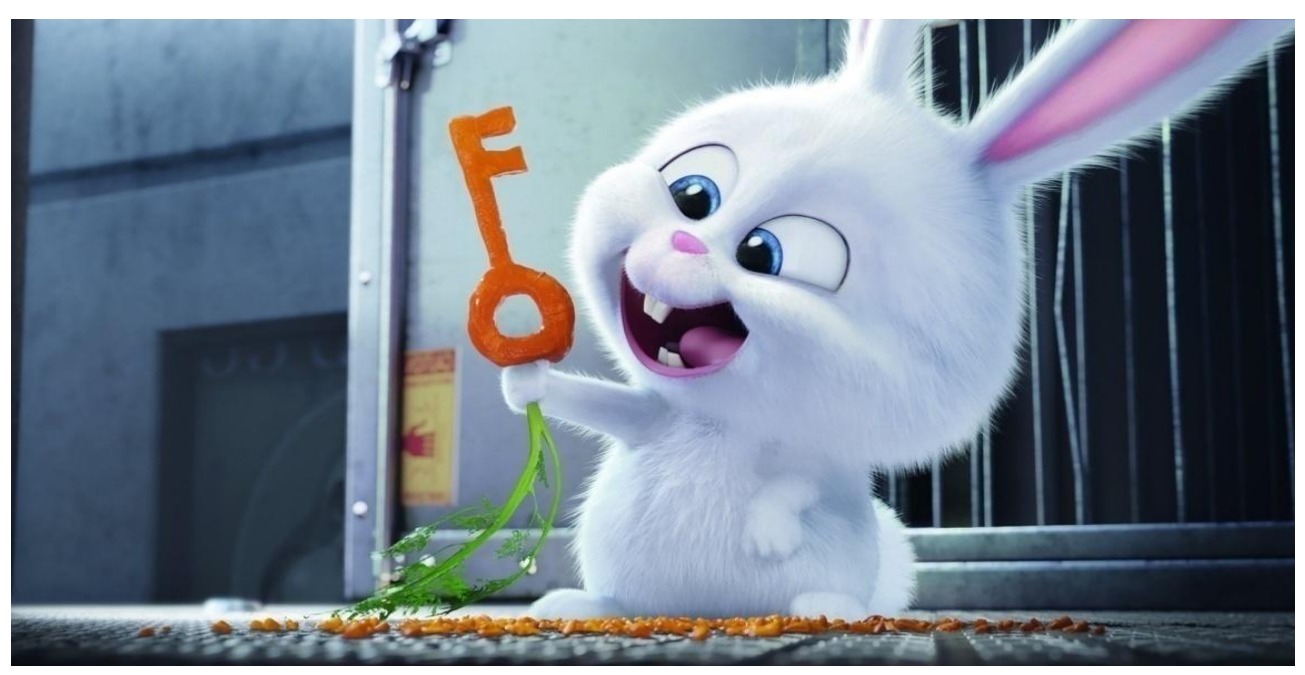
KUPPAM ENGINEERING COLLEGE



ONLINE COURSE : COVID-19 ZOOM VIDEO CLASSES

SUBJECT : MICROPROCESSOR & MICROCONTROLLER

BY

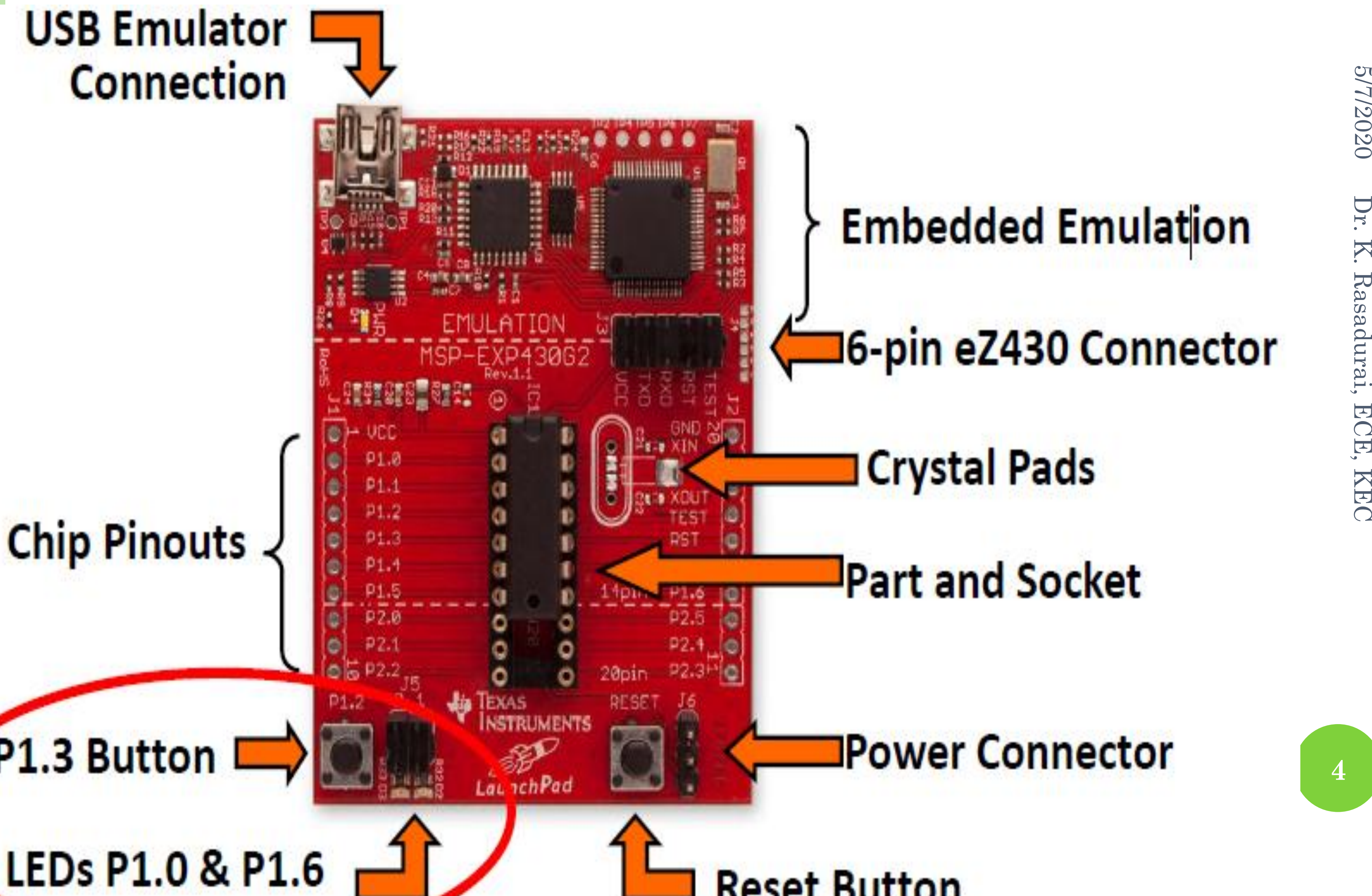
Dr. K. RASADURAI
Department of Electronics and Communications Engineering
KUPPAM ENGINEERING COLLEGE,
Kuppam – 517425, Chittoor Dist., Andhra Pradesh

CONTENT

1. **I/O ports pull up/down resistors concepts**
2. **Interrupts and interrupt programming**
3. **Watchdog timer**
4. **System clocks**
5. **Low Power aspects of MSP430:**
   a. **low power modes,**
   b. **Active vs Standby current consumption**
6. **FRAM vs Flash for low power & reliability**
7. **Timer & Real Time Clock (RTC)**
8. **PWM control, timing generation and measurements**
9. **Analog interfacing and data acquisition:**
   a. **ADC and Comparator in MSP430,**
   b. **Data transfer using DMA.**

GPIO – REGISTERS OR DIGITAL I/O

MSP 430 - General Purpose IO
Launch Pad Development Board



1. General Purpose INPUT OUTPUT (GPIO)

- one of the simplest integrated peripherals of the MSP430.

- General Purpose 8-bit Input Output (GPIO).

- The Input / Output (I/O) ports can be configured as interruptible or non-interruptible.

- Additionally, the port pins can be individually configured for general-purpose use, or

- as special function I/Os, such as USARTs, comparator signals and ADCs.

General Purpose IO

MSP430 device, there can be up to ten 8-bit digital Input/Output (I/O) ports, named P1 to P10.

Typically, the MSP430 I/O ports P1 and P2 have interrupt capability.

Each interrupt on these I/O lines can be individually configured to provide an interrupt on a rising edge or falling edge of an input signal.

All I/O lines with interrupt capacity use a single interrupt vector.

The available digital I/O pins are:

-**eZ430-F2013 MSP430 USB Stick: 10 pins - Port P1 (8 bits) and Port P2 (2 bits)**

- **eZ430-RF2500 MSP430 USB Stick: 32 pins - Ports P1 to P4 (8 bits)**

- **MSP430FG4618/F2013 Experimenter board: 80 pins – Ports P1 to P10 (8 bits). Ports P7/P8 and P9/P10 can be accessed as 16- bit values (words) as ports PA and PB respectively.**

MSP 430 GPIO

Each of these I/O ports has the following capacity:

-Independently programmable

-Combined input, output, and interrupt conditions;

- Edge-selectable interrupt inputs for all the 8 bits of ports P1 and P2

- Read/write access to port-control registers supported by all two or one-address instructions

- Each I/O has an individually programmable pullup / pulldown resistor (2xx family only).

Individually configured as special functions I/O, for example:

- USART – Universal Synchronous/Asynchronous Receive/Transmit

- Comparative signals

- Analogue-to-Digital converter

- Amongst others…

GPIO – REGISTERS

Independent of the I/O port type

- (non-interruptible: P3 and others) or
- interruptible (P1 and P2),

1. Direction Registers (PxDIR)
2. Input Registers (PxIN)
3. Output Registers (PxOUT)
4. Pull-up/Pull-down Resistor Enable Registers (PxREN)
5. Output Drive Strength Registers (PxDS)
6. Function Select Registers: (PxSEL) and (PxSEL2)
7. Interruptible ports (P1 and P2)
   a. Interrupt Edge Select Registers (PxIES)
   b. Interrupt Flag Registers (PxIFG)

GPIO – Registers …

1. Direction Registers (PxDIR)

- Read/write 8-bit registers;
- Selects the direction of the corresponding I/O pin, regardless of the selected function of the pin (general purpose I/O or as a special function I/O);
- Other module functions must be set as required by the other modules;

PxDIR configuration:

- **Bit = 1: The port pin is set up as an output;**
- **Bit = 0: the port pin is set up as an input.**

2. Input Registers (PxIN)

- Each bit of these read-only registers reflects the input signal at the corresponding I/O pin (pin configured as general purpose I/O);

PxIN configuration:

- **Bit = 1: The input is high;**
- **Bit = 0: The input is low;**
- Tip: Avoid writing to these read-only registers because it will result in increased current consumption.

