segment of the memory where the data is stored. Data segment register contains the base address or starting address of data segment.

Extra Segment Register (ES):

This register also refers to a segment in the memory which is another data segment in the memory. Extra segment register contains the base address or starting address of extra segment.

Stack Segment Register (SS):

This is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data. Stack segment register contains the base address or starting address of stack segment.

Instruction pointer (IP):

Instruction pointer is a pointer register. Instruction pointer is a 16 bit register which is used to store the offset address of the next instruction that is to be executed.

Address summer or Adder:

Adder is used to generate 20 bit address using two 16 bit addresses. Each address in the physical memory is called physical address. To access an operand (either data or the instruction) from a particular segment of the memory, the 8086 has to first calculate the physical address of that operand.

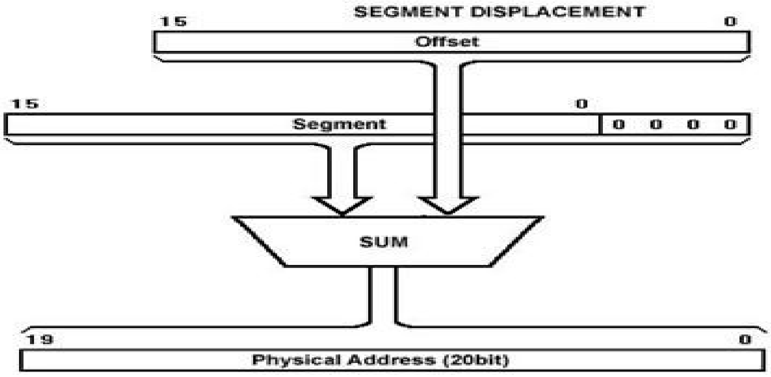
To perform this task, the 8086 adds the base address of the corresponding segment with the offset address, which may be the content of a register, 8 bit or 16 bit displacement in the given instruction, or the combination of both, depending upon the addressing modes used by the instruction.

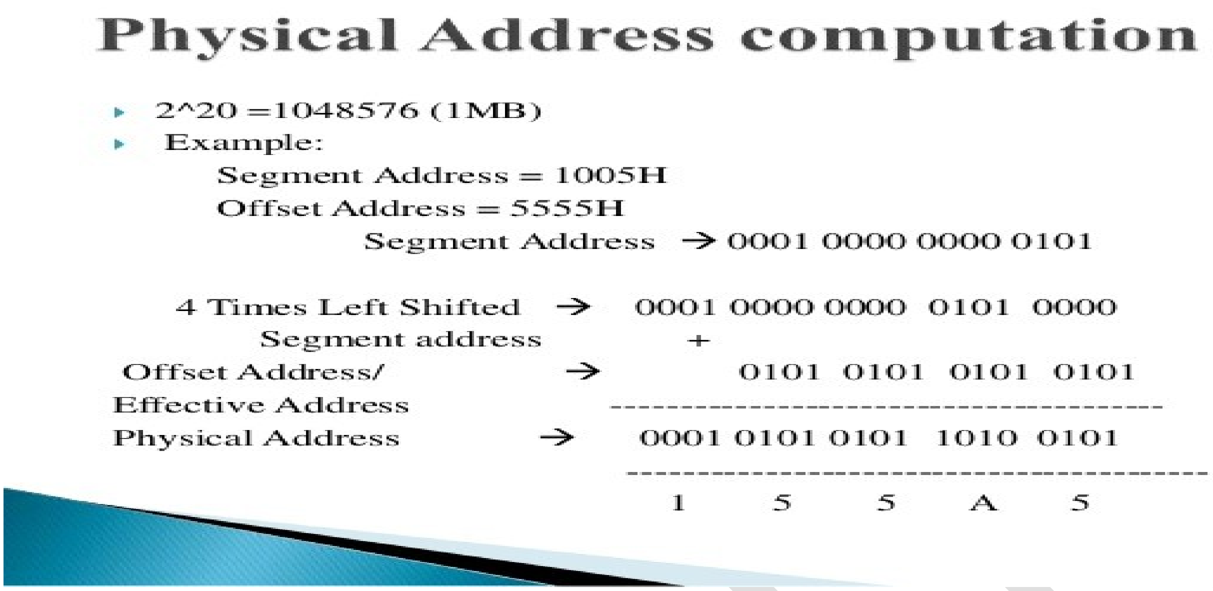
In order to calculate the physical address, two types of addresses are required. They are

- Segment address
- Offset address

Segment registers contains the segment address which is nothing but base address or starting address of particular segment. The segment address can be obtained by appending four zeros to the right portion of the content of particular segment register, which is same as appending the hexadecimal digit 0. It is also equivalent to shifting the content of the segment register left by four bit positions.

Offset address is defined as the distance of required memory locations in segment from the base address. Offset address may be the content of a register, 8 bit or 16 bit displacement in the given instruction, or the combination of both. Offset address is also known as effective address.





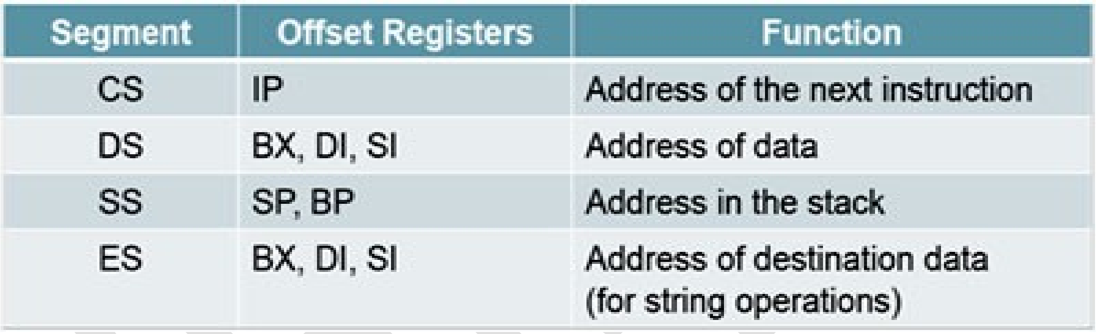
Physical Address computation

- 2^20 =1048576 (1MB)
- Example:

```
          Segment Address = 1005H
          Offset Address = 5555H
                    Segment Address   → 0001 0000 0000 0101

   4 Times Left Shifted   →    0001 0000 0000  0101  0000
         Segment address            +
 Offset Address/            →        0101  0101  0101  0101
 Effective Address              -------------------------------------
 Physical Address          →     0001 0101 0101  1010  0101
                                 ----------------------------------------
                                   1     5     5     A     5
```

The default offset addresses for particular segments are as shown in figure.



Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

Register organization:

In 8086 microprocessor there are two types of registers. They are

- General purpose registers
- Special purpose registers

General purpose registers

Execution unit consists of four 16-bit general purpose registers. They are AX, BX, CX and DX registers. These 16 bit registers can be used as two 8-bit registers. The general purpose registers are either used for holding the data, variables and intermediate results temporarily or for other purpose like counter or for storing offset address for some particular addressing modes etc.

AX Register

AX register is used as accumulator. Accumulator register consists of two 8- bit registers AL and AH, which can be combined together and used as a 16-bit register.

AL contains lower order data and AH contains higher order data.

Accumulator can be used for I/O operations and used in multiply and division operations.

BX Register

BX register is known as Base register. Base register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register. BX register is used to hold offset address for forming physical address in case of addressing modes.

CX Register

CX register is known as Count Register. Count register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register. CX register is used to hold the count value while executing the repeated string instructions and loop instructions. CL is used to hold the count value while executing the shift /rotate instructions. The count value indicates the number of times the same code has to be executed when LOOP instruction is used, and the number of times the data has to be shifted when rotate/ shift instruction is used.

DX Register

DX register is known as Data Register. Data register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register. Data register can be used as a port number in I/O operations when executing IN/OUT instructions. Data register is used to hold a part of result during multiplication operation and a part of dividend before a division operation.

Special purpose registers:

There are three types of special purpose registers. They are pointer registers, index registers and flag registers.

Pointer Registers

The 16 bit pointers registers in 8086 microprocessor are IP, BP, and SP. Among pointer registers SP and BP registers are located in EU and IP register is located in BIU. The main purpose of pointer registers is to hold the offset address. The pointer registers can also be used as general purpose registers.

BP Register

BP register is known as base pointer. Base pointer is a 16 bit register and used to hold the offset address of the data to be read from or written into the stack segment.

SP Register

SP Register is known as stack pointer. SP Register is a 16 bit register and used to hold the offset address of the data stored in the top of the stack. Sp is used along with SS register to decide the address at which the data is to be pushed or popped during the execution of PUSH/ POP instructions.

IP Register

IP Register is known as instruction pointer. It is 16 bit register which is used to store the offset address of the next instruction that is to be executed.

Index Registers

Index registers are 16 bit registers which are used to store the offset address. Index registers are SI and DI registers. These registers are also used as general purpose registers.

SI Register

SI Register is known as source index register or source register which contains the source address of the data segment. It is a 16 bit register. SI register is used in string manipulations.

DI Register

DI Register is known as destination index register or destination register which contains the destination address of the data segment / extra segment. It is a 16 bit register. DI register is used in string manipulations.

Flag Register

Flag register indicates the status of the ALU results. A 16 flag register is used in 8086. It is divided into two parts

(a) Conditional code or status flags

(b) Machine control flags

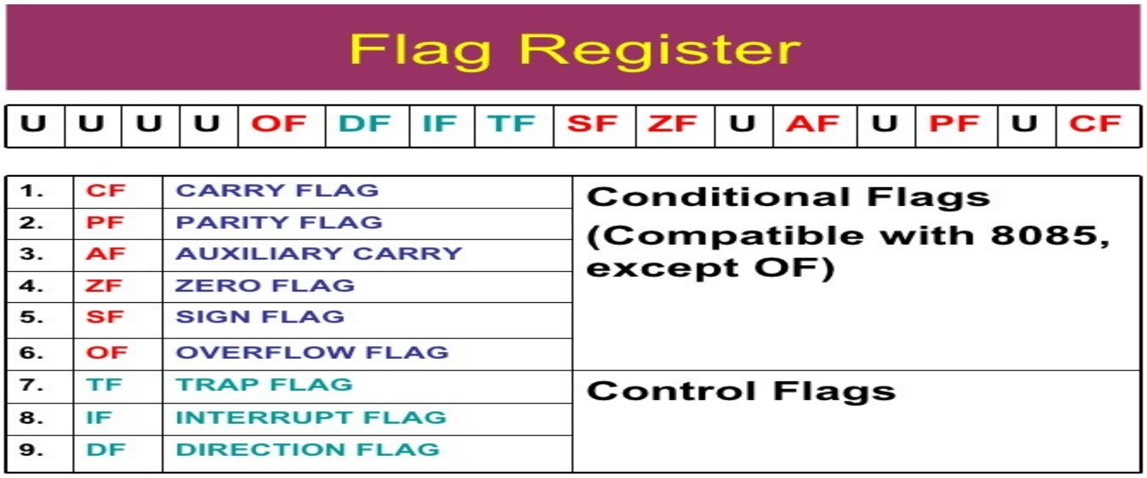
Condition code flag

The condition code flag is the lower byte of the 16-bit flag register. It contains 6 flags namely carry flag (CF), parity flag (PF), auxiliary carry flag (AF), zero flag (ZF), sign flag (SF) and overflow flag (OF).

Control flag

The control flag is the higher byte of the flag register. It contains three flags namely direction flag (*D*), interrupt flag (*I*) and trap flag (*T*).

The complete bit configuration of 8086 is shown in the figure.



Flag Register

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

1.	CF	CARRY FLAG	Conditional Flags (Compatible with 8085, except OF)
2.	PF	PARITY FLAG	
3.	AF	AUXILIARY CARRY	
4.	ZF	ZERO FLAG	
5.	SF	SIGN FLAG	
6.	OF	OVERFLOW FLAG	
7.	TF	TRAP FLAG	Control Flags
8.	IF	INTERRUPT FLAG	
9.	DF	DIRECTION FLAG	

Carry Flag: Carry flag is set (CF=1), when there is a carry of MSB is generated in case of addition or borrow in case of subtraction. Otherwise it is said to be reset (CF=0).

Parity Flag: Parity flag is set (PF=1), if the lower byte of the result contains even number of 1's. If there is odd number of 1's in the result it is said to be reset (PF=0).

Sign Flag: Sign flag is set (SF=1), when the result of any computation is negative, otherwise it is said to be reset (SF=0).

Zero Flag: Zero flag is set (ZF=1), if the result of computation is zero. Otherwise it is said to be reset (ZF=0).

Auxiliary Carry Flag: Auxiliary flag is set (AF=1), if there is a carry is generated in the lowest nibble, i.e, bit three during addition, or borrow is generated in the lowest nibble, i.e, bit three, during subtraction. Otherwise it is said to be reset (AF=0).

Over flow Flag: Overflow flag is set (OF=1), if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, and then the overflow will be set. Otherwise it is said to be reset (OF=0).

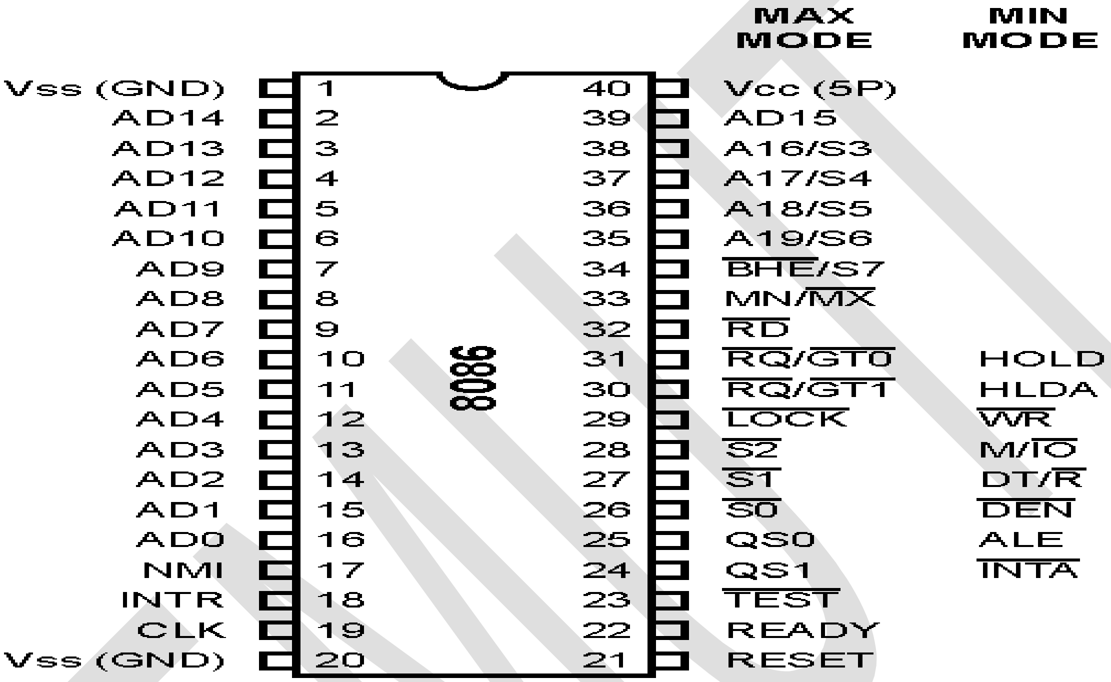
Trap Flag: Trap flag is set (TF=1), when the processor enters the single step execution mode. Otherwise it is said to be reset (TF=0).

Interrupt Flag: Interrupt flag is set (IF=1), when the interrupt requests are recognized by the microprocessor, otherwise they are said to be reset (IF=0).

Direction Flag: Direction flag is used by string instructions. This flag is said to be set (DF=1), when the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

Direction flag is reset (DF=0), when the string is processed from the lowest address to the highest address, i.e., auto incrementing mode.

8086 Pin diagram



POWER SUPPLY :

8086 uses 5V DC power supply at V$_{CC}$ pin 40, and uses ground at V$_{SS}$ pin 1 and 20 for its proper operation.

CLOCK SIGNAL:

Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

ADDRESS/DATA BUS (AD0-AD15):

These are 16 address/data bus which are multiplexed. AD0-AD7 carries low order byte data and AD8-AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

ADDRESS/STATUS BUS (A16-A19/S3-S6):

These are 4 address/status buses which are multiplexed. During the first clock cycle, it carries 4-bit address and later it carries status signals. The status of the interrupt enable flag bit (displayed on S5) is updated at the beginning of each clock cycle. The status line S_6 is always low. The S4 and S3 combinedly indicate which segment register is presently being used for memory access.

S4	S3	Function
0	0	Extra segment
0	1	Stack segment
1	0	Code segment
1	1	Data segment

S7/$\overline{BHE}$:

BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using higher order data bus D8-D15. It also indicates whether whole word or byte is transferred. During the first clock cycle $\overline{BHE}$ is a low and remaining cycle status signal S_7 is available.

BHE bar	S_7	Function
0	0	Whole word
0	1	Upper byte from or to odd address
1	0	Upper byte from or to even address
1	1	None

RD:

It is available at pin 32 and is used to read the data for Read operation. It performs read operation either from memory or IO device.

READY:

It is available at pin 32. It is an acknowledgement signal from slow I/O devices that data transfer is completed. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

RESET:

It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

INTR:

It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine the availability interrupt. If interrupt request is pending the processor enters into interrupt acknowledge cycle. This can be done by enabling the interrupt flag. INTR is a level triggered interrupt.

NMI:

It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor. A transition from low to high initiates the interrupt response at the end of the instruction.

RESET:

This input causes the processor to terminate the current activity and start execution from FFFF0 H. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronized.

TEST bar:

This input is examined by a 'WAIT' instruction. If the TEST input goes low, execution will continue, else, the processor remains in an idle state.

MN/*MX bar:*

It stands for Minimum/Maximum mode and is available at pin 33. It indicates which type of mode the processor is to operate. When it is high, it operates in the minimum mode and when it is low it operates in maximum mode.

INTA bar:

It is an interrupt acknowledgement signal and is available at pin 24. When the microprocessor receives this signal, it has to accept the interrupt.

ALE:

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address bus.

DEN bar:

It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver and indicates that data is available on the data bus. The transceiver is a device used to separate data from the multiplexed address/data bus.

DT/R bar:

It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transreceiver. When it is high, data is transmitted out and when it is low data is received.

M/$\overline{IO}$**:**

This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.

WR bar:

It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.

HOLD:

This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.HOLD signal is used in DMA (Direct Memory Access) operation.

HLDA:

It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal. This acknowledgement signal is sent by the processor to other devices when it receives HOLD request signal. HLDA signal is used in DMA (Direct Memory Access) operation.

QS_1 and QS_0:

These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table

QS_0	QS_1	Status
0	0	No operation
0	1	First byte of op code from the queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

S_0, S_1, S_2:

These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status –

S_2	S_1	S_0	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Op code fetch
1	0	1	Memory read

1	1	0	Memory write
1	1	1	Passive

LOCK:

When this signal is active, it indicates to the other processers to prevent the access of system bus. It is activated using the LOCK prefix on any instruction and remains active until completion of the next instruction.

RQ/GT$_1$ and RQ/GT$_0$:

These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus in case of DMA (Direct Memory Access) operations. When the RQ$_0$/RQ1 request signal is received by processor, then processor sends acknowledgment signal through GT$_0$/GT$_{1.}$ RQ/GT$_0$ has a higher priority than RQ/GT$_1$.

Timing and control signals:

Timing and control signals are minimum mode pin signals and maximum mode pin signals.

Minimum mode pin signals

- HOLD
- HLDA
- WR BAR
- M/$\overline{IO}$
- DT/R BAR
- DEN BAR
- INTA BAR

Maximum mode pin signals

- RQ/GTO BAR
- RQ/GT1 BAR
- LOCK
- S2 BAR
- S1 BAR
- S0 BAR

- QS1
- QS0

System timing diagrams:

The timing diagram provides information about the various conditions of signals such as high/low, when a machine cycle is being executed. Without the knowledge of timing diagram it is not possible to match the peripheral devices to the microprocessors. These peripheral devices includes memories, ports etc. Such devices can only be matched with microprocessors with the help of timing diagram.

Machine cycle*:* A basic microprocessor operation such as reading a byte from memory or writing a byte to a port is called a machine cycle (Bus cycle). A machine cycle consists of at least 4 clock cycles/ clock states (T-states) for accessing the data.

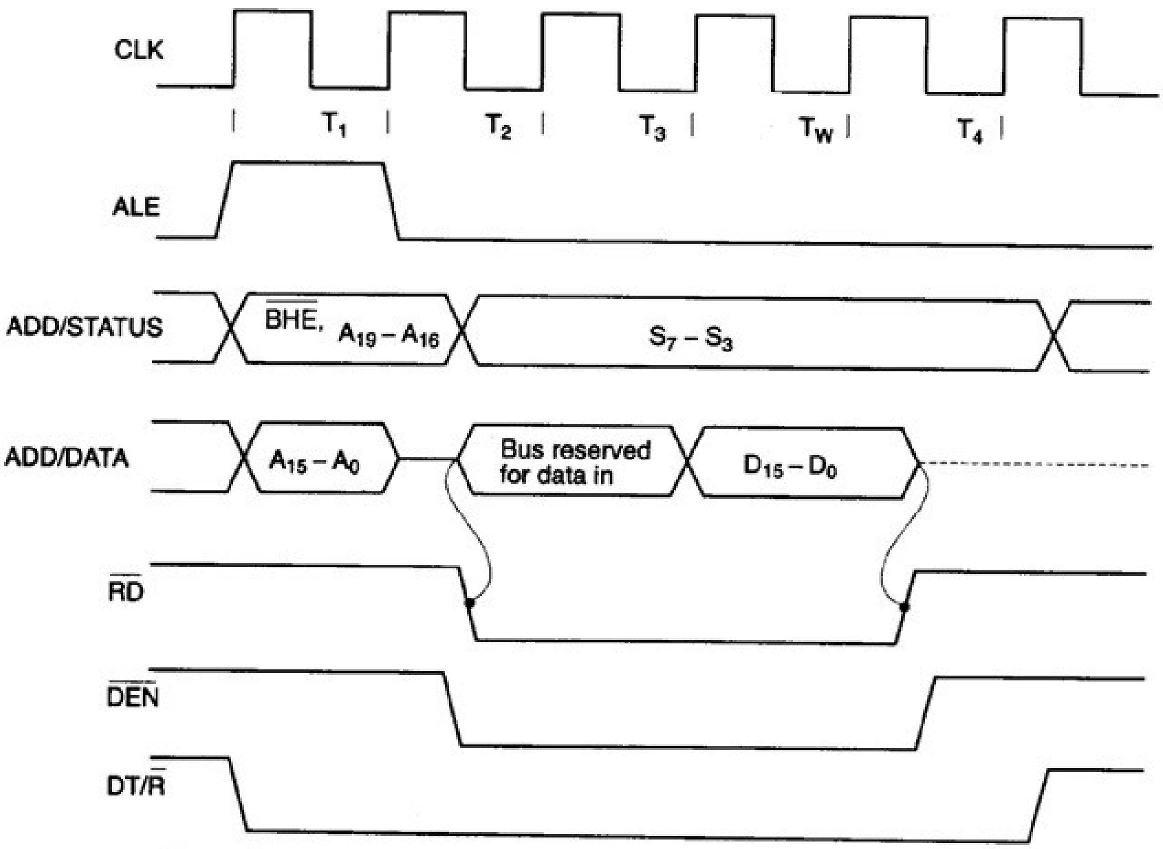
Instruction cycle: The time a microprocessor requires to fetch and execute an entire instruction is referred to as an instruction cycle. An instruction cycle consists of one or more machine cycles.

T-state*:* T-state is nothing but one subdivision of the operation performed in one clock period. These subdivisions are internal state of the microprocessor synchronized with system clock.

So, an instruction cycle is made up of machine cycles, and a machine cycle is made up of states. The time for the state is determined by the frequency of the clock signal.

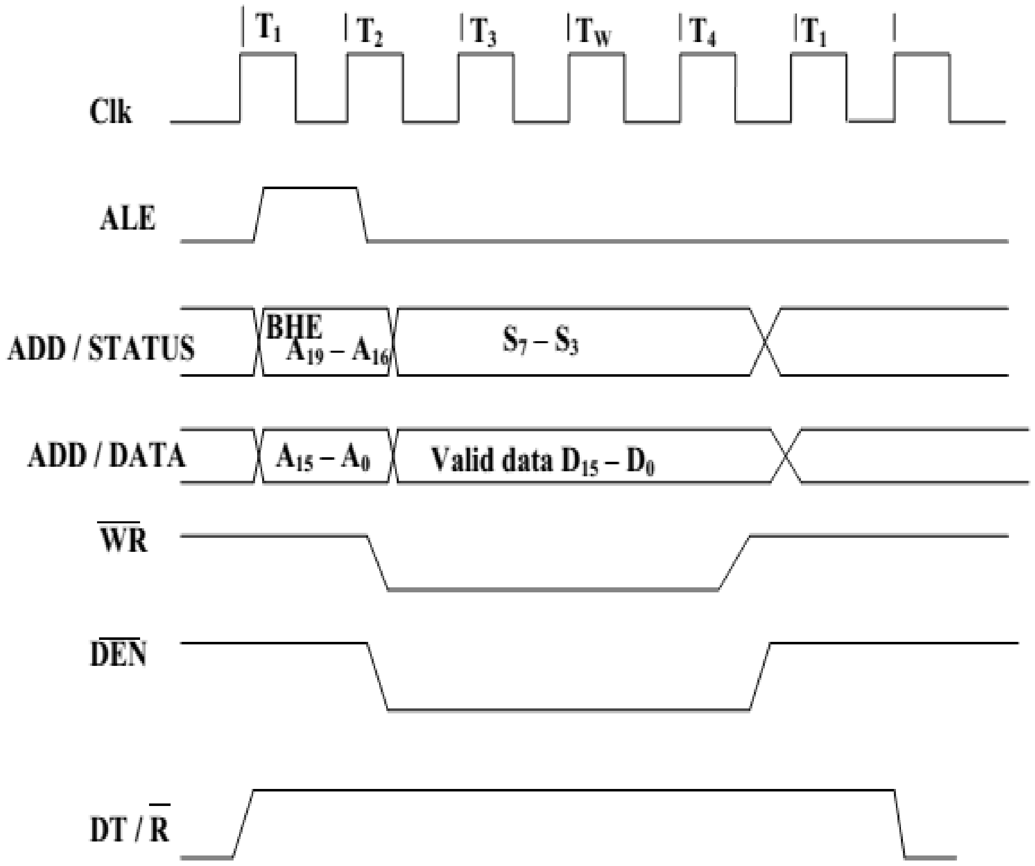
The 8086 has a combined address and data bus commonly referred as a time multiplexed address and data bus. The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40 pin standard DIP package. The bus can be de multiplexed using a few latches and transreceivers, when ever required. Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T1, T2, T3, and T4. The address is transmitted by the processor during T1. It is present on the bus only for one cycle.

Read cycle



- The read cycle begins in T1 with the assertion of address latch enable (ALE) signal and also M / $\overline{IO}$ signal. During the negative going edge of this signal, the valid address is latched on the local bus. From T1 to T4, the M/$\overline{IO}$ signal indicate a memory or I/O operation is to be performed..
- $\overline{BHE}$ and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
- At T2, the address is removed from the local bus and is sent to the output. The read ($\overline{RD}$) control signal is also activated in T2.
- The read ($\overline{RD}$) signal causes the address device to enable its data bus drivers. After $\overline{RD}$ 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.
- DEN bar indicates valid data is available on data bus.
- DT/R bar indicates whether data is transmitted or received to memory/port._

Write cycle



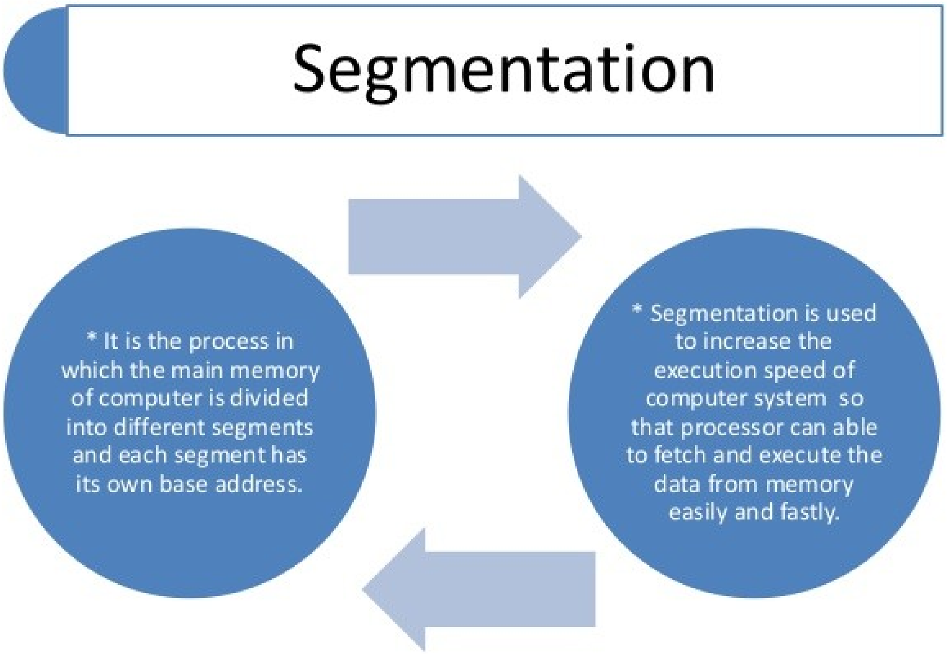
- A write cycle also begins with the assertion of ALE and the emission of the address. The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O operation. In T2, after sending the address in T1, the processor sends the data to be written to the addressed location.
- $\overline{BHE}$ and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
- DEN bar indicates valid data is available on data bus.
- DT/R bar indicates whether data is transmitted or received to memory/port.
- The data remains on the bus until middle of T4 state. $\overline{WR}$ becomes active at the beginning of T2.

Memory Segmentation:

8086microprocessor has 20 it address bus. So that it can address 1 MB of memory and each memory location is addressed by 20 bit address.

To hold 20 bit address there must be 20 bit register available within 8086 microprocessor but 8086 only has 16 bit registers. So 20 bit address cannot be stored inside 16 bit register. To avoid this problem memory segmentation is used in 8086 microprocessor.

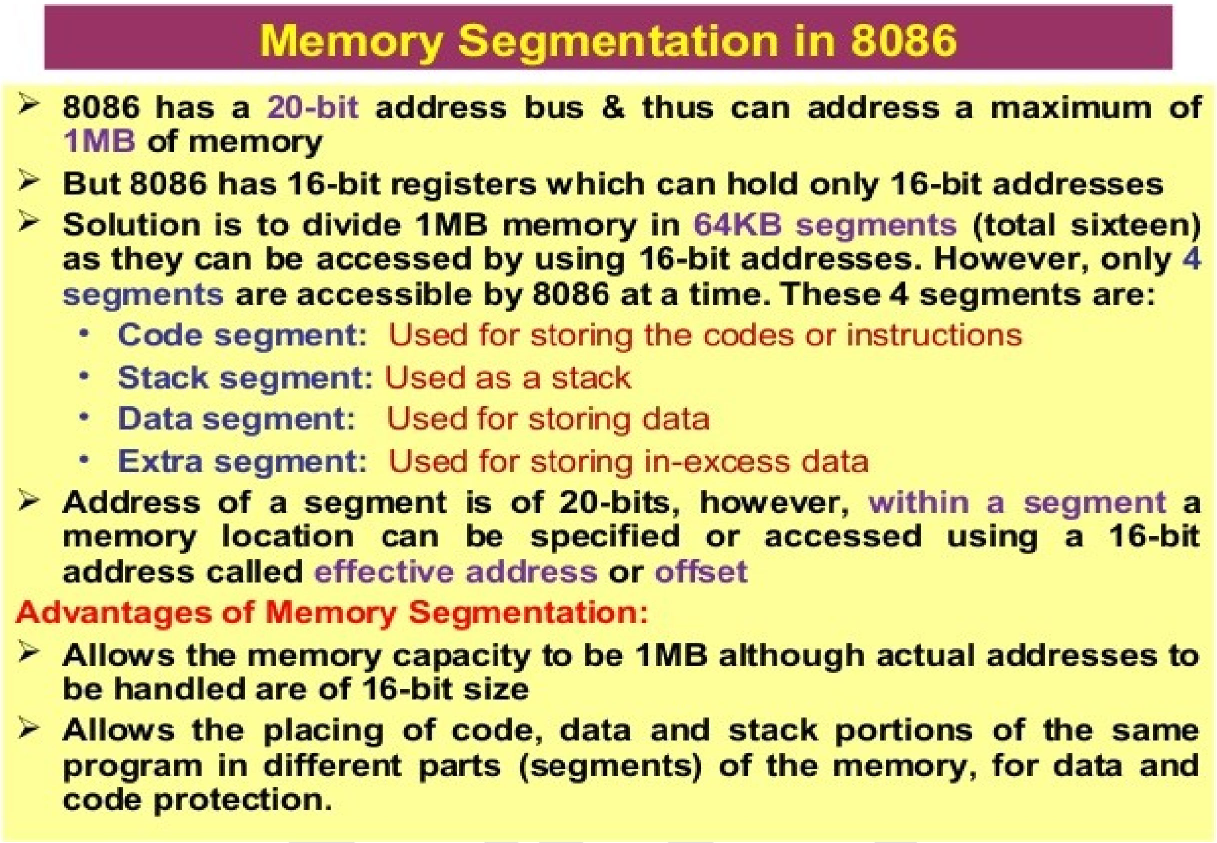
The process of dividing the main memory into segments is known as memory segmentation. The size of each segment is 64 KB. The complete 1 MB of memory is divided into 16 logical segments (1 MB/64 KB = 16).The 8086 microprocessor can able to access only 4 logical segments at a time.



Segmentation

* It is the process in which the main memory of computer is divided into different segments and each segment has its own base address.

* Segmentation is used to increase the execution speed of computer system so that processor can able to fetch and execute the data from memory easily and fastly.

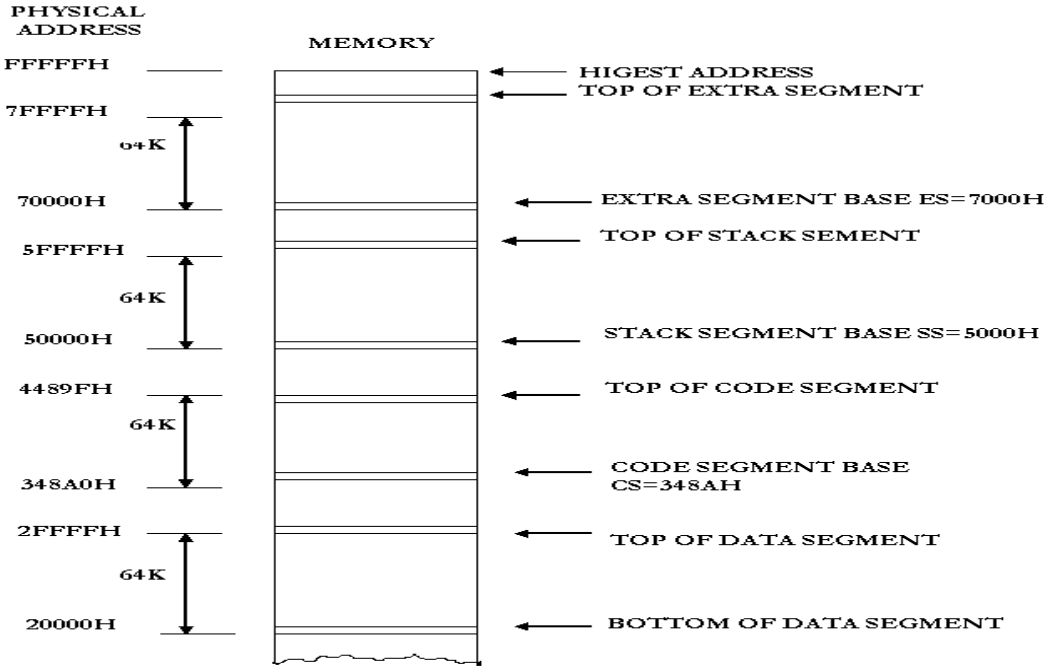


Memory Segmentation in 8086

- 8086 has a 20-bit address bus & thus can address a maximum of 1MB of memory
- But 8086 has 16-bit registers which can hold only 16-bit addresses
- Solution is to divide 1MB memory in 64KB segments (total sixteen) as they can be accessed by using 16-bit addresses. However, only 4 segments are accessible by 8086 at a time. These 4 segments are:
  - Code segment: Used for storing the codes or instructions
  - Stack segment: Used as a stack
  - Data segment: Used for storing data
  - Extra segment: Used for storing in-excess data
- Address of a segment is of 20-bits, however, within a segment a memory location can be specified or accessed using a 16-bit address called effective address or offset

Advantages of Memory Segmentation:

- Allows the memory capacity to be 1MB although actual addresses to be handled are of 16-bit size
- Allows the placing of code, data and stack portions of the same program in different parts (segments) of the memory, for data and code protection.



Interrupt structure of 8086

INTERRUPT

In general, the process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt. The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor.

OCCURRENCE OF INTERRUPT

The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,
ii) by an internal signal generated by a special instruction in the program,
iii) by an internal signal generated due to occurrence of some condition.

EXTERNAL SIGNAL

An 8086 can get interrupt from an external signal applied to the Non Mask able Interrupt (NMI) input pin, or the interrupt (INTR) input pin.

SPECIAL INSTRUCTION

8086 supports a special instruction, INT to execute special program. At the end of interrupt service routine, execution is usually returned to the interrupted program.

CONDITION PRODUCED BY THE INSTRUCTION

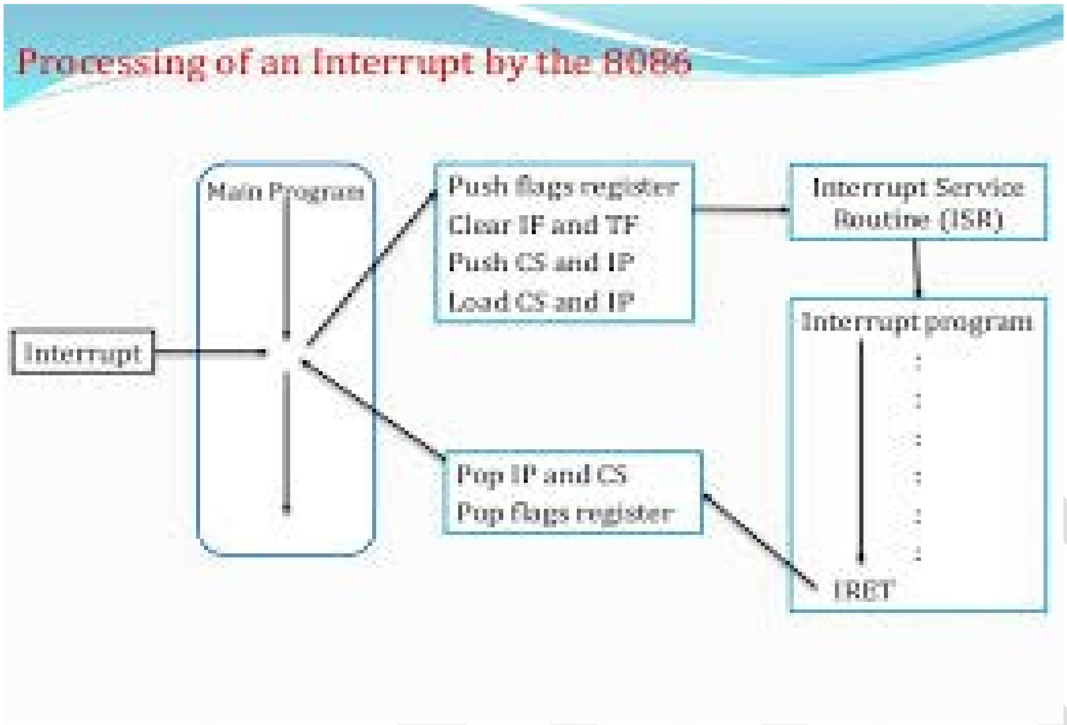
An 8086 is interrupted by the some condition produced in the 8086 by the execution of an instruction.

For example divide by zero: program execution will automatically be interrupted if you attempt to divide an operand by zero.

At the end of each instruction cycle 8086 checks to see if there is any interrupt request. If so, 8086 responds to interrupt by performing series of actions.

Interrupt process

When a microprocessor receives an interrupt signal it stops executing current normal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine. (ISR). At the end of ISR, the stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution from the point {instruction) where it was interrupted.



INTERRUPT CYCLE OF 8086

1. It decrements stack pointer by 2 and pushes the flag register on the stack.
2. It disables the INTR interrupt input by clearing the interrupt flag (IF) in flag register.
3. It resets the trap flag in the flag register.
4. It decrements the stack pointer by 2 and pushes the current code segments register contents on the stack.
5. It decrements the stack pointer by 2 and pushes the current instruction pointer contents on the stack.
6. It does an indirect far jump at the start of the procedure by loading the CS and IP values for the start of the interrupt service routine (ISR).

An IRET instruction at the end of the interrupt service procedure returns execution to the main program.

CLASSIFICATION OF INTERRUPTS

In general the interrupts can be classified in the following three ways:

1. Hardware and software interrupts
2. Vectored and Non Vectored interrupt:
3. Mask able and Non Mask able interrupts.

HARDWARE INTERRUPTS

The interrupts initiated by external hardware by sending an appropriate signal to the Interrupt pin of the processor is called hardware interrupt.

The 8086 processor has two Interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

SOFTWARE INTERRUPTS

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt.

The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255.

VECTORED INTERRUPTS

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to vector address is predefined by the manufacturer of processors. (In these vector addresses the interrupt service subroutines (ISR) are stored).

NON-VECTORED INTERRUPTS

In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt. In 8086 there are 17 vectored interrupts. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).

MASKABLE AND NON MASKABLE INTERRUPTS

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling.

In 8086 the interrupt flag (IF) can be set to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts.

The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR.

In 8086 processor all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable.

INTERRUPT VECTOR TABLE

The first 1Kbyte of memory of 8086 (00000 to003FF) is set aside as a table for storing the starting addresses of Interrupt Service Routines (ISR). The starting address of an ISR is often called the **Interrupt Vector** or **Interrupt Pointer.** The block of memory is referred as **Interrupt Vector Table.**

When 8086 responds to an interrupt, it gets the new values of CS and IP registers from four memory addresses. Since 4 bytes are required to store the CS and IP values for each interrupt service routines.

The figure shows how 256 interrupts are arranged in the memory table. Each interrupt type is given a number from 0 to 255 and the address of each interrupt is found by multiplying the type by 4.

Example: For type 11, the interrupt address is 11*4=44=0002C H
For type 2, the interrupt address is 2*4=8=00008H

These 256 interrupts are groups as three types of interrupts, namely

- Dedicated interrupts
- Reserved interrupts
- Available interrupts

Dedicated interrupts:

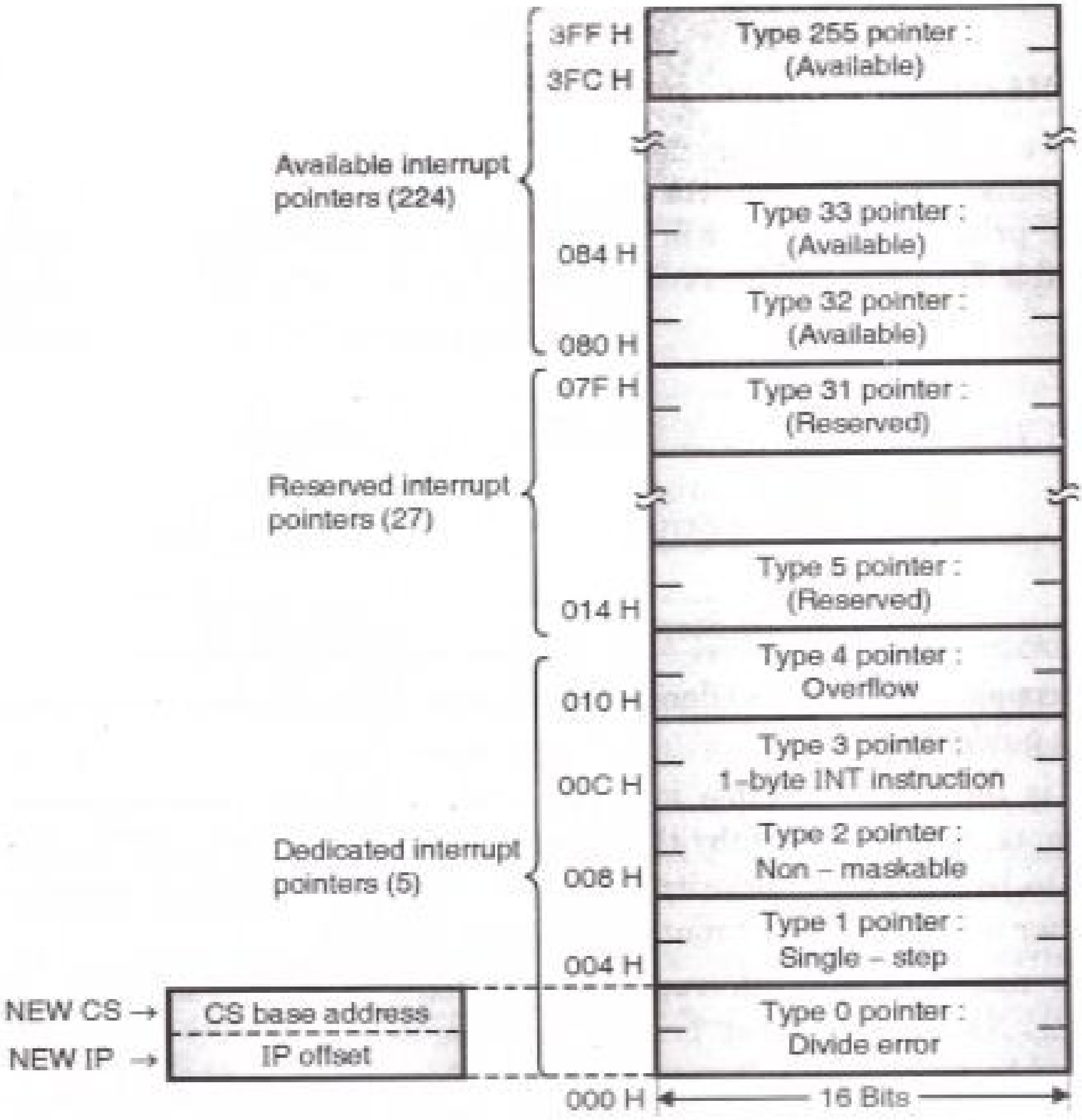
There are 5 dedicated interrupts in 8086 microprocessor. The interrupts from type 0 to type 4 are known as dedicated interrupts. These interrupts are dedicated to particular application so it is named as dedicated interrupts.

Reserved interrupts:

There are 27 reserved interrupts in 8086 microprocessor. The interrupts from type 5 to type 31 are known as reserved interrupts. These interrupts are reserved for future purpose, so it is named as reserved interrupts.

Available interrupts:

There are 224 reserved interrupts in 8086 microprocessor. The interrupts from type 32 to type 255 are known as reserved interrupts. These interrupts are available for users, so it is named as available interrupts.


3FF H
3FC H
Type 255 pointer : (Available)
Available interrupt pointers (224)
Type 33 pointer : (Available)
084 H
Type 32 pointer : (Available)
080 H
07F H
Type 31 pointer : (Reserved)
Reserved interrupt pointers (27)
Type 5 pointer : (Reserved)
014 H
Type 4 pointer : Overflow
010 H
Type 3 pointer : 1-byte INT instruction
00C H
Dedicated interrupt pointers (5)
Type 2 pointer : Non – maskable
008 H
Type 1 pointer : Single – step
004 H
NEW CS →
CS base address
NEW IP →
IP offset
Type 0 pointer : Divide error
000 H
16 Bits

Interrupt vector table

The above figure represents interrupt vector table. In that interrupt vector table , only first 5 types are dedicated interrupt pointers, the next 27 interrupt types, from 5 to 31, are reserved by INTEL for the use in future microprocessors. The upper 224 interrupt types, from 32 to 255, are available for user for hardware or software interrupts.

8086 INTERRUPT TYPES

1. Divide by Zero interrupt(Type 0)

When the quotient from either a DIV or IDIV instruction is too large to fit in the result register, 8086 will automatically execute type 0 interrupt.

2. **Single Step Interrupt(Type 1)**

The type 1 interrupt is then single step trap. In the single step mode, the system will execute one instruction and wait for further direction from user. Then user can examine the contents of registers and memory locations and if they are correct, user can tell the system to execute the next instruction. This feature is useful for debugging assembly language program.

An 8086 system is used in single step mode by setting the trap flag. If trap flag is set, the 8086 will automatically execute a type 1 interrupt after execution of each instruction.

3. **Non Maskable Interrupt (Type 2)**

As the name suggests, this interrupt cannot be disabled by any software instruction. This interrupt is activated by low to high transition on NMI input pin. In response 8086 will do a type 2 interrupt.

The characteristics of NMI are as follows:

- These interrupts are known as non-maskable types.
- These interrupts have highest priority than INTR interrupt.
- These interrupts are edge triggered which are low to high transition.
- In order to function they must remain high for at least 2 cycles of CLK.
- These interrupts do not send out any form of acknowledgements.

4. **Breakpoint Interrupt (Type 3)**

The type 3 interrupt is used to implement breakpoint function in the system. This type of interrupt indicates termination of the program. The type 3 interrupt is produced by execution of INT 3 instruction.

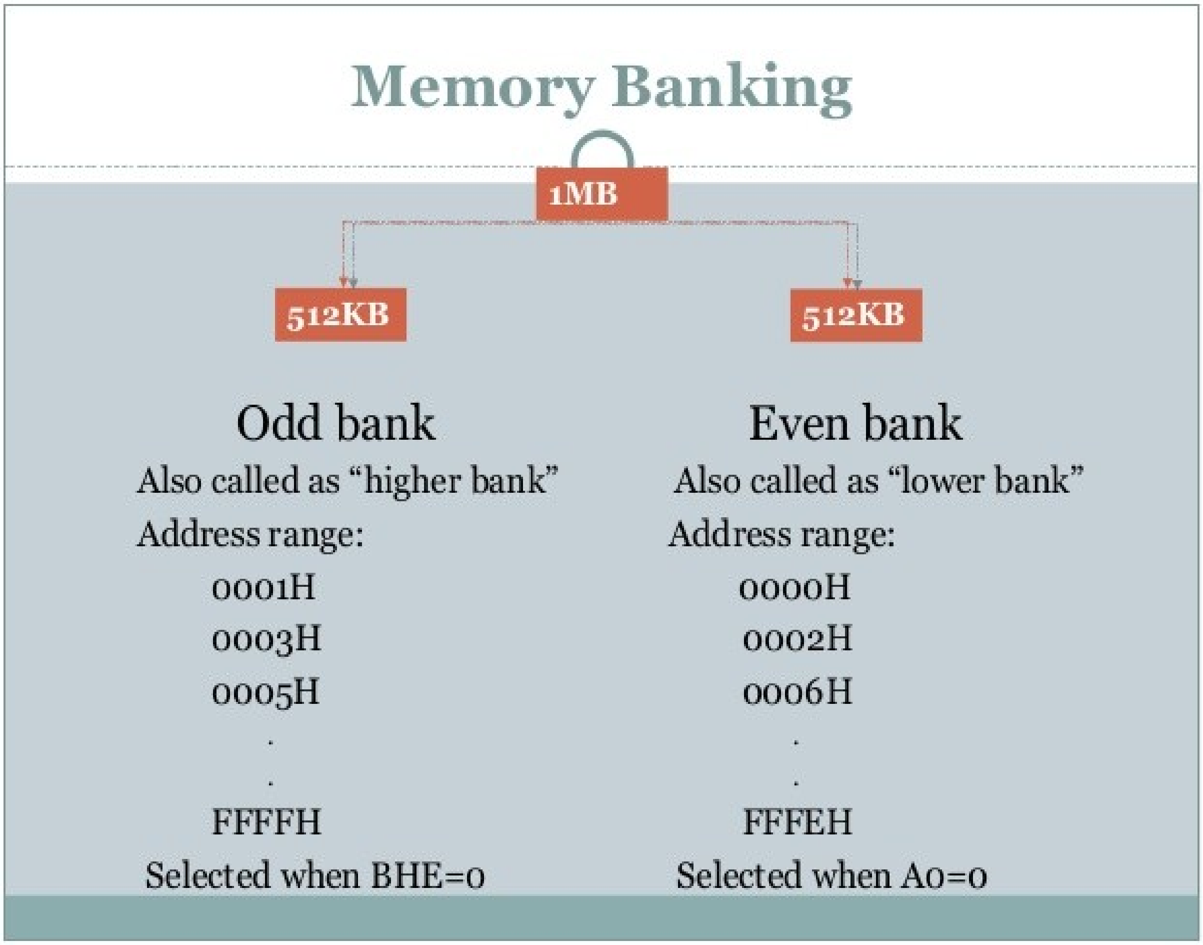
5. **Overflow Interrupt (Type 4)**

The type 4 interrupt is used to check overflow condition after any signed arithmetic operation in the system. The 8086 overflow flag, OF, will be represented in the destination register or memory location.

Memory organization & memory banks accessing

8086 is a 16 bit microprocessor and hence can access two bytes of data in one memory. But commercially available memory chips are only one byte size, ie. They can store only one byte in a memory location. Obviously, to store 16 bit data, two successive memory locations are used and lower byte of 16 bit data can be stored in first memory location while the second byte is stored in next memory location.

8086 microprocessor can address 1 MB of memory. This 1 MB (1024KB) of memory is physically organized into 2 memory banks namely even addressed memory bank and odd addressed memory bank with each 512 KB of size.



Memory Banking

1MB

512KB

512KB

Odd bank

Also called as "higher bank"

Address range:

0001H

0003H

0005H

.

.

FFFFH

Selected when BHE=0

Even bank

Also called as "lower bank"

Address range:

0000H

0002H

0006H

.

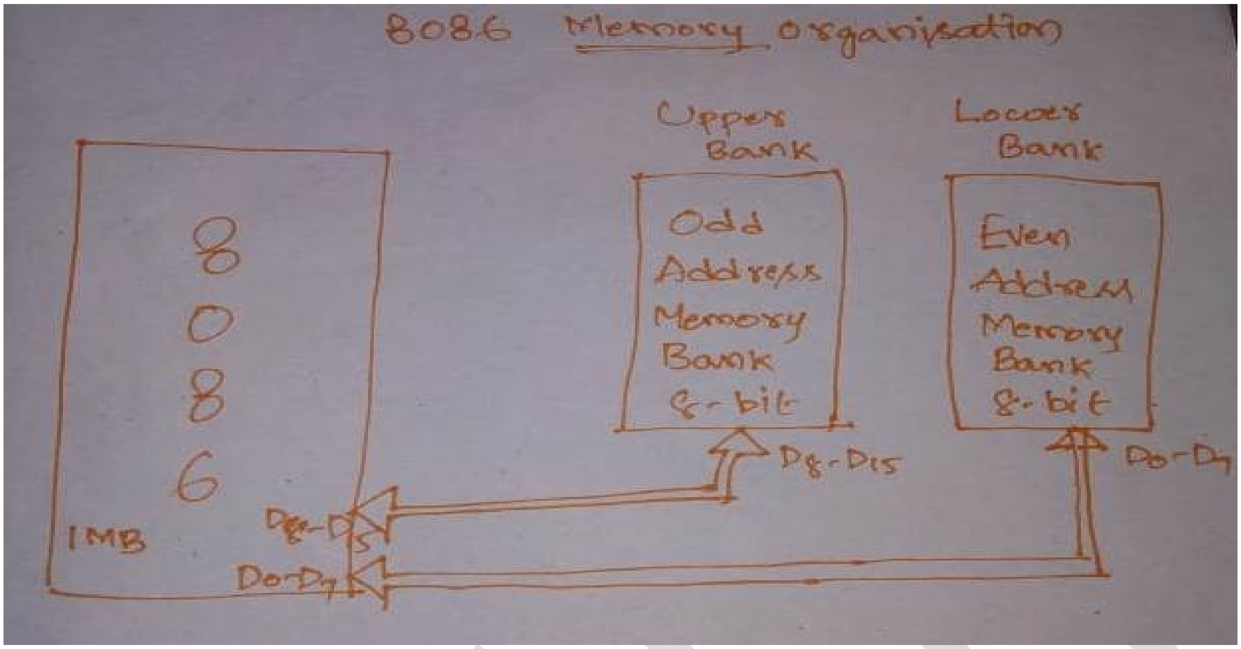
.

FFFEH

Selected when A0=0

Even address memory bank

Even address memory bank is also called as lower bank. It is 8 bit memory bank. It stores the data with even addresses like 00000H, 00002H, and 00004 H and so on. The data with even address is transferred on lower order data bus D0 – D7.



Odd address memory bank

Odd address memory bank is also called as higher bank. It is 8 bit memory bank. It stores the data with odd addresses like 00001H, 00003H, and 00005 H and so on. The data with odd address is transferred on lower order data bus D8 – D15.

The processor provides two enable signals, BHE and A_0, to selectively allow reading from or writing into either an odd byte location, even byte location, or both.

If A_0=0 and BHE bar=1 then even memory bank is selected. Similarly if A_0=1 and BHE bar=0 then odd memory bank is selected.

Thus 8086 transfers a 16 bit data to/from memory by selecting both memory banks for 16 bit operation. For 8 bit operation, even memory bank and odd memory bank are to be used based on even address and odd address.

UNIT-II

8086 Addressing modes

The addressing mode indicates the way of locating the data or operands in the memory. It also specifies the way in which the data or operand is accessed from the memory.

Types of addressing modes

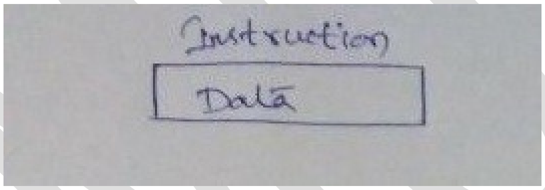
- Immediate addressing mode
- Register addressing mode
- Data memory addressing mode
- Program memory addressing mode
- Stack memory addressing mode

Immediate addressing mode

In immediate addressing mode, the data is specified in the instruction. The data can be 8 bit or 16 bit data.

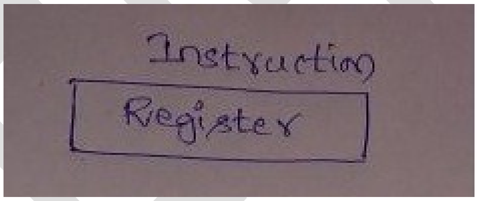
Example:

- MOV AL, 50H; move the data 50H to AL
- MOV BX, 230A H; move the data 230aH to BX



Register addressing mode

In register addressing mode, the data is present in the register and this data is moved and the result is stored in the register.



Example:

- MOV AL, BL; Move the content of BL to AL
- MOV CX, BX; Move the content of BX to CX
- ADD CL,BL; Add the contents of CL and BL and store the result in CL
- ADD BX, DX; Add the contents of BX and DX and the results is stored in DX

Data memory addressing modes:

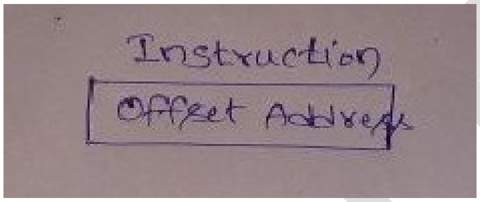
In data memory addressing mode the data can be accessed through offset address. The term effective address (EA) represents the offset address of the data within a segment, which is obtained by different methods, depending upon the addressing mode that is used in the instruction.

Let us consider the various register values for discussion of data memory addressing modes-

REGISTER	CS	DS	SS	ES	BX	BP	SI	DI
VALUE	1000H	3000H	4000H	6000H	2000H	1500H	2500H	3500H

1. Direct addressing mode:

In this addressing mode, the 16 bit offset address of the data within the segment is directly given in the instruction. 20 bit physical address is calculated with the help of offset address then the data can be accessed from this 20 b9t address.



Example:

a) MOV AL, [1000H]

In this instruction, the effective address is 1000H. The destination register (AL) can be accessed from the memory at the given location by calculating 20 bit physical address.

Memory address= segment address + offset address
Memory address= DS*10H + offset address
Memory address = 3000*10 + 1000 = 31000 H

Here the data segment is considered as default segment when data transfer instructions are used. The data available at the address (31000 H) is moved into AL and hence data can be accessed.

b) MOV BX, [5000H]

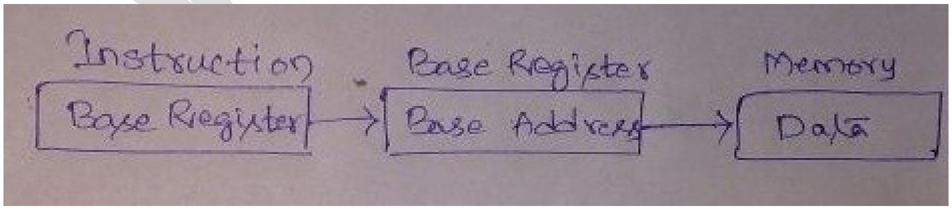
In this instruction, the effective address is 5000H. The destination register (BX) can be accessed from the memory at the given location by calculating 20 bit physical address.

Memory address= segment address + offset address
Memory address= DS*10H + offset address
Memory address = 3000*10 + 2000 = 32000 H

Here the data segment is considered as default segment when data transfer instructions are used. The data available at the address (32000 H) is moved into BX and hence data can be accessed.

2. Base addressing mode:

In base addressing mode, the offset address is specified in terms of base registers. The effective address is obtained by adding the contents of BX or BP registers. When BX register is present in the instruction, the data can be accessed from the data segment and when BP register is present, the data can be accessed from the stack segment. Finally 20 bit address is calculated and from that memory location data can be accessed.



Example:

a) MOV CL, [BX]

In this instruction the effective address is obtained from BX register. BX=2000H
Memory address=DS*10 + [BX] = 3000*10+2000=32000H
The byte from the memory address 32000H is read and stored in CL register.

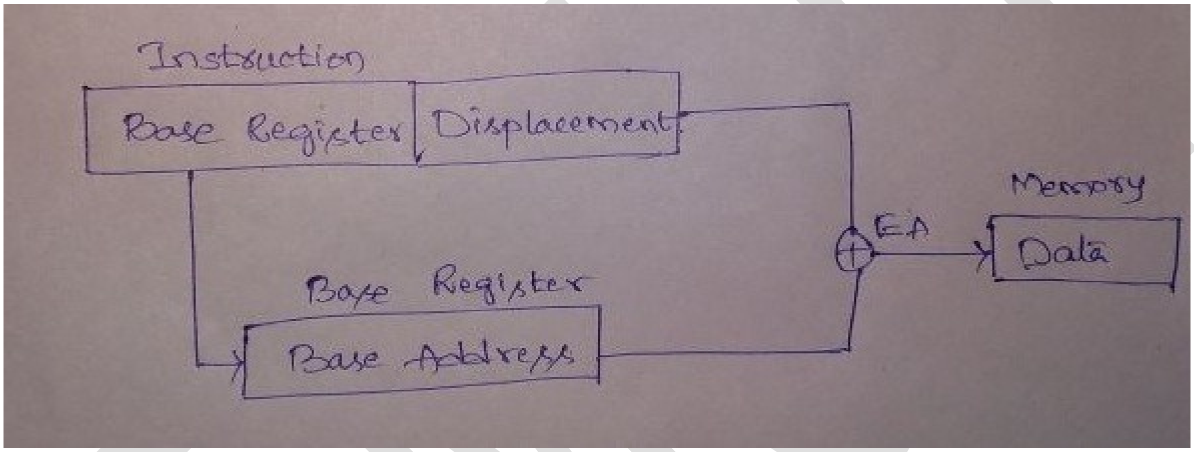
b) MOV DX, [BP]

In this instruction the effective address is obtained from BP register. BP=1500H
Memory address=SS*10 + [BP] = 4000*10+1500=41500H
The byte from the memory address 41500H is read and stored in DX register.

3. Relative Base addressing mode:

In relative base addressing mode, the offset address is specified through base registers and displacement. The effective address is obtained from adding the contents of base registers and 8 bit or 16 bit displacement. The 8 bit displacement can have the values from -128 to +127 and 16 bit displacement can have values from -32767 to +32767. Finally 20 bit address is calculated and from that memory location data can be accessed.

-



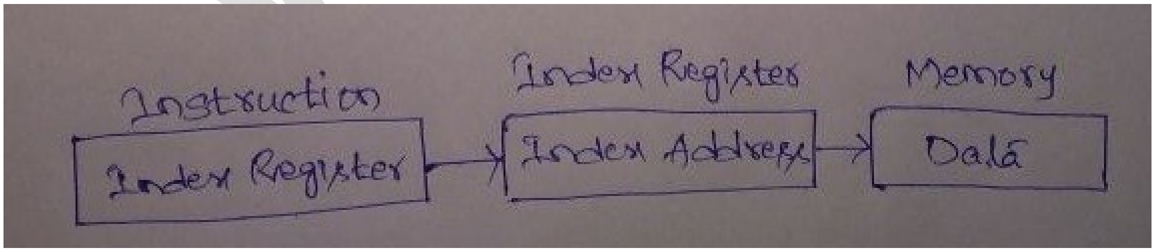
Example:

a) MOV AX, [BX + 5]

In this instruction the effective address is obtained by adding contents of BX register (2000) with displacement (5)
Memory address=DS*10 + [BX] + 5 = 3000*10+2000 + 5=32005H
The byte from the memory address 32005H is read and stored in AX register.

4. Index addressing mode:

In index addressing mode, the offset address is specified in terms of index registers. The effective address is obtained by adding the contents of SI or DI registers. The data is taken from data segment. Finally 20 bit address is calculated and from that memory location data can be accessed.



Example:

c) MOV CX, [DI]

In this instruction the effective address is obtained from DI register. DI=3500H
Memory address=DS*10 + [DI] = 3000*10+3500=33500H
The byte from the memory address 33500H is read and stored in CX register.

d) MOV BL, [SI]

In this instruction the effective address is obtained from SI register. SI=2500H
Memory address=DS*10 + [SI] = 3000*10+2500=32500H
The byte from the memory address 32500H is read and stored in BL register.

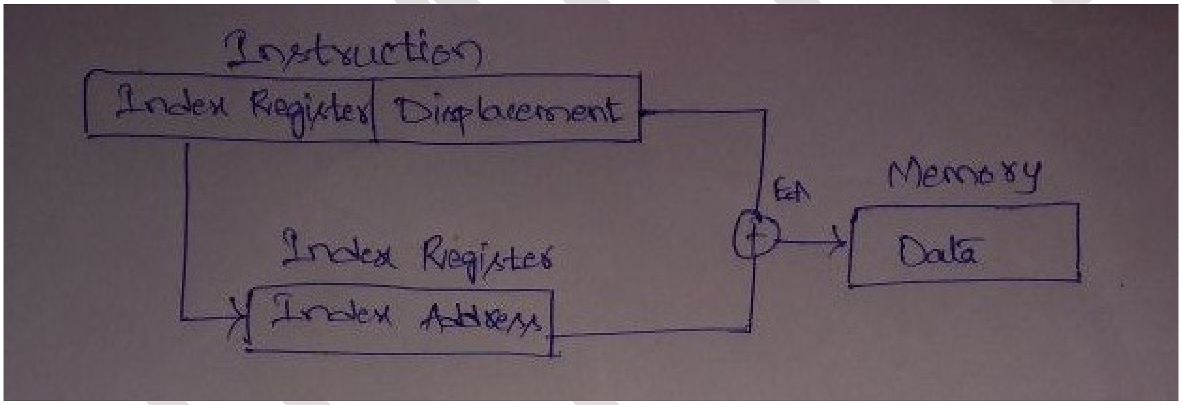
5. Relative Index addressing mode:

In relative index addressing mode, the offset address is specified through index registers and displacement. The effective address is obtained from adding the contents of index registers and 8 bit or 16 bit displacement. The 8 bit displacement can have the values from -128 to +127 and 16 bit displacement can have values from -32767 to +32767. Finally 20 bit address is calculated and from that memory location data can be accessed.

Example:

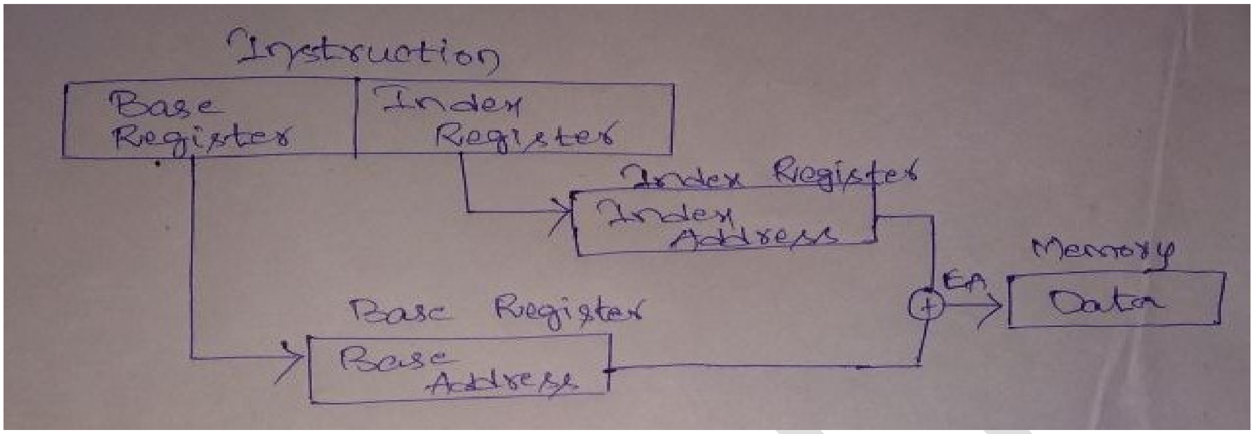
b) MOV CL, [DI + 10H]

In this instruction the effective address is obtained by adding contents of DI register (3500) with displacement (10)
Memory address=DS*10 + [DI] + 10 = 3000*10+3500 + 10=33510H
The byte from the memory address 33510H is read and stored in CL register.



6. Based Indexed addressing mode:

In based indexed addressing mode, the offset address is specified through base registers and index registers. The effective address is obtained from adding the contents of base registers and index registers. Finally 20 bit address is calculated and from that memory location data can be accessed.



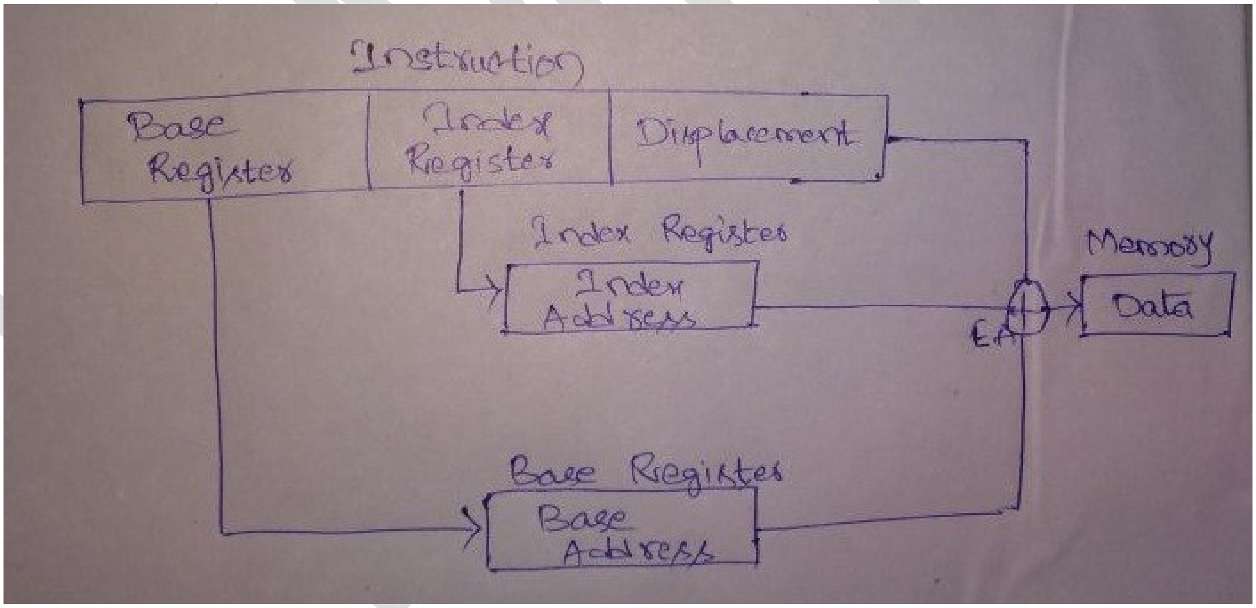
Example:

a) MOV AX, [BX + SI]

In this instruction the effective address is obtained by adding contents of BX register (2000) and SI register (2500).
Memory address=DS*10 + [BX] + [SI] = 3000*10+2000 + 2500=34500H
The byte from the memory address 34500H is read and stored in AX register.

7. Relative Based Indexed addressing mode:

In relative based indexed addressing mode, the offset address is specified through base registers, index registers and displacement. The effective address is obtained from adding the contents of base registers, index registers and displacement. Finally 20 bit address is calculated and from that memory location data can be accessed.



Example:

a) MOV AX, [BX + SI + 50 H]

In this instruction the effective address is obtained by adding contents of BX register (2000), SI register (2500) and 50H.
Memory address=DS*10 + [BX] + [SI] +50H= 3000*10+2000 + 2500 + 50=34550H
The byte from the memory address 34500H is read and stored in AX register.

Program memory addressing modes:

The program memory addressing modes are used with the JMP and CALL instructions. These addressing modes are related to branching instructions, where the flow of control is transferred from one location to another location. The data can be accessed from memory based on control transfer.

Based on the control transfer there are two types of addressing modes, they are

- Intra segment addressing mode
- Inter segment addressing mode

Intra segment addressing mode:

If the flow of control is transferred within the segment then it is said to be intra segment addressing mode.

There are two types of intra segment addressing modes, they are

- Direct Intra segment addressing mode
- Indirect Intra segment addressing mode

Direct Intra segment addressing mode

In this addressing mode, the address to which the control is to be transferred lies within the same segment in the instruction and appears directly in the instruction as an immediate displacement value. In this addressing mode the displacement is calculated relative to the contents of instruction pointer (IP).

The effective address to which control will be transferred is given by the sum of 8 or 16 bit displacement and current IP contents. In case of jump instruction, if displacement is 8 bit (-127 to +128) then it is short jump and if the displacement is 16 bits (-32678 to +32677) then it is long jump.

Example:

JMP SHORT LABEL

In direct Intra segment addressing mode:

In this addressing mode, the address to which the control is to be transferred lies within the same segment in the instruction and but it is passed indirectly in the instruction. In this addressing mode the branch address is found as the contents of register or memory location.

Example:

JMP [BX]

Inter segment addressing mode:

In this addressing mode the flow of control is transferred from one segment to another segment.
There are two types of inter segment addressing modes. They are

- Inter segment direct addressing mode
- Inter segment indirect addressing mode

Inter segment direct addressing mode

In this addressing mode, the flow of control is transferred from one segment to another segment and it is specified directly in the instruction. The contents of CS and IP are specified directly in the instruction.

Example:

JMP 5000H, 2000H

Inter segment indirect addressing mode:

In this addressing mode, the flow of control is transferred from one segment to another segment and it is passed to instruction indirectly ie; the contents of memory block containing four bytes ie; IP (LSB), IP (MSB), CS (LSB), CS (MSB) sequentially.

Example:

JMP [BX]

Stack memory addressing mode:

The stack is used to hold data temporarily during program execution and also store the return address for procedures and interrupt service routines (ISR). The stack memory is a last in first out (LIFO) memory. Data are placed into stack using the PUSH instruction and taken out using the POP instruction. The CALL instruction uses the stack to hold the return address for procedure and RET instruction is used to remove return address from the stack.

The stack segment is maintained by two types of registers; they are pointer register and stack segment register. Data is pushed into or popped from the stack as word. Whenever a word is pushed into stack, the higher order 8 bits are placed into memory location specified by SP and then the lower order bits are placed into memory specified by SP. SP is decremented by two locations.

Since SP gets decremented for every push operation, the stack segment is said to be growing downwards, as for push operations, data are stored in the lower memory addresses in stack segment.