GPIO – REGISTERS….

3. Output Registers (PxOUT)

- The output registers are read-write. Each bit of these registers reflects the value written to the corresponding output pin. PxOUT configuration:
  - **Bit = 1: The output is high**
  - **Bit = 0: The output is low**

- The 2xx family provides the additional feature that each I/O has a pullup/pulldown resistor that can be individually programmed. If the pin's ullup/pulldown resistor is enabled, the corresponding bit in the PxOUT register selects the pull-up or pull-down:
  - **Bit = 1: The pin is pulled up**
  - **Bit = 0: The pin is pulled down**

GPIO – REGISTERS….

4. **Pull-up/Pull-down Resistor Enable Registers (PxREN)**

- Applies to the 2xx family only.

- Each bit of this register enables or disables the pullup / pulldown resistor of the corresponding I/O pin.

- PxREN configuration:

 Bit = 1: Pullup/pulldown resistor enabled

 Bit = 0: Pullup/pulldown resistor disabled

5. Output Drive Strength Registers (PxDS)

- Each bit in each PxDS register selects either full drive or reduced drive strength. Default is reduced drive strength.

PxDS configuration:

- **Bit = 0: Reduced drive strength**
- **Bit = 1: Full drive strength**

GPIO – REGISTERS….

6. **Function Select Registers: (PxSEL) and (PxSEL2)**

- Some port pins are multiplexed with other peripheral module Functions.

- The bits: (PxSEL) and (PxSEL2 – 2xx family and some devices of the 47x(x) family), are used to select the pin function: I/O general-purpose port or peripheral module function.

PxSEL configuration:

- **Bit = 0: I/O function is selected for the pin;**
- **Bit = 1: Peripheral module function is selected for the pin.**

Function Select Registers: (PxSEL) and (PxSEL2)…………..

The 2xx family devices provide the PxSEL2 bit to configure additional features of the device. The PxSEL and PxSEL2 combination provides the following configuration of the 2xx devices:

- **Bit = 0: I/O function is selected for the pin;**
- **Bit = 1: Peripheral module function is selected for the pin.**

PxSEL	PxSEL2	Pin Function
0	0	Selects general-purpose I/O function
0	1	Selects the primary peripheral module function
1	0	Reserved (See device-specific data sheet)
1	1	Selects the secondary peripheral module function

GPIO – REGISTERS….

7. Interruptible ports (P1 and P2)

Each pin of ports P1 and P2 is able to generate an interrupt request (pin is interruptible) and is configured using the PxIFG, PxIE, and PxIES registers. The port makes use of all the same configuration registers as non-interruptible ports (as described above), but with three additional registers:

Interrupt Enable (PxIE)

- Read-write register to enable interrupts on individual pins;
- PxIE configuration:
- Bit = 1: The interrupt is enabled;
- Bit = 0: The interrupt is disabled.
- Each PxIE bit enables the interrupt request associated with the corresponding PxIFG interrupt flag;
- Writing to PxOUT and/or PxDIR can result in setting PxIFG.

GPIO – REGISTERS….

7. Interruptible ports (P1 and P2)

a. Interrupt Edge Select Registers (PxIES)

- This read-write register selects the transition on which an interrupt occurs for the corresponding I/O pin
- PxIES configuration:
  * Bit = 1: Interrupt flag is set on a high-to-low transition
  * Bit = 0: Interrupt flag is set on a low-to-high transition

b. Interrupt Flag Registers (PxIFG)

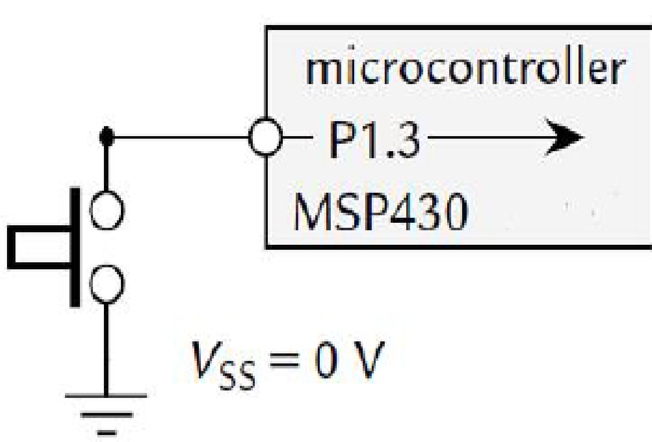
- The bit of this read-write register is set automatically when the programmed signal transition (edge) occurs on the corresponding I/O pin, provided that the corresponding PxIE bit and the GIE bit are set
- Each PxIFG flag can be set by software, enabling an interrupt generated by software
- Each PxIFG flag must be reset with software
- PxIFG configuration:
  * Bit = 0: No interrupt is pending
  * Bit = 1: An interrupt is pending

PROBLEM WITH INPUT USING A BUTTON

- When the button is pressed down (closed), MSP430 will detect a 0 from the pin.
- When the button is up (open), what will MSP430 detect?

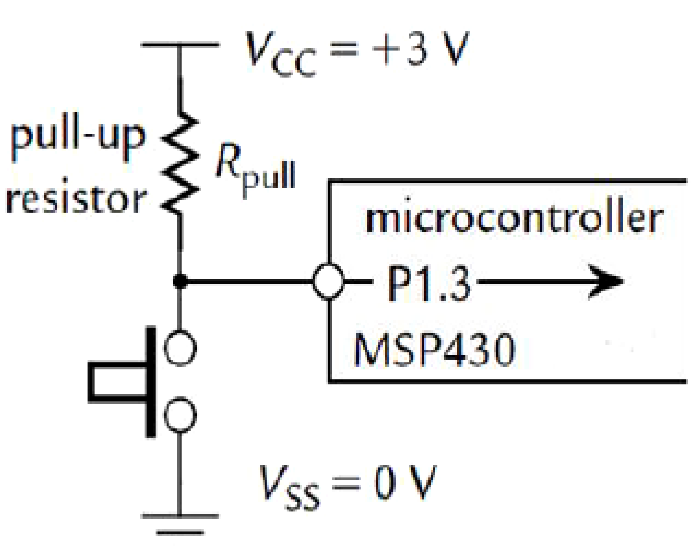
Ans.: random value

→ Floating



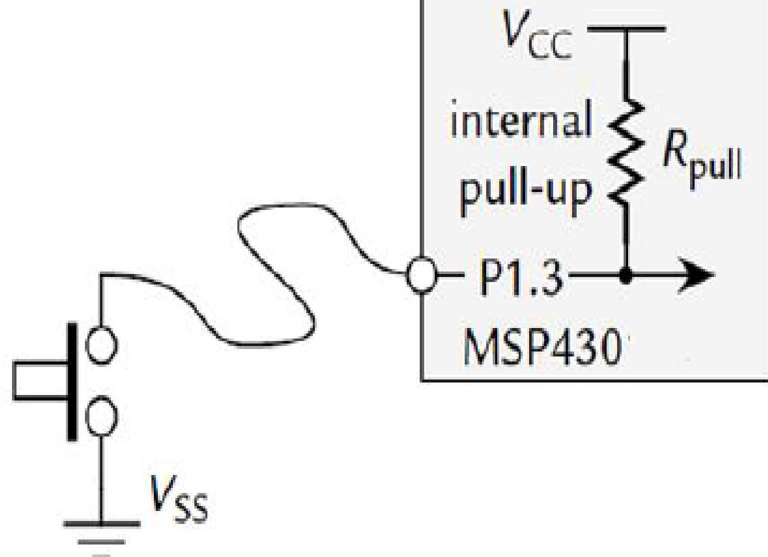
Typical Way of Connecting a Button

- The *pull-up resistor* R_{pull} holds input at logic 1 (voltage V_{CC}) while button is up and logic 0 or V_{SS} when button is down → *active low*
  - A wasted current flows through R_{pull} to ground when button is down
  - This is reduced by making R_{pull} large enough, typically *33 kΩ*



Typical Way of Connecting a Button

- MSP430 offers internal pull-up/down
- PxREN register selects whether this resistor is used (1 to enable, 0 to disable)
  - When enabled, the corresponding bit of PxOUT register selects whether the resistor pulls the input up to V_{CC} (1) or down to V_{SS} (0)



Sample Code for Input (MSP430g2553)

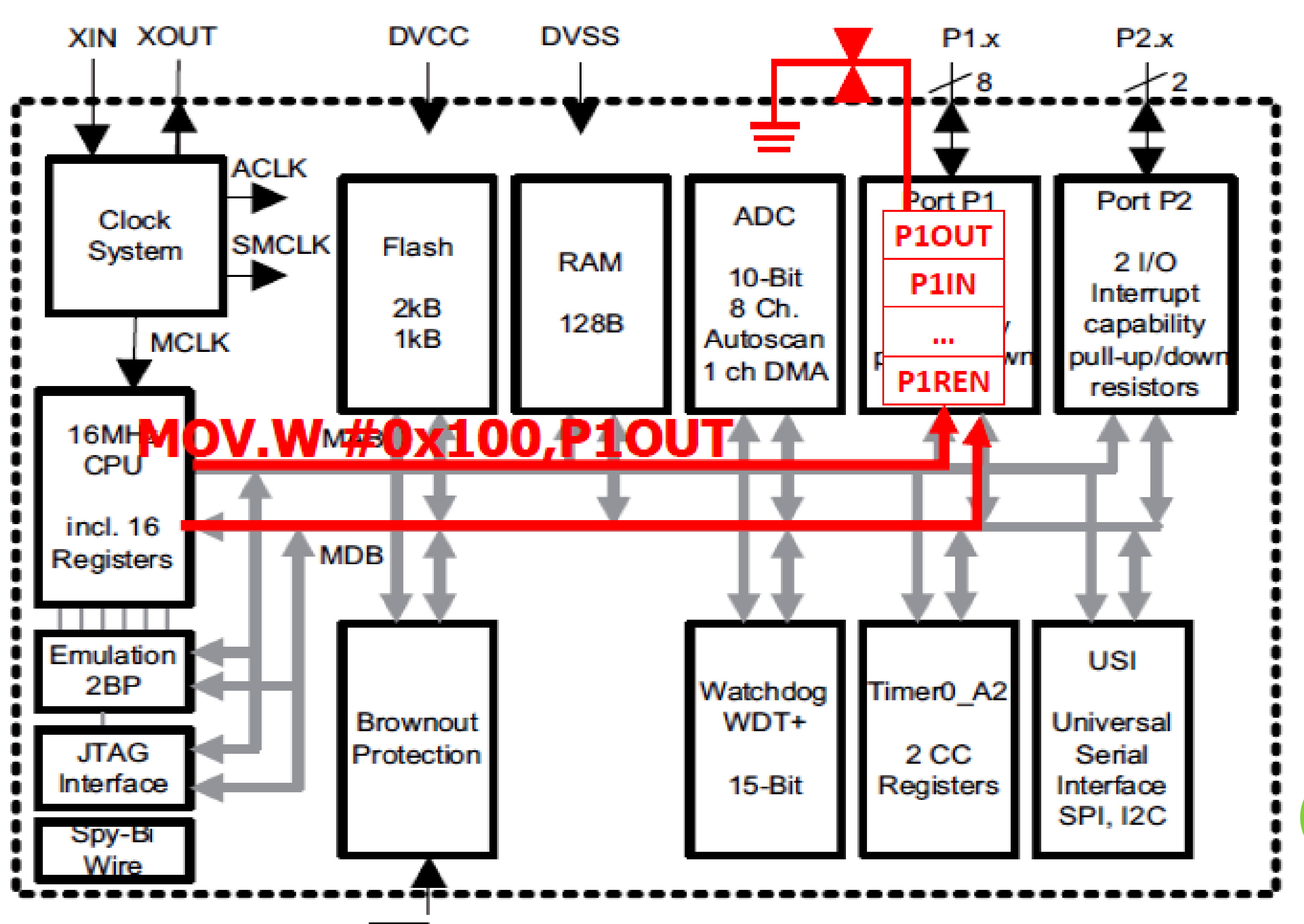
```
#include <msp430g2553.h>
#define LED1 BIT0    //P1.0 to red LED
#define B1 BIT3      //P1.3 to button

void main(void){
  WDTCTL = WDTPW + WDTHOLD; //Stop watchdog timer
  P1OUT |= LED1 + B1;
  P1DIR = LED1; //Set pin with LED1 to output
  P1REN = B1;   //Set pin to use pull-up resistor
  for(;;){   //Loop forever
    if((P1IN & B1) ==0){  //Is button down or not
      P1OUT &= ~LED1; // Turn LED1 off  }
    else{
      P1OUT |= LED1; // Turn LED1 on }
  }
}
```

Recall: Sample Code for Output

```
#include <msp430.h>
void main(void) {
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog
  P1DIR |= 0x41;  // set P1.0 & 6 to outputs
  for(;;) {
     volatile unsigned int i;
     P1OUT ^= 0x41; // Toggle P1.0 & 6
     i = 50000;      // Delay
     do (i--);
     while (i != 0);
  }
}
```

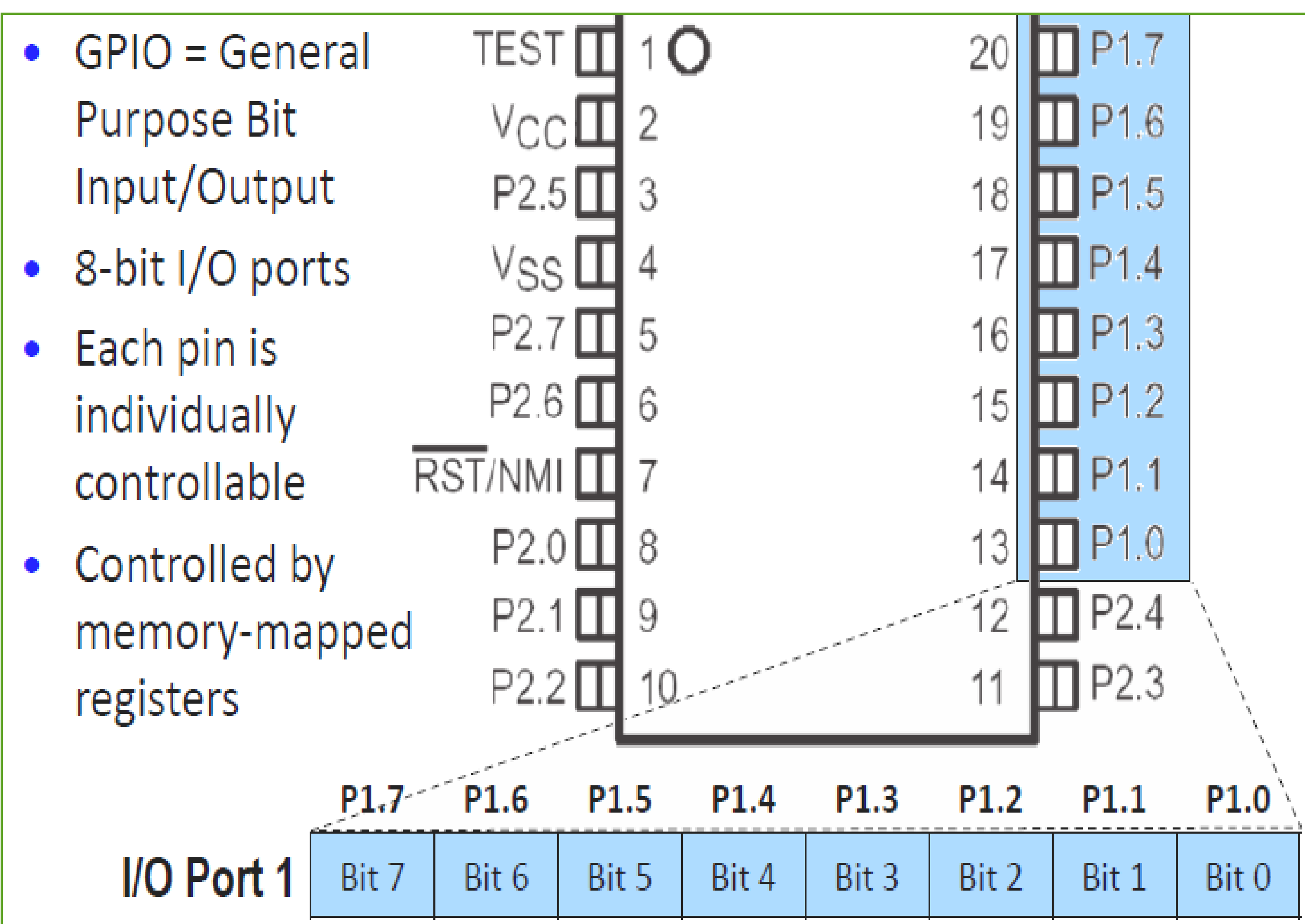
General Purpose IO
Recall: Memory-Mapped I/O



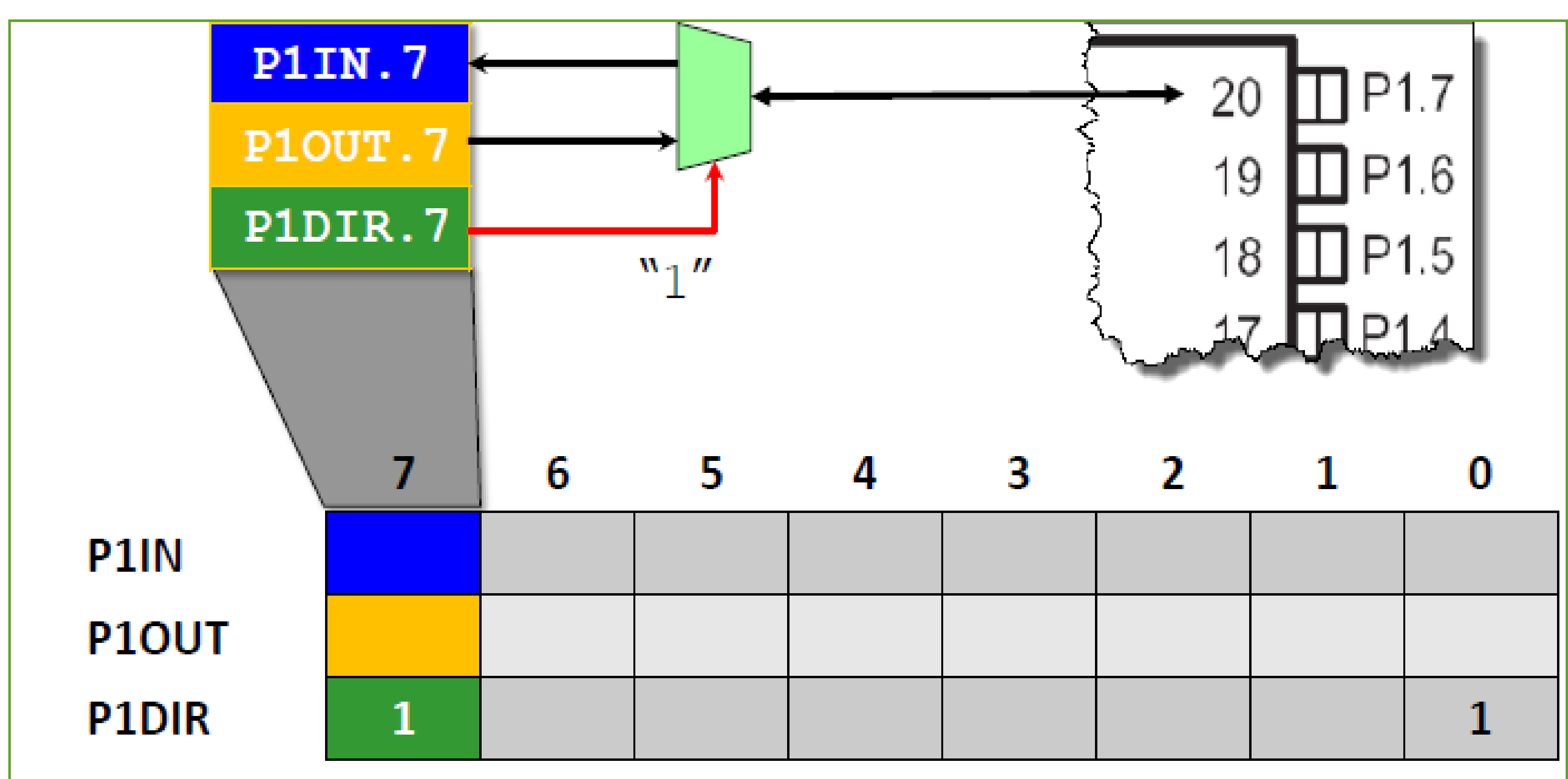
General Purpose IO
Configuring the I/O Ports

Registers (Mem Addr)	Functions	Descriptions
P1IN (0x0020)	Port 1 input	This is a read-only register that reflects the current state of the port's pins.
P1OUT (0x0021)	Port 1 output	The values written to this read/write register are driven out to corresponding pins when they are configured to output.
P1DIR (0x0022)	Port 1 data direction	Bits written as 1 (0) configure the corresponding pins for output (input).
P1SEL (0x0026)	Port 1 function select	Bits written as 1 (0) configure corresponding pins for use by the specialized peripheral (for general-purpose I/O).
P1REN (0x0027)	Port 1 resistor enable	Bits set in this register enable pull-up/down resistors on the corresponding I/O pins.