Whenever a word is popped from stack, the lower order 8 bits of word are removed from memory location specified by SP and then the higher order bits are removed from memory specified by SP. SP is incremented by two locations.

Example:

```
PUSH AX
PUSH DS
POP ES
POP [BX]
```

Instruction Set of 8086

The 8086 instructions are categorized into the following main types.

i. Data Transfer / Copy Instructions
ii. Flag transfer instructions
iii. Arithmetic and Logical Instructions
iv. Branch Instructions
v. Loop Instructions
vi. Machine Control Instructions
vii. Flag Manipulation Instructions
viii. Shift and Rotate Instructions
ix. String Instructions

-

Data Copy / Transfer Instructions:

MOV:

This instruction copies a word or a byte of data from some source to a destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number.

The general format of MOV instruction is **MOV destination, source**

Example:

```
MOV AX, BX
MOV AX, 5000H
MOV AX, [SI]
MOV AX, [2000H]
MOV [734AH], BX
MOV DS, CX
MOV CL, [357AH]
```

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.

Example:

```
PUSH AX
PUSH DS
 PUSH [5000H]
```

POP: Pop from Sack

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. The stack pointer is incremented by 2.

Example:

```
 POP AX
POP DS
POP [5000H]
```

XCHG: Exchange byte or word

This instruction exchanges the contents of the specified source and destination operands. Exchange instruction can exchange the contents of a register from the contents of a memory location. The source and destination must be either byte or word.

Example:

```
XCHG [5000H], AX
XCHG BX, AX
```

XLAT: Translate byte using look-up table

The XLAT instruction is used to translate a byte in AL from one code to another code. The instruction replaces a byte in AL register with the byte in the memory at [BX], which is one of the data items present in a look up table.

Before XLAT instruction is executed, the look up table containing the desired codes must be put in data segment and offset address of the starting location of the look up table is stored in BX. The code byte to be translated is put in AL. when XLAT is executed now; it adds the content of AL with BX to find the offset address of the data in look up table. Further the byte in that offset address will be get copied to AL.

Example:

```
 LEA BX, TABLE1
MOV AL, 04H
XLAT
```

IN: Copy a byte or word from specified port to accumulator.

The IN instruction copies data from a port to the AL or AX register. If 8 bit port is read, the data is stored in AL and if 16 bit port is read, the data is stored in AX.

The IN instruction has two formats. They are fixed port and variable port.

In fixed port type IN instruction, the 8 bit address of the port is specified directly in the instruction. With this form, any one of possible ports can be addressed.

Example:

```
IN AL, 03H
IN AX, 40H
```

For variable port type IN instruction, the port address is loaded into DX register before the IN instruction. Since DX is 16 bit register, the port address can be any one of the number between 0000H to FFFF H. Hence 65536 ports can be able to address.

Example:

```
MOV DX, 0FE H
IN AL, DX
IN AX, DX
```

OUT: Copy a byte or word from accumulator specified port.

The OUT instruction transfers a byte from AL or a word from AX to specified port. Similar to IN instruction, OUT instruction has two forms- fixed and variable ports.

Example:

```
 OUT 03H, AL
OUT DX, AX
```

LEA: Load effective address of operand in specified register.

The general format of LEA instruction is **LEA destination, source.** This instruction loads the effective address of the source operand into the destination operand.

Example:

LEA BX, ADR

LEA SI, ADR [BX]_

LDS: Load pointer to DS

This instruction loads DS register and the specified destination register in the instruction with the content of memory location specified as a source in the instruction.

Example:

LDS BX, 5000H_

LES: Load pointer to ES

This instruction loads ES register and the specified destination register in the instruction with the content of memory location specified as a source in the instruction.

Example:

LES BX, 5000H_

Flag transfer instructions:

LAHF: Load (copy to) AH with the low byte the flag register.

[AH] [Flags low byte]

This instruction copies the lower order byte of the flag register into AH.

SAHF: Stores (copy) AH register to low byte of flag register.

[Flags low byte] [AH]

This instruction stores the content of AH in lower order of the flag register.

PUSHF: Copy flag register to top of stack.

This instruction copies the flag register contents to stack.

POPF: Copy word at top of stack to flag register.

This instruction copies the contents from stack to flag register.

Arithmetic Instructions:

The 8086 provides many arithmetic operations: addition, subtraction, negation, multiplication and comparing two values.

ADD:

This instruction adds an immediate data or the contents of memory location specified in the register (source) to the contents of another register (destination) or memory location. The result is in the destination. The source and destination can be either register or memory location but can't be both memory locations.

Example:

ADD AX, 0100H

ADD AX, BX

ADD AX, [SI]

ADD AX, [5000H]

ADD [5000H], 0100H

ADC: Add with Carry

This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.

Example

ADC 0100H

ADC AX, BX

ADC AX, [SI]

ADC AX, [5000]

ADC [5000], 0100H

SUB: Subtract

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. The source and destination can be either register or memory location but can't be both memory locations.

Example:

SUB AX, 0100H

SUB AX, BX

SUB AX, [5000H]

SUB [5000H], 0100H

SBB: Subtract with Borrow

The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand

Example:

SBB AX, 0100H

SBB AX, BX

SBB AX, [5000H]

SBB [5000H], 0100H

INC: Increment

This instruction increases the contents of the specified Register or memory location by 1. Immediate data cannot be operand of this instruction.

Example

INC AX

INC [BX]

INC [5000H]

DEC: Decrement

The decrement instruction subtracts 1 from the contents of the specified register or memory location.

Example

DEC AX

DEC [5000H]

NEG: Negate

The negate instruction forms 2's complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

Example

NEG AL

AL = 0011 0101 35H

Replace number in AL with its 2's complement

AL = 1100 1011 = CBH

CMP: Compare

This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from destination operand but it does not store the result in destination.

Example

CMP BX, 0100H

CMP AX, 0100H

CMP [5000H], 0100H

CMP BX, [SI]

CMP BX, CX

MUL: Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte in source with unsigned byte in AL or multiplies unsigned word in source with unsigned word in AX. The source can be either register or memory location. The result will be stored in AX or DX register.

Example

MUL BH

MUL [DI]

IMUL: Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX. The source can be either register or memory location. The result is stored in AX or DX.

Example

IMUL BH

IMUL CX

IMUL [SI]

CBW: Convert Signed Byte to Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Example

MOV AX, 4255H	AH=42, AL=55
CBW	AX=0055 H

In this example, AL=55H is equivalent to 01010101H which means sign bit is 0. Hence after CBW the sign bit is extended in to AH register. So AH=0055H.

CWD: Convert Signed Word to Double Word

This instruction sets all the bits in DX register to the same sign bit of AX register.

Example

MOV DX, 0000H,	0000	0000	0000	0000
MOV AX, F4ECH,	1111	0100	1110	1100
CWD				

Result: DX = 1111 1111 1111 1111

In the above example, the sign bit in AX is 1 so the signed bit is extended to DX register. So DX=1111 1111 1111 1111

DIV: Unsigned division

This instruction is used to divide an unsigned word by a byte. The word must be AX and byte can be from a register or memory location. The quotient is stored in AL and remainder is stored in AH.

This instruction is used to divide an unsigned double word by a word. The double word must be AX or DX and word can be from a register or memory location. The quotient is stored in AX and remainder is stored in DX.

Example

DIV CL ; Word in AX / byte in CL ;

Quotient in AL, remainder in AH

DIV CX ; Double word in DX and AX / word in CX, and Quotient in AX, remainder in DX

ASCII & BCD Instructions

Representation of decimal numbers is

- ASCII representation
- BCD representation
    - Unpacked BCD
    - Packed BCD

ASCII:

When a number is entered in keyboard, microprocessor accepts data in form of ASCII values. For 0 to 9 decimal digits the corresponding ASCII values are 30 to 39. 8086 allows adding these data as they enter the processor for decimal digits. For example, if 5 and 9 are entered through keyboard, 8086 accepts the data as 35 and 39 ASCII values.

For processing ASCII numbers the following instructions are used

- ASCII addition (AAA)
- ASCII subtraction (AAS)
- ASCII multiplication (AAM)
- ASCII division (AAD)

For processing BCD numbers the following instructions are used

- Packed BCD addition (DAA)
- Packed BCD subtraction (DAS)

AAA: ASCII Adjust After Addition

The AAA instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL. The AAA instruction converts the resulting contents of Al to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4 bits of AL to check whether it contains valid BCD number in the range 0 to 9 and if it is greater than 9, then 6 is added to it and AH is cleared. If carry is generated it is added to AH.

AAS: ASCII Adjust AL after Subtraction

This instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. The procedure is similar to AAA instruction except for the subtraction of 06 from AL.

AAM: ASCII Adjust after Multiplication

This instruction, after execution, converts the product available In AL into unpacked BCD format. The AAM instruction, multiplies the two unpacked BCD operands and result is stored in AX.

AAD: ASCII Adjust before Division

This instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. In the instruction sequence, this instruction appears Before DIV instruction.

DAA: Decimal Adjust Accumulator

This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or AF is set, it will add 6 to lower nibble in AL. after adding 6 to lower nibble of AL, if the upper nibble of AL is greater than 9 or carry flag is set, DAA adds 60 to AL.

DAS: Decimal Adjust after Subtraction

This instruction converts the result of the subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only. If the lower nibble is greater than 9, after subtraction or AF is set, it will subtracts 6 to lower nibble in AL. after subtracting 6 to lower nibble of AL, if the upper nibble of AL is greater than 9 or carry flag is set, DAA subtracts 60 to AL.

Logical Instructions

AND: Logical AND

This instruction bit by bit ANDs the source operand that may be an immediate register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Example

AND AX, 0008H

AND AX, BX

OR: Logical OR

This instruction bit by bit ORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Example

OR AX, 0008H

OR AX, BX

NOT: Logical Invert

This instruction complements the contents of an operand register or a memory location, bit by bit.

Example

NOT AX

NOT [5000H]

XOR: Logical Exclusive OR

This instruction bit by bit XORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Example

XOR AX, 0098H

XOR AX, BX

TEST: Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this ANDing operation is not available for further use, but flags are affected.

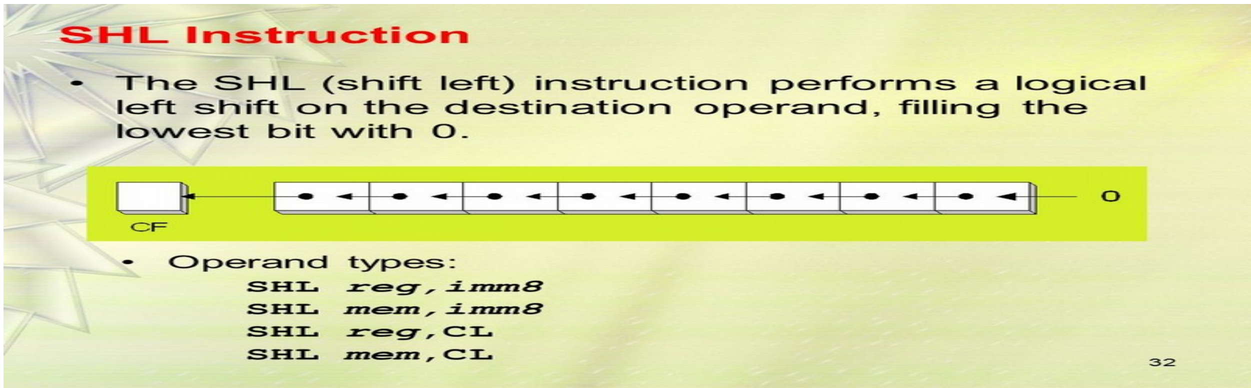
Example

TEST AX, BX

TEST [0500], 06H

SAL/SHL: SAL / SHL destination, count.

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag. The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts is indicated by count.



SHL Instruction

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.

- Operand types:

```
SHL reg,imm8
SHL mem,imm8
SHL reg,CL
SHL mem,CL
```

32

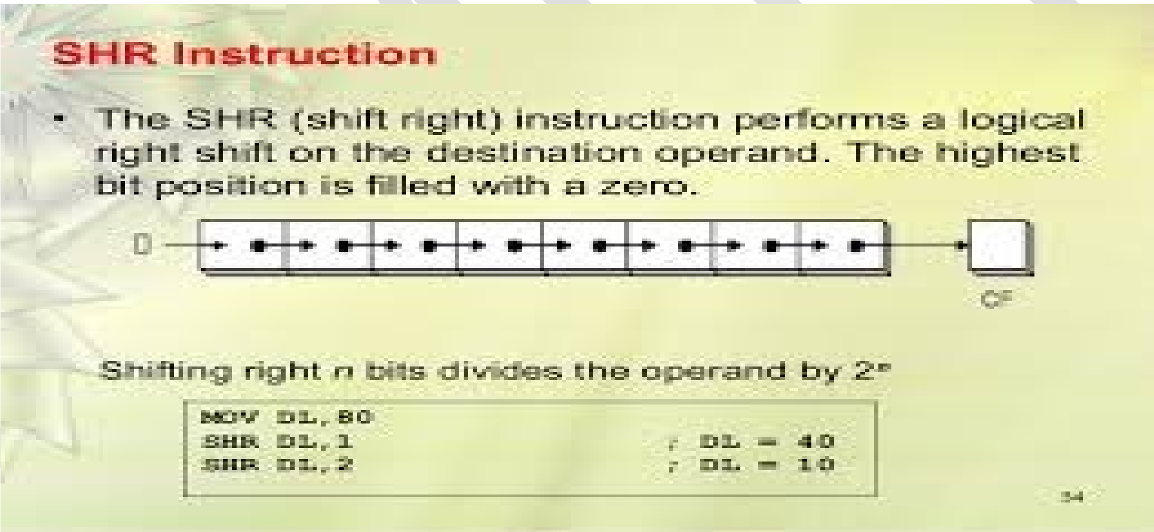
Example

SAL CX, 1

SAL AX, CL

SHR: SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word. It can be a register or in a memory location. The number of shifts is indicated by count.



SHR Instruction

- The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.

Shifting right *n* bits divides the operand by 2^n

```
MOV DL,80
SHR DL,1          ; DL = 40
SHR DL,2          ; DL = 10
```

34

Example

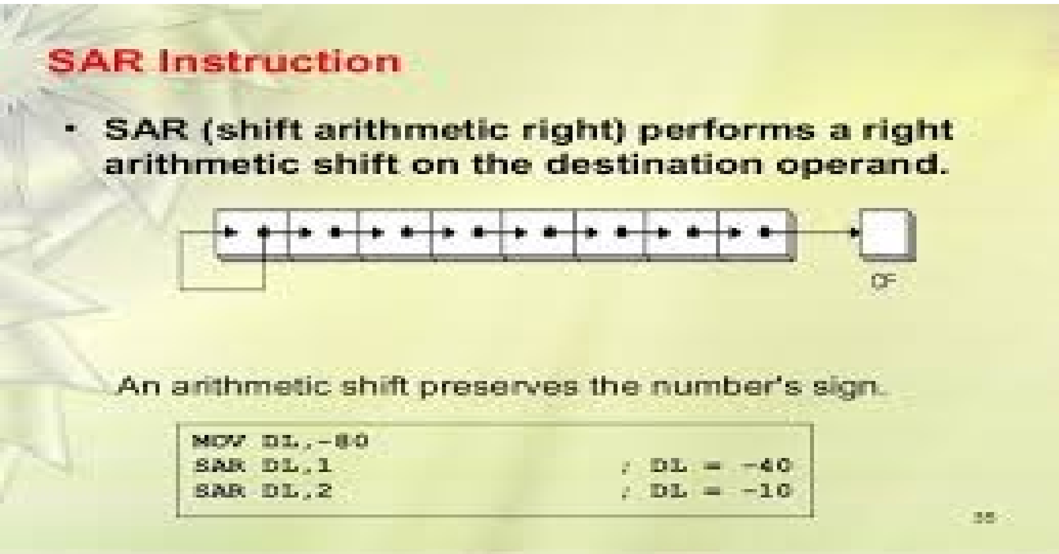
SHR CX, 1

MOV CL, 05H

SHR AX, CL

SAR: SAR destination, count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.



SAR Instruction

- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.

An arithmetic shift preserves the number's sign.

```
MOV DL,-80
SAR DL,1                  ; DL = -40
SAR DL,2                  ; DL = -10
```

35

Example

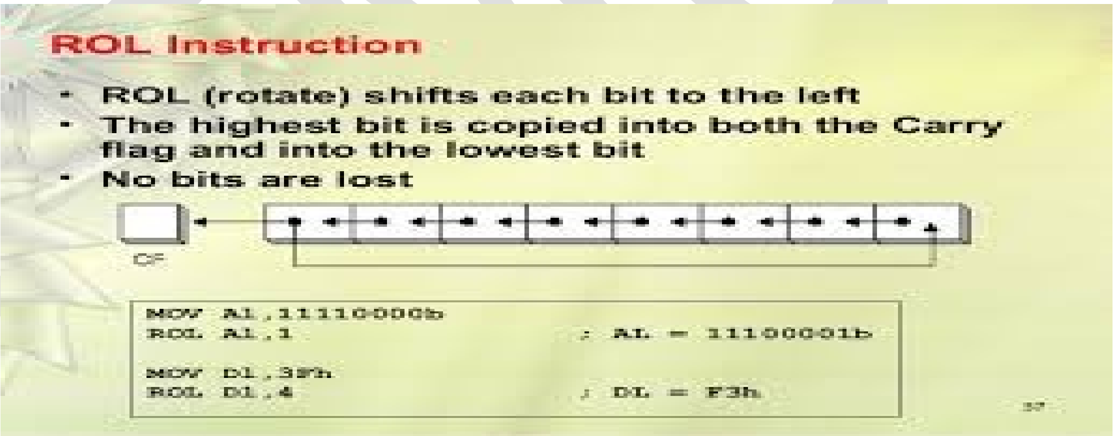
SAR BL, 1

MOV CL, 04H

SAR DX, CL

ROL Instruction: ROL destination, count

This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.



ROL Instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost

```
MOV AL,11110000b
ROL AL,1                  ; AL = 11100001b

MOV DL,3Fh
ROL DL,4                  ; DL = F3h
```

37

Example

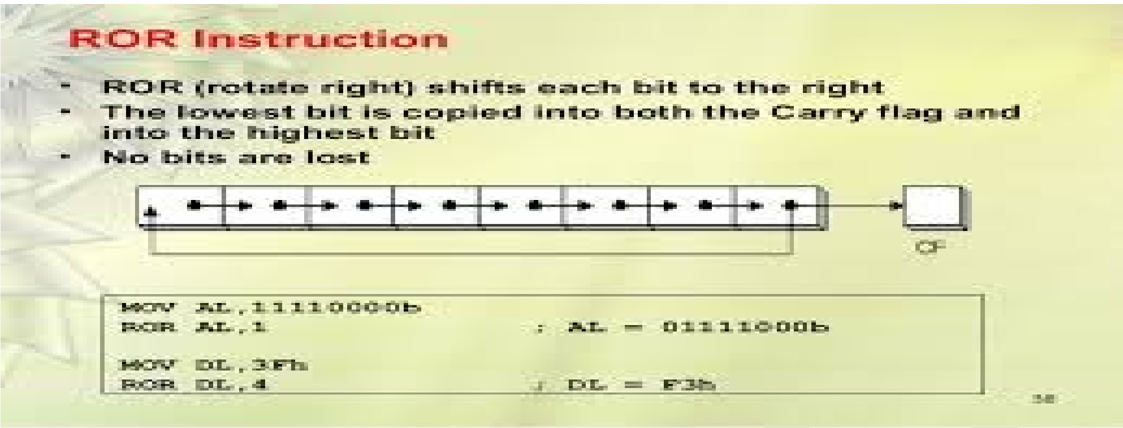
ROL CX, 1

MOV CL, 03H

ROL BL, CL

ROR Instruction: ROR destination, count

This instruction rotates all bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF.



ROR Instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost

```
MOV  AL,11110000b
ROR  AL,1               ; AL = 01111000b

MOV  DL,3Fh
ROR  DL,4               ; DL = F3h
```

Example

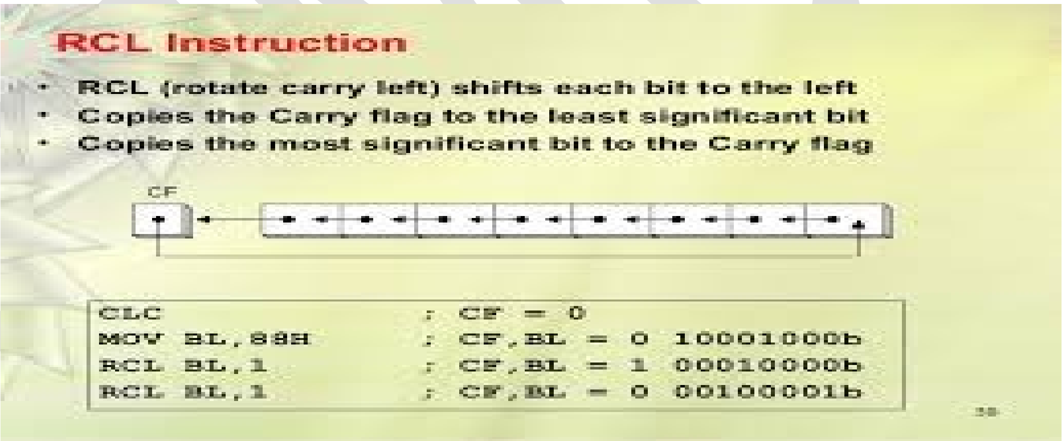
ROR CX, 1

MOV CL, 03H

ROR BL, CL

RCL Instruction: RCL destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is place as new LSB.



RCL Instruction

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag

```
CLC               ; CF = 0
MOV  BL,88H       ; CF,BL = 0 10001000b
RCL  BL,1         ; CF,BL = 1 00010000b
RCL  BL,1         ; CF,BL = 0 00100001b
```

Example

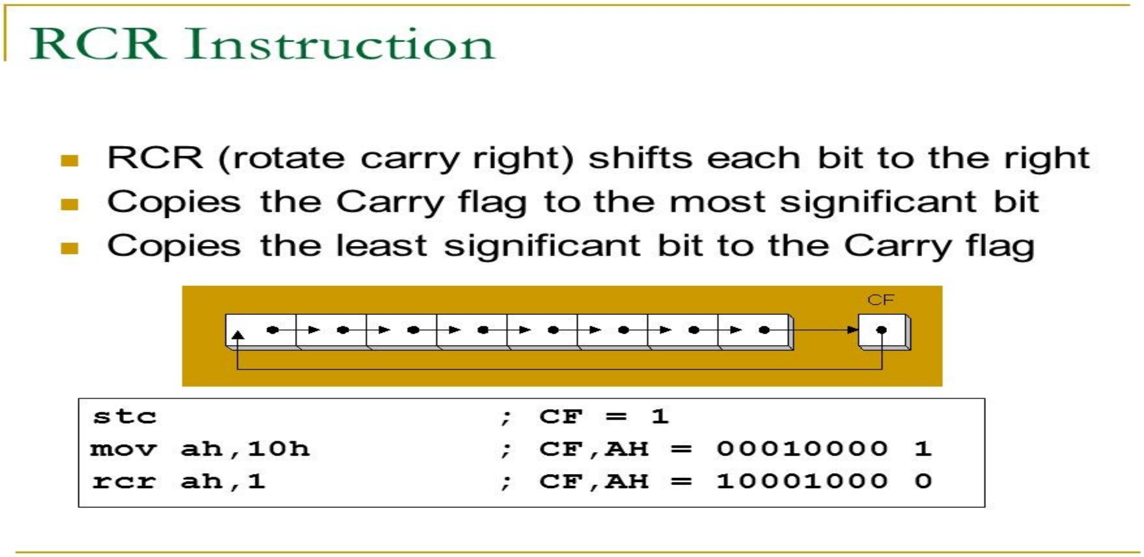
RCL CX, 1

MOV CL, 04H

RCL AL, CL

RCR Instruction: RCR destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the right along with the carry flag. LSB is placed as a new carry and previous carry is place as new MSB.



RCR Instruction

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag

```
stc                    ; CF = 1
mov ah,10h             ; CF,AH = 00010000 1
rcr ah,1               ; CF,AH = 10001000 0
```

<u>Example</u>

RCR CX, 1

MOV CL, 04H

RCR AL, CL

<u>Branch Instructions:</u>

Branch Instructions transfers the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location to be transferred.

The Branch Instructions are classified into two types

i. Unconditional Branch Instructions.
ii. Conditional Branch Instructions.

<u>Unconditional Branch Instructions:</u>

In Unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

<u>CALL:</u> Unconditional Call

This instruction is used to call a Subroutine (Procedure) from a main program. Address of procedure may be specified directly or indirectly. There are two types of procedure depending upon whether it is available in the same segment or in another segment.

i. Near CALL
ii. For CALL

On execution this instruction stores the incremented IP & CS onto the stack and loads the CS & IP registers with segment and offset addresses of the procedure to be called.

RET: Return from the Procedure.

At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with Flags are retrieved into the CS, IP and Flag registers from the stack and execution of the main program continues further.

INT N: Interrupt Type N.

In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When INT N instruction is executed, the type byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from memory block in 0000 segment.

INTO: Interrupt on Overflow

This instruction is executed, when the overflow flag OF is set. This is equivalent to a Type 4 Interrupt instruction.

JMP: Unconditional Jump

This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement. No Flags are affected by this instruction.

IRET: Return from ISR

When it is executed, the values of IP, CS and Flags are retrieved from the stack to continue the execution of the main program.

LOOP: LOOP Unconditionally

This instruction executes the part of the program from the Label or address specified in the instruction up to the LOOP instruction CX number of times. After each iteration, CX is decremented automatically and JUMP IF NOT ZERO structure.

Conditional Branch Instructions

When this instruction is executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the op code is satisfied. Otherwise execution continues sequentially.

JZ/JE Label Transfer execution control to address ‘Label’, if ZF=1.

JNZ/JNE Label Transfer execution control to address ‘Label’, if ZF=0

JS Label Transfer execution control to address ‘Label’, if SF=1.

JNS Label Transfer execution control to address ‘Label’, if SF=0.

JO Label Transfer execution control to address ‘Label’, if OF=1.

JNO Label Transfer execution control to address ‘Label’, if OF=0.

JNP Label Transfer execution control to address ‘Label’, if PF=0.

JP Label Transfer execution control to address ‘Label’, if PF=1.

JB Label Transfer execution control to address ‘Label’, if CF=1.

JNB Label Transfer execution control to address ‘Label’, if CF=0.

JCXZ Label Transfer execution control to address ‘Label’, if CX=0

Conditional LOOP Instructions

LOOPZ / LOOPE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

LOOPNZ / LOOPENE

Label Loop through a sequence of instructions from label while ZF=1 and CX=0.

String Manipulation Instructions

A series of bytes or words available in memory at consecutive locations is referred as String byte or String word. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent.

The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

I. Starting and End Address of the String.
II. Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. If it is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two.

REP: Repeat Instruction Prefix

This instruction is used as a prefix to other instructions, the instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one).

i. REPE / REPZ - repeat operation while equal / zero.
ii. REPNE / REPNZ - repeat operation while not equal / not zero. These are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB / MOVSW: Move String Byte or String Word

Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents.

CMPS: Compare String Byte or String Word

The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. The contents of source location are subtracted from the destination location when CMP instruction is executed. If both the byte or word strings are equal, zero Flag is set. The REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP Prefix is False.

SCAN: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES: DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string. Whenever a match to the specified operand is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

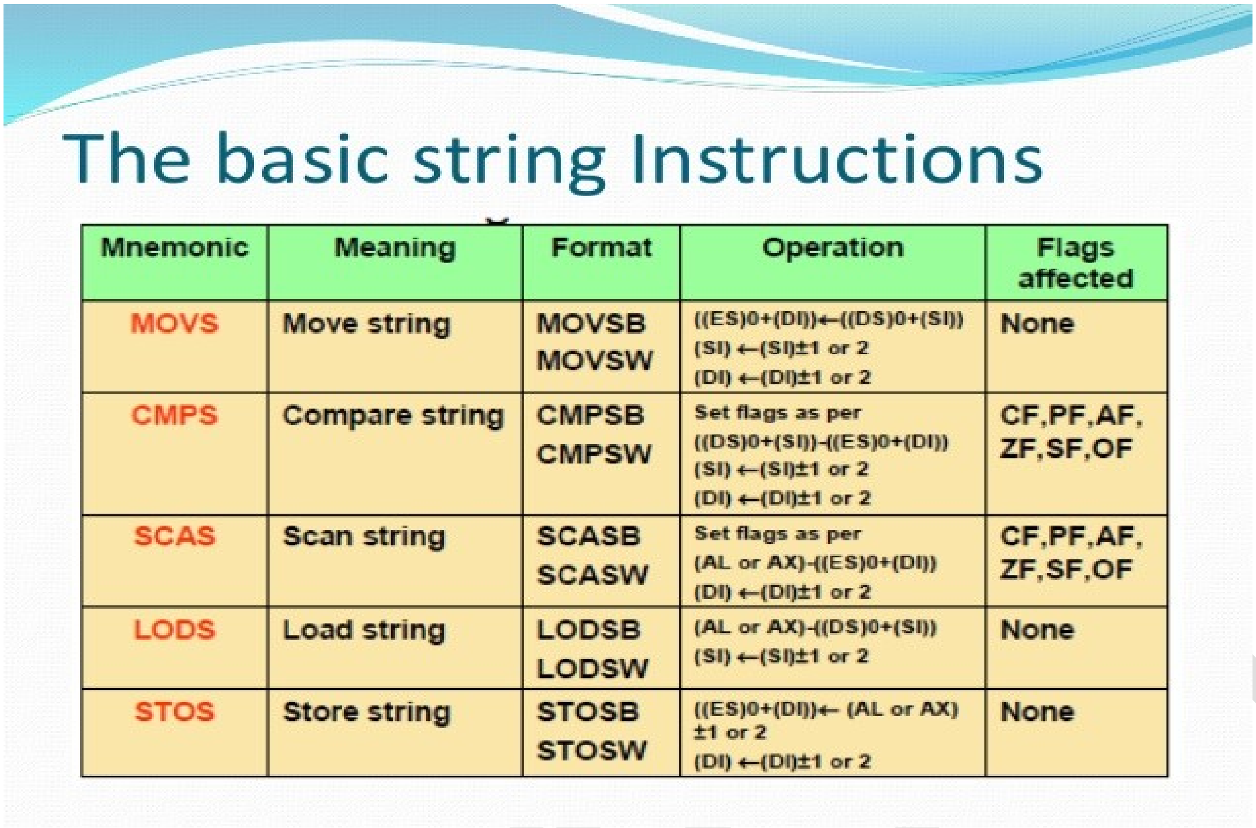
LODS: Load String Byte or String Word

The LODS instruction loads the AL / AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF, if it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other Flags are affected by this instruction.

STOS: Store String Byte or String Word

The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly; No Flags are affected by this instruction.

The direction Flag controls the String instruction execution, the source index SI and Destination Index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode, SI and DI are decremented automatically after each iteration. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically after each iteration.



The basic string Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
MOVS	Move string	MOVSB MOVSW	((ES)0+(DI))←((DS)0+(SI)) (SI) ←(SI)±1 or 2 (DI) ←(DI)±1 or 2	None
CMPS	Compare string	CMPSB CMPSW	Set flags as per ((DS)0+(SI))-((ES)0+(DI)) (SI) ←(SI)±1 or 2 (DI) ←(DI)±1 or 2	CF,PF,AF, ZF,SF,OF
SCAS	Scan string	SCASB SCASW	Set flags as per (AL or AX)-((ES)0+(DI)) (DI) ←(DI)±1 or 2	CF,PF,AF, ZF,SF,OF
LODS	Load string	LODSB LODSW	(AL or AX)-((DS)0+(SI)) (SI) ←(SI)±1 or 2	None
STOS	Store string	STOSB STOSW	((ES)0+(DI))← (AL or AX) ±1 or 2 (DI) ←(DI)±1 or 2	None

Flag Manipulation and Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These instructions are categorized into two types:

1. Flag Manipulation instructions.

2. Machine Control instructions.

Flag Manipulation instructions

The Flag manipulation instructions directly modify some of the Flags of 8086.

i. CLC – Clear Carry Flag. CF=0
ii. CMC – Complement Carry Flag.
iii. STC – Set Carry Flag. CF=1
iv. CLD – Clear Direction Flag. DF=0
v. STD – Set Direction Flag. DF=1
vi. CLI – Clear Interrupt Flag. IF=0
vii. STI – Set Interrupt Flag. IF=1

Machine Control instructions(processor control instructions)

The Machine control instructions control the bus usage and execution

i. WAIT – Wait for Test input pin to go low.
ii. HLT – Halt the process.
iii. NOP – No operation.
iv. ESC – Escape to external device like NDP

v. LOCK – Bus lock instruction prefix

WAIT Instruction:

When WAIT instruction is executed it takes microprocessor to idle condition. The CPU will not do any processing during this condition, until TEST pin becomes low or interrupt is received on INTR or NMI pin.

HLT Instruction:

When ever HLT instruction is executed, it causes 8086 to stop fetching and executing instructions. The 8086 will enter into halt state. The only way to get the processor out of halt state is the interrupt signal on INTR pin or interrupt signal on NMI pin or reset signal on RESET input.

NOP Operation:

This instruction simply takes three clock cycles and does no processing. After this it will execute the next instruction. This instruction is used to provide delay between two instructions.

ESC Instruction:

When ever this instruction is executed the microprocessor passes the information to other coprocessor.

LOCK Instruction:

This instruction is used as prefix. This prefix makes sure that during execution of the instruction, control of system is not taken by other microprocessor. It prevents the access of system bus from others.

Assembly language programming:

Assembly language programs can be converted into machine language programs either by manually finding the op code for each instruction in the program or by using assembler and executing in 8086 based systems.

Since manually finding the op code of each instruction of 8086 is time consuming, the assembler is normally used to converting assembly language programs to machine language programs.

Many assemblers, such as Microsoft Macro Assembler (MASM), Turbo Assembler (TASM), and DOS Assembler are used to convert the 8086 assembly language programs into machine language programs.

While using these assemblers, the assembly language program is written using assembler directives. Assembler directives are the commands to the assembler to indicate the size of variable, number of bytes or words to be reserved in the memory, value of a constant, name of the segment, etc., in the program.

Assembler directives are the alphabet strings which are given to the assembler as hints or directions while converting assembly language programming to machine language programming.

There are two types of statements in an assembly language programming. They are

1. Instructions: These are translated into object codes by the assembler.
2. Directives: These give directions to the assembler during assembly process.

Types of assemblers:

There are two types of assemblers. They are

- **One pass assembler:** The assembler scans the program statements one by one from left to right starting with first statement to last statement indicated by end statement. This is called the pass. The purpose of first pass is to provide the assembler the location of labels.
- **Two pass assembler:** The assembler scans the program statements twice. The first scan consists of labels of addresses. The second scan translates the assembly program to binary codes.

Assembly language program development tools:

Editor:

An editor is a program which allows creating the file containing the assembly language statements. The assembly programs are saved as .ASM file.

Assembler:

An assembler is a program used to translate assembly language programs to machine language programs. It produces . OBJ file.

Linker:

A linker is a program used to join several file into large object file. It produces . EXE file so that the program becomes executable.

Locator:

A locator is a program which is used to assign the specific address of where the segments of object code are to be loaded into memory. It converts .EXE file to .BIN file.

Debugger:

A debugger is a program which allows loading .OBJ code program into system memory, executing program and troubleshooting.

Emulator:

An emulator is a mixture of hardware and software. It is used to test and debug the hardware and software of external system.

Assembler Directives

In assembly language programming while converting assembly language program to machine language some errors may occur. To overcome these errors hints or directions are to be given to the assembler. These directions or hints are given through Assembler directives.

Assembler directives are alphabet strings which are given to assembler as directions.

DB - Defined Byte.
DD - Defined Double Word
DQ - Defined Quad Word
DT - Define Ten Bytes
DW - Define Word

DB - DB directive is used to declare a byte type variable or to store a byte in memory location.

Example:

1. PRICE DB 49h, 98h, 29h ;

   Declare an array of 3 bytes, named as PRICE and initialize.
2. NAME DB ‘ABCDEF’ ;

   Declare an array of 6 bytes and initialize with ASCII code for letters
3. TEMP DB 100 DUP(?) ;

   Set 100 bytes of storage in memory and give it the name as TEMP, but leave the 100 bytes uninitialized. Program instructions will load values into these locations.

DW - The DW directive is used to define a variable of type word or to reserve storage location of type word in memory.

Example:

MULTIPLIER DW 437Ah;

This declares a variable of type word and named it as MULTIPLIER. This variable is initialized with the value 437Ah when it is loaded into memory to run.

EXP1 DW 1234h, 3456h, 5678h ; this declares an array of 3 words and initialized with specified values.

STOR1 DW 100 DUP (0);

Reserve an array of 100 words of memory and initialize all words with 0000.Array is named as STOR1.

DD - The DW directive is used to define a variable of type double word or to reserve storage location of type double word in memory.

DQ - The DW directive is used to define a variable of type quad word or to reserve storage location of type quad word in memory.

DT - The DW directive is used to define a variable of type ten words or to reserve storage location of type ten words in memory.

END - 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. Carriage return is required after the END directive.

ENDP - ENDP directive is used along with the name of the procedure to indicate the end of a procedure to the assembler

Example:

SQUARE_NUM PROCE; It start the procedure some steps to find the square root of a number

SQUARE_NUM ENDP

Hear it is the End for the procedure

ENDS - This ENDS directive is used with name of the segment to indicate the end of that logic segment.

Example:

CODE SEGMENT;

Hear it start the logic segment containing code. Some instructions statements to perform the logical operation

CODE ENDS;

End of segment named as CODE

EVEN - This 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 is at even address 8086 can read a memory in 1 bus cycle. If the word starts at an odd address, the 8086 will take 2 bus cycles to get the data. A series of words can be read much more quickly if they are at even address. When EVEN is used the location counter will simply incremented to next address and NOP instruction is inserted in that incremented location.

Example:

DATA1 SEGMENT;

Location counter will point to 0009 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(0);

Array of 100 words will start from an even address for quicker read DATA1 ENDS

EQU - This EQU directive is used to give a name to some value or to a symbol. Each time the assembler finds the name in the program, it will replace the name with the value or symbol you given to that name.

Example:

FACTOR EQU 03H;

ASSUME

The ASSUME directive is used to tell the assembler that the name of the logical segment should be used for a specified segment. The 8086 works directly with only 4 physical segments: a Code segment, a data segment, a stack segment, and an extra segment.

Example:

ASUME CS: CODE

This tells the assembler that the logical segment named CODE contains the instruction statements for the program and should be treated as a code segment.

ASUME DS: DATA
This tells the assembler that for any instruction which refers to a data in the data segment, data will found in the logical segment DATA.

GROUP - The GROUP directive is used to group the logical segments named after the directive into one logical group segment.

LABLE- This directive indicates that name is assigned to the current memory location.

NAME- This directive indicates the name given to the program module.

OFFSET – This directive determines offset address of the segment.

INCLUDE - This INCLUDE directive is used to insert a block of source code from the named file into the current source module.

ORG- This directive is used to indicate the starting memory location of the program.

PROC - The PROC directive is used to identify the start of a procedure. The term near or far is used to specify the type of the procedure.

Example:
SMART PROC FAR;

This identifies that the start of a procedure named as SMART and instructs the assembler that the procedure is far.

SMART ENDP

This PROC is used with ENDP to indicate the break of the procedure.

PTR - This PTR operator is used to declare the type of label, variable, or memory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, particular label, variable, or memory operand is treated as an 8 bit quantity, while if the prefix is WORD, it is treated as 16 bit quantity.

Example:
INC BYTE PTR [BX];

Increment the byte pointed to by BX. This PTR operator can also be used to override the declared type of variable.

SEGMENT- This directive indicates the start of the logical segment.

PUBLIC - The PUBLIC directive is used to instruct the assembler that a specified name or label will be accessed from other modules.

Example:
PUBLIC DIVISOR, DIVIDEND;

These two variables are public so these are available to all modules. If an instruction in a module refers to a variable in another assembly module, we can access that module by declaring as EXTRN directive.

SHORT – This type of directive tells assembler that only 1 byte of displacement is required to jump instruction.

Example:
JMP SHORT MULTIPLY

TYPE - TYPE operator instructs the assembler to determine the type of a variable and determines the number of bytes specified to that variable.

Example:
Byte type variable – assembler will give a value 1
Word type variable – assembler will give a value 2
Double word type variable – assembler will give a value 4
ADD BX, TYPE WORD_ ARRAY;

MACRO and ENDM:

Suppose a number of instructions occur repeatedly in the main program, the program becomes lengthy. In such situations a macro definition is assigned with repeatedly appearing string of instructions. The process of assigning a label or macro name to repeatedly appearing string of instructions is called macro definition. The macro name is then used throughout the main program to refer to that string of instructions.

A MACRO can be defined anywhere in a program, using the directive MACRO and ENDM. The ENDM directive marks the end of the instructions or statements assigned to macro name.

EXTRN and PUBLIC:

The directive EXTRN (external) informs the assembler that the procedures, labels, and names declared after this directive have already been defined in some other segments and in the segments where they actually appear, they must be declared public, using public directive.

LOCAL:

The label, variables, constants or procedures declared LOCAL in a module are to be used only by that particular module.

Global:

The label, variables, constants or procedures declared GLOBAL in a module are to be used by other modules of the program.

Modular programming

In a program it may be necessary to perform a particular a task repeatedly. The formulation of complex programs from number of complex sequences called program modules which performs a well defined task is referred as modular programming.

Large programs are broken into small segments called modules. Each module implements a specific function and has its own code segment and data segment.

Procedures are useful in such situations. A procedure is a group of instructions that usually performs a task. It forms a reusable section of software which is stored once in memory. This saves memory space and makes it easier to develop the software. A procedure is a sequence of instructions, which can be employed repeatedly within a longer program.

To make software development faster, it is better to develop and test in the form of small program segments. This splitting does not cause any loss of capability to the program. The advantages of procedures over single programs are-

1. It reduces the code length and memory requirement
2. It reduces development time as the modifications in the procedures are localized, which can be debugged and tested separately.
3. It supports modular programming methodology and improves the legibility of a programs the flow of logic is well defined.
4. Since it is possible to develop library of procedures for most commonly used task, which can be shared by users from library, unnecessary duplication of codes can be avoided. A list of library modules may be only requirement for the user.
5. Procedure provides a flexible and convenient way of exchanging information between the application programs.

In order to handle procedures, operations required by the processor systems are invoking or calling a procedure (called program) and returning from the module back to main program (calling program).

It is necessary to provide the input parameters to the procedure when it is called and to return back to the result after execution. The number of input/output parameters passed to and from the procedure can vary. Even the parameters to be passed may be control/status information which may not have peripheral for data transfer.

Since the main module and procedure use processor registers/RAM without any difference, they can be used to hold the input as well as output parameter information. If data is small in amount to exchange there is no problem but if large amount of data is to be exchanged then information must be stored in memory location and it is necessary to pass address pointer to the module. To reduce this problem, stack can be used for passing the parameter.

The procedure may be classified as follows:

1. Intra segment procedure(Near procedure)
2. Inter segment procedure (Far procedure)
3. Re – entrant procedure
4. Recursive procedure

Intra segment procedure (Near procedure)

These procedures defined in the same segment of the main program module. They can be identified by near directive.

Inter segment procedure (Far procedure)

These procedures are not defined in main program module but they are defined in other segment. They can be identified by far directive.

Re – entrant procedure

They define the procedure that can be interrupted, used and re entered without losing data.

Recursive procedure

These procedures that can be called by themselves and are often used to work with complex data structures called trees.

The instructions that are used to handle the procedures:

1. CALL
2. RET
3. RETF
4. RETN

CALL Instruction:

When CALL instruction is executed, the instruction performs the following two operations:

1) Stores the return address to which the procedure will return to the execution.
2) Modified the contents of instruction pointer register so that it points to the starting address of the procedure, depending upon whether it is an intra segment or an inter segment call.

Types of CALL instructions:

Direct CALL:

If the procedure is in the same segment then the processor produces the starting address of the procedure by adding 16 bit displacement contained in the instruction to the contents of IP.

If the procedure is in another segment then IP and CS register contents are changed to transfer control to the procedure.

In Direct CALL:

If the procedure is in the same segment then the processor produces the starting address of the procedure by adding 16 bit value specified by any of the general purpose register in the instruction to the contents of IP.

If the procedure is in another segment then it replaces IP and CS register contents with 16 bit values from memory locations whose address is specified by MOD type in the instruction.

RET Instruction:

The RET instruction is the last instruction in the procedure. At the end of the procedure, the value saved in the stack is loaded back in the IP register to return execution to the calling program so that the control is transferred to the main program. The assembler will automatically code a near RET for a near procedure and a far RET for far procedure.

RETF:

This instruction copies the return values from the stack back into the IP and CS registers to transfer control to next line in main program.

RETN:

This instruction copies the return values from the stack back into the IP register.

Macro

A macro is a group of instructions that performs a task. It is inserted in the program during the assembly process. Macro instructions are provided in the program by the assembler at a point where they are invoked by using their names. A macro is a sequence of code that needs to be written only once but basic structure can repeated several times within a module by giving its name.

The code to be repeated is called the prototype code, and the prototype code along with the statements for referencing and terminating is called the macro definition. The procedure for using a macro is to give macro definition and then declare it at various appropriate points within a program by placing the statement macro calls. When a macro is encountered by the assembler, the assembler replaces the call with the macro’s code. It is proceed by a macro definition and completed by a macro terminator.

The macro and endm directives are used to define a macro sequence. The first statement of a macro contains the name and any parameters associated with it. It is termed as definition. The last statement endm is called terminator. All the statements between name and terminator define a macro body. When a macro is to be used its name is written. This is called macro call. The assembler replaces the call with the code. This is called macro expansion.

Example:

```
PUSH_ALLMACRO       ; Definition
PUSH AX             ; Macro body
PUSH BX
PUSH CX
PUSH DX
PUSH DI
PUSH SI
ENDM                ; Terminator
```

Comparison between procedure and macro

PROCEDURE	MACRO
• Procedures are called during program execution	• Macros are inserted during assembly process
• Procedures are assembled and executed separately	• Macros cannot be executed separately
• Procedures reduces memory requirements	• There will be no change in memory requirements for macros
• Procedures can be defined anywhere and in any segment	• Macros can be defined in the same program
• Procedure requires a special CALL statement	• Macros can be used just by using its name
• Procedure can be used by any assembler	• Macros can be used if assembler supports MACRO features
• In procedures parameters can be passed through registers and memory or stack	• In macros parameters can passed as a part of the statement that calls MACRO
• In procedure machine code is put only	• In macros machine code is generated

once in memory	each time when it is called
• Procedures can be accessed by CALL and RETURN mechanism during program execution	• Macros can be accessed during assembly process when a name given to it is defined.

UNIT III

Introduction to microcontrollers

A microcontroller is a special digital processor system which is designed for specific application. Microcontroller is a single chip micro computer.

A microcomputer consists of digital computer and a microprocessor along with memory, I/O devices, CPU, I/O ports, timers/counters, ADC, DAC, serial ports, interrupt logic, oscillator circuit and functional blocks.

A microcontroller is a highly integrated single chip, which consists of on chip CPU (Central Processing Unit), RAM (Random Access Memory), EPROM/PROM/ROM (Erasable Programmable Read Only Memory), I/O (input/output) – serial and parallel, timers, interrupt controller.

S .No	Microprocessor	Microcontroller
1	A microprocessor is a general purpose device which is a single VLSI chip called a CPU.	A microcontroller is a dedicated chip which is also called single chip computer.
2	A microprocessor do not contain on chip I/O Ports, Timers, Memories etc.	A microcontroller includes RAM, ROM, serial and parallel interface, timers, interrupt circuitry (in addition to CPU) in a single chip.
3	It has many instructions to move the data between memory and CPU.	It has one or two instructions to move the data between memory and CPU.
4	Microprocessor instructions are mainly nibble or byte addressable	Microcontroller instructions are both bit addressable as well as byte addressable.
5	Access times for memory and I/O devices are more.	Access times for built-in memory and I/O devices are less.
6	Microprocessor based system design is complex and expensive	Microcontroller based system design is rather simple and cost effective
7	Microprocessor based system requires more hardware.	Microcontroller based system requires less hardware reducing PCB size and increasing reliability.
8	It has single memory for data and code.	It has separate memory map for data and code.
9	Less number of pins are multifunctional.	More number of pins are multifunctional.

Computer architectures

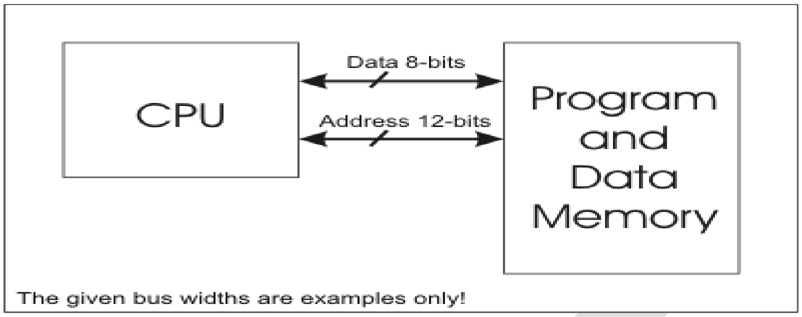
RISC and CISC CPU architectures

RISC	**CISC**
Instruction takes one or two cycles	Instruction takes multiple cycles
Only load/store instructions are used to access memory	In additions to load and store instructions, memory access is possible with other instructions also.
Instructions executed by hardware	Instructions executed by the micro program
Fixed format instruction	Variable format instructions
Few addressing modes	Many addressing modes
Few instructions	Complex instruction set
Most of the have multiple register banks	Single register bank
Highly pipelined	Less pipelined
Complexity is in the compiler	Complexity in the micro program

Harvard & Von- Neumann CPU architecture

Von Neumann Architecture:

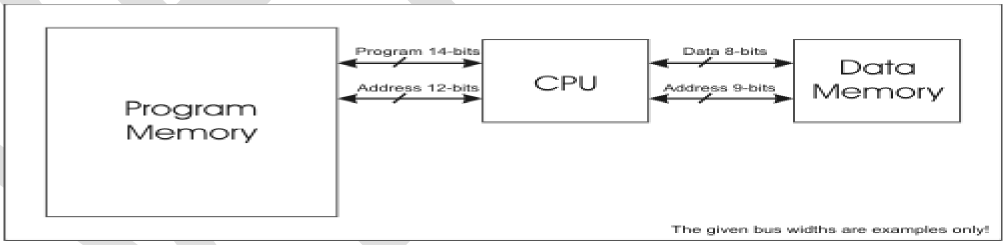
- It is named after the mathematician and early computer scientist John Von Neumann.
- The computer has single storage system (memory) for storing data as well as program to be executed.
- It consists of a single set of address/data buses between CPU and memory
- Processor needs two clock cycles to complete an instruction. Pipelining the instructions is not possible with this architecture.



- In the first clock cycle the processor gets the instruction from memory and decodes it. In the next clock cycle the required data is taken from memory. For each instruction this cycle repeats and hence needs two cycles to complete an instruction.

Harvard Architecture:

- The name is originated from "Harvard Mark I" a relay based old computer.
- The computer has two separate memories for storing data and program.
- It consists of two sets of address/data buses between CPU and memory
- Processor can complete an instruction in one cycle if appropriate pipelining strategies are implemented.
- In the first stage of pipeline the instruction to be executed can be taken from program memory. In the second stage of pipeline data is taken from the data memory using the decoded instruction or address.
- Most of the modern computing architectures are based on Harvard architecture. But the number of stages in the pipeline varies from system to system.



Features of MSP 430

MSP is a Mixed Signal Processor developed by Texas Instrument.

It is a 16 bit processor with Von-Neumann based architecture.

It is specifically designed for low power applications.

It has RISC (Reduced Instruction Set Computer) CPU.

It has 16 bit data bus and address bus.

Registers in CPU are 16 bit which can be interchangeably used for storing data and address.

It operates at 16 MHz frequency.

MSP 430 has RAM size of 128 bytes and 2 KB of Flash ROM.

All the blocks in the device are connected with memory address bus and memory data bus.

MSP 430 can drive numerous portable devices directly.

MSP 430 supports big endian and little endian architecture orderings.

It supports serial communication protocols like I^{2}C, SPI, and UART.

It has two 8 bit I/O ports.

Several features of MSP 430 make it suitable for low power and portable applications.

It is very easy to put the device in low power mode by controlling the bits in status register.

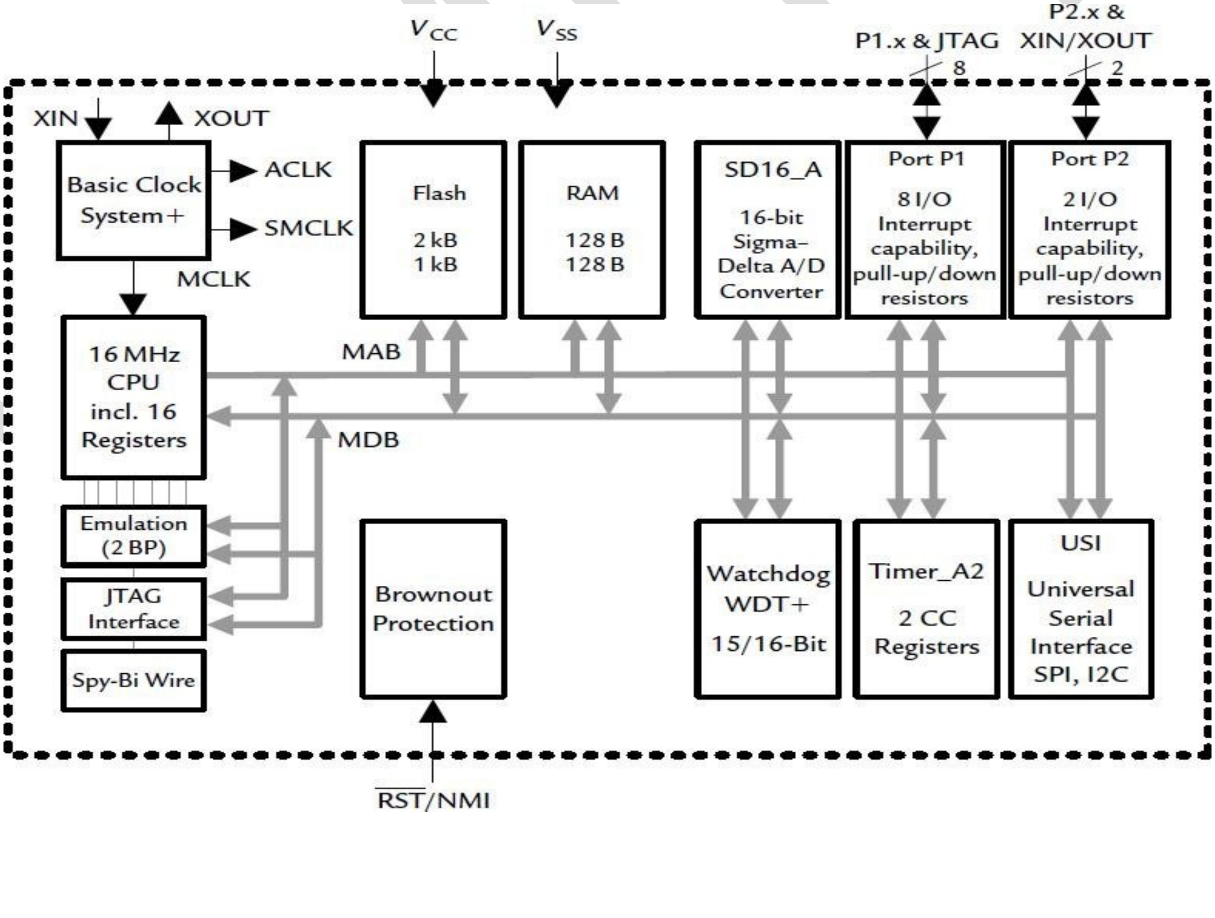
MSP 430 can be awakened by interrupt and return automatically to its low power mode after handling the low power mode.

There is wide choice of clocks.

Many peripherals can access without CPU for more time.

Many portable devices like LCD can directly access MSP 430.

Architecture of MSP 430



The MSP 430 architecture has following blocks.

Clock system
Flash memory
RAM
CPU
Emulation
JTAG Interface
SPY-Bi Wire
Watch dog timer
Brown out protection
Port
A/D converter
USI/SPI/I^{2}C
Timer registers

Clock system

The clock system of MSP 430 provides three clocks namely Master clock (MCLK), Subsystem Master Clock (SMCLK) and Auxiliary clock (ACLK).

- The master clock is used by the CPU and few peripherals.
- SMCLK is distributed to peripherals.
- ACLK is also distributed to peripherals.

MCLK & SMCLK run at same frequencies which are at mega hertz range. ACLK is derived from a watch crystal and therefore runs at lower frequency.

Memory

- Code access is always performed on even addresses.
- Data can be accessed as bytes or words.
- The addressable memory space is 64 KB

Flash Memory

Flash memory is a widely used, reliable, and flexible nonvolatile memory to store software code and data in a microcontroller. Word or byte tables can be stored and used without the need to copy the tables to RAM before using them.

Flash memory can be both programmed and electrically erased.

RAM

Random access memory is used to store both code and data. The size of RAM is 128 bytes.RAM starts at 0200h. End address depends on the amount of RAM present and varies by device.

The CPU Core

The CPU consists of an instruction decoder, arithmetic logic unit, and a register file (group of registers). The instruction decoder is responsible for translating the numeric program instructions into processor actions, the arithmetic logic unit carries out additions, subtractions, logical operations and so on. The register file consists of 16, 16 bit registers (a register is simply a storage location inside the processor that can hold a single 16 bit number). The registers are numbered R0 to R15. The first 4 of these (R0 to R3) have special purpose, the remainder, R4 to R15 are for general purpose.

Registers:

MSP 430 has sixteen 16-bit registers. These registers do not have address in the main memory map. First four registers have dedicated alternate functions and the remaining 12 registers are used as working registers for general purposes.

15	... bits...	0
R0/PC	program counter	0
R1/SP	stack pointer	0
R2/SR/CG1	status register	
R3/CG2	constant generator	
R4	general purpose	
⋮		
R15	general purpose	

Program counter, PC:

This contains the address of the next instruction to be executed.

Stack pointer, SP:

MSP430 uses the top (high addresses) of the main RAM as stack memory. The stack pointer holds the address of the most recently added word and is automatically adjusted as the stack grows downward in memory or shrinks upward.

Status register, SR:

This contains a set of flags (single bits), whose functions fall into three categories. The most commonly used flags are C, Z, N, and V, which give information about the result of the last arithmetic or logical operation. The Z flag is set if the result was zero and cleared if it was nonzero, for instance. Setting the GIE bit enables maskable interrupts. The final group of bits is CPUOFF, OSCOFF, SCG0, and SCG1, which control the mode of operation of the MCU. All systems are active when all bits are clear.

15	···	9	8	7	6	5	4	3	2	1	0
reserved			V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C

The C, Z, N, and V bits are affected by many of the operations performed by the ALU

- The carry bit C is set when an extra bit generated from the result of an arithmetic operation. The carry flag also takes part in rotations and shifts. It is sometimes used as temporary storage to pass a bit from one register to another or to a subroutine.
- The zero flag Z is set when the result of an operation is 0. A common application is to check whether two values are equal: They are subtracted and the Z bit is tested to see whether the result is 0, which shows that the values are the same.
- The negative flag N is made equal to the MSB of the result, which indicates a negative number if the values are signed.

- The signed overflow flag V is set when the result of a signed operation has overflowed, even though a carry may not be generated.
- Enable Interrupts

  Setting the general interrupt enable or GIE bit enables maskable interrupts, provided that the individual sources of interrupts have themselves been enabled. Clearing the bit disables all maskable interrupts. There are also non maskable interrupts, which cannot be disabled with GIE.

Control of Low-Power Modes:

The CPUOFF, OSCOFF, SCG0, and SCG1 bits control the mode of operation of the MCU. All systems are fully operational when all bits are clear. Setting combinations of these bits puts the device into one of its low-power modes (LPM0 – 4).

Constant generator:

This provides the six most frequently used values so that they need not be fetched from memory whenever they are needed. It uses both R2 and R3 to provide a range of useful values by exploiting the CPU's addressing modes.

General purpose registers:

The remaining 12 registers, R4–R15, are general working registers. They may be used for either data or addresses because both are 16-bit values, which simplify the operation significantly.

Emulator:

Emulator, JTAG interface & Spy-Bi wire are used to communicate with desktop computer when downloading program and debugging the program. Emulator and simulator is used for checking the operation of the processor.

JTAG:

JTAG Joint Test Action Group is a standardized interface to MSP 430 microcontroller with desktop computer that can used for controlling & monitoring during the programmer system testing and debugging. JTAG uses 16,24 wires for its operation.

Spy-Bi wire:

Spy-Bi wire is used to used to download the software. It requires only two wires. It is similar to JTAG interface which is also used to communicate MSP430 with desktop computer.

Watchdog timer:

Watchdog timer is a safety feature, which resets the processor if any errors or malfunction occurs in the system. The main purpose of Watchdog timer is to protect the system against failure of the software such as hang or entering into the infinite loop etc. If the selected time interval expires, a system reset is generated.

Brown out protection:

Brown out protection is used to protect the system if any voltage fluctuations occur in the system to dangerous level.

I/O ports:

I/O ports are used to communicate with the MSP430 by the user or external environment. MSP430 is having no. of GPIO ports and each port size is 8-bit. Each I/O pin is individually configurable as I/P or O/P pin.

A/D converters:

Analog signals are converted to digital signals by using ADC. MSP 430 utilizes sigma delta analog to digital converters to convert analog signal to digital signal.

Timers:

Timers are used to generate delays. It can also be used as counters which perform counting operations.

Power supply:

There are GND & VCC connections for power supply. For ground it has 0 V and for VCC it provides + 5V. MSP 430 is applicable to low power applications it can operate up to 1.8 V to 3.6 V.

All the blocks in MSP 430 are connected through Memory address bus (MAB) and Memory data bus (MDB). MAB is unidirectional and MDB is bidirectional.

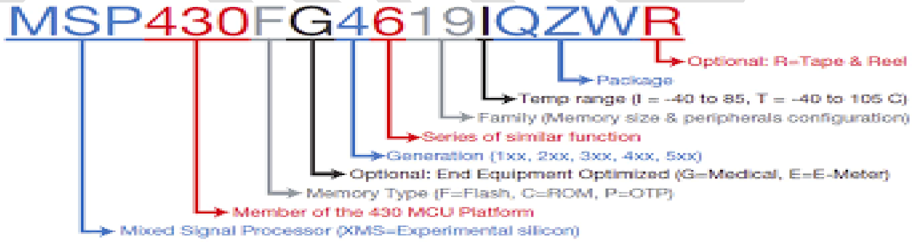
Advantages:

- Stack processing capability
- RISC architecture
- Real time capability with ultra low power consumption
- Digitally controlled oscillator controlled

Disadvantages:

Number of pins are more
Block diagram is complex
High power mode will be operated

MSP 430 FAMILY VARIANTS/ SERIES


MSP430FG4619IQZWR
Optional: R=Tape & Reel
Package
Temp range (I = -40 to 85, T = -40 to 105 C)
Family (Memory size & peripherals configuration)
Series of similar function
Generation (1xx, 2xx, 3xx, 4xx, 5xx)
Optional: End Equipment Optimized (G=Medical, E=E-Meter)
Memory Type (F=Flash, C=ROM, P=OTP)
Member of the 430 MCU Platform
Mixed Signal Processor (XMS=Experimental silicon)

Features of Various MSP430 Families

MSP430x1xx series

Power specification overview, as low as:

- µA RAM retention
- 0.7 µA real-time-clock mode
- 200 µA / MIPS active
- Features fast wakeup from standby mode in less than 6 µs.

Device parameters

- Flash options: 1–60 KB
- ROM options: 1–16 KB
- RAM options: 128 B–10 KB
- GPIO options: 14, 22, 48 pins

- ADC options: Slope, 10 & 12bitSAR
- Other integrated peripherals: 12bit DAC, up to 2 16bit timers, watchdog timer, brownout reset, SVS, USART module (UART, SPI), DMA, 16×16 multiplier, Comparator A, temperature sensor
- Applications: stepper motor controller, wireless mouse, electric meter.

MSP430F2xx series

Power specification overview, as low as:

- µA RAM retention
- 0.3 µA standby mode (VLO)
- 0.7 µA real-time-clock mode
- 220 µA / MIPS active
- Feature ultrafast wakeup from standby mode in less than 1 µs

Device parameters

- Flash options: 1–120 KB
- RAM options: 128 B – 8 KB
- GPIO options: 10, 11, 16, 24, 32, and 48 pins
- ADC options: Slope, 10 & 12bit SAR, 16 & 24bit Sigma Delta
- Other integrated peripherals: operational amplifiers, 12bit DAC, up to 2 16-bit timers, watchdog timer, brownout reset, SVS, USI module (I²C, SPI), USCI module, DMA, 16×16 multiplier, Comparator_A+, temperature sensor
- Applications:e-metering

MSP430G2xx series

Ultra Low Power, as low as (@2.2 V):

- 0.1µA RAM retention
- 0.4 µA Standby mode (VLO)
- 0.7 µA real-time-clock mode
- 220 µA / MIPS active
- UltraFast wake-up from Standby Mode in <1 µs

Device parameters

- Flash options: 0.5–56 KB
- RAM options: 128 B–4 KB
- GPIO options: 10, 16, 24, 32 pins
- ADC options: Slope, 10bit SAR
- Other integrated peripherals: Capacitive Touch I/O, up to 3 16-bit timers, watchdog timer, brownout reset, USI module (I²C, SPI), USCI module, Comparator_A+, Temp sensor

MSP430x3xx series

Power specification overview, as low as:

- 0.1µA RAM retention

- 0.9 µA real-time-clock mode
- 160 µA / MIPS active
- Features fast wakeup from standby mode in less than 6 µs.

Device parameters:

- ROM options: 2–32 KB
- RAM options: 512 B–1 KB
- GPIO options: 14, 40 pins
- ADC options: Slope, 14bit SAR
- Other integrated peripherals: LCD controller, multiplier

MSP430x4xx series

Power specification overview, as low as:

- 0.1µA RAM retention
- 0.7 µA real-time-clock mode
- 200 µA / MIPS active
- Features fast wakeup from standby mode in less than 6 µs.

Device parameters:

- Flash/ROM options: 4 – 120 KB
- RAM options: 256 B – 8 KB
- GPIO options: 14, 32, 48, 56, 68, 72, 80 pins
- ADC options: Slope, 10 & 12bit SAR, 16bit Sigma Delta
- Other integrated peripherals: SCAN_IF, ESP430, 12bit DAC, Op Amps, RTC, up to 2 16bit timers, watchdog timer, basic timer, brownout reset, SVS, USART module (UART, SPI), USCI module, LCD Controller, DMA, 16×16 & 32x32 multiplier, Comparator_A, temperature sensor, 8 MIPS CPU Speed
- Applications: energy metering, low power applications

MSP430x5xx series

Power specification overview, as low as:

- 0.1µA RAM retention
- 2.5 µA real-time-clock mode
- 165 µA / MIPS active
- Features fast wakeup from standby mode in less than 5 µs.

Device parameters:

- Flash options: up to 512 KB
- RAM options: up to 66 KB
- ADC options: 10 & 12bit SAR
- GPIO options: 29, 31, 47, 48, 63, 67, 74, 87 pins
- Other integrated peripherals: High resolution PWM, 5 V I/O's, USB, up to 4 16-bit timers, watchdog timer, Real Time Clock, brownout reset, SVS, USCI module, DMA, 32x32 multiplier, Comp B, temperature sensor.

- Applications: USB, RF connectivity and consumer electronics.

MSP430x6xx series

Power specification overview, as low as:

- 0.1µA RAM retention
- 2.5 µA real-time-clock mode
- 165 µA / MIPS active
- Features fast wakeup from standby mode in less than 5 µs

Device parameters:

- Flash options: up to 512 KB
- RAM options: up to 66 KB
- ADC options: 12bit SAR
- GPIO options: 74 pins
- Other integrated peripherals: USB, LCD, DAC, Comparator_B, DMA, 32x32 multiplier, power management module (BOR, SVS, SVM, LDO),watchdog timer, RTC, Temp sensor.

MSP430x5xx Series Block Diagram

Unified clock system:

The unified clock system consists of three clock signals namely Master clock, Sub system master clock and Auxiliary clock.

- The master clock is used by the CPU and few peripherals.
- SMCLK is distributed to peripherals.
- ACLK is also distributed to peripherals.

MCLK & SMCLK run at same frequencies which are at mega hertz range. ACLK is derived from a watch crystal and therefore runs at lower frequency.

Power management module:

The power management module is used for supervision of power supply voltages.

DMA (Direct Memory Access):

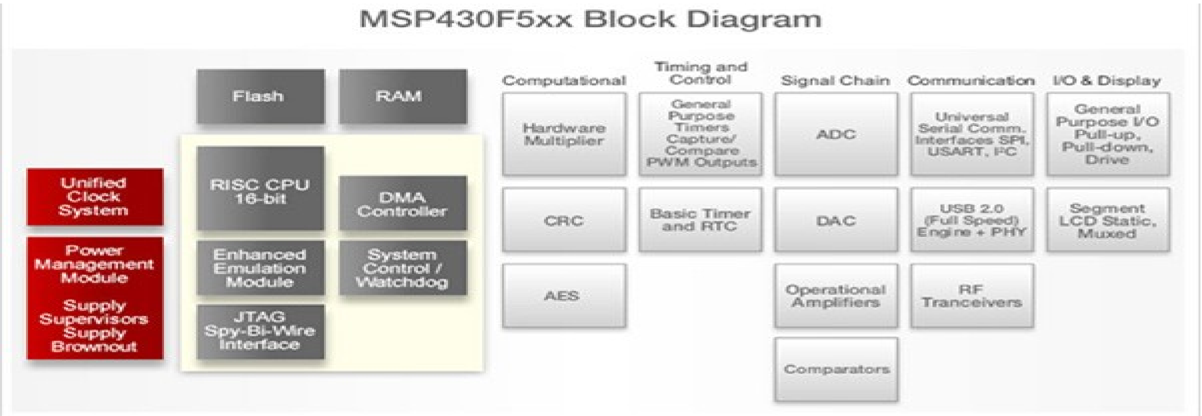
Direct memory access is used when huge amount of data is to be transferred without CPU from memory to I/O.

Watchdog Timer Module:

Watchdog timer is used when fault occurs in the system then watchdog timer resets automatically the system. The main purpose of watchdog timer is to protect the system against failure of software.

Hardware multiplier:

The multiplication operation is supported by a dedicated peripheral module. The module performs operations with 32-, 24-, 16-, and 8-bit operands. The module supports signed and unsigned multiplication as well as signed and unsigned multiply-and-accumulate operations.


MSP430F5xx Block Diagram
Flash
RAM
Unified Clock System
Power Management Module
Supply Supervisors Supply Brownout
RISC CPU 16-bit
DMA Controller
Enhanced Emulation Module
System Control / Watchdog
JTAG Spy-Bi-Wire Interface
Computational
Hardware Multiplier
CRC
AES
Timing and Control
General Purpose Timers Capture/ Compare PWM Outputs
Basic Timer and RTC
Signal Chain
ADC
DAC
Operational Amplifiers
Comparators
Communication
Universal Serial Comm. Interfaces SPI, USART, I²C
USB 2.0 (Full Speed) Engine + PHY
RF Tranceivers
I/O & Display
General Purpose I/O Pull-up, Pull-down, Drive
Segment LCD Static, Muxed

CRC (Cyclic Redundant Check):

CRC is used to perform error correction and detection in the system. CRC is used to check the errors.

JTAG:

JTAG is an interface between MSP430 and PC. It is used for programming, monitoring, debugging and testing the programs.

SPY BI Wire:

SPY BI wire is used to download the software and run the program to interface MSP430 with PC. It is similar to JTAG.

AES (Advanced Encryption Standard):

Advanced encryption standard is used to protect system by support the system by providing security.

Real Time Clock:

The RTC module can be used as a general-purpose 32-bit counter (counter mode) or as an integrated real-time clock (RTC) (calendar mode). In counter mode, the RTC also includes two independent 8-bit timers that can be cascaded to form a 16-bit timer/counter. Both timers can be read and written by software.

GPIO (General purpose I/O):

General purpose Input / Output is used to interact with hardware.

SCI (Serial Communication Interface):

SCI supports serial communication using 3 protocols.

- I^{2}C
- SPI
- UART

Operational Amplifier:

OP Amps are used to perform the arithmetic & logical operations and also amplifies the signal.

Comparator:

Comparator is used to compare the two signals and produce the sum and difference signals.

RF Transceivers:

RF Transceivers are used for supporting the wireless communication.

Segmented Display:

Segmented display is used to display the signal.

Universal Serial Bus (USB)

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)
  - 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 V_{BUS}
  - 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
- Internal 48-MHz USB clock
  - Integrated programmable PLL
  - Highly-flexible input clock frequencies for use with lowest-cost crystals
- 2KB of dedicated USB buffer space for endpoints
- 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

I/O Ports and Registers:

- I/O ports are used to communicate with the MSP430 by the user or external environment.
- MSP430 is having no. of GPIO ports (number will vary with family) and each port size is 8-bit.
- Each I/O pin is individually configurable as I/p or o/p pin.
- Each I/O pin is having separate pull-up and pull-down resistors, data input and output registers.
- All the I/O pins can be configured as digital I/O pin or module pin.
- P1 & P2 ports having interrupt capability.
- P1 & P2 interrupts are prioritized with P1.0 is highest and P1.7 is lowest. Similarly for P2
- Both P1 & P2 having separate interrupt vector registers P1IV and P2IV respectively.

CPU:

CPU has a 16 bit ALU, 4 Special function registers, 12 general function registers

- 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.
- 12 general purpose registers saves CPU cycles by allowing the storage of frequently used values and variables instead of using RAM.
- Registers can hold a word of 16bits

Program counter:

- The 16-bit Program Counter (PC/R0) points to the next instruction to be read from memory and executed by the CPU.

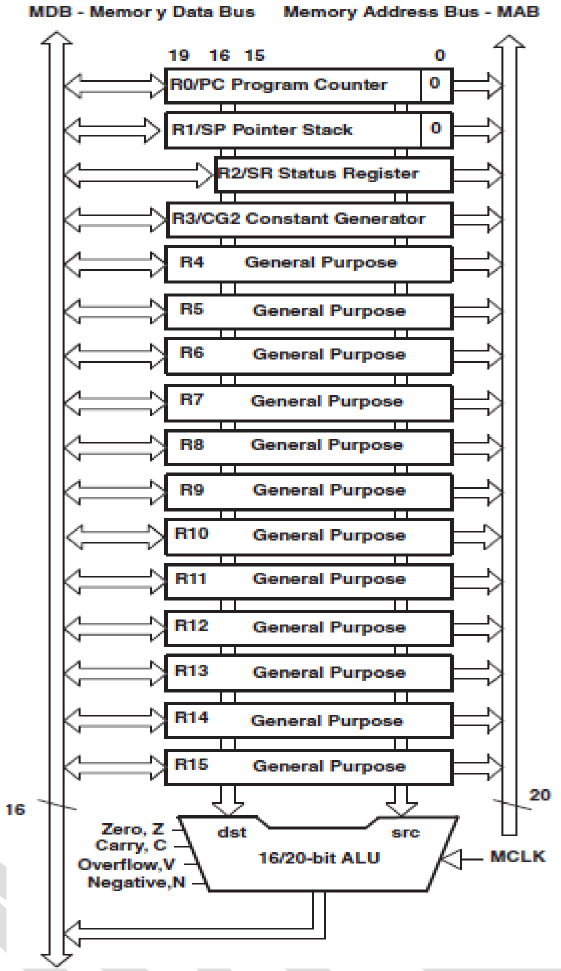
- The Program counter is implemented by the number of bytes used by the instruction (2, 4, or 6 bytes, always even).
- It is important to note that the PC is aligned at even addresses, because the instructions are 16 bits, even though the individual memory addresses contain 8-bit values.

Stack Pointer:

- When a subroutine is called the CPU jumps to the subroutine, executes the code then returns to instruction after the call instruction.
- It must keep track of the contents of the PC before jumping to the subroutine, so that it can return afterward.
- This is the primary purpose of the stack.
- Some processors use separate hardware for the stack.
- But MSP430 uses the top of Main RAM.
- The Stack Pointer (SP/R1) is located in R1.
- The SP holds the address of most recently added word and is automatically grows downward in memory.
- Stack can be used by user to store data for later use(instructions: store by PUSH, retrieve by POP.
- 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)
- Stack can be used by subroutine calls to store the program counter value for return at subroutine's end (RET).