MSP430 GPIO

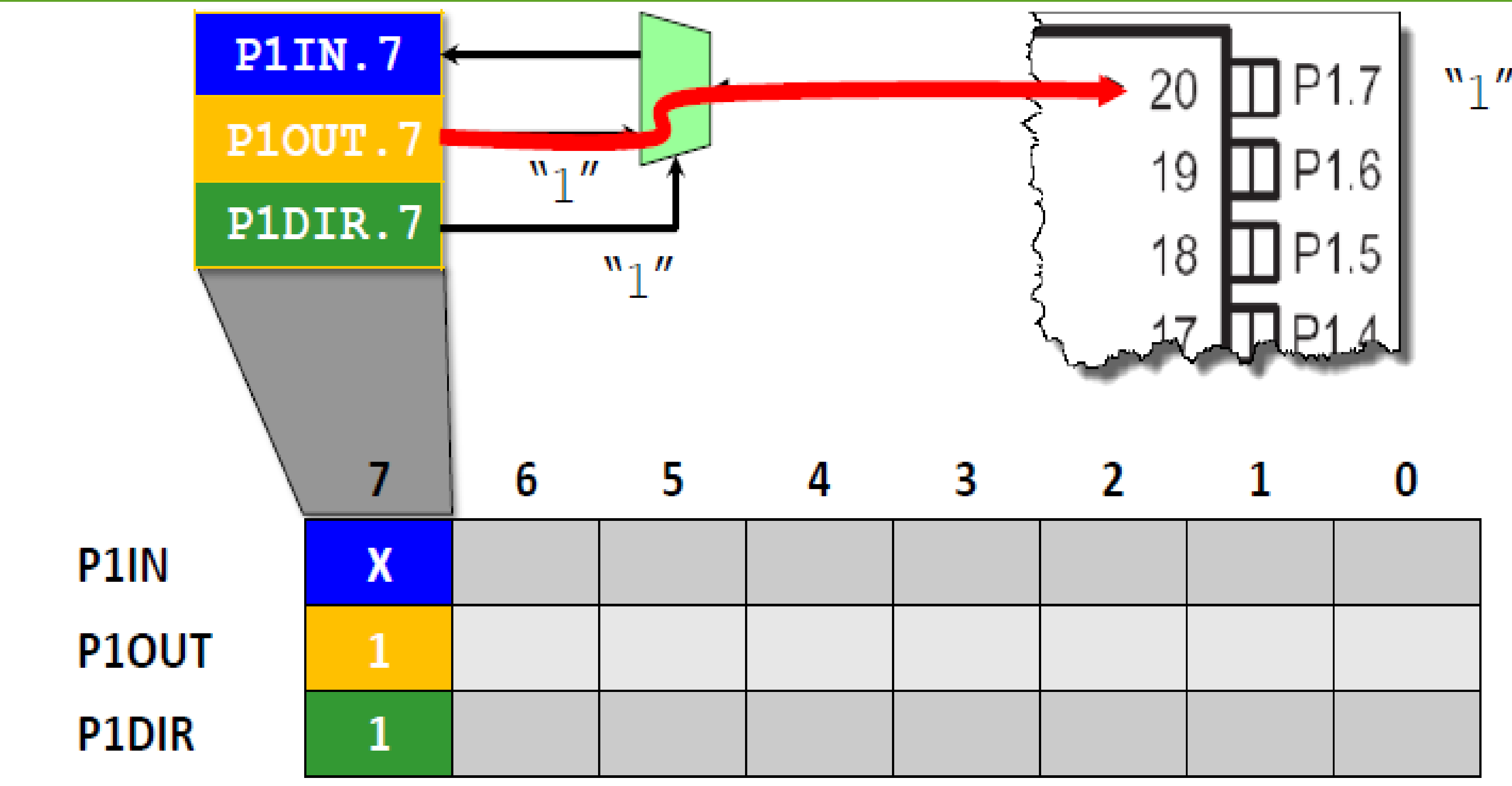


PxDIR (Pin Direction): Input or Output



- PxDIR.y: 0 = input 1 = output
- Register example: `P1DIR &= 0x81;`

MSP 430 GPIO - Output



- PxOUT.y: 0 = low 1 = high
- Register example: `P1OUT &= 0x80;`

GPIO - Sample Code for Input

```
#include <msp430g2231.h>
// Pins for LED and button on port 1
#define LED1 BIT0    //P1.0 to red LED
#define B1 BIT3      //P1.3 to button

void main(void){
  WDTCTL = WDTPW + WDTHOLD; //Stop watchdog timer
  P1OUT |= LED1;  //Preload LED1 on
  P1DIR = LED1;   //Set pin with LED1 to output
  for(;;){   //Loop forever
    if((P1IN & B1) ==0){  //Is button down or not
      P1OUT &= ~LED1; //
    else{
      P1OUT |= LED1; // T
  }
}
```

P1OUT is not initialized and must be written before configuring the pin for output.

Table 1-2. Digital I/O Registers

Offset	Acronym	Register Name	Type	Access	Reset	Section
0Eh	P1IV	Port 1 Interrupt Vector	Read only	Word	0000h	Section 1.4.1
0Eh	P1IV_L		Read only	Byte	00h	
0Fh	P1IV_H		Read only	Byte	00h	
1Eh	P2IV	Port 2 Interrupt Vector	Read only	Word	0000h	Section 1.4.2
1Eh	P2IV_L		Read only	Byte	00h	
1Fh	P2IV_H		Read only	Byte	00h	
00h	P1IN or PAIN_L	Port 1 Input	Read only	Byte		Section 1.4.9
02h	P1OUT or PAOUT_L	Port 1 Output	Read/write	Byte	undefined	Section 1.4.10
04h	P1DIR or PADIR_L	Port 1 Direction	Read/write	Byte	00h	Section 1.4.11
06h	P1REN or PAREN_L	Port 1 Resistor Enable	Read/write	Byte	00h	Section 1.4.12
08h	P1DS or PADS_L	Port 1 Drive Strength	Read/write	Byte	00h	Section 1.4.13
0Ah	P1SEL or	Port 1 Port Select	Read/write	Byte	00h	Section 1.4.14
	PFREN_L					
08h	P11DS or PFDS_L	Port 11 Drive Strength	Read/write	Byte	00h	Section 1.4.13
0Ah	P11SEL or PFSEL_L	Port 11 Port Select	Read/write	Byte	00h	Section 1.4.14
00h	PAIN	Port A Input	Read only	Word		
00h	PAIN_L		Read only	Byte		
01h	PAIN_H		Read only	Byte		
02h	PAOUT	Port A Output	Read/write	Word	undefined	
02h	PAOUT_L		Read/write	Byte	undefined	
03h	PAOUT_H		Read/write	Byte	undefined	
04h	PADIR	Port A Direction	Read/write	Word	0000h	

0Ah	PFSEL_L		Read/write	Byte	00h
0Bh	PFSEL_H		Read/write	Byte	00h
00h	PJIN	Port J Input	Read only	Word	
00h	PJIN_L		Read only	Byte	
01h	PJIN_H		Read only	Byte	
02h	PJOUT	Port J Output	Read/write	Word	undefined
02h	PJOUT_L		Read/write	Byte	undefined
03h	PJOUT_H		Read/write	Byte	undefined
04h	PJDIR	Port J Direction	Read/write	Word	0000h
04h	PJDIR_L		Read/write	Byte	00h
05h	PJDIR_H		Read/write	Byte	00h
06h	PJREN	Port J Resistor Enable	Read/write	Word	0000h
06h	PJREN_L		Read/write	Byte	00h
07h	PJREN_H		Read/write	Byte	00h

Interrupts vector Register P1IV

15	14	13	12	11	10	9	8
P1IV							
r0	r0	r0	r0	r0	r0	r0	r0
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
P1IV							
r0	r0	r0	r-0	r-0	r-0	r-0	r0

P1 IV Register Description

Bit	Field	Type	Reset	Description
15-0	P1IV	R	0h	Port 1 interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Port 1.0 interrupt; Interrupt Flag: P1IFG.0; Interrupt Priority: Highest 04h = Interrupt Source: Port 1.1 interrupt; Interrupt Flag: P1IFG.1 06h = Interrupt Source: Port 1.2 interrupt; Interrupt Flag: P1IFG.2 08h = Interrupt Source: Port 1.3 interrupt; Interrupt Flag: P1IFG.3 0Ah = Interrupt Source: Port 1.4 interrupt; Interrupt Flag: P1IFG.4 0Ch = Interrupt Source: Port 1.5 interrupt; Interrupt Flag: P1IFG.5 0Eh = Interrupt Source: Port 1.6 interrupt; Interrupt Flag: P1IFG.6 10h = Interrupt Source: Port 1.7 interrupt; Interrupt Flag: P1IFG.7; Interrupt Priority: Lowest

Interrupts vector Register P2IV

15	14	13	12	11	10	9	8
P2IV							
r0	r0	r0	r0	r0	r0	r0	r0
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
P2IV							
r0	r0	r0	r-0	r-0	r-0	r-0	r0

P2 IV Register Description

Bit	Field	Type	Reset	Description
15-0	P2IV	R	0h	Port 2 interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Port 2.0 interrupt; Interrupt Flag: P2IFG.0; Interrupt Priority: Highest 04h = Interrupt Source: Port 2.1 interrupt; Interrupt Flag: P2IFG.1 06h = Interrupt Source: Port 2.2 interrupt; Interrupt Flag: P2IFG.2 08h = Interrupt Source: Port 2.3 interrupt; Interrupt Flag: P2IFG.3 0Ah = Interrupt Source: Port 2.4 interrupt; Interrupt Flag: P2IFG.4 0Ch = Interrupt Source: Port 2.5 interrupt; Interrupt Flag: P2IFG.5 0Eh = Interrupt Source: Port 2.6 interrupt; Interrupt Flag: P2IFG.6 10h = Interrupt Source: Port 2.7 interrupt; Interrupt Flag: P2IFG.7; Interrupt Priority: Lowest

P1 IES Register

P1IES Register

7	6	5	4	3	2	1	0
P1IES							
rw	rw	rw	rw	rw	rw	rw	rw

P1IES Register Description

Bit	Field	Type	Reset	Description
7-0	P1IES	RW	undefined	Port 1 interrupt edge select 0b = P1IFG flag is set with a low-to-high transition. 1b = P1IFG flag is set with a high-to-low transition.

P1 IE Register

P1IE Register

7	6	5	4	3	2	1	0
P1IE							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

P1IE Register Description

Bit	Field	Type	Reset	Description
7-0	P1IE	RW	0h	Port 1 interrupt enable 0b = Corresponding port interrupt disabled 1b = Corresponding port interrupt enabled

P1 IFG Register

P1IFG Register

7	6	5	4	3	2	1	0
P1IFG							
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

P1IFG Register Description

Bit	Field	Type	Reset	Description
7-0	P1IFG	RW	0h	Port 1 interrupt flag 0b = No interrupt is pending 1b = Interrupt is pending

INTERRUPTS

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 input from humans. This saves the overhead of regular polling.**
- **Waking the CPU from sleep. (a low-power mode and can be awakened only by an interrupt).**
- **Calls to an operating system. A substitute is for software to set an unused interrupt flag for one of the peripherals, such as port P1 or P2.**

INTERRUPTS - HANDLED

- **The code that was interrupted can be resumed without error. (the values in the CPU registers must be restored).**

- **The hardware can take two extreme approaches to this:**
  - **Copies of all the registers are saved on the stack automatically as part of the process for entering an interrupt.**
  - **The opposite approach is for the hardware to save only the absolute minimum, which is the return address in the PC as in a subroutine.**
  - **This is much faster but it is up to the user to save and restore values of the critical registers, notably the status register.**

What Happens when an Interrupt Is Requested?

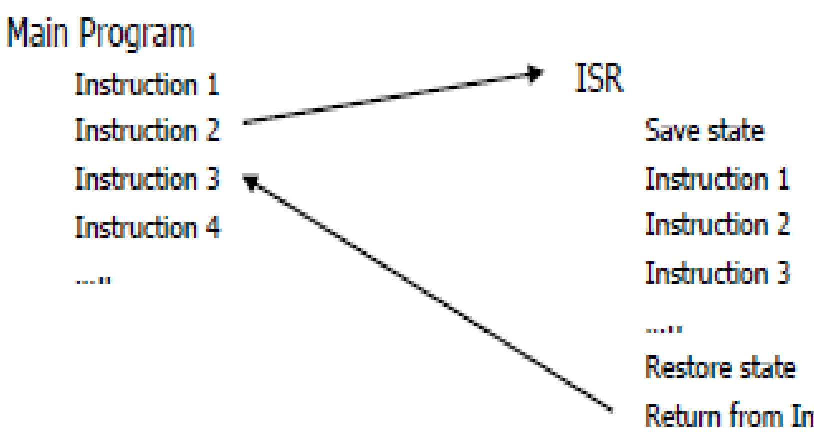
Hardware then performs the following steps to launch the ISR:

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 are waiting for service.**
5. **The interrupt request flag is cleared automatically for vectors that have a single source. Flags remain set for servicing by software if the vector has multiple sources, which applies to the example of TAIFG.**
6. **The SR is cleared, which has two effects. First, further maskable interrupts are disabled because the GIE bit is cleared; nonmaskable interrupts remain active. Second, it terminates any low-power mode**
7. **The interrupt vector is loaded into the PC and the CPU starts to execute the interrupt service routine at that address.**

This sequence takes six clock cycles in the MSP430 before the ISR commences. The stack at this point is shown in Figure 6.5. The position of SR on the stack is important if the low-power mode of operation needs to be changed.

MSP430 - Interrupts

- Reaction to something in I/O (human, comm link)
- Usually asynchronous to processor activities
- "interrupt handler" or "interrupt service routine" (ISR) invoked to take care of condition causing interrupt
  - Change value of internal variable (count)
  - Read a data value (sensor, receive)
  - Write a data value (actuator, send)



MSP430 - Interrupts

- Interrupts *preempt* normal code execution
  - Interrupt code runs in the *foreground*
  - Normal (e.g. `main()`) code runs in the *background*
- Interrupts can be *enabled* and *disabled*
  - *Globally*
  - *Individually* on a per-peripheral basis
  - *Non-Maskable* Interrupt (NMI)
- The occurrence of each interrupt is *unpredictable*
  - *When* an interrupt occurs
  - *Where* an interrupt occurs
- Interrupts are associated with a variety of on-chip and off-chip peripherals.
  - Timers, Watchdog, D/A, Accelerometer
  - NMI, change-on-pin (Switch)

MSP430 - INTERRUPTS

- Interrupts commonly used for
  - Urgent tasks w/higher priority than main code
  - Infrequent tasks to save polling overhead
  - Waking the CPU from sleep
  - Call to an operating system (software interrupt).
- Event-driven programming
  - The flow of the program is determined by events—i.e., sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads.
  - The application has a main loop with event detection and event handlers.

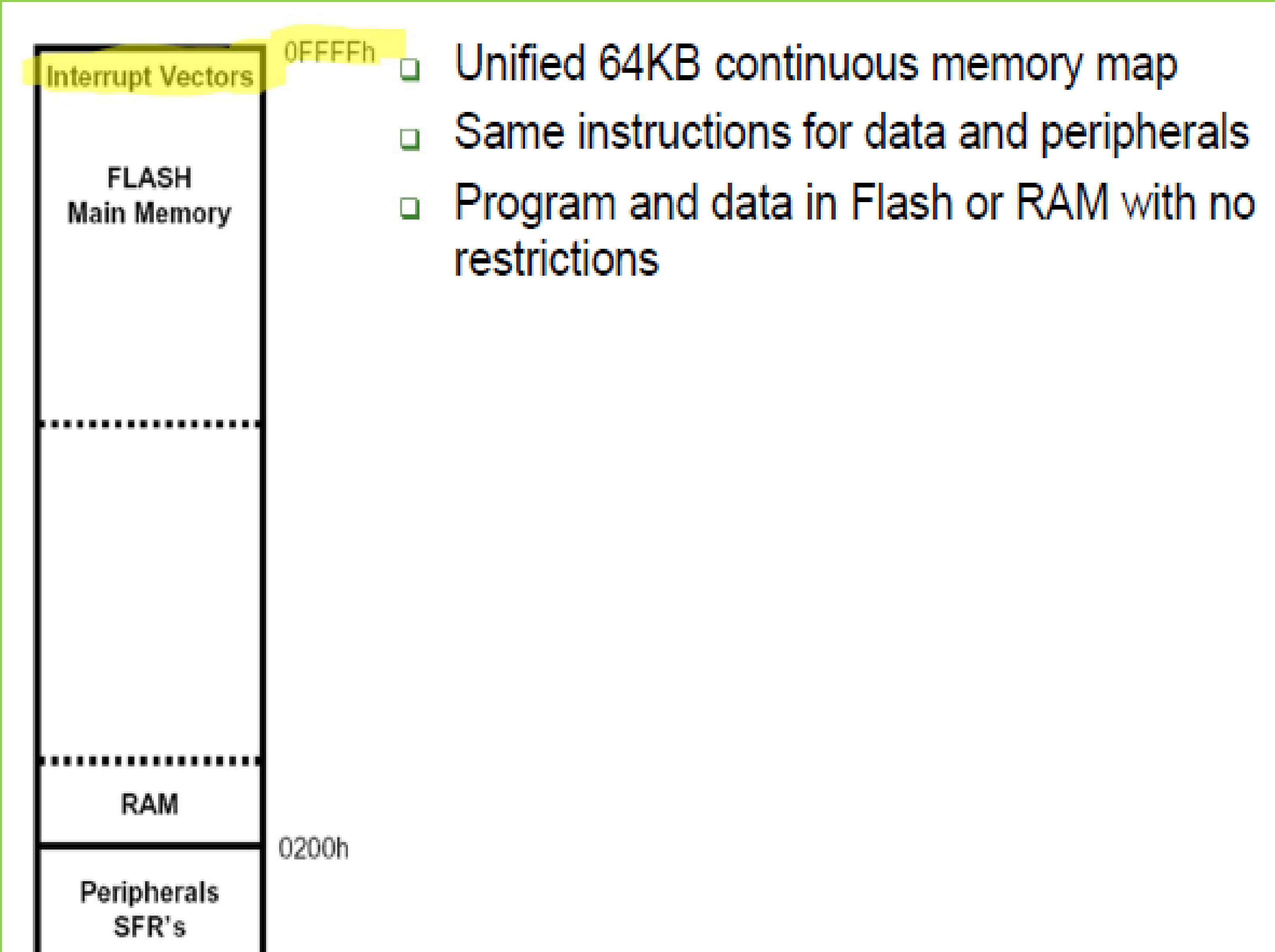
INTERRUPT FLAGS

- Each interrupt has a flag that is raised (set) when the interrupt occurs.
- Each interrupt flag has a corresponding enable bit – setting this bit allows a hardware module to request an interrupt.
- Most interrupts are ***maskable***, which means they can only interrupt if
  1) enabled and
  2) the general interrupt enable (GIE) bit is set in the status register (SR).