Status registers:

The Status Register (SR/R2) is a 16 bit register, and it stores the status 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.



<u>R2/R3: Constant Generator Registers (CG1/CG2)</u>

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.

<u>General Purpose registers</u>

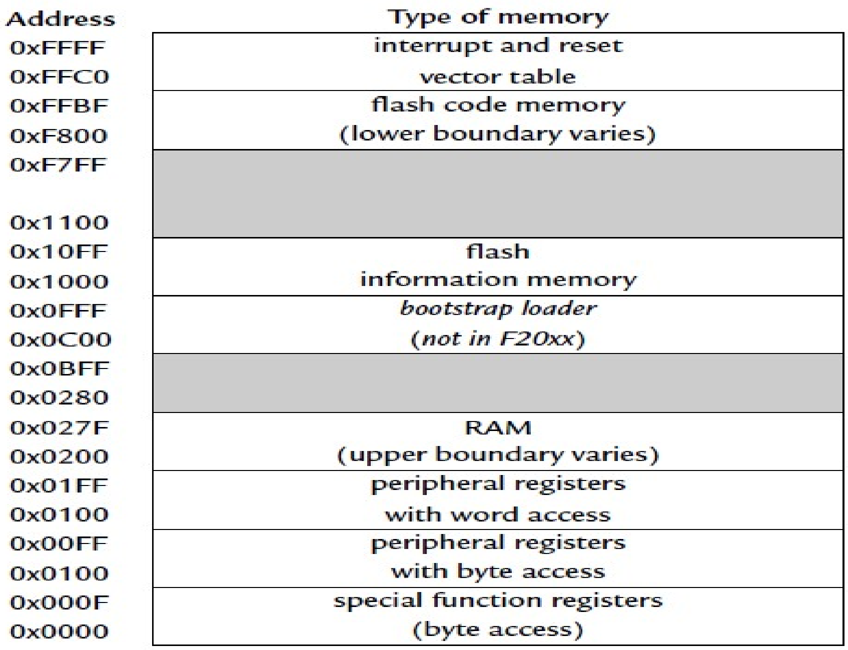
- The remaining 12 registers R4-R15 are general working registers
- They may be used for either data or address because both are 16-bit values, which simplify the operation.

<u>Address space:</u>

The MSP430x5xx family has a 1-MB unified memory map with expanded peripheral space over previous families.

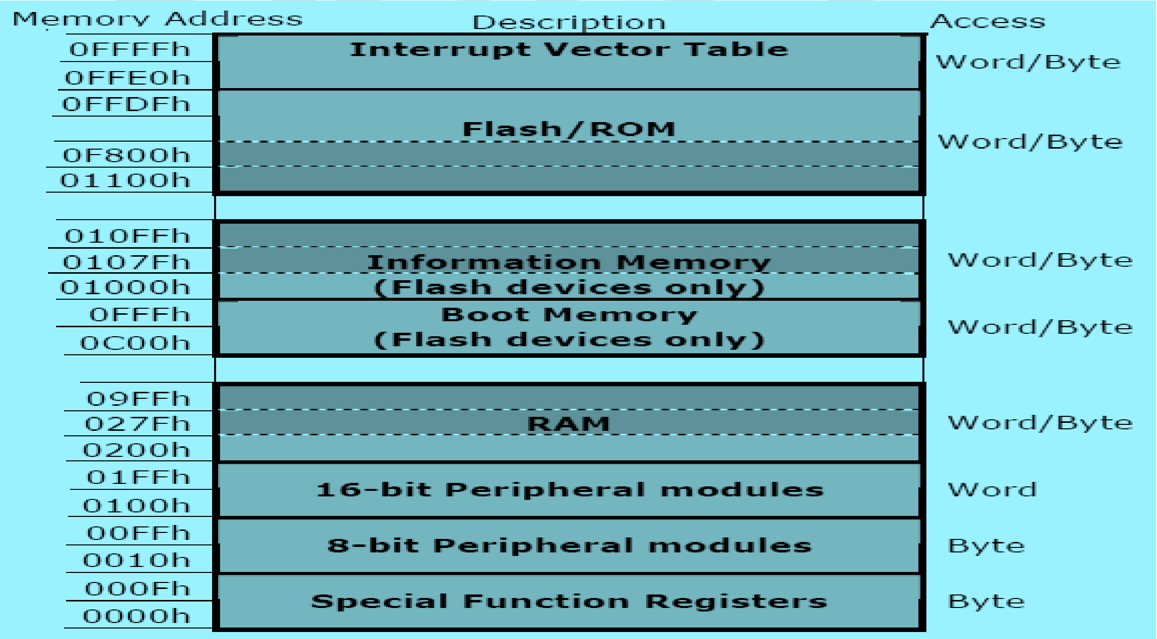
Segment	Address	Size
Peripherals	00000h – 00FFFh	4KB
Boot Memory	01000h – 017FFh	2KB
Information memory	01800h – 019FFh	512 Bytes
RAM	01C00h – 05BFFh	16KB

Code memory	05C00h – 45BFFh	256KB
Interrupt vectors	0FF80h – 0FFFFh	128 Bytes



Address	Type of memory
0xFFFF 0xFFC0	interrupt and reset vector table
0xFFBF 0xF800	flash code memory (lower boundary varies)
0xF7FF 0x1100	
0x10FF 0x1000	flash information memory
0x0FFF 0x0C00	*bootstrap loader (not in F20xx)*
0x0BFF 0x0280	
0x027F 0x0200	RAM (upper boundary varies)
0x01FF 0x0100	peripheral registers with word access
0x00FF 0x0100	peripheral registers with byte access
0x000F 0x0000	special function registers (byte access)

Memory map of the MSP430F2013, based on the data shee



Memory Address	Description	Access
0FFFFh 0FFE0h	Interrupt Vector Table	Word/Byte
0FFDFh 0F800h 01100h	Flash/ROM	Word/Byte
010FFh 0107Fh 01000h	Information Memory (Flash devices only)	Word/Byte
0FFFh 0C00h	Boot Memory (Flash devices only)	Word/Byte
09FFh 027Fh 0200h	RAM	Word/Byte
01FFh 0100h	16-bit Peripheral modules	Word
00FFh 0010h	8-bit Peripheral modules	Byte
000Fh 0000h	Special Function Registers	Byte

- All memory, including RAM, Flash/ROM, information memory, special function registers (SFRs), and peripheral registers are mapped into a single, contiguous address space.
- 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 be accessed from **even addresses**
- The interrupt vector table is mapped at the very end of memory space (upper 16 words of Flash/ROM), in locations 0FFE0h through to 0FFFEh. The priority of the interrupt vector increases with the word address. It is used to store the addresses of interrupts.
- The start address of Flash/ROM depends on the amount of Flash/ROM present on the device. The start address varies between01100h (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.
- 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
- 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.
- 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 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.
- The Special Function Registers (SFRs) are located at memory addresses from 0000h to 000Fh. SFRs must be accessed using byte instructions only.

ADDRESSING MODES OF MSP430:

Addressing mode means the way of specifying the operands in an instruction. The MSP430 has seven addressing modes to interact with the CPU registers. All 16 of these are treated on an almost equal basis, including the four special-purpose registers R0–R3 or PC, SP, SR/CG1, and CG2.

The MSP430 supports seven addressing modes for the source operand and four addressing modes for the destination operand. They are

- Register mode
- Indexed mode
- Symbolic mode
- Absolute mode
- Indirect register mode

- Indirect auto increment mode
- Immediate mode

Register Mode

This uses one or two of the registers in the CPU. It is the most straightforward addressing mode and is available for both source and destination.

Ex: MOV.W R4, R5 - Move (copy) the contents of source (register R4) to destination (R5).

The registers are specified in the instruction word; no further data are needed. It is also the fastest mode and this instruction takes only 1 cycle.

Any of the 16 registers can be used for either source or destination but there are some special cases:

- The PC is incremented by 2 while the instruction is being fetched, before it is used as a source.
- The constant generator CG2 reads 0 as a source.
- Both PC and SP must be even because they address only words, so the LSB is discarded if they are used as the destination.
- SR can be used as a source and destination in almost the usual way although there are some details about the behavior of individual bits.

For byte instructions,

- Operands are taken from the lower byte; the upper byte is not affected.
- The result is written to the lower byte of the register and the upper byte is cleared.

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 is addressed by adding a constant base address to the contents of a CPU register; the value in the register is not changed.

Ex : MOV.B 3(R5), R4

Move (copy) the contents at source address (3+R5) to destination (register R4)

Indexed addressing can be used for the source, destination, or both. The base addresses, just the single value 3 here because only one address is indexed, are stored in the words following the instruction. They cannot be produced by the constant generator.

Symbolic Mode

Symbolic mode allows the assignment of labels to fixed memory locations, so that those locations can be addressed directly with the assigned label name. For example, suppose that a program uses the variable LoopCtr, which occupies a word. The following instruction stores the value of LoopCtr in R6 using symbolic mode.

Ex: **mov.w** LoopCtr ,R6 *; load word LoopCtr into R6*

Absolute Mode

It is similar to Symbolic mode, with the difference that the label is preceded by "&". The word following the instruction contains the absolute address.

Ex: **mov.b** &P1IN ,R6 *; load byte P1IN into R6*

Where P_1IN is the absolute address of the register. This addressing mode is used for special function and peripheral registers, whose addresses are fixed in the memory map. This addressing mode can be used for both source and destination operands.

The instruction in above example is written in indexed mode as below:

mov.b P1IN(SR),R6 *; load byte P1IN into R6*

Here PIN is already the complete address required so it should be added to a register that contains 0. The MSP430 mimics this by using the status register SR which provides a constant 0 i.e., it behaves as CG1 when it is used as the base for indexed addressing.

Register indirect mode

This is available only for the source operand and is shown by the symbol @ in front of a register, @Rn. It means that the contents of Rn are used as the *address* of the operand. In other words, Rn holds a pointer rather than a value.

Ex: **mov.w** @R5 ,R6 *; load word from address (R5)=4 into R6*

The address of the source is 4, which is in R5. Thus a word is loaded from address 4 into R6. The value in R5 is unchanged. This has exactly the same effect as indexed addressing with a base address of 0 but saves a word of program memory, which also makes it faster.

Indirect addressing cannot be used for the destination so indexed addressing must be used instead by emulating the instruction as below.

mov.w R6 ,0(R5) *; store word from R6 into address 0+(R5)=4*

The penalty is that a word of 0 must be stored in the program memory and fetched from it. The constant generator cannot be used.

Indirect Autoincrement Mode

This is also applicable only for the source operand and the format is specified with a symbol '@' in front of the register and + sign after it, such as @Rn+. Here, Rn register value is used as data pointer and increments the register content after the operation by 1 (for byte operations) or 2 (for word operation).

Ex: **mov.w** @R5+,R6

A word is loaded from address 4 into R6 and the value in R5 is incremented to 6 because a word (2 bytes) was fetched.

This mode cannot be used for the destination. Instead the main instruction must use indexed mode with an offset of 0, followed by an explicit increment of the register by 1 or 2. The reverse of this move therefore needs two instructions:

mov.w R6 ,0(R5) *; store word from R6 into address 0+(R5)=4*

incd.w R5 *; R5 += 2*

Immediate Mode

In this addressing mode, the immediate data is specified as the operand in the instruction i.e., the data is readily available as a part instruction that will be fetched from the memory for the operation.

Immediate mode is used to assign constant values to registers or memory locations. The immediate data cannot be a destination operand.

Ex: **mov.w** #0900h,R5

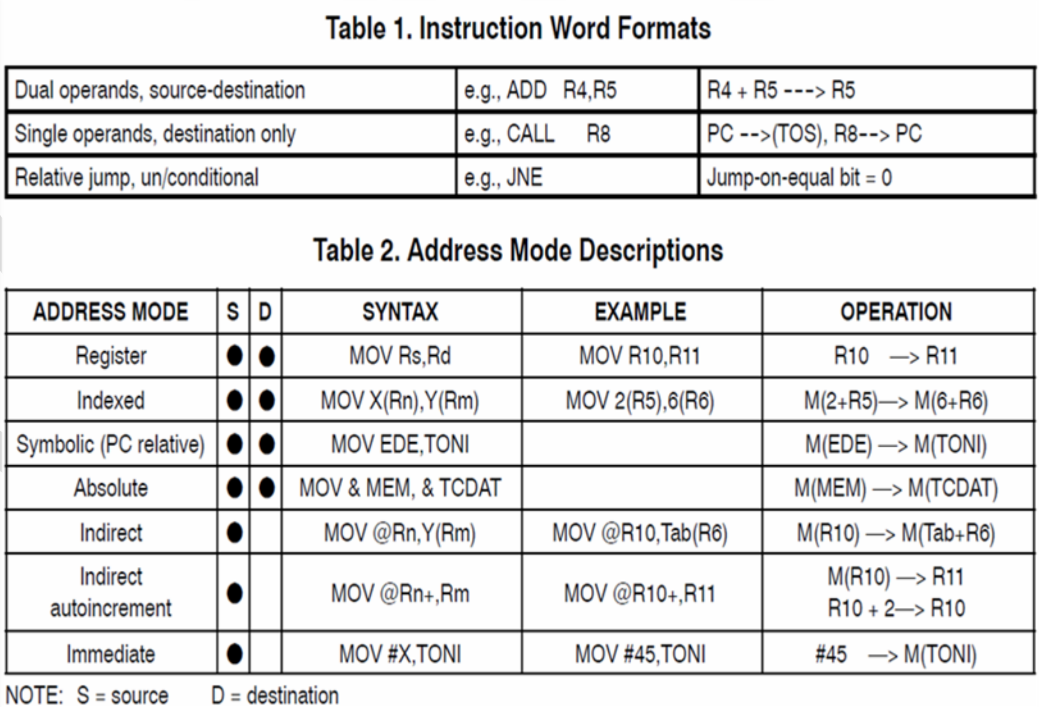


Table 1. Instruction Word Formats

Dual operands, source-destination	e.g., ADD R4,R5	R4 + R5 ---> R5
Single operands, destination only	e.g., CALL R8	PC -->(TOS), R8--> PC
Relative jump, un/conditional	e.g., JNE	Jump-on-equal bit = 0

Table 2. Address Mode Descriptions

ADDRESS MODE	S	D	SYNTAX	EXAMPLE	OPERATION
Register	●	●	MOV Rs,Rd	MOV R10,R11	R10 —> R11
Indexed	●	●	MOV X(Rn),Y(Rm)	MOV 2(R5),6(R6)	M(2+R5)—> M(6+R6)
Symbolic (PC relative)	●	●	MOV EDE,TONI		M(EDE) —> M(TONI)
Absolute	●	●	MOV & MEM, & TCDAT		M(MEM) —> M(TCDAT)
Indirect	●		MOV @Rn,Y(Rm)	MOV @R10,Tab(R6)	M(R10) —> M(Tab+R6)
Indirect autoincrement	●		MOV @Rn+,Rm	MOV @R10+,R11	M(R10) —> R11 R10 + 2—> R10
Immediate	●		MOV #X,TONI	MOV #45,TONI	#45 —> M(TONI)

NOTE: S = source D = destination

On–chip Peripherals:

FRAM Technology:

Ferro electric Random Access Memory is a memory technology that contains the best features of flash and SRAM. It is a non volatile like flash but it is fast & low power writes.

Hardware Multiplier:

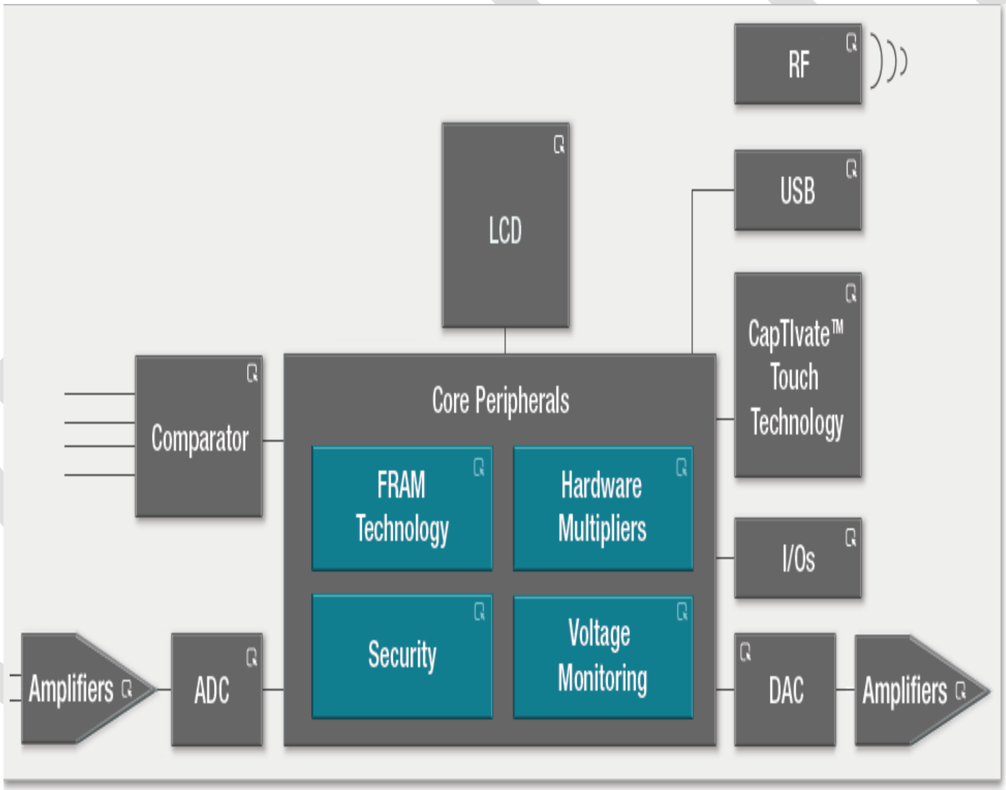
The MSP430 low power microcontroller offers 16 bit and 32 bit multiplication modules on select devices. These peripherals can be used while the microcontroller is in low power modes. It is independent of CPU. It also supports signed and unsigned multiplications.

Security:

The MSP 430 advanced microcontrollers provides embedded security systems that allow customers to prevent, detect and respond to unintended or malicious behavior, including MCU reverse engineering. These secure microcontroller features include advanced encryption security (AES) hardware accelerations, IP encapsulation, memory protection, anti tampering, FRAM debug lockout, BSL password protection.

Voltage monitoring:

The first primary functions of voltage monitoring are to generate a supply voltage for core logic, the second function is to provide several mechanisms for supervising & monitoring of both voltage applied to the device (DVCC) and voltage generated for core (VCORE). The PMM (power monitoring module) uses an integrated lo drop voltage regulator (LDO) to produce a secondary core voltage from primary one applied to the device.



CapTI Vate ™ touch technology:

MSP MCUs with FRAM and CapTIVate technology are the most noise immune capacitive touch MCUs.

16 & 24 bit Sigma Delta Converters:

The MSP 430 microcontroller consists up to 7 independent sigma delta ADC. This ADC is based on second order over sampling sigma delta modulators & digital decimation filters. The decimation filters are combo type filters with selectable oversampling ratios up to 256.

Analog comparators:

This module supports precision slope ADCs, supply voltage supervision & monitoring of external analog signals.

Operational amplifiers:

OP amps supports front end analog signal conditioning prior to ADC. OP amp output slew rate can be configured.

LCD drivers:

MSP 430 supports up to 320 segments individual blinking segments control. This controller includes a proven core that has been optimized for low power modes.

Inputs/Outputs:

MSP 430 is designed to support a variety of need dependent upon specific applications or pin configurable settings. I/O pins may e multiplexed with multiple peripherals providing layout and peripheral flexibility.

USB:

Development is made easy with USB developer's package & tools like MSP 430 also uses USB vender.

Instruction set:

Generally instruction set is having

- Op Code
- Source Operand
- Destination Operand

Byte, word and address instructions are accessed using the **.B, .W or .A** extensions. Byte instructions are used to access byte data or byte peripherals. Word instructions are used to access word data or word peripherals. If no extension is used, the instruction is a word instruction.

There are two types of instructions. They are

- Core instructions
- Emulated instructions

Core instructions that have unique op-codes are decoded by CPU. The Emulated instructions are instructions that make code easier to write and read data but don't have op-codes themselves instead they are replaced automatically by the assembler with an equivalent core instruction.

There are three core-instruction formats:

- Double-operand
- Single-operand
- Program flow control

Movement instructions:

There is only the one 'mov' instruction to move data. It can address all of memory as either source or destination, including both registers in the CPU and the whole memory map.

mov . w src , dst

Here w denotes that the operations can use either bytes or words

Stack Operations:

Stack operations are used to transfer data to stack. The data can be inserted into stack by performing PUSH operation and the data can be deleted from stack by performing POP operation.

push .w src ; push data onto stack

pop .w dst ; pop data off stack.

Arithmetic instructions with two operands:

operation	operands	action	

add .w	src , dst	**add**	*dst + = src*
addc .w	src , dst	add with carry	*dst += (src + C)*
adc .w	dst	add carry bit	*dst + = C*
sub .w	src , dst	subtract	*dst - = scr*
subc .w	src , dst	subtract with borrow	*dst - = (scr +~C)*
sbc .w	dst	Subtract borrow bit	*dst - = ~C - Emulated*
cmp .w	src , dst	compare , set flags only	*(dst – scr)*

<u>Arithmetic instructions with one operand:</u>

All these instructions are emulated, which means the operands is always in destination.

operation	**operands**	**action**	
clr .w	**dst**	clear	dst = 0
dec .w	**dst**	decrement	dst - -
decd .w	**dst**	double decrement	dst- = 2
inc .w	**dst**	increment	dst ++
incd .w	**dst**	double increment	dst+ = 2
tst .w	**dst**	test (compare with 0)	(dst – 0)

<u>Decimal arithmetic instructions:</u>

Decimal arithmetic instructions are used when BCD values are operands.

operation	**operands**	**action**	
dadd .w	src , dst	Decimal add with carry	dst+ = src +c
dadc .w	dst	Decimal add carry bit	dst+ =c

<u>Logical instructions with two operands:</u>

<u>operation</u>	**<u>operands</u>**	**<u>action</u>**	
and .w	src , dst	Bitwise AND	dst&=scr
xor .w	src ,dst	Bitwise XOR	dst^=scr
bit .w	src , dst	Bitwise test set flag only	(dst&scr)
bis .w	Src, dst	Bit set	dst|=scr
bic .w	src , dst	Bit clear	dst&=~scr

<u>Logical instructions with one operand:</u>

There is only one instruction of this type. invert ‘inv’ instruction , also known as ones complement ,which changes all 0 bits to 1 and 1s to 0

inv.w dst ; invert bits

Byte Manipulation:

These instructions do not need a suffix because the size of. the operands is fixed. **Swpb src**

; swap upper and lower bytes (word only)

Sxt src *;* extend sign of lower byte (word only)

Operations on bits in status register

operation	action	
clrc	*Clear carry bit*	C=0
clrn	*Clear negative bit*	N=0
clrz	*Clear zero bit*	Z=0
setc	*Set carry bit*	C=1
setn	*Set negative bit*	N=1
setz	*Set zero bit*	Z=1
dint	*Disable general interrupts*	GIE=0
eint	*Enable general interrupts*	GIE=1

Shift and rotate instructions:

There are three types of shifts

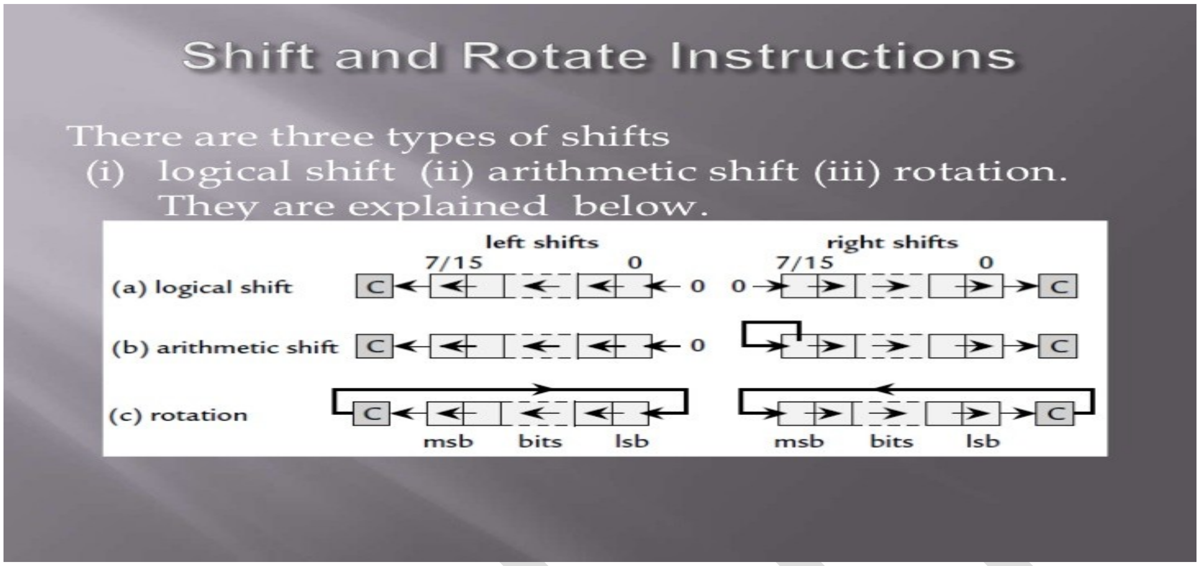
- Logical shift: Inserts zeroes for both right and left shifts.
- arithmetic shift: Inserts zeroes for left shifts but the most significant bit, which carries the sign, is replicated for right shifts

  Ex : rla dst *; arithmetic shift left*

  rra src *; arithmetic shift right.*
- Rotation: Does not introduce or lose any bits; bits are moved out of one end of the register are passed around to the other.

  Ex: rlc dst ; rotate left through carry.

  rrc src *; rotate right through carry*



Shift and Rotate Instructions

There are three types of shifts
(i) logical shift (ii) arithmetic shift (iii) rotation.
They are explained below.

operation	operand	action
rla	dst	Arithmetic shift left (Emulated)
rra	scr	Arithmetic shift right
rlc	dst	Rotate left through carry (emulated)
rrc	scr	Rotate right through carry

Instruction based of program flow control

Sub Routines, interrupts and Branches

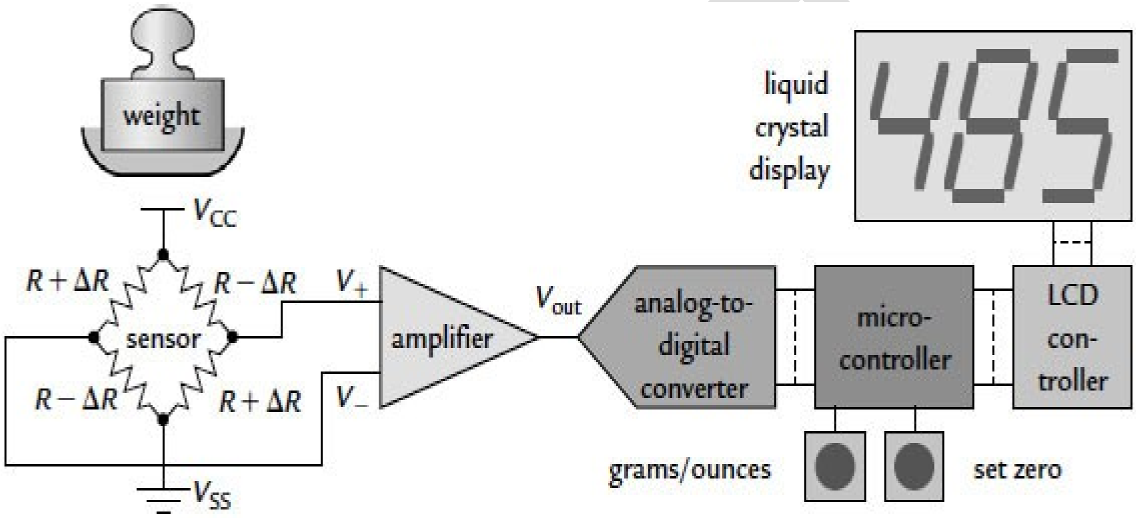
operation	operand	action	
br	src	Branch (go to)	emulated
call	src	Call subroutine	
ret		Return from subroutine	emulated
reti		Return from interrupt	
nop		No operation	emulated

Jumps, Unconditional and Conditional

operation	operand	action	
jc	label	Jump if carry set	C=0 (**jhs**)
jnc	label	Jump if carry not set	C=1 (**jlo**)
jn	label	Jump if negative	N=1
jz	label	Jump if zero	Z=0 (**jeq**)

jnz	label	Jump if not zero	Z=1 (**jne**)
jeq	label	Jump if equal	dst = scr
jne	label	Jump if not equal	dst != scr
jhs	label	Jump if higher or same	dst >= scr
jlo	label	Jump if lower	dst < scr

Sample embedded system using MSP 430 microcontroller:



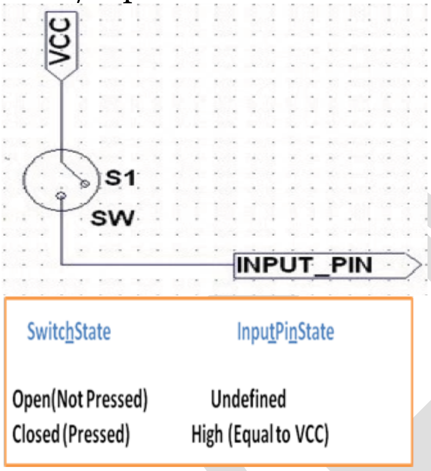
- The sensor has four resistive elements arranged as a Wheatstone bridge. Ideally, this is balanced when there is no load, giving V+ = V-.
- Two of the resistances increases and two decreases when a weight is placed on the scale pan, driving the bridge out of balance.
- A differential amplifier magnifies the difference in voltage between its input terminals, giving V_{out} = A(V+ - V-), where A is the gain.
- The analog output of the amplifier is converted to a binary value in an analog-to-digital converter.
- The microcontroller multiplies the input by an appropriate factor so that the display gives the weight in grams or ounces and subtracts an offset so that the display reads zero when no weight is present. It also reads the buttons and supervises the complete system.
- There is a serial interface between the microcontroller and the liquid crystal display, which has a built-in controller.
- The entire system can be built from a sensor, an MSP 430, a simple LCD without a controller and a couple of decoupling capacitors.
- The MSP430x4xx family drives segmented LCD’s directly which eradicates the requirement for a controller.
- Numerous devices include ADC’s with high resolution, differential inputs which would work directly from sensor without the need for an amplifier.
- The microcontroller can also manage the power drawn by the circuit. So that the processor would be switched off when it was not necessary and the complete shut down after a period of inactivity.

UNIT -IV

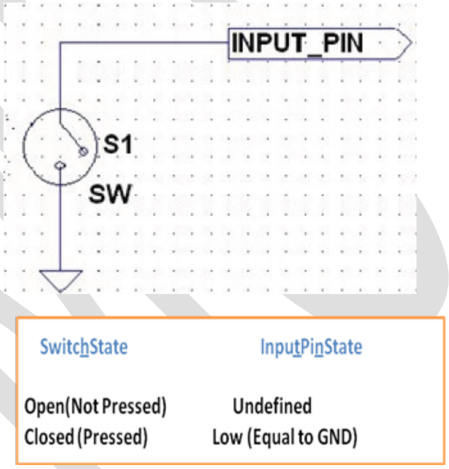
I/O Ports Pull up/down Registers:

- I/O ports are used to communicate with the MSP430 by the user or external environment.
- MSP430 is having no. of GPIO ports (number will vary with family) and each port size is 8-bit.
- Each I/O pin is individually configurable as I/p or o/p pin.
- Each I/O pin is having separate pull-up and pull-down resistors, data input and output registers.
- All the I/O pins can be configured as digital I/O pin or module pin.
- P1 & P2 ports having interrupt capability.
- P1 & P2 interrupts are prioritized with P1.0 are highest and P1.7 is lowest. Similarly for P2
- Both P1 & P2 having separate interrupt vector registers P1IV and P2IV respectively.

The I/O pin structure is shown below.



SwitchState	InputPinState
Open(Not Pressed)	Undefined
Closed(Pressed)	High (Equal to VCC)



SwitchState	InputPinState
Open(Not Pressed)	Undefined
Closed(Pressed)	Low (Equal to GND)

PxDIR (Direction Register): Each bit in each PxDIR register selects the direction of the corresponding I/O pin, regardless of the selected function for the pin. PxDIR bits for I/O pins that are selected for other functions must be set as required by the other function. By default all port pins are input pins.

Bit = 0: Port pin is switched to input direction
Bit = 1: Port pin is switched to output direction

PxIN (Input Register): Each bit in each PxIN register reflects the value of the input signal at the corresponding I/O pin when the pin is configured as I/O function. These registers are read only.

Bit = 0: The input is low
Bit = 1: The input is high

PxOUT (Output Register): Each bit in each PxOUT register is the value to be output on the corresponding I/O pin when the pin is configured as I/O function, output direction.

Bit = 0: The output is low
Bit = 1: The output is high

If the pin is configured as I/O function, input direction and the pullup/pulldown resistor is enabled, the corresponding bit in the PxOUT register selects pullup or pulldown.

Bit = 0: The pin is pulled down
Bit = 1: The pin is pulled up

PxREN (Resistor Enable Register): Each bit in each PxREN register enables or disables the pullup/pulldown resistor of the corresponding I/O pin. The corresponding bit in the PxOUT register selects if the pin is pulled up or pulled down.

Bit = 0: Pullup/pulldown resistor disabled
Bit = 1: Pullup/pulldown resistor enabled

PxSEL (Selection Register): Port pins are often multiplexed with other peripheral module functions. Each PxSELx bit is used to select the pin function - I/O port or peripheral module function.

Bit = 0: I/O Function is selected for the pin
Bit = 1: Peripheral module function is selected for the pin

Setting PxSELx = 1 does not automatically set the pin direction. Other peripheral module functions may require the PxDIRx bits to be configured according to the direction needed for the module function.

PxIE (Interrupt Enable): This read-write register enables interrupts on individual pins. Interrupts on the pins are enabled when their corresponding bits in this register are set. This register is cleared on reset. By default, interrupts are disabled on the port pins.

PxIES (Interrupt Edge Select): This read-write register selects the transition on which an interrupt occurs. If set, interrupt occurs on a high-to-low transition on the corresponding pin. If cleared, an interrupt occurs on a low-to-high transition on the corresponding pin.

PxIFG (Interrupt Flag): Whenever interrupt occurs on the pin then corresponding bit position is set to ‘1’. This register is cleared by the software before executing RETI instruction in the ISR.

Bit = 0: No interrupt is pending
Bit = 1: An interrupt is pending

Interrupts:

An interrupt is breaking normal program execution. When ever an interrupt occurs processor runs ISR program. Interrupt can be enabled or disabled. Interrupts can occurred can be unpredictable when and where it occurs. Interrupts can occur globally, individually or through NMI.

Interrupts are commonly used for a range of applications:

- Urgent tasks that must be executed promptly at higher priority than the main code. However, it is even faster to execute a task directly by hardware if this is possible.
- Infrequent tasks, such as handling slow i/o devices. This saves the overhead of regular polling.
- Waking the CPU from sleep. This is particularly important in the MSP430, which typically spends much of its time in a low-power mode and can be awakened only by an interrupt.

The code to handle an interrupt is called an *interrupt service routine* (ISR). It looks superficially like a function but there are a few crucial modifications. The feature that interrupts arise at unpredictable times means that an ISR must carry out its action and clean up thoroughly so that the main code can be resumed without error—it should not be able to tell that an interrupt occurred.

Each interrupt has a flag, which is raised (set) when the condition for the interrupt occurs.

The MSP430 uses *vectored* interrupts, which means that the address of each ISR—its vector—is stored in a *vector table* at a defined address in memory. In most cases each vector is associated with a unique interrupt but some sources share a vector.

Each interrupt vector has a distinct priority, which is used to select which vector is taken if more than one interrupt is active when the vector is fetched. The priorities are fixed in hardware and cannot be changed by the user. They are given simply by the address of the vector: A higher address means a higher priority. The reset vector has address 0xFFFE, which gives it the top priority, followed by 0xFFFC for the single nonmaskable interrupt vector. The vectors for the maskable interrupts depend on the peripherals in a particular device and are listed in a table of *Interrupt Vector Addresses*.

Types of interrupts

The MSP430 offers various interrupt sources, both internal and external. There are three types of interrupts:

- Reset
- Non-maskable interrupts (NMI)
- Maskable interrupts by GIE.

Each one of these interrupts has a priority, determining which interrupt is taken when more than one interrupt is pending at any one time. The main difference between non-maskable interrupts and maskable interrupts is the fact that the non-maskable interrupt (NMI) cannot be disabled by the General Interrupt Enable (GIE) bit in the Status Register (SR). NMIs are used for high priority events such as emergency shutdown of a machine.

Most interrupts are *maskable*, which means that they are effective only if the general interrupt enable (GIE) bit is set in the status register (SR). They are ignored if GIE is clear. Therefore both the enable bit in the module and GIE must be set for interrupts to be generated.

Reset*:*

A *reset* is a sequence of operations that puts the device into a well-defined state, from which the user's program may start. This is obviously necessary when power is first applied. A reset is also generated if the device detects a serious fault in hardware or software from which the user's program cannot be expected to recover. A peculiarity of the MSP430 is that it has two levels of reset, depending on whether the reset was caused by hardware or software.

Power-on Reset (POR)

This is generated by the following severe conditions related to *hardware*:

- The device is powered up. More generally, a POR is raised if the supply voltage drops to so low a value that the device may not work correctly: a brownout.

- A low external signal on the RST/NMI pin resets the device if the pin is configured for the reset function rather than the non-maskable interrupt. The reset function RST is active by default.
- Larger variants have a more comprehensive supply voltage supervisor (SVS). This is configurable, unlike the brownout detector. It sets the SVSFG flag if the voltage falls below the programmed level and can optionally reset the device.