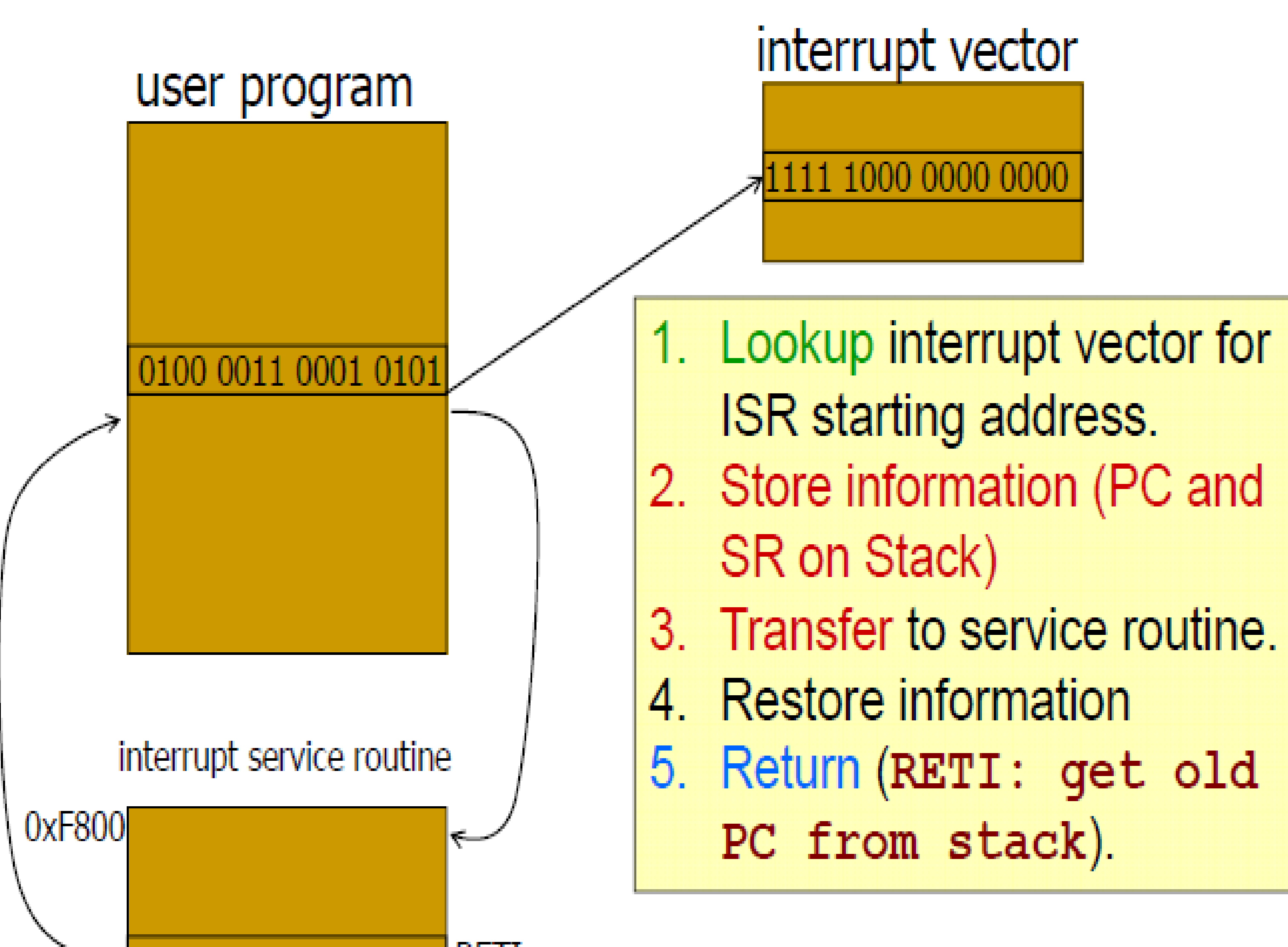
Interrupt vectors

- The CPU must know where to fetch the next instruction following an interrupt.
- The address of an ISR is defined in an *interrupt vector*.
- The MSP430 uses *vectored interrupts* where each ISR has its own vector stored in a *vector table* located at the end of program memory.
- Note: The *vector table* is at a fixed location (defined by the processor data sheet), but the ISRs can be located anywhere in memory.

INTERRUPT MEMORY



Serving Interrupt request



1. Lookup interrupt vector for ISR starting address.
2. Store information (PC and SR on Stack)
3. Transfer to service routine.
4. Restore information
5. Return (**RETI: get old PC from stack**).

MSP430x2xx Interrupt Vectors

Table 7. Interrupt Sources   Higher address -> higher priority

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-up External reset Watchdog Timer+ Flash key violation PC out-of-range[1]	PORIFG RSTIFG WDTIFG KEYV See [2]	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG[2][3]	(non)-maskable, (non)-maskable, (non)-maskable	0FFFCh	30
			0FFFAh	29
			0FFF8h	28
Comparator_A+ (MSP430F20x1)	CAIFG[4]	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer_A2	TACCR0 CCIFG[4]	maskable	0FFF2h	25
Timer_A2	TACCR1 CCIFG.TAIFG[2][4]	maskable	0FFF0h	24
			0FFEEh	23
			0FFECh	22
ADC10 (MSP430F20x2)	ADC10IFG[4]	maskable	0FFEAh	21
SD16_A (MSP430F20x3)	SD16CCTL0 SD16OVIFG, SD16CCTL0 SD16IFG[2][4]	maskable		
USI (MSP430F20x2, MSP430F20x3)	USIIFG, USISTTIFG[2][4]	maskable	0FFE8h	20
I/O Port P2 (two flags)	P2IFG.6 to P2IFG.7[2][4]	maskable	0FFE6h	19
I/O Port P1 (eight flags)	P1IFG.0 to P1IFG.7[2][4]	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
See [5]			0FFDEh to 0FFC0h	15 to 0, lowest

MSP430F2274 Address Space

Memory	Size	Address	Description	Access
Flash	32KB	0xFFFF 0xFFC0	Interrupt Vector Table	Word
		0xFFBF 0x8000	Program Code	Word/Byte
SRAM	1KB	0x05FF 0x0200	Stack	Word/Byte
	256	0x01FF 0x0100	16-bit Peripherals Modules	Word
	240	0x00FF 0x0010	8-bit Peripherals Modules	Byte
	16	0x000F 0x0000	8-bit Special Function Registers	Byte

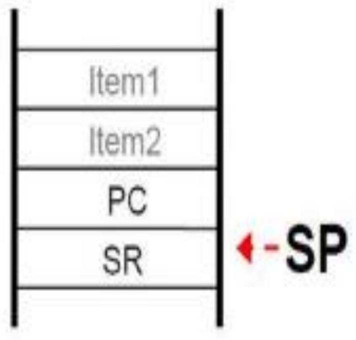
Processing an Interrupt………

1) Current instruction completed
2) MCLK started if CPU was off
3) Processor pushes program counter on stack
4) Processor pushes status register on stack
5) Interrupt w/highest priority is selected
6) Interrupt request flag cleared if single sourced
7) Status register is cleared
   - Disables further maskable interrupts (GIE cleared)
   - Terminates low-power mode
8) Processor fetches interrupt vector and stores it in the program counter
9) User ISR must do the rest!

INTERRUPT STACK

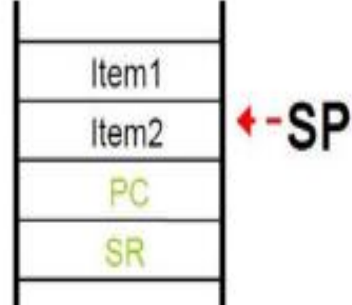
Item1
Item2
SP

Prior to Interrupt Service Routine (=ISR)



ISR hardware - automatically

- Program Counter (= PC) pushed
- Status Register (= SR) pushed
- Interrupt vector moved to PC
- **GIE, CPUOFF, OSCOFF and SCG1 cleared**
- IFG flag cleared on single source flags



reti - automatically

- SR popped - *original*
- PC popped

Interrupt Service Routines

- Look superficially like a subroutine.
- However, unlike subroutines
  - ISR's can execute at unpredictable times.
  - Must carry out action and thoroughly clean up.
  - Must be concerned with shared variables.
  - Must return using ***reti*** rather than ***ret***.
- ISR must handle interrupt in such a way that the interrupted code can be resumed without error
  - Copies of all registers used in the ISR must be saved (preferably on the stack)

Interrupt Service Routines

- Well-written ISRs:
  - Should be *short* and *fast*
  - Should affect the rest of the system *as little as possible*
  - Require a *balance* between doing very little – thereby leaving the background code with lots of processing – and doing a lot and leaving the background code with nothing to do
- Applications that use interrupts should:
  - Disable interrupts *as little as possible*
  - *Respond to interrupts* as quickly as possible

Interrupt Service Routines

- Interrupt-related runtime problems can be exceptionally hard to debug
- Common interrupt-related errors include:
  - Failing to *protect global variables*
  - Forgetting to actually *include the ISR* - no linker error!
  - Not testing or validating thoroughly
  - *Stack overflow*
  - Running out of *CPU horsepower*
  - Interrupting critical code
  - Trying to *outsmart the compiler*

RETURNING FROM ISR

- MSP430 requires 6 clock cycles before the ISR begins executing
  - The time between the interrupt request and the start of the ISR is called ***latency (plus time to complete the current instruction, 6 cycles, the worst case)***
- An ISR always finishes with the return from interrupt instruction (***reti***) requiring 5 cycles
  - The SR is popped from the stack
    - Re-enables maskable interrupts
    - Restores previous low-power mode of operation
  - The PC is popped from the stack
  - Note: if waking up the processor with an ISR, the new power mode must be set in the stack saved SR

Return From Interrupt

- Single operand instructions:

Mnemonic	Operation	Description
PUSH (.B or .W) src	SP-2→SP, src→@SP	Push byte/word source on stack
CALL dst	SP-2→SP, PC+2→@SP dst→PC	Subroutine call to destination
RETI	TOS→SR, SP+2→SP TOS→PC, SP+2→SP	Return from interrupt

- Emulated instructions:

Mnemonic	Operation	Emulation	Description
RET	@SP→PC SP+2→SP	MOV @SP+,PC	Return from subroutine
POP (.B or .W) dst	@SP→temp SP+2→SP temp→dst	MOV(.B or .W) @SP+,dst	Pop byte/word from stack to destination

Summary

- By coding efficiently you can run multiple peripherals at high speeds on the MSP430
- Polling is to be avoided – use interrupts to deal with each peripheral only when attention is required
- Allocate processes to peripherals based on existing (fixed) interrupt priorities - certain peripherals can tolerate substantial latency
- Use GIE when it's shown to be most efficient and the application can tolerate it – otherwise, control individual IE bits to minimize system interrupt latency.
- An interrupt-based approach eases the handling of *asynchronous* events

P1 AND P2 INTERRUPTS

- Only transitions (low to hi or hi to low) cause interrupts
- P1IFG & P2IFG (Port 1 & 2 Interrupt FlaG registers)
  - Bit 0: no interrupt pending
  - Bit 1: interrupt pending
- P1IES & P2IES (Port 1 & 2 Interrupt Edge Select reg)
  - Bit 0: PxIFG is set on low to high transition
  - Bit 1: PxIFG is set on high to low transition
- P1IE & P2IE (Port 1 & 2 Interrupt Enable reg)
  - Bit 0: interrupt disabled
  - Bit 1: interrupt enabled

Example P1 interrupt msp430x20x3_P1_02.c

```
#include  <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;   // Stop watchdog timer
  P1DIR |= 0x01;              // Set P1.0 to output direction
  P1IE |= 0x10;               // P1.4 interrupt enabled
  P1IES |= 0x10;              // P1.4 Hi/lo edge
  P1IFG &= ~0x10;             // P1.4 IFG cleared

  _BIS_SR(LPM4_bits + GIE);   // Enter LPM4 w/interrupt
}
// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
  P1OUT ^= 0x01;                             // P1.0 = toggle
  P1IFG &= ~0x10;                            // P1.4 IFG cleared
}
```

Ex: Timer interrupt: msp430x20x3_ta_03.c

```
#include <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                  // Stop WDT
  P1DIR |= 0x01;                             // P1.0 output
  TACTL = TASSEL_2 + MC_2 + TAIE;            // SMCLK, contmode, interrupt

  _BIS_SR(LPM0_bits + GIE);                  // Enter LPM0 w/ interrupt
}
// Timer_A3 Interrupt Vector (TAIV) handler
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
 switch( TAIV )
 {
   case  2: break;                           // CCR1 not used
   case  4: break;                           // CCR2 not used
   case 10: P1OUT ^= 0x01;                   // overflow
            break;
 }
```

Msp430x20x3_ta_06.c (modified, part 1)

```
#include  <msp430x20x3.h>
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;// Stop WDT
  P1DIR |= 0x01;            // P1.0 output
  CCTL1 = CCIE;             // CCR1 interrupt enabled
  CCR1 = 0xA000;
  TACTL = TASSEL_2 + MC_2; // SMCLK, Contmode
  _BIS_SR(LPM0_bits + GIE);// Enter LPM0 w/ int.
}
```

Servicing a timer interrupt; toggling pin in ISR

Msp430x20x3_ta_06.c (modified, part 2)

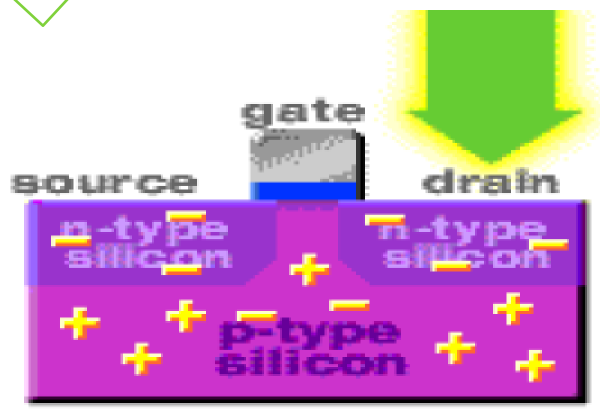
```
// Timer_A3 Interrupt Vector (TAIV) handler
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A(void)
{
  switch( TAIV )
  {
  case  2:              // CCR1
    {
    P1OUT ^= 0x01;   // Toggle P1.0
    CCR1 += 0xA000; // Add Offset to CCR1 == 0xA000
    }
           break;
  case  4: break;    // CCR2 not used
  case 10: break;    // overflow not used
 }
}
```

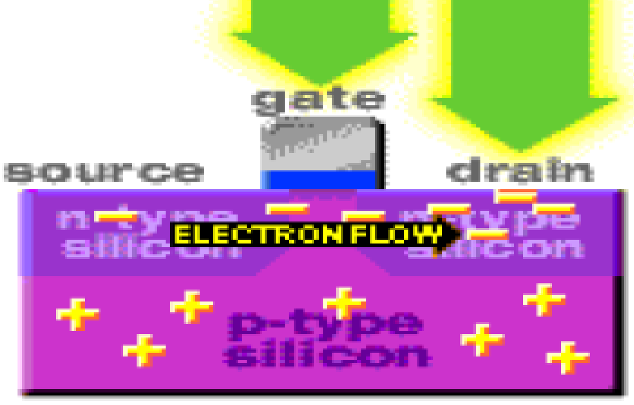
Low power modes :

Energy and Power

LOW POWER MODES

- Energy: ability to do work
  - Most important in battery-powered systems
- Power: energy per unit time
  - Important even in wall-plug systems---power becomes heat
- Power draw increases with...
  - Vcc
  - Clock speed
  - Temperature





Switching consumes power → dynamic power
- Switching slower, consume less power
- Smaller sizes reduce power to operate

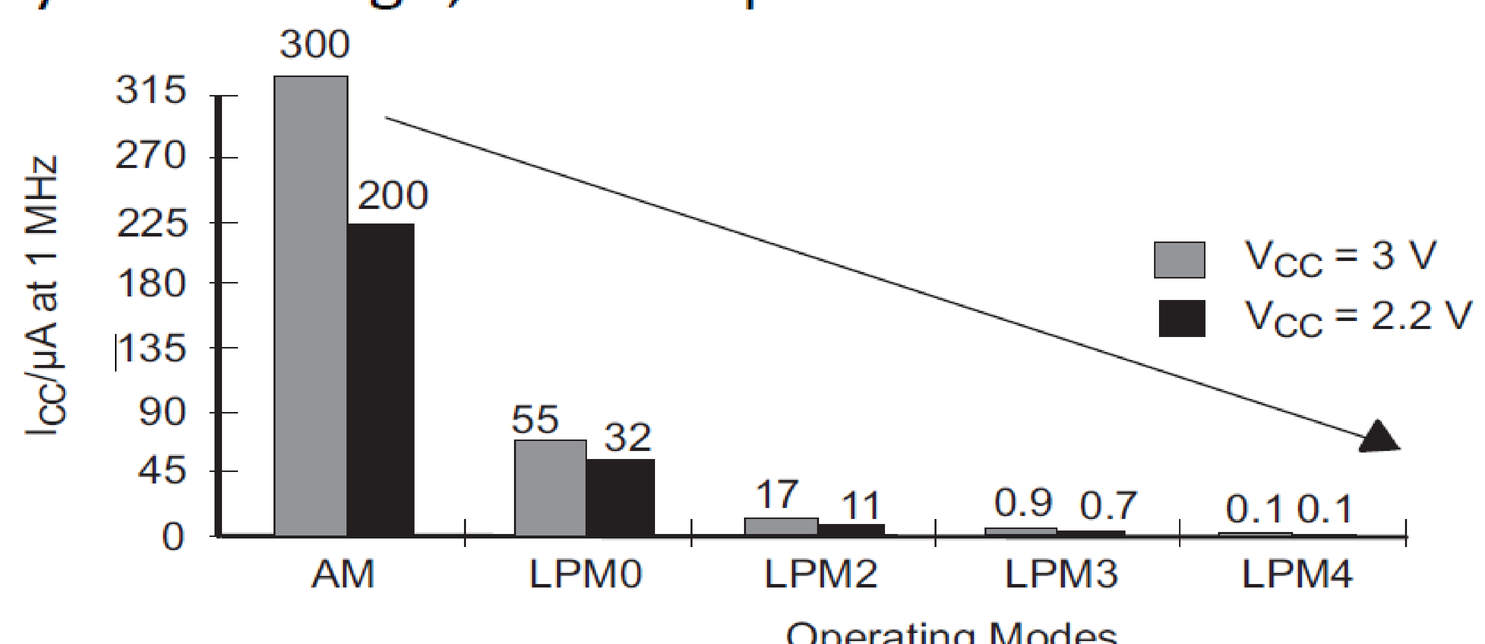
Leakage → static power

Low-Power Optimization

Why low power?

- Portable and mobile devices are getting popular, which have limited power sources, e.g., battery
- Energy conservation for our planet
- Power generates heat → low carbon

Power optimization becomes a new dimension in system design, besides performance and cost



Put the system in low-power modes and/or use low-power modules as much as possible

How?

- Provide clocks of diff. frequencies → frequency scaling
- Lower supplied voltage → voltage scaling
- Turn off clocks when no work to do → clock gating
- Use interrupts to wake up the CPU, return to sleep when done (another reason to use interrupts)
- Switched on peripherals only when needed
- Use low-power integrated peripheral modules in place of software, e.g., low-power DSP

- Clock gating for synchronous sequential logic:
  - Disable the clock so that flip-flops will hold their states forever and the whole circuit will not switch