Power-up Clear (PUC)

This always follows a power-on reset. It is also generated when *software* appears to be out of control in the following ways:

- The watchdog timer overflows in watchdog mode. The watchdog is active by default and must either be disabled or regularly cleared before it rolls over.
- An attempt is made to write to the watchdog control register WDTCTL without the correct password 0x5A (available as the symbol WDTPW) in the upper byte. A reset is triggered even if the watchdog is disabled or in interval timer mode.
- The registers for the flash memory controller, FCTL*n*, are protected by a password in the same way as WDTCTL. The value is 0xA5, available as the constant FWKEY. This is to protect runaway software from corrupting the stored program.
- In newer devices, a PUC is triggered by an attempt to fetch an instruction from the range of addresses reserved for peripheral registers or to read unimplemented memory.

Non-maskable interrupts (NMI)

A non-maskable interrupt can be generated by following sources:

- An edge on the RST/NMI pin when configured in NMI mode
- An oscillator fault occurs
- An access violation to the flash memory

Maskable Interrupts

Maskable interrupts are caused by peripherals with interrupt capability. Each maskable interrupt source can be disabled individually by an interrupt enable bit, or all maskable interrupts can be disabled by the general interrupt enable (GIE) bit in the status register (SR).

Interrupt Processing:

When an interrupt is requested from a peripheral and the peripheral interrupt enable bit and GIE bit are set, the interrupt service routine is requested. Only the individual enable bit must be set for non-maskable interrupts to be requested.

Interrupt Acceptance:

1. Any currently executing instruction is completed if the CPU was active when the interrupt was requested. MCLK is started if the CPU was off.
2. The PC, which points to the next instruction, is pushed onto the stack.
3. The SR is pushed onto the stack.
4. The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.

5. The interrupt request flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.
6. The SR is cleared. This terminates any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.
7. The content of the interrupt vector is loaded into the PC: the program continues with the interrupt service routine at that address.

The time taken to start the execution of ISR after an interrupt request is called 'interrupt latency'. MSP430 takes six clock cycles before the ISR commences. The stack at this point is shown in Figure.

Return from Interrupt:

An interrupt service routine must always finish with the special *return from interrupt* instruction reti, which has the following actions:

1. The SR pops from the stack. All previous settings of GIE and the mode control bits are now in effect, regardless of the settings used during the interrupt service routine. In particular, this re-enables maskable interrupts and restores the previous low-power mode of operation if there was one.
2. The PC pops from the stack and execution resumes at the point where it was interrupted. Alternatively, the CPU stops and the device reverts to its low-power mode before the interrupt.

This takes a further five cycles in the MSP430. The stack is restored to its state before the interrupt was accepted.

Interrupt Nesting

Interrupt nesting is enabled if the GIE bit is set inside an interrupt service routine. When interrupt nesting is enabled, any interrupt occurring during an interrupt service routine will interrupt the routine, regardless of the interrupt priorities.

WATCHDOG TIMER:

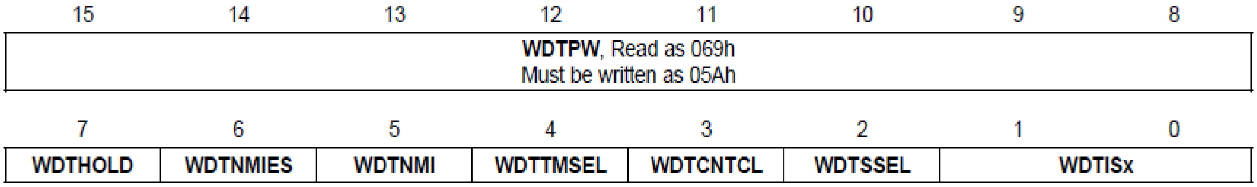
The main purpose of Watchdog timer is to protect the system against failure of the software such as hang or entering into the infinite loop etc. If the selected time interval expires, a system reset is generated. If the watchdog function is not needed in an application, the module can be configured as an interval timer and can generate interrupts at selected time intervals.

Features*:*

- Four software selectable time intervals
- Watchdog timer mode
- Interval timer mode
- Access to WDT control register is password protected
- Selectable clock source
- Can be stopped to conserve power
- Clock-fail safe feature

Watchdog timer control register

The operation of watchdog timer is controlled by 16-bit register, WDTCTL.



15	14	13	12	11	10	9	8
WDTPW, Read as 069h Must be written as 05Ah							

7	6	5	4	3	2	1	0
WDTHOLD	WDTNMIES	WDTNMI	WDTTMSEL	WDTCNTCL	WDTSSEL	WDTISx	

The upper byte is having a password to protect from unexpected write into control register. For write operation, WDTCTL password is 5Ah and read operation returns 69h into the upper byte. The lower byte of WDTCTL contains control bits to configure the timer.

WDTHOLD: This bit stops the operation of the watchdog counter. The clock multiplexer is disabled and the counter stops incrementing. It holds the last value until the hold bit is reset and the operation continues. It is cleared by the PUC signal.

0: The WDT is fully active.
1: The counter is stopped.

WDTNMIES: Watchdog timer NMI edge select. This bit selects the interrupt edge for the NMI interrupt when WDTNMI =1. Modifying this bit can trigger an NMI.

0: NMI on rising edge
1: NMI on falling edge

WDTNMI: Watchdog timer NMI select. This bit selects the function for the RST/NMI pin.

0: Reset function
1: NMI function

WDTTMSEL: Watchdog timer mode select

0: Watchdog mode
1: Interval timer mode

WDTCNTCL: Watchdog timer counter clear. Setting WDTCNTCL = 1 clears the count value to 0000h. WDTCNTCL is automatically reset.

0: No action
1: WDTCNT = 0000h

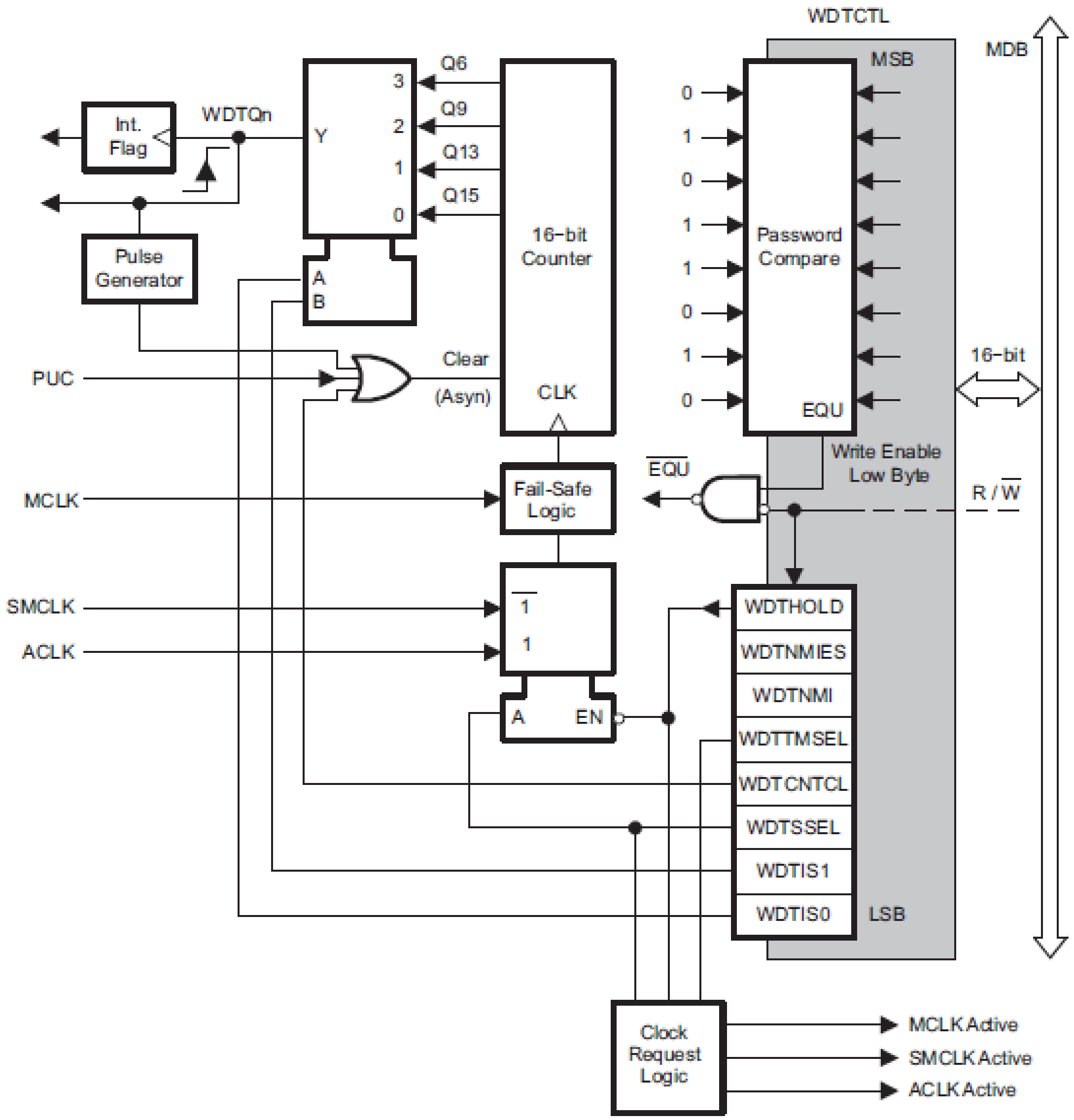
WDTSSEL: Watchdog timer clock source select

0: SMCLK
1: ACLK

WDTISx: Watchdog timer interval select. These bits select the watchdog timer interval to set the WDTIFG flag and/or generate a PUC.

00 Watchdog clock source /32768
01 Watchdog clock source /8192
10 Watchdog clock source /512
11 Watchdog clock source /64

The WDT block diagram is shown in below figure



Watchdog Timer Operation

The WDT module can be configured as either a watchdog or interval timer with the WDTCTL register. The WDTCTL register also contains control bits to configure the RST/NMI pin.

Watchdog Timer Counter

The watchdog timer counter (WDTCNT) is a 16-bit up-counter that is not directly accessible by software. The WDTCNT is controlled and time intervals are selected through the watchdog timer control register WDTCTL. The WDTCNT can be sourced from ACLK or SMCLK. The clock source is selected with the WDTSSEL bit.

Watchdog Mode

By default, the WDT module is configured in the watchdog mode with an initial 32768 cycle reset interval using the SMCLK. The user must setup, halt, or clear the WDT prior to the expiration of the initial reset interval otherwise, another PUC will be generated. When the WDT is configured to operate in watchdog mode, either writing to WDTCTL with an incorrect password, or expiration of the selected time interval triggers a PUC. A PUC resets the WDT to its default condition and configures the RST/NMI pin to reset mode.

Interval Timer Mode

Setting the WDTTMSEL bit to 1 selects the interval timer mode. This mode can be used to provide periodic interrupts. In interval timer mode, the WDTIFG flag is set at the expiration of the selected time interval. A PUC is not generated in interval timer mode at expiration of the selected timer interval and the WDTIFG enable bit WDTIE remains unchanged.

When the WDTIE bit and the GIE bit are set, the WDTIFG flag requests an interrupt. The WDTIFG interrupt flag is automatically reset when its interrupt request is serviced, or may be reset by software. The interrupt vector address in interval timer mode is different from that in watchdog mode.

Watchdog Timer Interrupts

The WDT uses two bits in the SFRs for interrupt control.

- The WDT interrupt flag, WDTIFG, located in IFG1.0
- The WDT interrupt enable, WDTIE, located in IE1.0

Watchdog mode: The WDTIFG flag sources a reset vector interrupt. If WDTIFG=1, the WDT initiates the reset condition, either by timing out or by a password violation. The user has control of the reset source.
Interval mode: The WDTIFG flag is set after the selected time interval and requests a WDT interval timer interrupt if the WDTIE and the GIE bits are set. When the interrupt is serviced, the
WDTIFG flag is reset automatically. It also can be reset using software.

Watchdog Timer Clock Fail-Safe Operation

This includes fail-safe logic to preserve the watchdog's clock. For example, if ACLK is the WDT clock source, it does not let the device enter LPM4 because that would disable its clock. Therefore it is not possible to use LPM4 with WDT active; the watchdog 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. This feature is not available when WDT is in interval timer mode.

SYSTEM CLOCKS:

The MSP430 devices have a clock system that allows the CPU and the peripherals to operate from different clock sources.

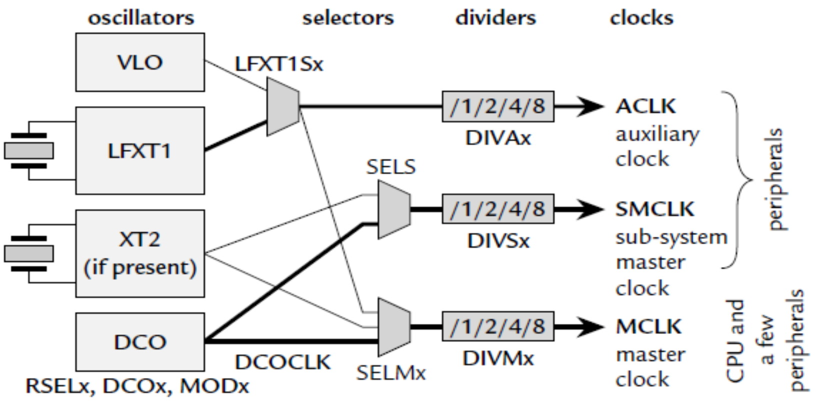
System clock have conflicting requirements due to following reasons:

- Low power
- Fast start/stop
- Accurate
- High performance

Devices often run in low power modes until some event occurs, then system must wakeup and handles events rapidly.
Two kinds of clocks are needed:

- Fast clock: Start/ Stop rapidly; clock need not be accurate
- Slow clock: Runs continuously to monitor real time; uses less power to be accurate

The simplified block diagram of clock system is shown below:



The clock sources from these oscillators can be selected to generate a range of different clock signals: Master clock (MCLK), Sub-system master clock (SMCLK) and auxiliary clock (ACLK). Each of these clock signals can be internally divided by 1, 2, 4 or 8, before being made available to the CPU and peripheral devices.

- Master clock, MCLK is used by CPU and a few peripherals and is sourced by any of these oscillators (LFXT1 or XT2 or DCO). By default, it is sourced by DCO. The CPUOff bit is located in the status register and halts the selected clock source.
- Sub-Master clock, SMCLK is used by peripherals which is sourced by any of these oscillators (XT2 or DCO). The SCG1 bit is located in the status register and halts the selected clock source.
- The auxiliary clock (ACLK) is the buffered output of the LFXT1 oscillator and provides clock signals for peripheral modules. The control bit XTS selects whether the LFXT1 oscillator operates as a low-frequency (LF) crystal oscillator or as a high-frequency (XT1) oscillator.

The Basic Clock Module includes different oscillators to provide different clocks for the CPU and peripherals and is given below:

1. Very Low Frequency Oscillator (VLO)
2. Low or High Frequency Crystal Oscillator (LFXT1)
3. High Frequency Oscillator (XT2)
4. Digitally Controlled Oscillator (DCO)

1. Very Low Frequency Oscillator (VLO):

The internal very-low-power low-frequency oscillator (VLO) provides a typical frequency of 12 kHz without requiring a crystal. The OSCOFF bit disables the VLO for LPM4. The LFXT1 crystal oscillators are disabled when the VLO is selected reducing current consumption. The VLO consumes no power when not being used.

2. Low or High Frequency Crystal Oscillator (LFXT1):

This is available in all the devices. The LFXT1 oscillator supports ultra-low current consumption using a 32 KHz. A watch crystal connects to XIN and XOUT without any other external components.

The LFXT1 oscillator also supports high-speed crystals or resonators in the range 450 kHz to 16 MHz The high-speed crystal or resonator connects to XIN and XOUT and requires external capacitors on both terminals.

The LFXT1 oscillator can be powered down (bit OSCOFF=1) if it is not used to source MCLK and can use clock signal from an external oscillator (XIN pin).

3. High Frequency Oscillator (XT2):

Some devices have a second crystal oscillator, XT2. XT2 sources XT2CLK and its characteristics are identical to LFXT1 in HF mode. The XT2OFF bit disables the XT2 oscillator if XT2CLK is not used for MCLK or SMCLK.

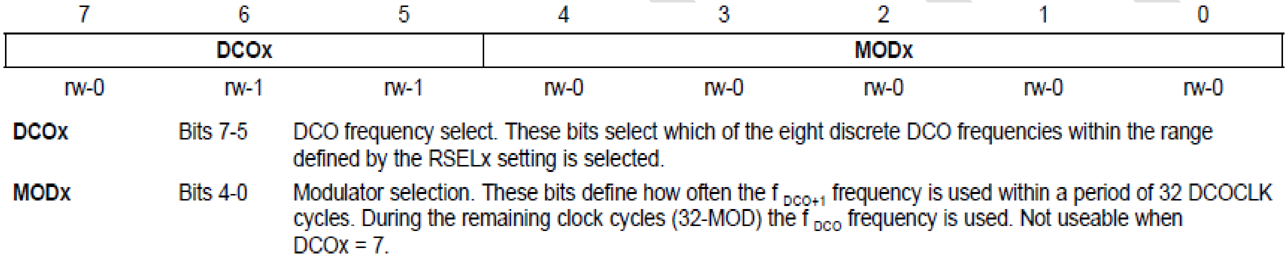
After applying V_{CC} or after restarting the crystal oscillator, the OFIFG is set to '1' and requests NMI if OFIE is active. The OFIFG becomes inactive when XT2 starts oscillating after 100 clock cycles (at 1MHz).

4. Digitally Controlled Oscillator (DCO):

The DCO is an oscillator with RC-type whose frequency is programmed by a current, which is selected by RSEL. DCO is available in all devices.

Basic Clock Module Control Registers:

1. DCOCTL, DCO control Register

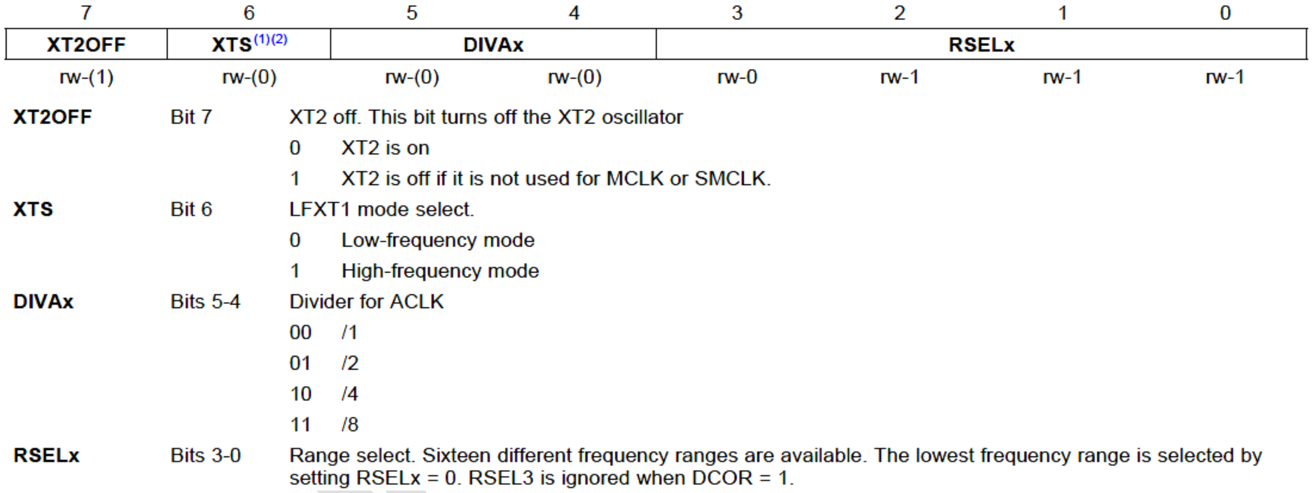


7	6	5	4	3	2	1	0
DCOx			MODx				
rw-0	rw-1	rw-1	rw-0	rw-0	rw-0	rw-0	rw-0

DCOx Bits 7-5 DCO frequency select. These bits select which of the eight discrete DCO frequencies within the range defined by the RSELx setting is selected.

MODx Bits 4-0 Modulator selection. These bits define how often the f_{DCO+1} frequency is used within a period of 32 DCOCLK cycles. During the remaining clock cycles (32-MOD) the f_{DCO} frequency is used. Not useable when DCOx = 7.

2. BCSCTL1, Basic Clock System Control Register 1



7	6	5	4	3	2	1	0
XT2OFF	XTS[(1)(2)]	DIVAx		RSELx			
rw-(1)	rw-(0)	rw-(0)	rw-(0)	rw-0	rw-1	rw-1	rw-1

XT2OFF Bit 7 XT2 off. This bit turns off the XT2 oscillator

0 XT2 is on

1 XT2 is off if it is not used for MCLK or SMCLK.

XTS Bit 6 LFXT1 mode select.

0 Low-frequency mode

1 High-frequency mode

DIVAx Bits 5-4 Divider for ACLK

00 /1

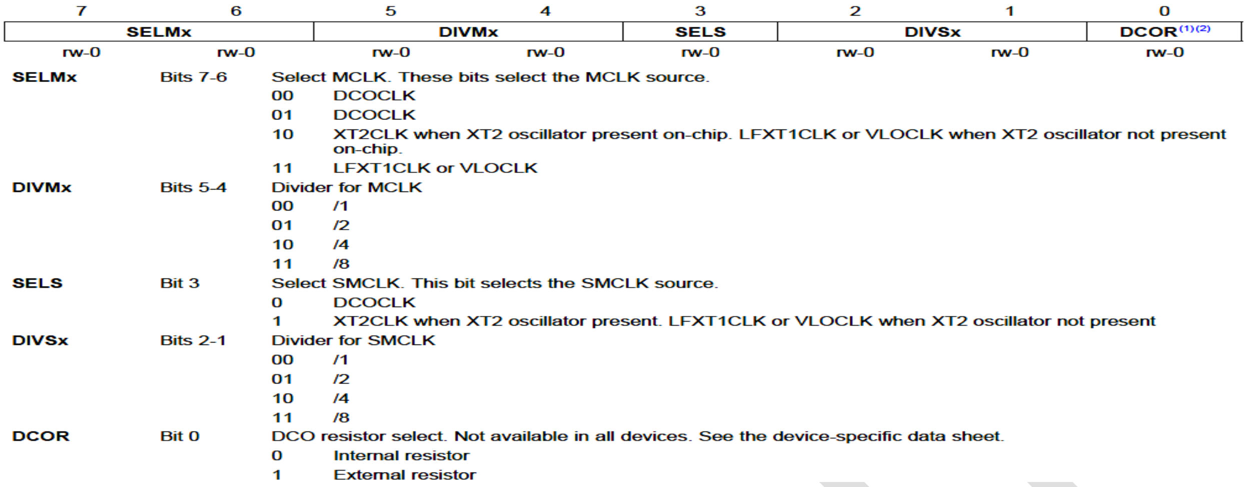
01 /2

10 /4

11 /8

RSELx Bits 3-0 Range select. Sixteen different frequency ranges are available. The lowest frequency range is selected by setting RSELx = 0. RSEL3 is ignored when DCOR = 1.

3. BCSCTL2, Basic Clock System Control Register 2



7	6	5	4	3	2	1	0
SELMx		DIVMx		SELS	DIVSx		DCOR[1][2]
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

SELMx Bits 7-6 Select MCLK. These bits select the MCLK source.
00 DCOCLK
01 DCOCLK
10 XT2CLK when XT2 oscillator present on-chip. LFXT1CLK or VLOCLK when XT2 oscillator not present on-chip.
11 LFXT1CLK or VLOCLK

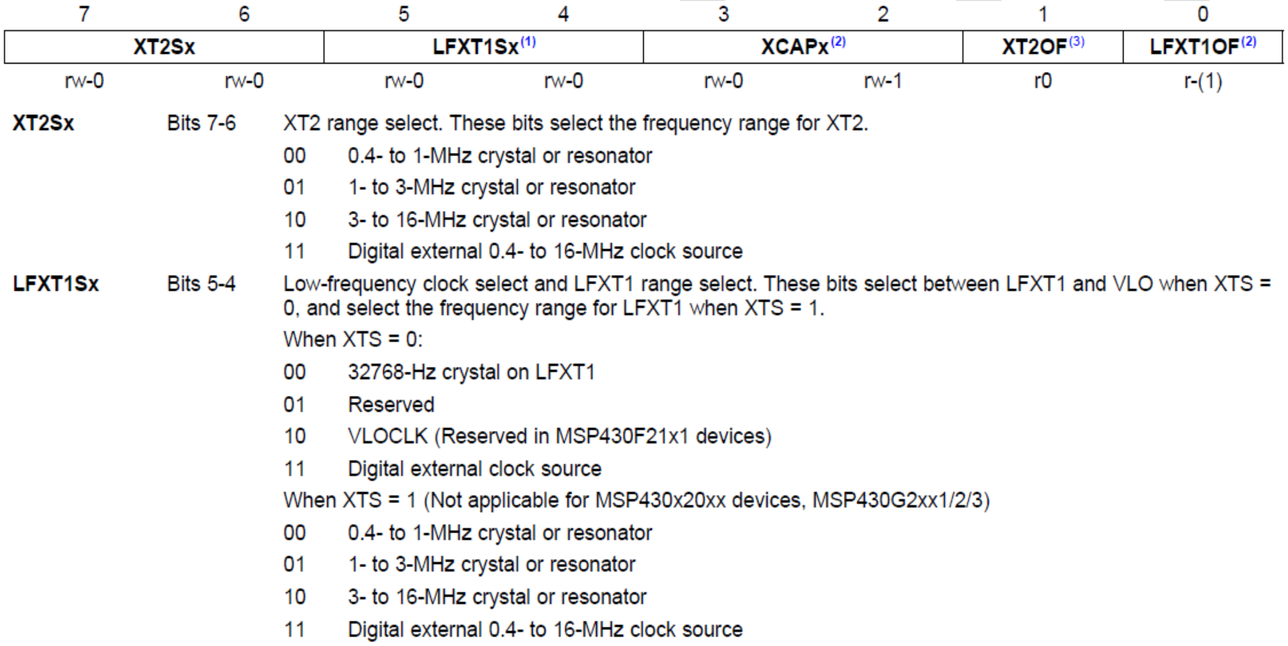
DIVMx Bits 5-4 Divider for MCLK
00 /1
01 /2
10 /4
11 /8

SELS Bit 3 Select SMCLK. This bit selects the SMCLK source.
0 DCOCLK
1 XT2CLK when XT2 oscillator present. LFXT1CLK or VLOCLK when XT2 oscillator not present

DIVSx Bits 2-1 Divider for SMCLK
00 /1
01 /2
10 /4
11 /8

DCOR Bit 0 DCO resistor select. Not available in all devices. See the device-specific data sheet.
0 Internal resistor
1 External resistor

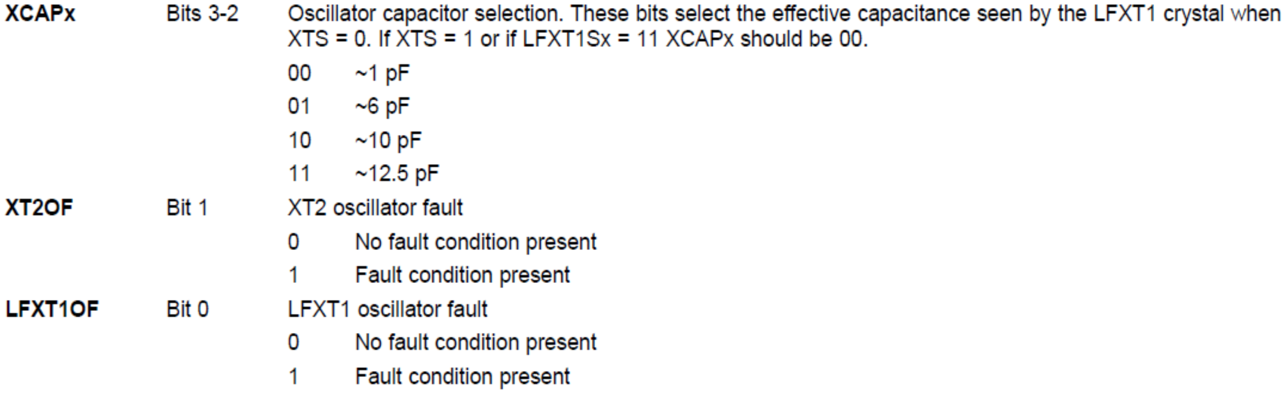
4. BCSCTL3, Basic Clock System Control Register 3



7	6	5	4	3	2	1	0
XT2Sx		LFXT1Sx[1]		XCAPx[2]		XT2OF[3]	LFXT1OF[2]
rw-0	rw-0	rw-0	rw-0	rw-0	rw-1	r0	r-(1)

XT2Sx Bits 7-6 XT2 range select. These bits select the frequency range for XT2.
00 0.4- to 1-MHz crystal or resonator
01 1- to 3-MHz crystal or resonator
10 3- to 16-MHz crystal or resonator
11 Digital external 0.4- to 16-MHz clock source

LFXT1Sx Bits 5-4 Low-frequency clock select and LFXT1 range select. These bits select between LFXT1 and VLO when XTS = 0, and select the frequency range for LFXT1 when XTS = 1.
When XTS = 0:
00 32768-Hz crystal on LFXT1
01 Reserved
10 VLOCLK (Reserved in MSP430F21x1 devices)
11 Digital external clock source
When XTS = 1 (Not applicable for MSP430x20xx devices, MSP430G2xx1/2/3)
00 0.4- to 1-MHz crystal or resonator
01 1- to 3-MHz crystal or resonator
10 3- to 16-MHz crystal or resonator
11 Digital external 0.4- to 16-MHz clock source



XCAPx Bits 3-2 Oscillator capacitor selection. These bits select the effective capacitance seen by the LFXT1 crystal when XTS = 0. If XTS = 1 or if LFXT1Sx = 11 XCAPx should be 00.
00 ~1 pF
01 ~6 pF
10 ~10 pF
11 ~12.5 pF

XT2OF Bit 1 XT2 oscillator fault
0 No fault condition present
1 Fault condition present

LFXT1OF Bit 0 LFXT1 oscillator fault
0 No fault condition present
1 Fault condition present

Control of clock module with status register:

CPUOFF-disables MCLK, stops CPU and any other peripherals that use MCLK.

SCG0-disables SMCLK and stops other peripherals that use SMCLK.
SCG1-diasbles DC generator for DCO.
OSCOFF-disables VLO & LFXT1

MSP430 Low Power Modes

The MSP430 is designed from the ground up for low power. This includes both design and process implementation. Despite this, the bulk of power savings is realized by placing the MSP430 in various power saving modes. We first have to understand what consumes the most current in the MSP430. This breaks down as follows:

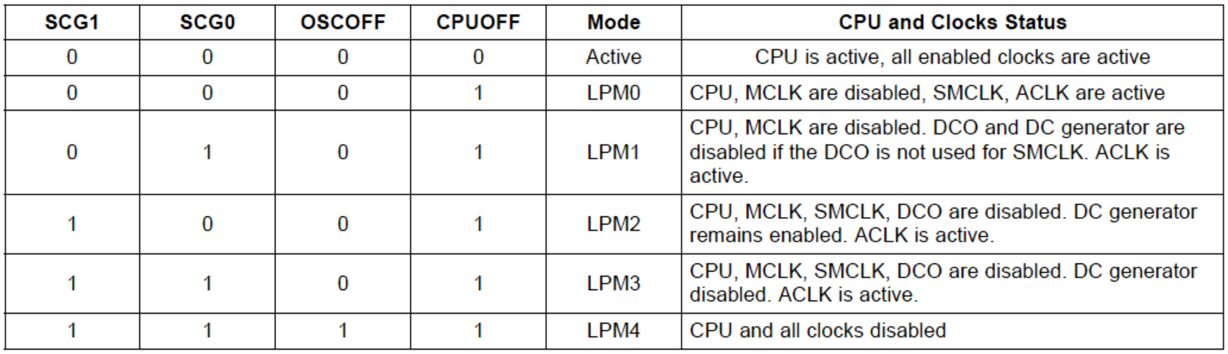
- MSP430 CPU draws the most, proportional to the frequency at which it is running.
- Clocks and Oscillators, especially high speed clocks.
- Modules and Peripherals

Saving power comes down to shutting off as many modules, peripherals, and clocks as we can, for as long as possible without violating the application time requirements. It is also important to reduce the speed of the CPU as it can significantly affect power consumption. The trade-off is that processing will take longer.

MSP430 peripherals are also designed for low power. For example, the ADC will shut off automatically after the conversions are finished. The issue is always what to turn off and when, also when to wake them up. Interrupts play a central role because they enable the MSP430 to go to a deep sleep and wake up if events have occurred, either external as detected by GPIOs or internally generated by the peripherals.

There are four bits that control the CPU and modes of operation present in SR: CPUOff, OscOff, SCG0 and SCG1. The major advantage of including the operating mode bits in the status register is that the present state of the operating condition is saved onto the stack during an interrupt service request. As long as the stored status register information is not altered, the processor continues (after RETI) with the same operating mode as before the interrupt event.

These LPM modes refer to the individual bits in the Status Register. By setting and clearing the bits in the SR, one can turn off CPU and clocks resulting in certain Low Power Modes.



SCG1	SCG0	OSCOFF	CPUOFF	Mode	CPU and Clocks Status
0	0	0	0	Active	CPU is active, all enabled clocks are active
0	0	0	1	LPM0	CPU, MCLK are disabled, SMCLK, ACLK are active
0	1	0	1	LPM1	CPU, MCLK are disabled. DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active.
1	0	0	1	LPM2	CPU, MCLK, SMCLK, DCO are disabled. DC generator remains enabled. ACLK is active.
1	1	0	1	LPM3	CPU, MCLK, SMCLK, DCO are disabled. DC generator disabled. ACLK is active.
1	1	1	1	LPM4	CPU and all clocks disabled

Entering into Low Power modes:

To enter into lo power mode the bis (bit set) instruction is used.
bis.w, # LPM3, SR; enters into low power mode 3

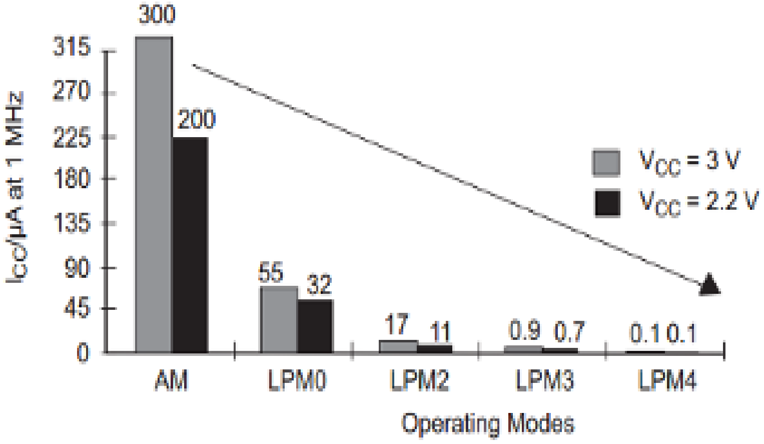
Exiting from Low Power Modes:

To exit from low power mode bic instruction is used.
Bic.w, # GIE ! LPM3, SR; enter into normal mode
Interrupt to the system can also wakeup MSP 430 from low power mode.

Active Vs Standby current consumption:

Power consumption depends on clock frequency, ambient temperature, supply voltage, peripheral selection, input/output usage and memory type.

The power consumption profile of the various LPMs:



At 1MHz, we go from 300uA down to less than 1uA by switching to LPM3. This is what makes the MSP430 such a good microcontroller for low power applications. It can survive for years on batteries. But, to take advantage of this, we have to build an application that takes advantage of these low power modes and turns on only what's necessary when it's necessary.

<u>Low power operating modes program flow steps:</u>

1) Enable/disable CPUOFF, OSCOFF, SCG0, SCG1 bits in Status register.
2) LPM (low power mode) is active after writing to Status register.
3) CPU will suspend program execution.
4) Disables peripherals
    - Operating with any disabled clock
    - Individual control register setting
5) All I/O port pins and RAM/Registers are unchanged
6) Wakeup is possible through enabled interrupts
7) Enter into ISR
    - Operating mode is saved on stack
    - PC and SR is stored on stack
    - Interrupt vector is moved to PC
    - CPUOFF, OSCOFF, SCG0, SCG1 bits are reset
8) Return from ISR
    - Status register is popped from stack
    - Restore previous mode
9) Interrupt wakes up MSP 430

<u>FRAM Vs Flash for low power & reliability</u>

Ferroelectric Random Access Memory (FRAM) is an ultra-low power nonvolatile memory technology with write speeds similar to static RAM (SRAM). The technology has been in the industry for over a decade, implemented as stand-alone memory. FRAM's first introduction as an embedded memory in a general purpose ultra-low-power MCU was on Texas Instruments 16-bit MSP430™ product line, the MSP430FR57xx family.

Some of the key attributes of FRAM are:

• FRAM is nonvolatile; that is, it retains its contents on loss of power.

• FRAM is a true random-access memory. The memory is not segmented; addressing of data for read or writes at word or byte level happens directly in the same way as SRAM.

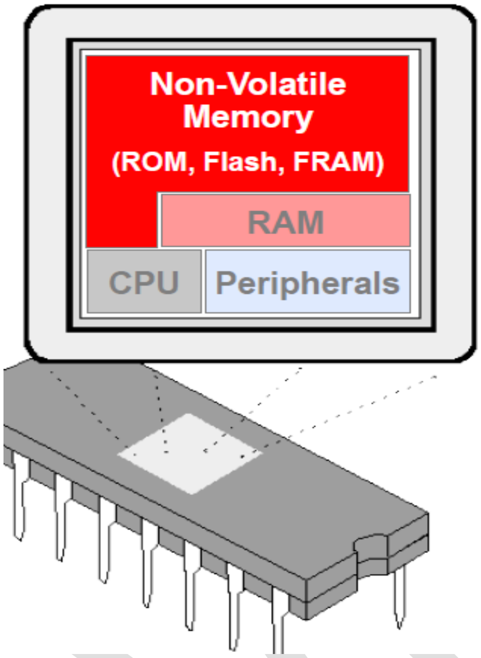
• The embedded FRAM on MSP430 devices can be accessed (read or write) at a maximum speed of 8 MHz Above 8 MHz, wait states are used when accessing FRAM.

• Writing to FRAM and reading from FRAM require no setup or preparation such as pre-erase before write or unlocking of control registers.

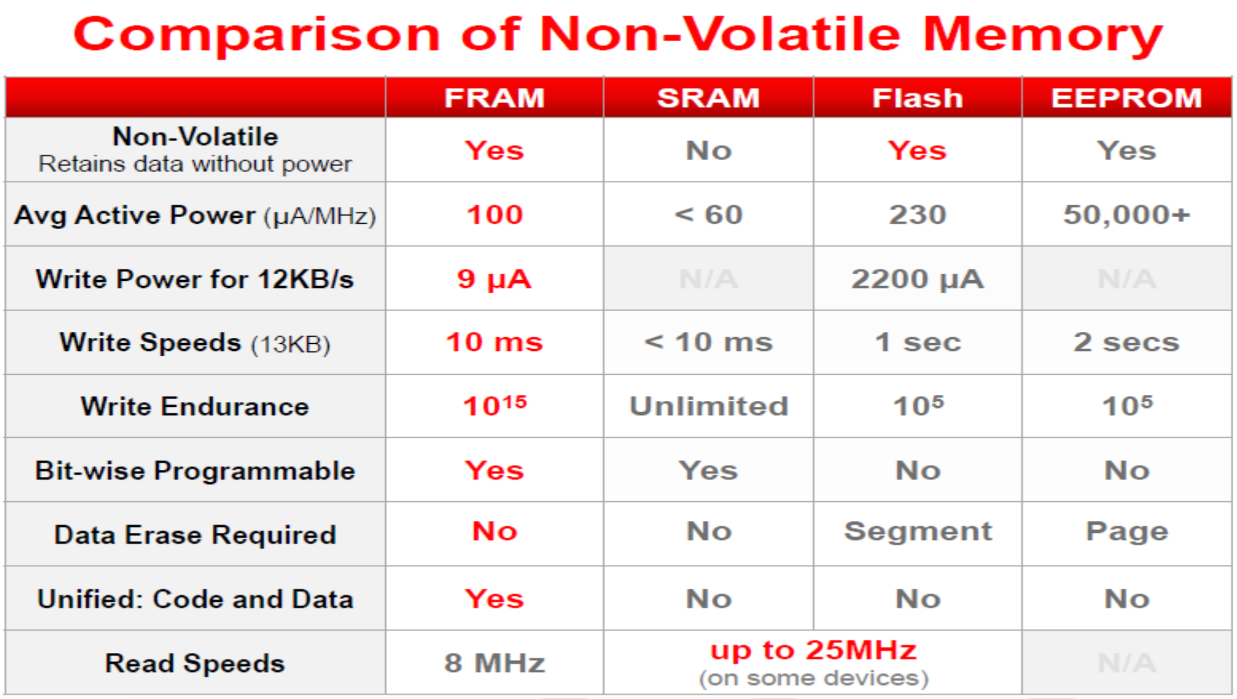
• FRAM write accesses are extremely low power, because writing to FRAM does not require a charge pump.

• FRAM writes can be performed across the full voltage range of the device.

• FRAM meets and exceeds reliability requirements on data integrity. It provides practically unlimited endurance for read and writes operations on the order of 1015 write or erase cycles.

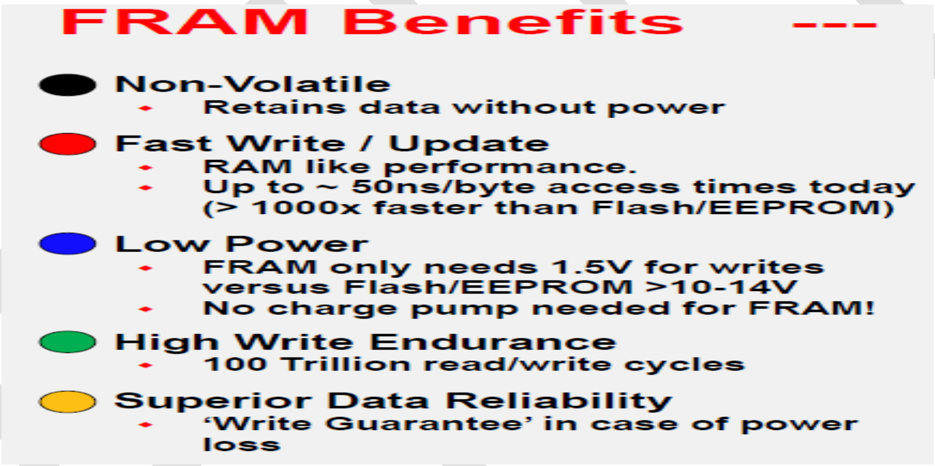


- Non-Volatile Memory (NVM) retains its information when powered down
- By contrast, Random Access Memory (RAM) needs power to keep its information
- NVM examples include:
  (MSP430 devices only use Flash or FRAM)
  - Flash Memory
  - FRAM (ferroelectric RAM)
  - ROM (read-only memory)
  - EEPROM (electrically erasable ROM)
  - Hard disk drives
- Flash & FRAM are in-system programmable, which means a program can rewrite them
- Typical MCU applications use:
  - NVM for program and calibration data
  - RAM for variables, stack and heap



Comparison of Non-Volatile Memory

	FRAM	SRAM	Flash	EEPROM
Non-Volatile Retains data without power	Yes	No	Yes	Yes
Avg Active Power (µA/MHz)	100	< 60	230	50,000+
Write Power for 12KB/s	9 µA	N/A	2200 µA	N/A
Write Speeds (13KB)	10 ms	< 10 ms	1 sec	2 secs
Write Endurance	10^{15}	Unlimited	10^{5}	10^{5}
Bit-wise Programmable	Yes	Yes	No	No
Data Erase Required	No	No	Segment	Page
Unified: Code and Data	Yes	No	No	No
Read Speeds	8 MHz	up to 25MHz (on some devices)		N/A



FRAM Benefits ---

- ● **Non-Volatile**
  - Retains data without power
- ● **Fast Write / Update**
  - RAM like performance.
  - Up to ~ 50ns/byte access times today (> 1000x faster than Flash/EEPROM)
- ● **Low Power**
  - FRAM only needs 1.5V for writes versus Flash/EEPROM >10-14V
  - No charge pump needed for FRAM!
- ● **High Write Endurance**
  - 100 Trillion read/write cycles
- ● **Superior Data Reliability**
  - 'Write Guarantee' in case of power loss

Timers:

Timers produce delays. These delays are unpredictable. So reference time is required. The reference of time can be produced by counter which counts fixed number of time. Actual time can be obtained by multiplying counter with clock interval time. The accuracy and stability of a clock should be critical.

Timers can be used for:

- Generates fixed time period events
- Allows a periodic wakeup from sleep mode
- Counts signal edges
- Replace delay loops

Timers can have different forms:

- Watchdog timers
- Basic Timer1
- Basic Timer2
- RTC

- Timer A
- Timer B

Watchdog Timers

The main purpose of Watchdog timer is to protect the system against failure of the software such as hang or entering into the infinite loop etc. If the selected time interval expires, a system reset is generated. If the watchdog function is not needed in an application, the module can be configured as an interval timer and can generate interrupts at selected time intervals.

Features*:*

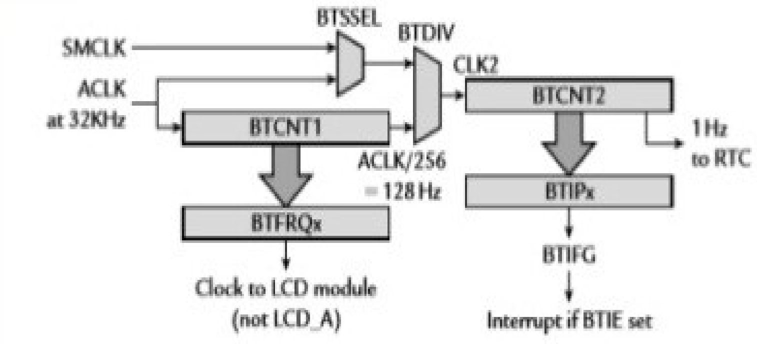
- Four software selectable time intervals
- Watchdog timer mode
- Interval timer mode
- Access to WDT control register is password protected
- Selectable clock source
- Can be stopped to conserve power
- Clock-fail safe feature

Basic Timer 1:

The Basic Timer 1 module is formed by **two independent 8-bit timers**: “Basic Timer 1 Counter 1 ” (BTCNT1) and “Basic Timer 1 Counter 2” (BTCNT2), which can be used in cascade to form a 16-bit timer.

Basic Timer 1 provides two or three functions in the MSP430:

- Clock for the LCD module
- Periodic Interrupts
- 1Hz clock signal to drive the Real-Time Clock
- Suitable for a RTC implementation
- Basic interval timer
- Simple interrupt capability



BTCNT1:

- Used to generate the frame frequency for the LCD controller:
- Read/write 8-bit register;
- Clock source: ACLK;

- Frame frequency selectable by software (BTFRFQx bits on the BTCTL register) based on the ACLK division (x);
- fLCD = ACLK/x.

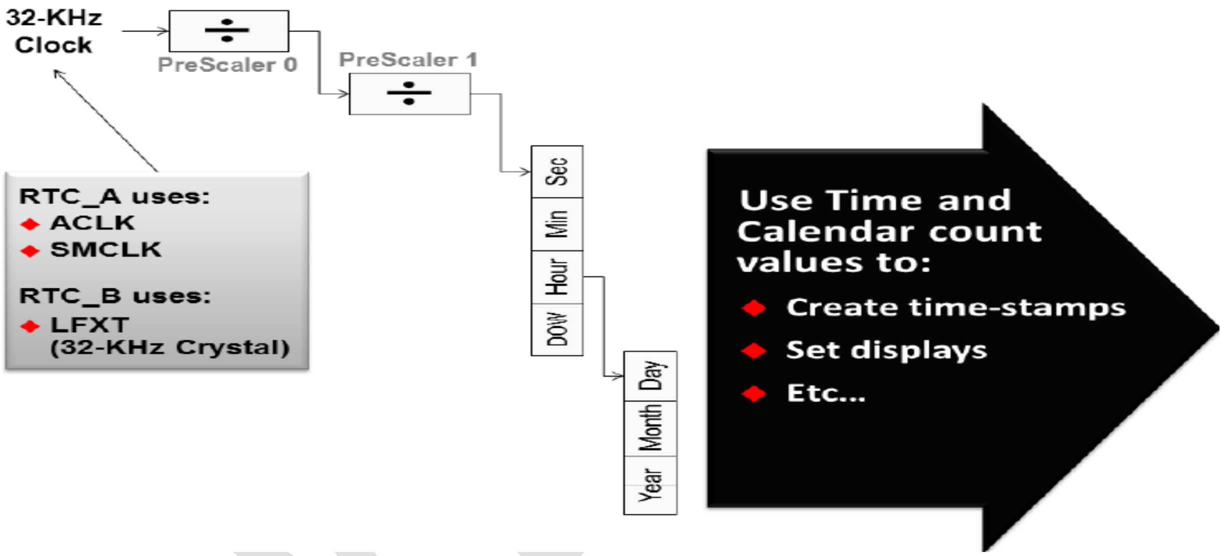
BTCNT2:

- Used as a programmable frequency divider with
- Interrupt capability, to provide periodic CPU interrupts and/or a Real Time Clock system.
- Read/write 8-bit register;
- Three possible clock sources: ACLK or SMCLK, or ACLK/256 when cascaded with BTCNT1 (selected by the BTSSEL and BTDIV bits on the BTCTL register);
- Sources the Basic Timer1 interrupt BTIFG, with an interval selected by the BTIPx bits in the BTCTL register;

Real Time Clock

Real Time Clock provides, real time clock and calender. RTC can be configurable with calender function or general purpose counter. It provides seconds, minutes, hours, days, months and years in RTC with calender function.RTC has interrupt capability. RTC mode can be selected in both binary and BCD format. RTC mode has programmable alarms.

RTC Block Diagram



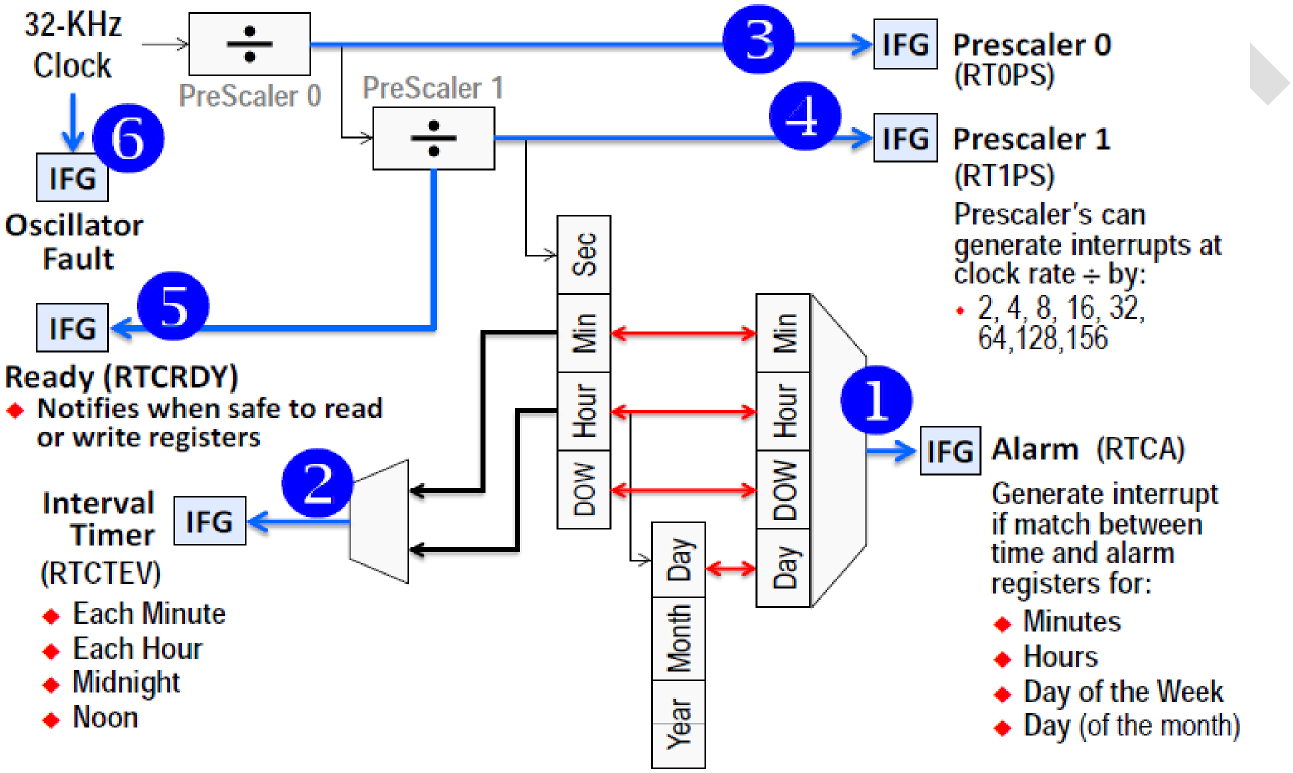
Real Time Clock Control Register (RTCCTL)

7	6	5	4	3	2	1	0
RTCBCD	RTCHOLD	RTCMODEx		RTCTEVx		RTCIE	RTCFG

The current time and date are held in a set of registers that contain the following bytes:

- Second (RTCSEC).
- Minute (RTCMIN).
- Hour (RTCHOUR), which runs from 0–23 (24-hour format).
- Day of week (RTCDOW), which runs from 0–6.
- Day of month (RTCDAY).
- Month (RTCMON).
- Year (RTCYEARL), assuming BCD format.
- Century (RTCYEARH), assuming BCD format.

RTC Interrupts



RTC modes:

RTC module can be configured in 2 modes:

- 32 bit general purpose Counter mode
- Calender mode

Counter mode can be selected when RTCMODE is RESET and calender mode is selected when RTCMODE is SET.

Counter Mode

•Counter can be incremented by the clock

•It can be sourced from ACLK,SMCLK,or prescaled versions of ACLK or SMCLK

•Prescaled version of ACLK or SMCLK are sourced from the prescale dividers•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

Calendar Mode

•RTCMODE is set
•It provides seconds, minutes,hours, day of week,day of month,month and year
•In selectable BCD or Hexadecimal Format
•Includes leap year algoritham[all years evenly divisible by four]
•RT0PS is sourced from ACLK[32.768KHz for RTC_A Calendar operation]
•RT1PS is cascaded with the output ACLK/256 of RT0PS

Real time clock alarm Function

•The RTC_A module provides for a flexible alarm system
•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.

Timers:

MSP430 microcontroller has following timer features

- 16-bit Timer/Counter
- 3-bit Prescaler
- 3 Capture/Compare Registers
- Four Modes
- Stop, Up, Continuous, Up/Down
- Selectable Clock Source
- Configurable Outputs
- PWM Capable

There are two main parts to the Timer hardware:

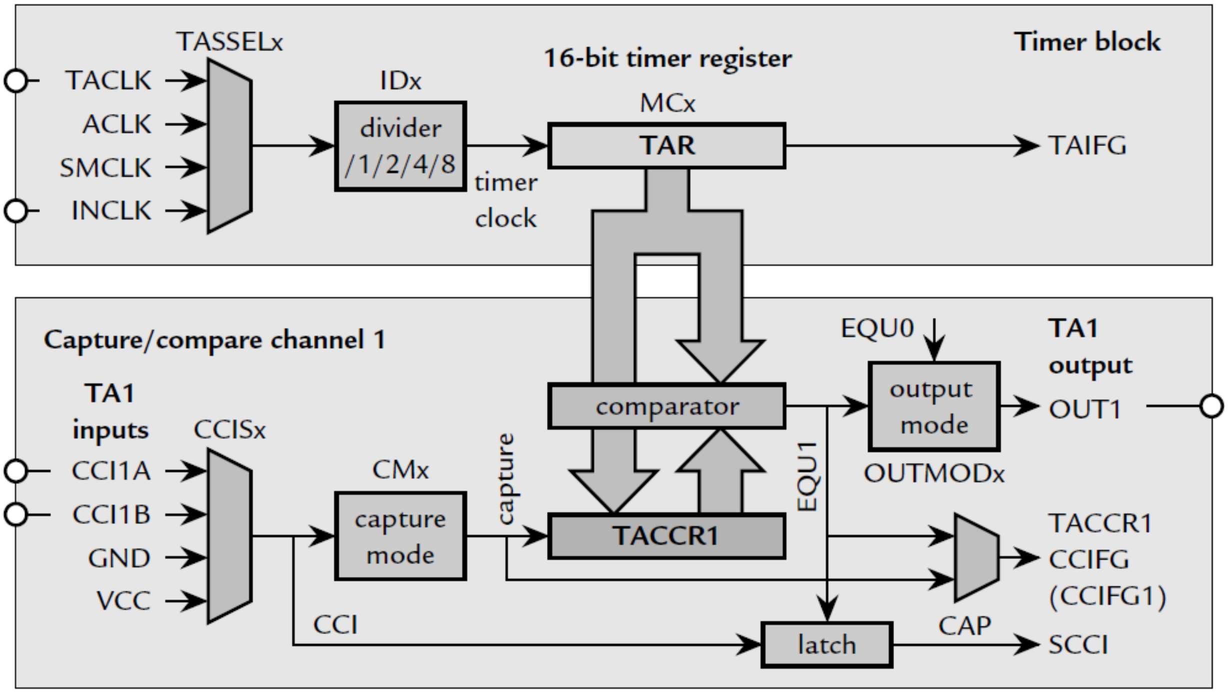
Timer block: The core, based on the 16-bit register TAR. There is a choice of sources for the clock, whose frequency can be divided down (prescaled). The timer block has no output but a flag TAIFG is raised when the counter returns to 0.

Capture/compare channels: In which most events occur, each of which is based on a register TACCRn. They all work in the same way with the important exception of TACCR0.

Each channel can

- **Capture** an input, which means recording the “time” (the value in TAR) at which the input changes in TACCRn; the input can be either external or internal from another peripheral or software.
- **Compare** the current value of TAR with the value stored in TACCRn and update an output when they match; the output can again be either external or internal.
- **Request an interrupt** by setting its flag TACCRn CCIFG on either of these events; this can be done even if no output signal is produced.
- **Sample** an input at a compare event; this special feature is particularly useful if Timer_A is

used for serial communication in a device that lacks a dedicated interface.



Timer Block:

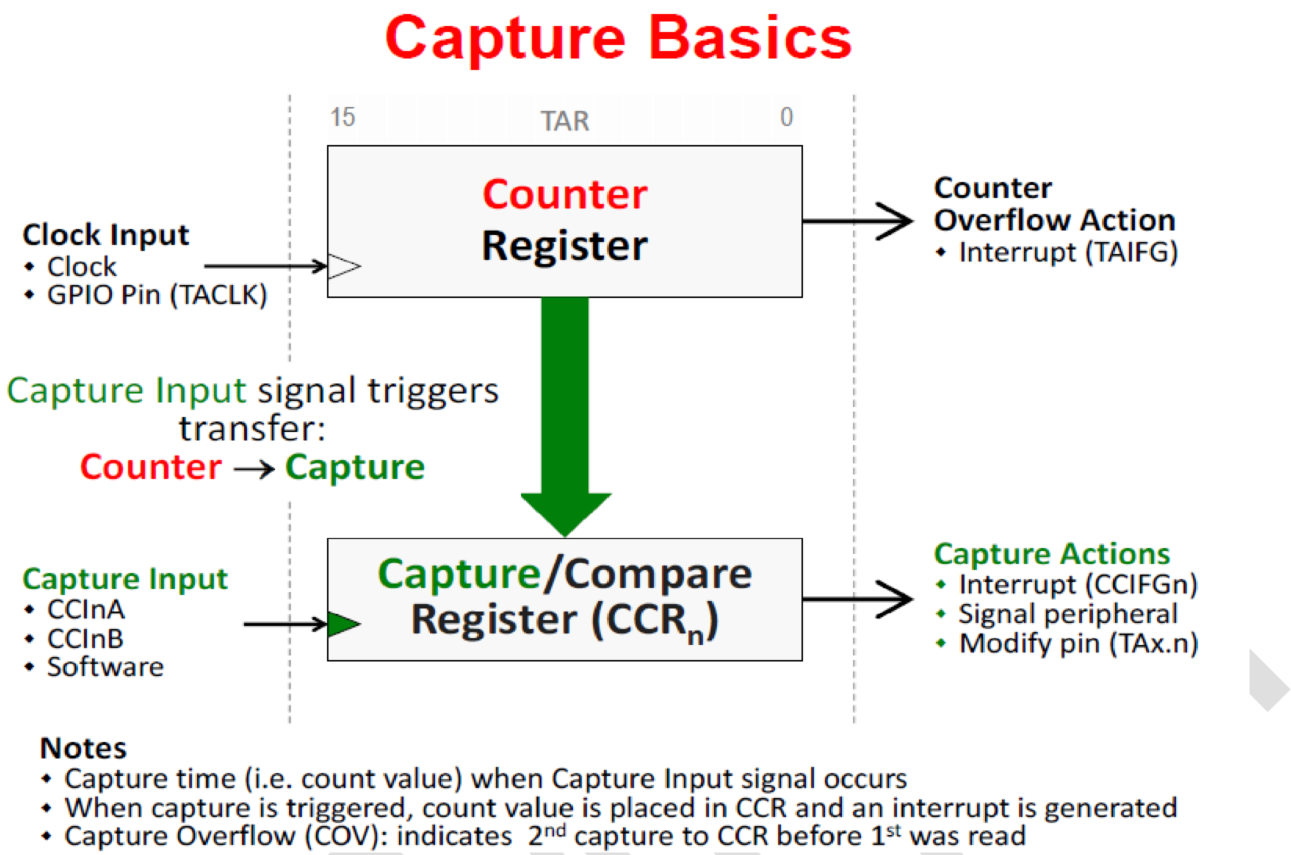
This contains the 16-bit timer register TAR, which is central to the operation of the timer. It is controlled by the register TACTL shown in Figure.

Remember that a timer is really no more than a counter and has no direct concept of time (the Real-Time Clock is an exception). It is the programmer's job to establish a relation between the value in the counter and real time. This depends essentially on the frequency of the clock for the timer. It can be chosen from four sources by using the TASSELx bits

- **SMCLK** is internal and usually fast (megahertz).
- **ACLK** is internal and usually slow, typically 32 KHz from a watch crystal .
- **TACLK** is external.
- **INCLK** is also external, sometimes a separate pin but often it is connected through an inverter to the pin for TACLK so that INCLK = $\overline{\text{TACLK.}}$

The frequency of the incoming clock can be divided down by 2, 4, or 8 if desired by configuring the IDx bits.

Capture: When a capture input signal occurs, a snapshot of the counter register is captured. It is copied into a capture register (CCR). Overflow occurs when indicate 2nd capture to CCR before 1st was read. There are 7 captures registers (CR0-CR6).

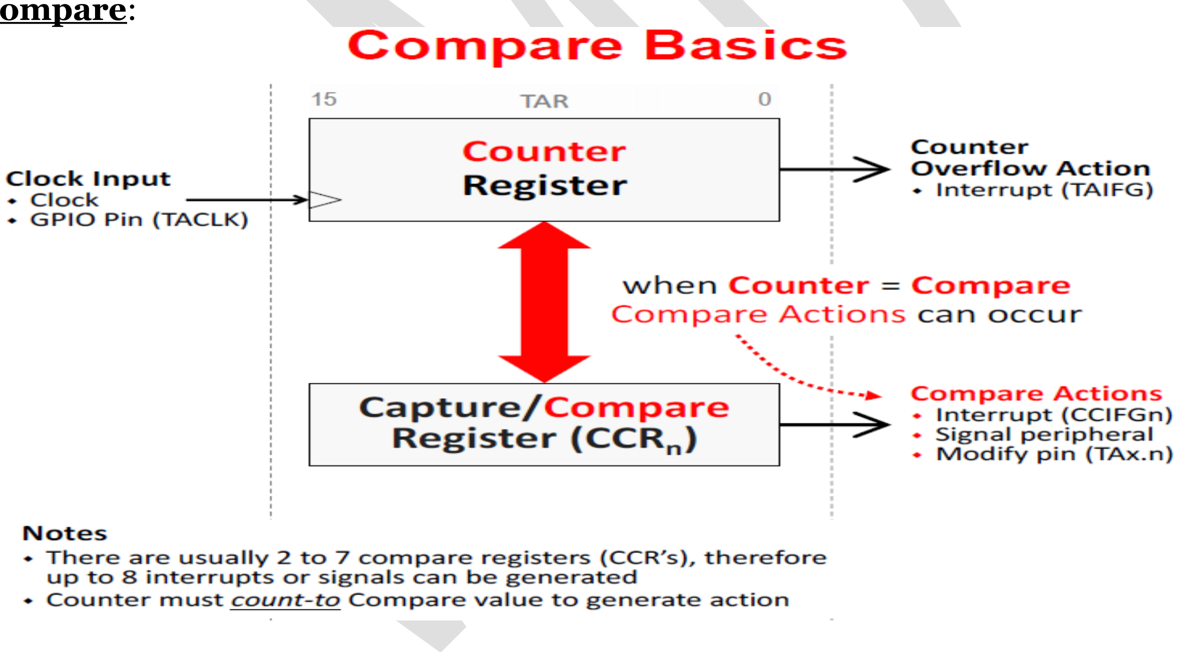


Capture Basics

Notes

- Capture time (i.e. count value) when Capture Input signal occurs
- When capture is triggered, count value is placed in CCR and an interrupt is generated
- Capture Overflow (COV): indicates 2nd capture to CCR before 1st was read

Compare:

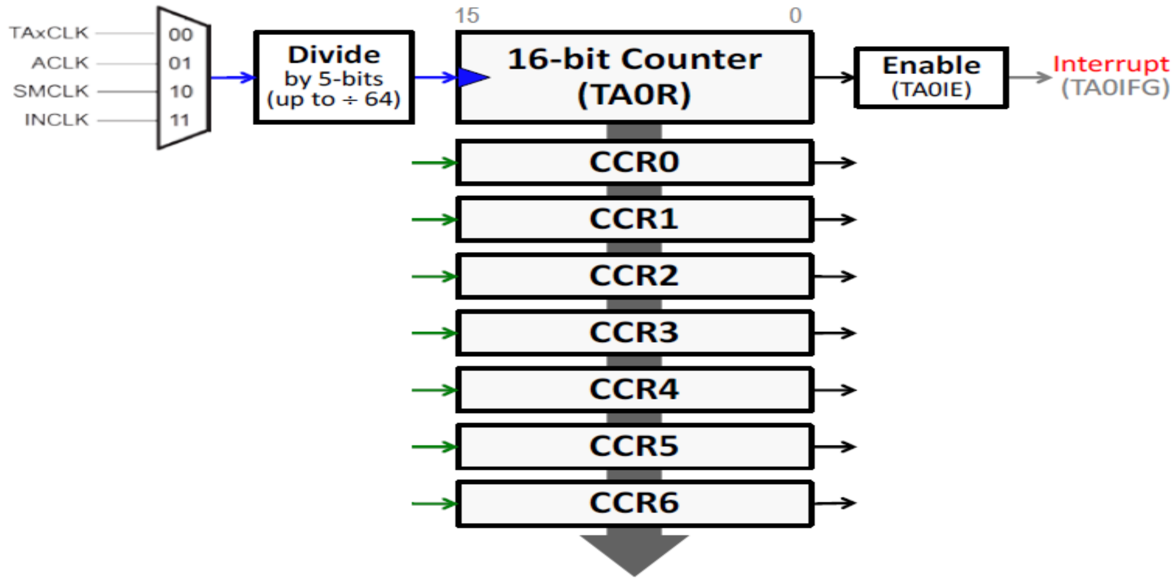


Compare Basics

Notes

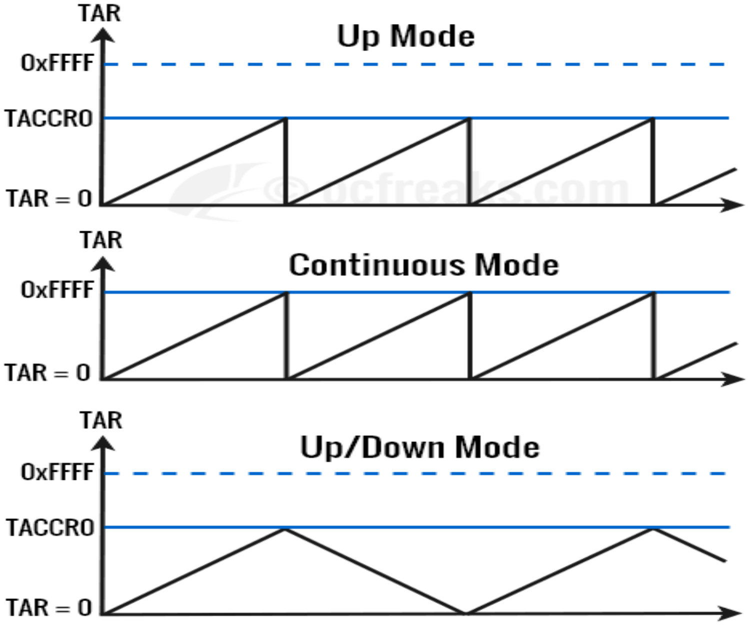
- There are usually 2 to 7 compare registers (CCR's), therefore up to 8 interrupts or signals can be generated
- Counter must *count-to* Compare value to generate action

Example: Timer0_A7

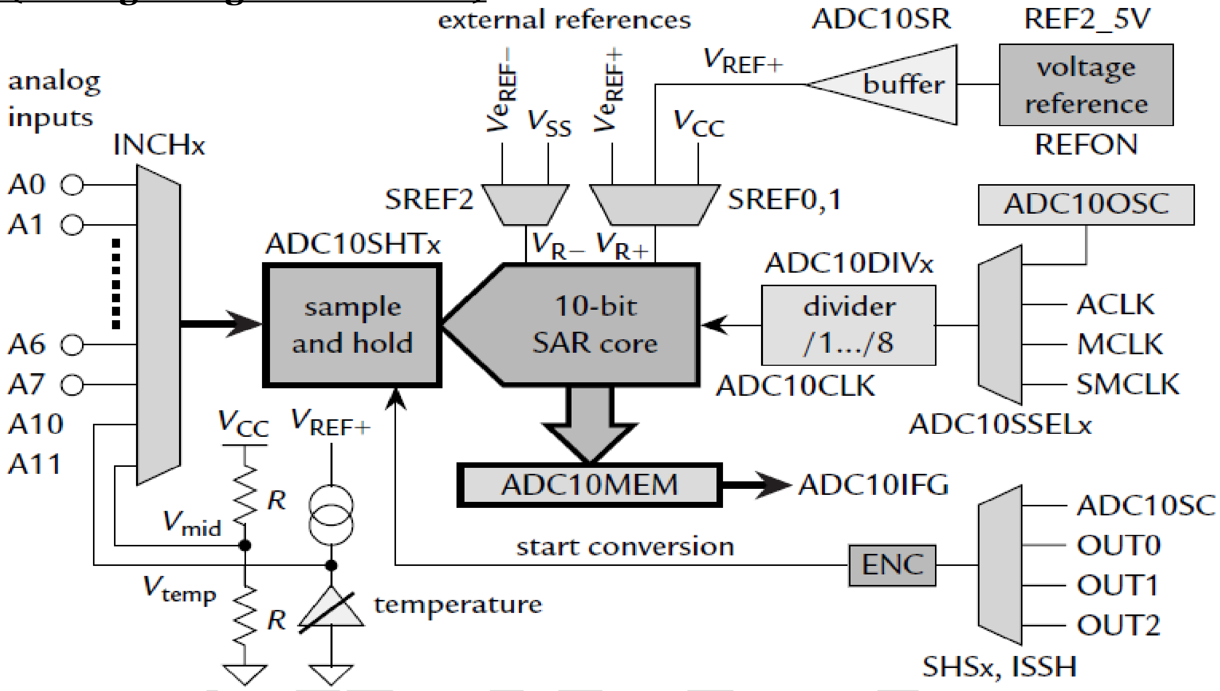


Timer modes:

MCx	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TACCR0.
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.



ADC (Analog to Digital Converter)



Core

- At the heart of the ADC10 is a 10-bit, switched-capacitor, SAR core.
- The ADC10 ON bit enables the core and a flag ADC10 BUSY is set while sampling and conversion is in progress.
- The result is written to ADC10 MEM in a choice of two formats, selected with the ADC10 DF bit.

Clock:

- It must lie within the range 0.45–6.3 MHz
- This can be taken from MCLK, SMCLK, ACLK, or the module’s internal oscillator ADC10OSC, selected with the ADC10SSELx bits.
- The internal oscillator runs nominally at 5 MHz but the specification gives a spread of to 6.3 MHz.
- The output of the divider is labeled ADC10CLK and feeds both the SAR core and sample-and-hold blocks.

Sample & Hold Unit:

This is shown separately in the block diagram but is presumably integrated into the switched-capacitor network.

- The time is chosen with the ADC10SHTx bits, which allow 4, 8, 16, or 64 cycles of ADC10CLK**.**

Input Selection

A multiplexer selects the input from eight external pins A0–A7 (more in larger MSP430s) and four internal connections.
Two of the internal connections are for optional, external reference voltages.
The other two internal connections are A10 to a temperature sensor and A11 to $V_{\mathrm{mid}} = ½ (V_{\mathrm{CC}} + V_{\mathrm{SS}})$, which is provided to monitor the supply voltage.

Conversion Trigger

- A conversion can be triggered in two ways provided that the ENC bit is set. The first is by setting the ADC10SC bit from software.
- Conversion is normally triggered by a rising edge on the input but this can be inverted with the ISSH bit.

Voltage References and Buffer

The ADC10 is slightly unusual because it uses references for both ends of the range; the lower limit need not be ground (V_{SS}).
There are two choices for the lower reference, selected with the SREF2 bit:

- The default is analog ground, AV_{SS}.
- An external reference $V_{\mathrm{eREF-}}$ can be applied directly to the SAR core without buffering.

There is more choice for the upper reference, selected with SREF0 and SREF1:

- The default is the analog supply voltage, AV_{CC}.
- An internal reference voltage $V_{\mathrm{REF+}}$ is enabled with REFON.
- An external reference $V_{\mathrm{eREF+}}$ can be applied directly to the SAR core.
- The external reference can also be buffered.
- No external capacitors are needed for the internal voltage reference, whether it is used internally or externally.

Interrupts

The interrupt flag ADC10IFG is raised when the result is written to ADC10MEM except when the DTC is used, when it is set after a block has been completed

Basic operation of the ADC 10:

1. Configure the ADC10, including the ADC10ON bit to enable the module. The ENC bit must be clear during this operation because most bits in ADC10CTL0 and ADC10CTL1 can be changed only when ENC = 0.
2. Set the ENC bit to enable a conversion. This cannot be done while the module is being configured in the previous step.
3. Trigger the conversion, either by setting the ADC10SC bit or by an edge from Timer_A.
4. The last two steps must be repeated for each conversion, which requires clearing and

setting the ENC bit again.

5. The input to the ADC10 is selected with the INCHx bits in ADC10CTL1.

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.

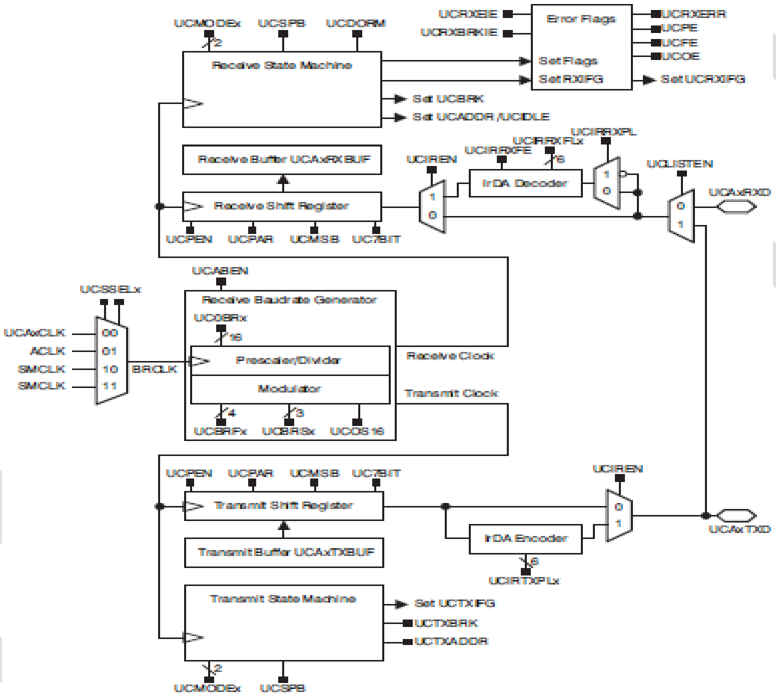


Figure 36-1. USCI_Ax Block Diagram – UART Mode (UCSYNC = 0)

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.

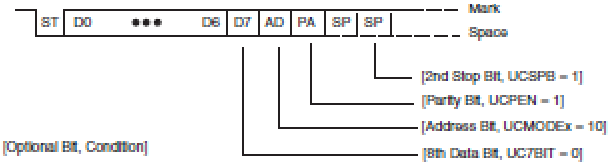

ST
D0
D6
D7
AD
PA
SP
SP
Mark
Space
[2nd Stop Bit, UCSPB = 1]
[Parity Bit, UCPEN = 1]
[Address Bit, UCMODEx = 10]
[8th Data Bit, UC7BIT = 0]
[Optional Bit, Condition]

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.

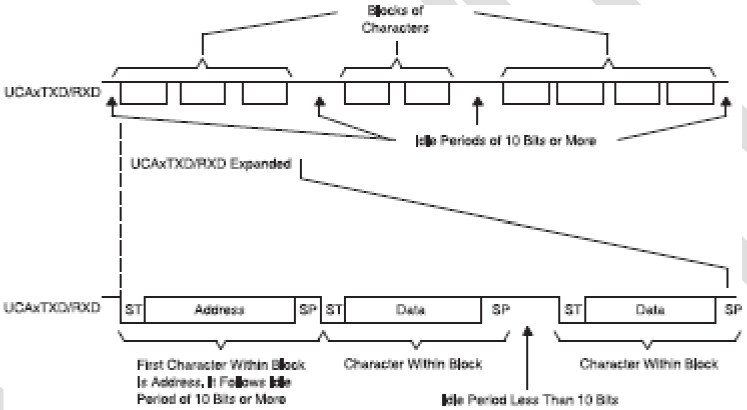

Blocks of Characters
UCAxTXD/RXD
Idle Periods of 10 Bits or More
UCAxTXD/RXD Expanded
UCAxTXD/RXD
ST
Address
SP
ST
Data
SP
ST
Data
SP
First Character Within Block Is Address. It Follows Idle Period of 10 Bits or More
Character Within Block
Character Within Block
Idle Period Less Than 10 Bits

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.

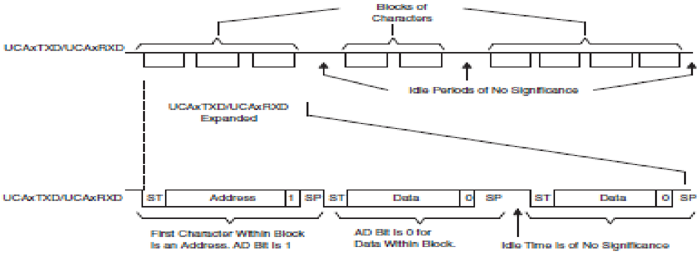

Blocks of Characters
UCAxTXD/UCAxRXD
Idle Periods of No Significance
UCAxTXD/UCAxRXD Expanded
UCAxTXD/UCAxRXD
ST
Address
1
SP
ST
Data
0
SP
ST
Data
0
SP
First Character Within Block Is an Address. AD Bit Is 1
AD Bit Is 0 for Data Within Block.
Idle Time Is of No Significance

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.

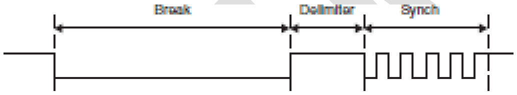


Figure 36-5. Auto Baud-Rate Detection – Break/Synch Sequence

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.

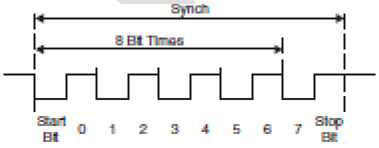


Figure 36-6. Auto Baud-Rate Detection – Synch Field

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.

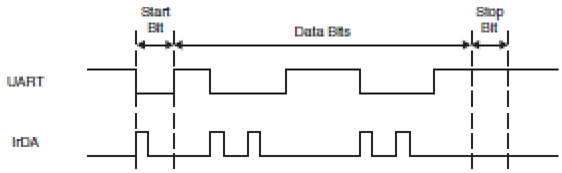


Figure 36-7. UART vs IrDA Data Format

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:

$UCIRTXPLx = tPULSE \times 2 \times fBRCLK – 1$ 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:

$UCIRRXFLx = (tPULSE − tWAKE) \times 2 \times fBRCLK – 4$

Where:

tPULSE = Minimum receive pulse width

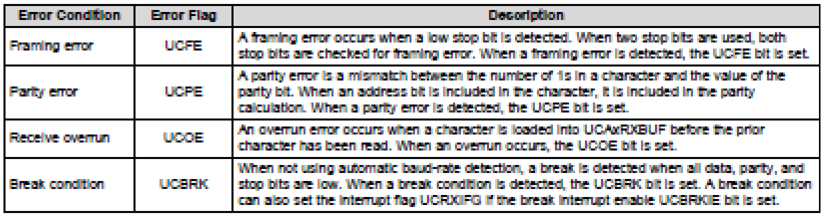
tWAKE = Wake time from any low-power mode. Zero when
the device is in active mode.

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).

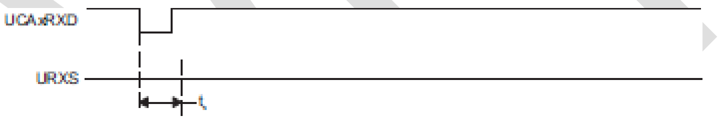


Figure 36-8. Glitch Suppression, USCI Receive Not Started

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.

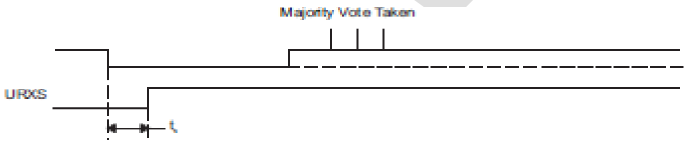


Figure 36-9. Glitch Suppression, USCI Activated

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.

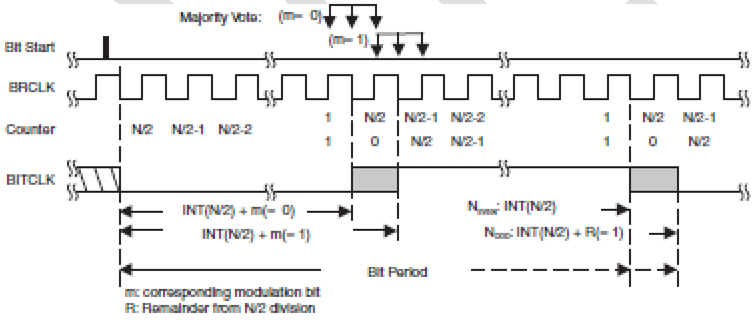
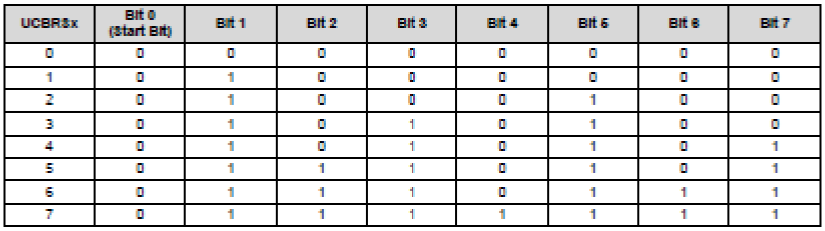


Figure 36-10. BITCLK Baud-Rate Timing With UCOS16 = 0

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



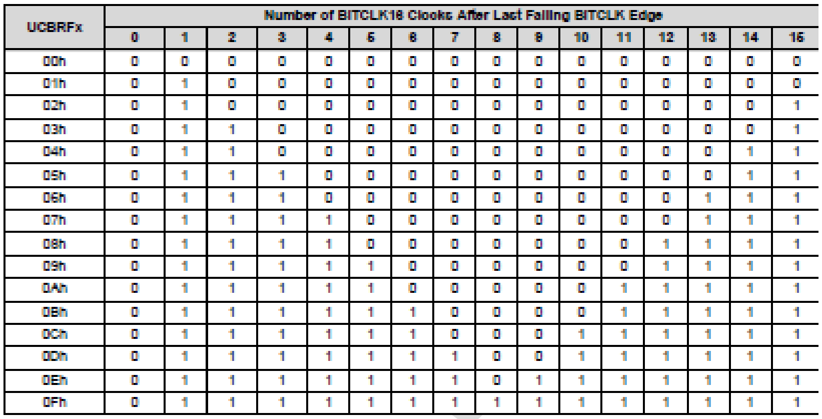
UCBRSx	Bit 0 (Start Bit)	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	1	0	0	0	1	0	0
3	0	1	0	1	0	1	0	0
4	0	1	0	1	0	1	0	1
5	0	1	1	1	0	1	0	1
6	0	1	1	1	0	1	1	1
7	0	1	1	1	1	1	1	1

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 COS16.

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.

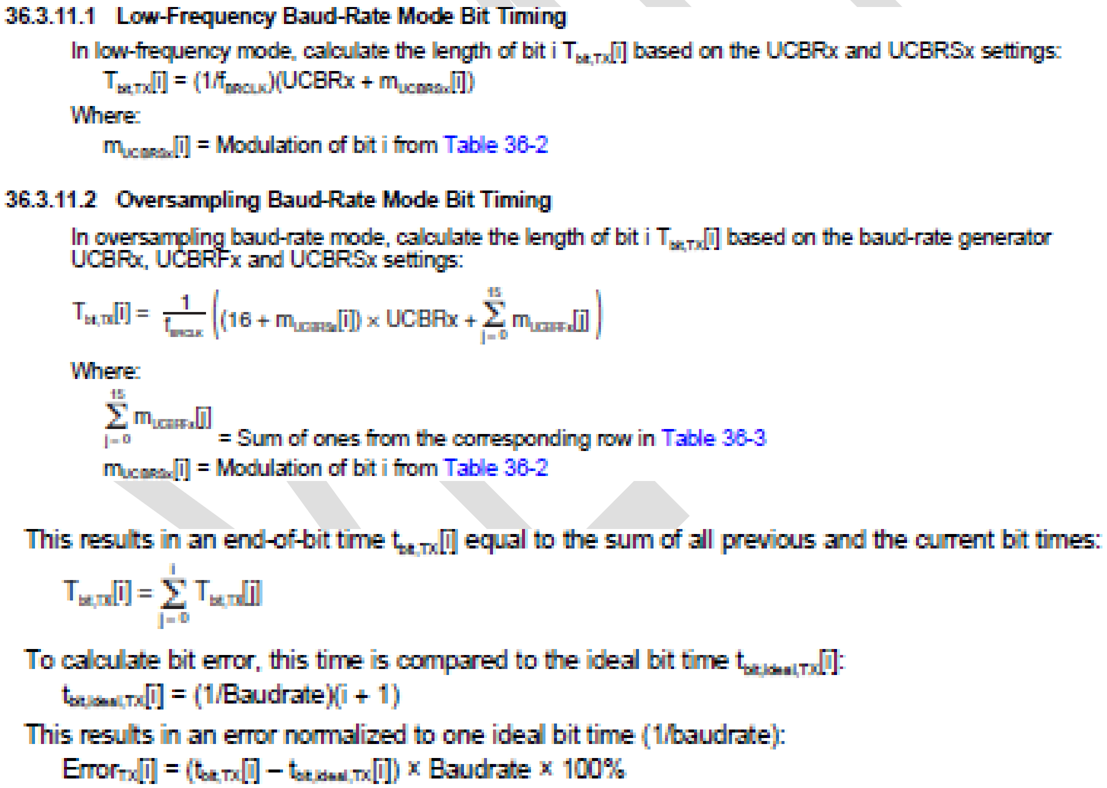
Oversampling Baud-Rate Mode Setting

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.

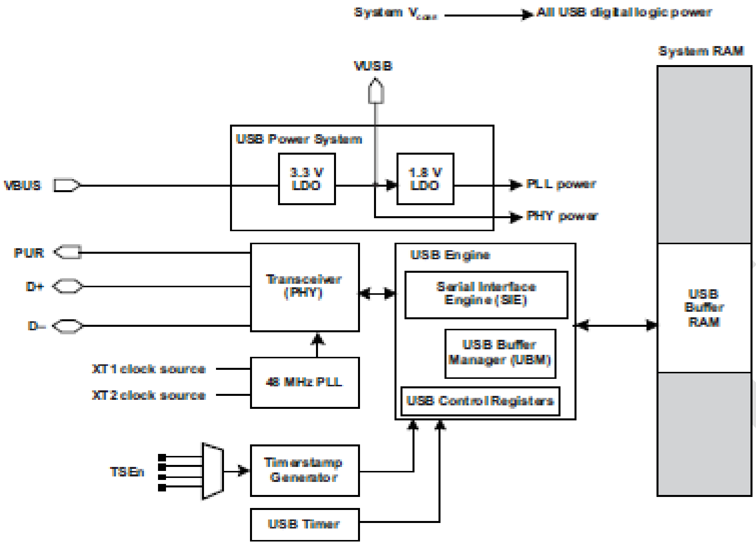


Figure 42-1. USB Block Diagram

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

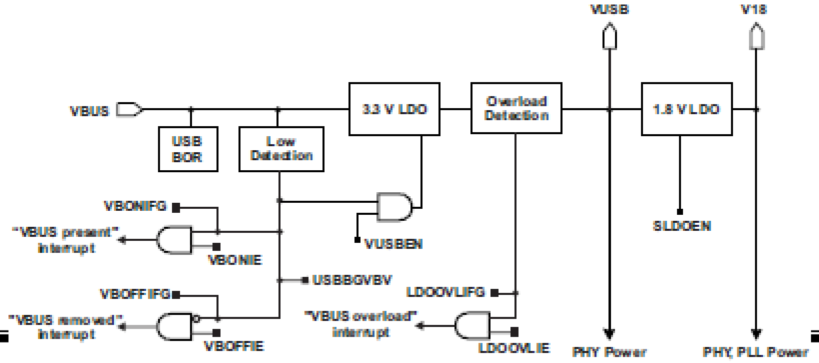


Figure 42-2. USB Power System

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

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.

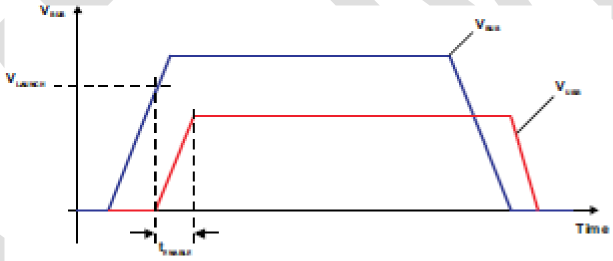


Figure 42-3. USB Power Up and Down Profile

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).

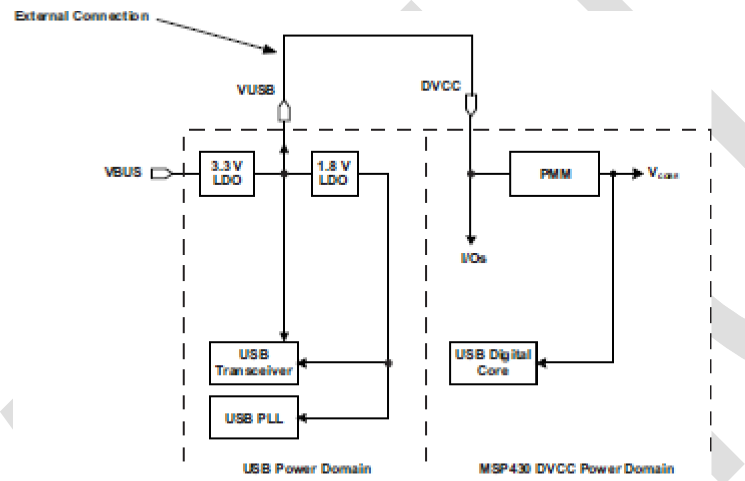


Figure 42-4. Powering Entire MSP430 From VBUS

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 over current condition on the 3.3-V LDO. When an over current 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

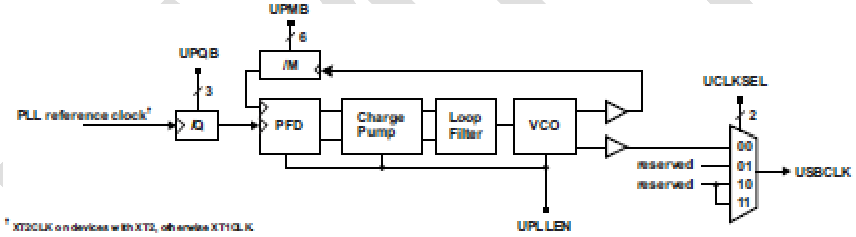


Figure 42-5. USB-PLL Analog Block Diagram

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.

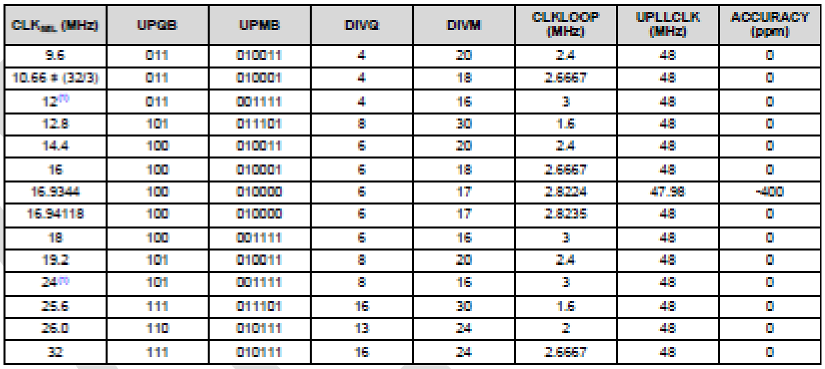
Modifying the Divider Values

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_{SEL} (MHz)	UPGB	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[1]	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[1]	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.

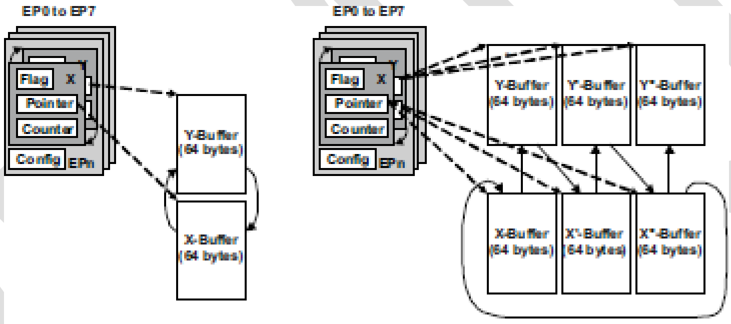


Figure 42-6. Data Buffers and Descriptors

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

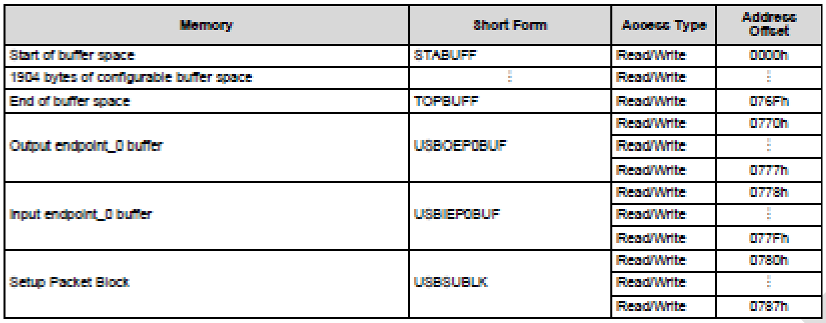
The USB buffer memory contains the data buffers for all endpoints and for SETUP packets. In that the buffers for endpoints 1 to 7 are flexible, there are USB buffer configuration registers that define them, and these too are in the USB buffer memory. (Endpoint 0 is defined with a set of registers in 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
- S

PI 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

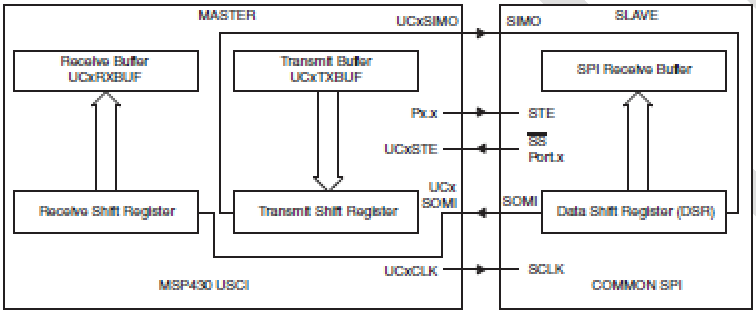


Figure 37-2. USCI Master and External Slave

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

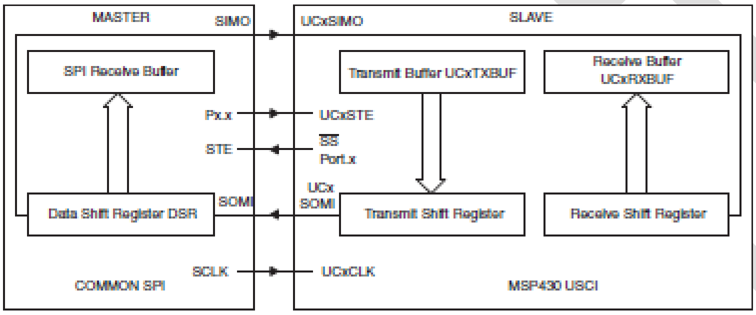


Figure 37-3. USCI Slave and External Master

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.

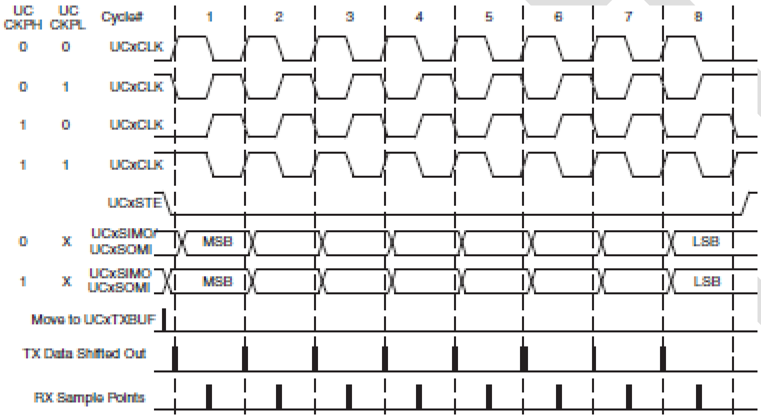


Figure 37-4. USCI SPI Timing With UCMSB = 1

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
- S
PI 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

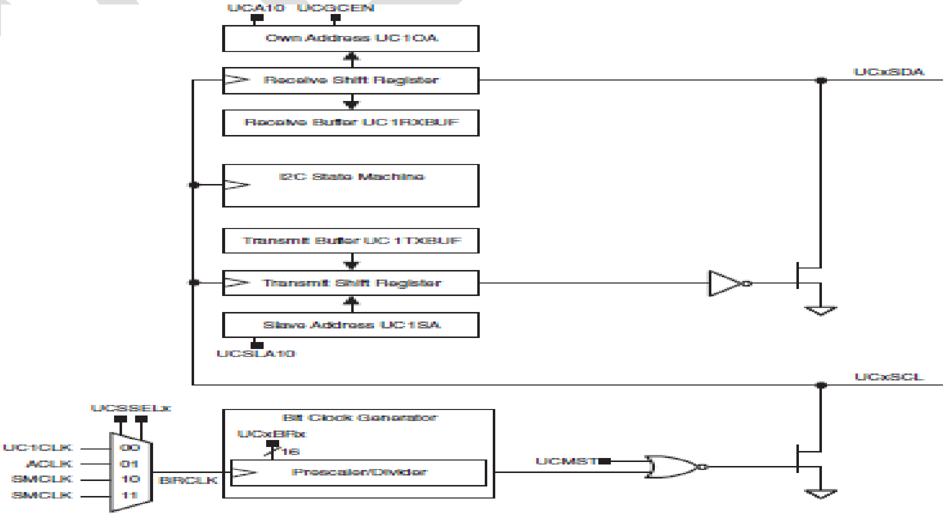


Figure 38-1. USCI Block Diagram – I²C Mode

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.

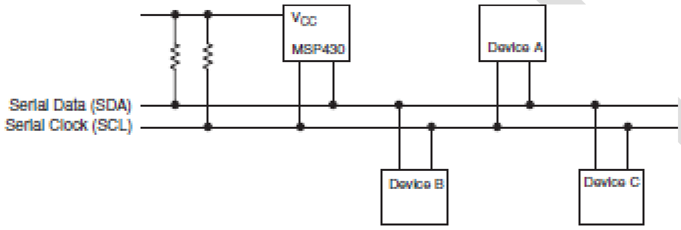


Figure 38-2. I²C Bus Connection Diagram

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.

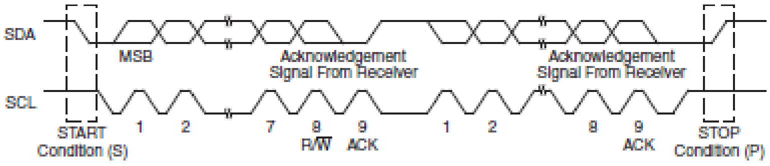

SDA
MSB
Acknowledgement Signal From Receiver
Acknowledgement Signal From Receiver
SCL
START Condition (S)
1
2
7
8
R/W
9
ACK
1
2
8
9
ACK
STOP Condition (P)

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.

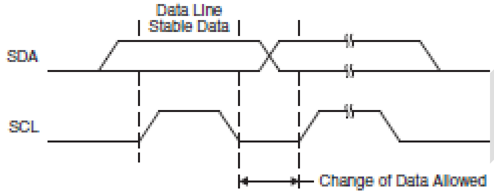


Figure 38-4. Bit Transfer on I²C Bus

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.

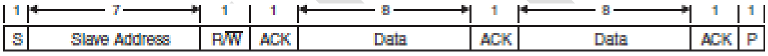


Figure 38-5. I²C Module 7-Bit Addressing Format

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.

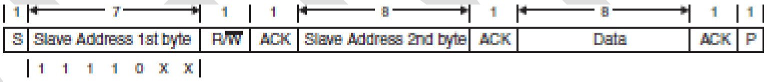


Figure 38-6. I²C Module 10-Bit Addressing Format

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

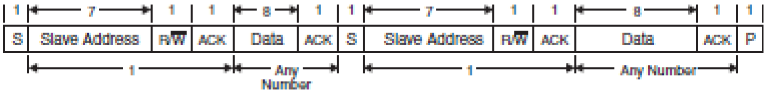


Figure 38-7. I²C Module Addressing Format With Repeated START Condition

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.

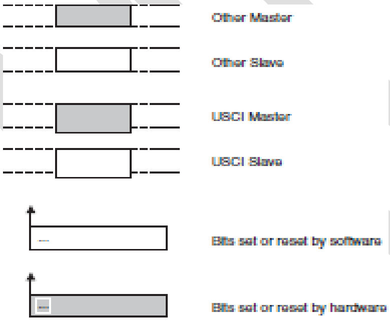


Figure 38-8. I^2C Time-Line Legend

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

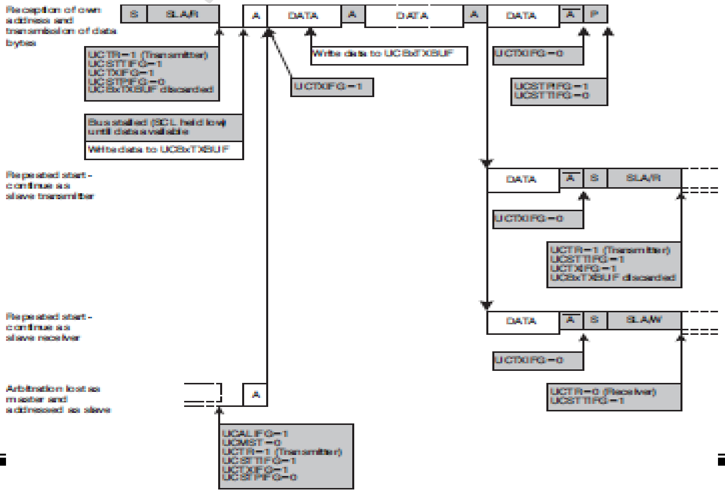


Figure 38-9. I^2C Slave Transmitter Mode

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.

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,

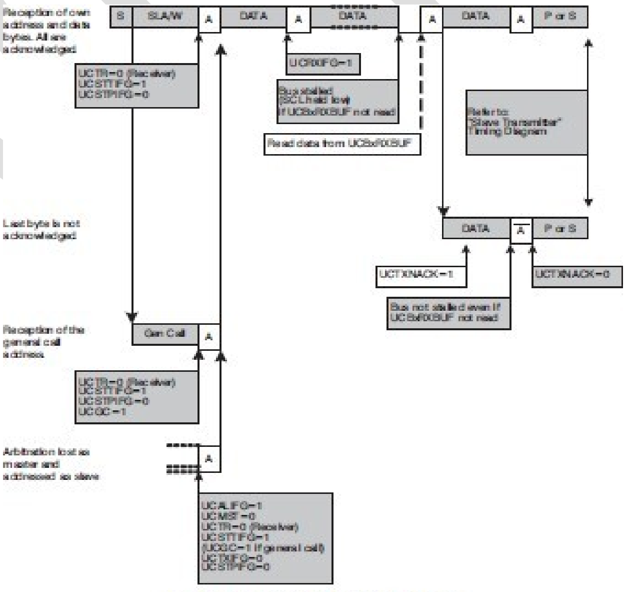

Reception of own address and data bytes. All are acknowledged
S
SLA/W
A
DATA
A
DATA
A
DATA
A
P or S
UCTR=0 (Receiver)
UCSTTIFG=1
UCSTPIFG=0
UCRXIFG=1
Bus stalled (SCL held low) if UCBxRXBUF not read
Read data from UCBxRXBUF
Refer to: "Slave Transmitter" Timing Diagram
Last byte is not acknowledged
DATA
A
P or S
UCTXNACK=1
UCTXNACK=0
Bus not stalled even if UCBxRXBUF not read
Reception of the general call address
Gen Call
A
UCTR=0 (Receiver)
UCSTTIFG=1
UCSTPIFG=0
UCGC=1
Arbitration lost as master and addressed as slave
A
UCALIFG=1
UCMST=0
UCTR=0 (Receiver)
UCSTTIFG=1
(UCGC=1 if general call)
UCTXIFG=0
UCSTPIFG=0

Figure 38-10. I^2C Slave Receiver Mode the USCI I2C state machine returns to its address reception state.

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.

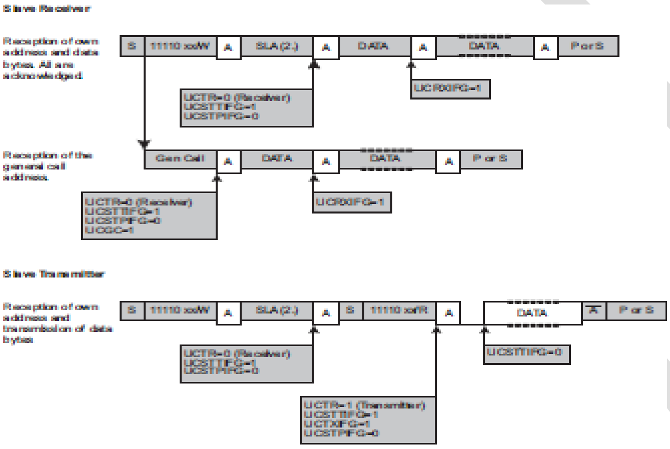


Figure 38-11. I²C Slave 10-Bit Addressing Mode

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.

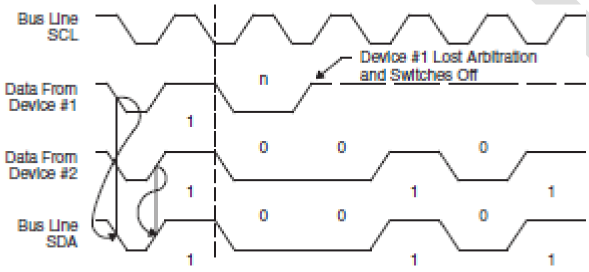


Figure 38-15. Arbitration Procedure Between Two Master Transmitters

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.

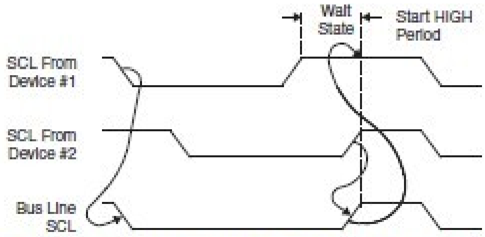


Figure 38-16. Synchronization of Two I²C Clock Generators During Arbitration

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.

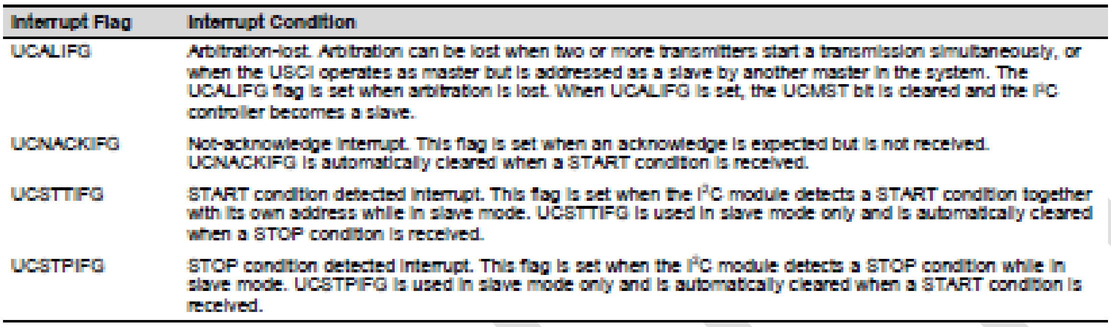
I2C Receive Interrupt Operation

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.

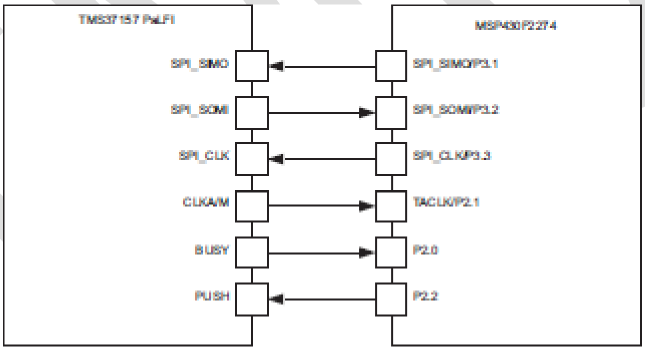


Figure 1. Block Diagram of Interface Between MSP430F2274 and TMS37157

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.

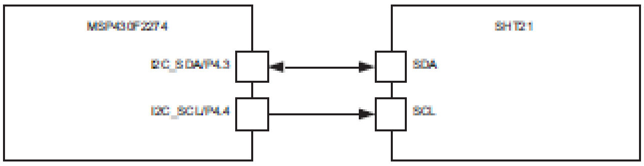


Figure 2. Block Diagram of Interface Between MSP430F2274 and SHT21

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.

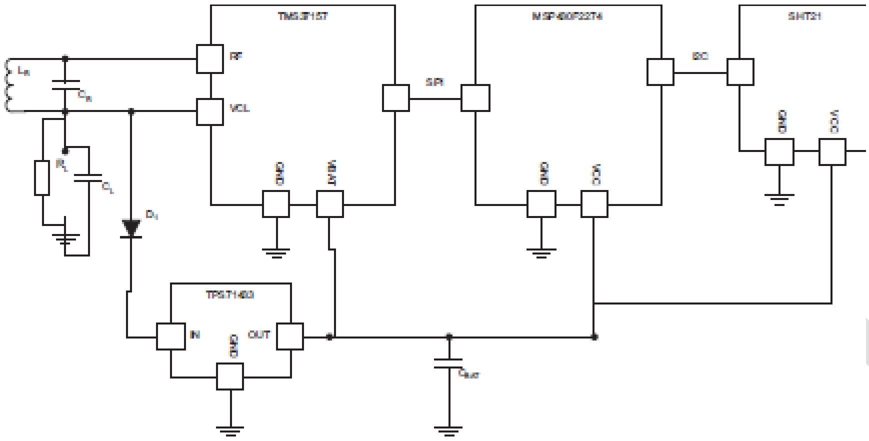


Figure 3. Principle Schematic of the Wireless Sensor

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

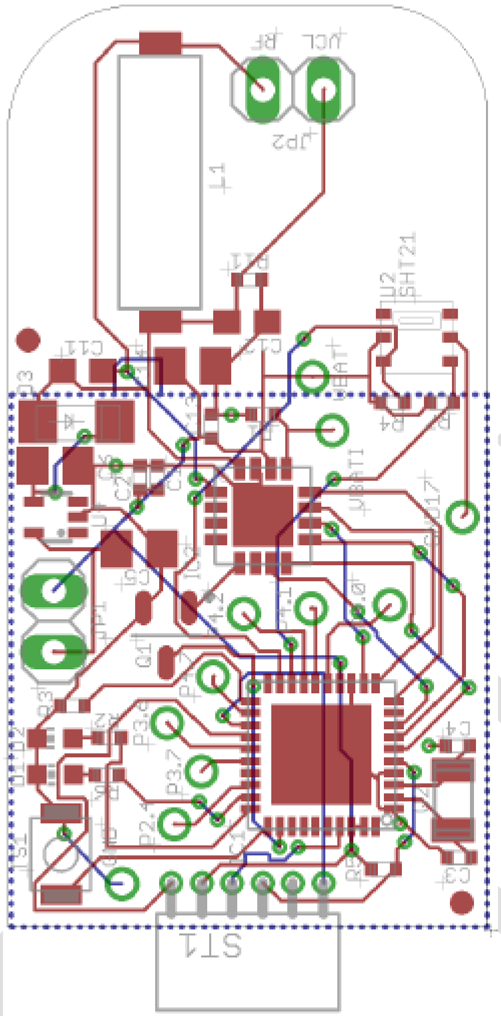


Figure 5. Eagle Layout of the Wireless Sensor

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.