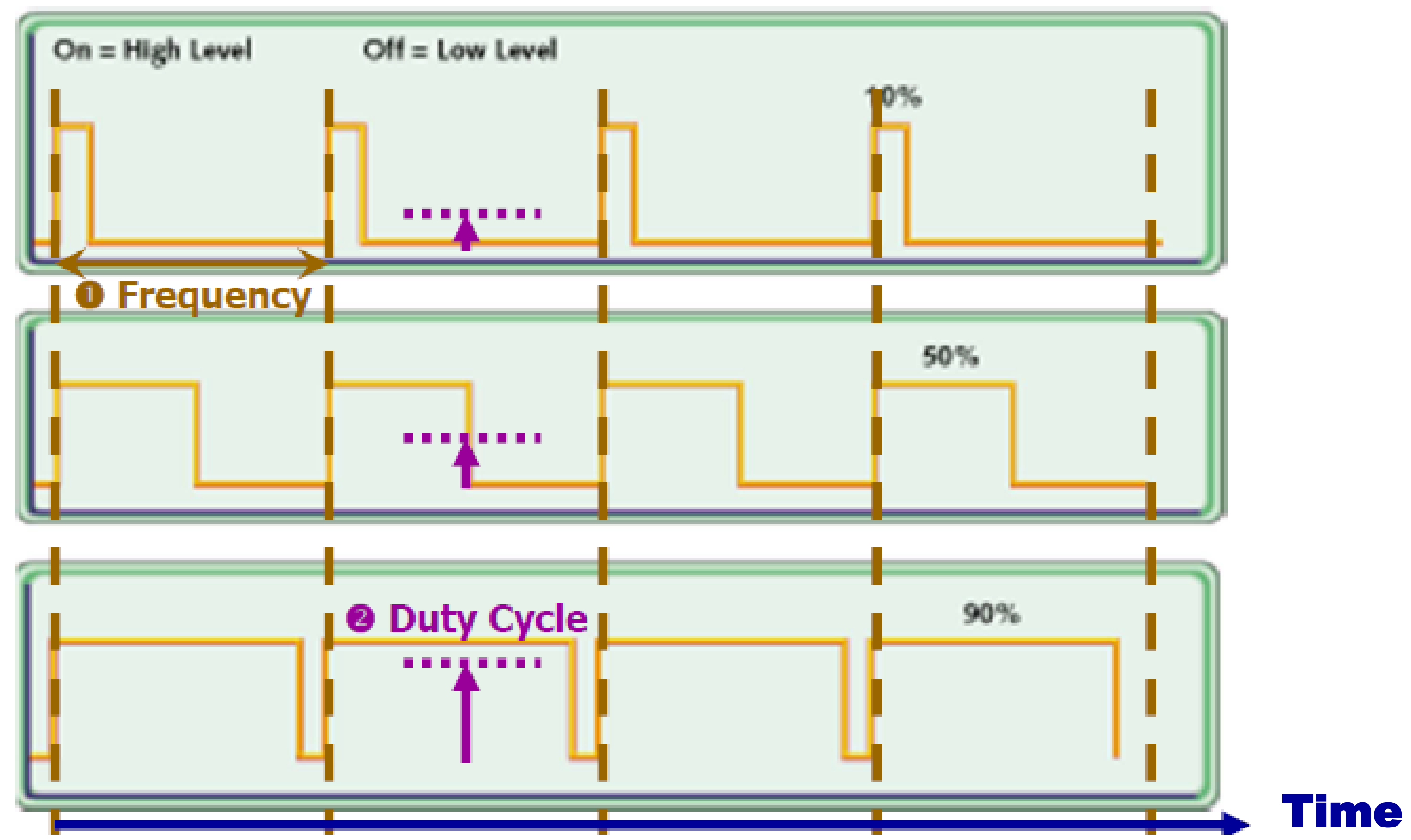
    → no dynamic power consumed

PWM Machines

Pulse Width Modulation (PWM)

- Pulse width modulation (PWM) is used to control analog circuits with a processor's digital outputs
- PWM is a technique of digitally encoding analog signal levels
  - The duty cycle of a square wave is modulated to encode a specific analog signal level
  - The PWM signal is still digital because, at any given instant of time, the full DC supply is either fully on or fully off
- The voltage or current source is supplied to the analog load by means of a repeating series of on and off pulses
- Given a sufficient bandwidth, any analog value can be encoded with PWM.

PWM – Frequency/Duty Cycle



Msp430x20x3_ta_16.c

PWM without the processor!

```
#include  <msp430x20x3.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                   // Stop WDT
  P1DIR |= 0x0C;                              // P1.2 and P1.3 output
  P1SEL |= 0x0C;                              // P1.2 and P1.3 TA1/2 options
  CCR0 = 512-1;                               // PWM Period
  CCTL1 = OUTMOD_7;                           // CCR1 reset/set
  CCR1 = 384;                                 // CCR1 PWM duty cycle
  TACTL = TASSEL_2 + MC_1;                    // SMCLK, up mode

  _BIS_SR(CPUOFF);                            // Enter LPM0
}
```

Multiple Oscillator Clock System

No crystal on eZ430 tools
Use VLO for ACLK
(mov.w #LFXT1S_2,&BCSCTL3)

Very Low-Power Oscillator (VLO)

ACLK 32 kHz

32.768 kHz

$f_{Crystal}$

Low-Power Peripherals

Control

MCLK 100 kHz - 25 MHz

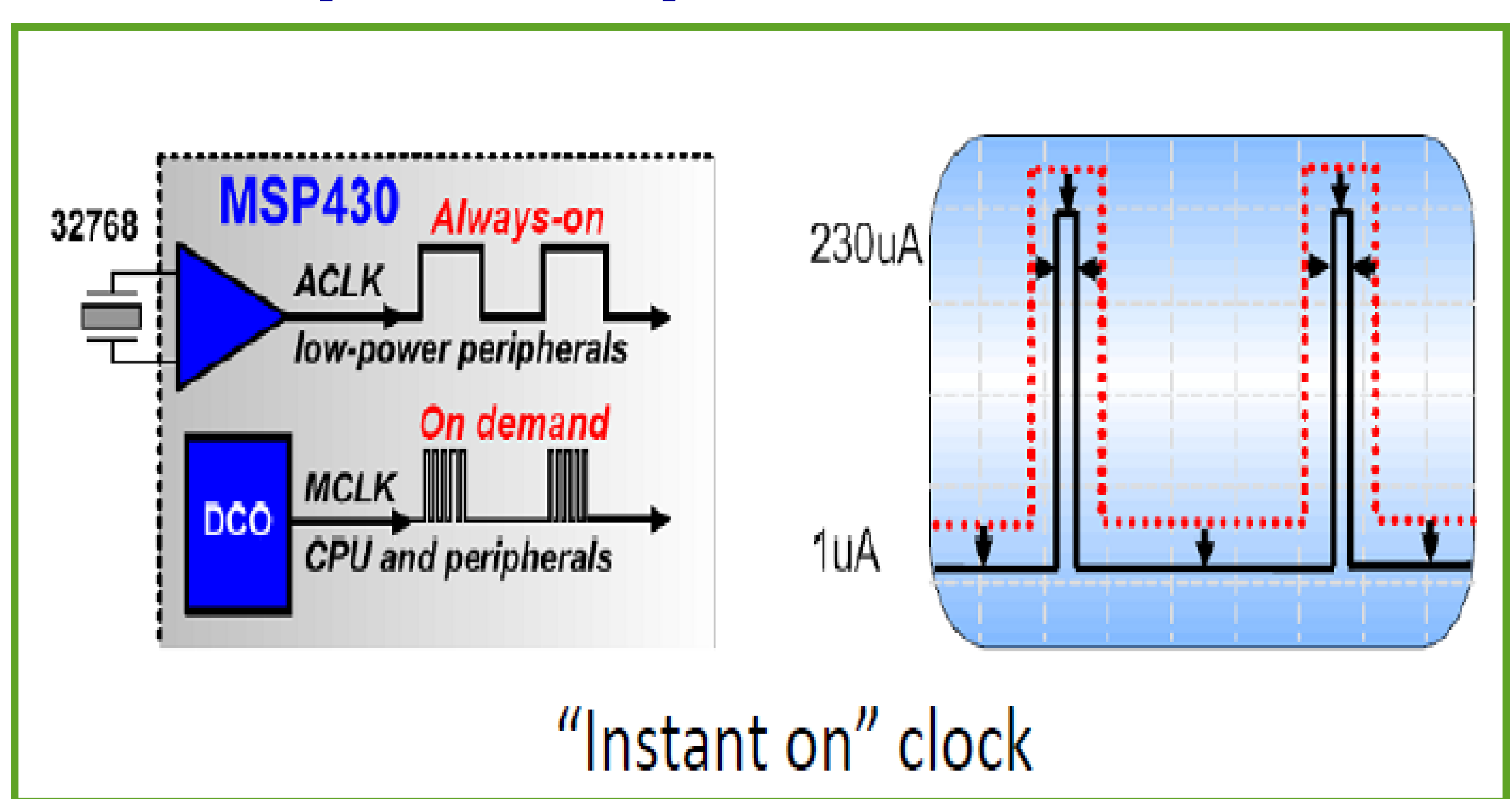
Digitally Controlled Oscillator

CPU and Peripherals

Lower Power Saving in MSP430

The most important factor for reducing power consumption is using the MSP430 clock system to maximize the time in LPM3

Finally, powering your system with lower voltages means lower power consumption as well.



LOW POWER MODES

CPU Off
DCO on
ACLK on
35uA

LPM0

fast clock from SMCLK and DCO

Stand-by
DCO off
ACLK on
0.8uA

LPM3

must maintain a real-time clock

Power Down
All clocks off
0.1uA

LPM4

- wakened only by an external signal, (RST/NMI)

Active Mode
DCO on
ACLK on
250uA

Active Mode

An interrupt automatically switches MSP430 …

LOW POWER MODES OF MSP430

- **Active mode:**

MSP430 starts up in this mode, which must be used when the CPU is required, i.e., to run code

An interrupt automatically switches MSP430 to active

Current can be reduced by running at lowest supply voltage consistent with the frequency of MCLK, e.g. VCC to 1.8V for fDCO = 1MHz

- **LPM0:**

CPU and MCLK are disabled

 Used when CPU is not required but some modules require a fast clock from SMCLK and DCO

- **LPM3:**

Only ACLK remains active

Standard low-power mode when MSP430 must wake itself at regular intervals and needs a (slow) clock

Also required if MSP430 must maintain a real-time clock

- **LPM4:**

CPU and all clocks are disabled

MSP430 can be wakened only by an external signal, e.g., RST/NMI, also called *RAM retention mode*

Principles of Low-Power Apps

- Maximize the time in LPM3 mode
- Use interrupts to wake the processor
- Switch on peripherals only when needed
- Use low-power integrated peripherals
- Timer_A and Timer_B for PWM
- Calculated branches instead of flag polling
- Fast table look-ups instead of calculations
- Avoid frequent subroutine and function calls
- Longer software routines should use single-cycle
- CPU registers

Controlling Low Power Modes

Through four bits in *Status Register (SR) in CPU*

- **SCG0 (System clock generator 0) : when set, turns off DCO, If DCOCLK is not used for MCLK or SMCLK**
- **SCG1 (System clock generator 1) : when set, turns off the SMCLK**
- **OSCOFF(Oscillator off): when set, turns off LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK**
- **CPUOFF (CPU off): when set, turns off the CPU**
- **All are clear in active mode**

Processor Clock Speeds

- Often, the most important factor for reducing power consumption is slowing the clock down
  - Faster clock = Higher performance, more power
  - Slower clock = Lower performance, less power
- Using assembly code:

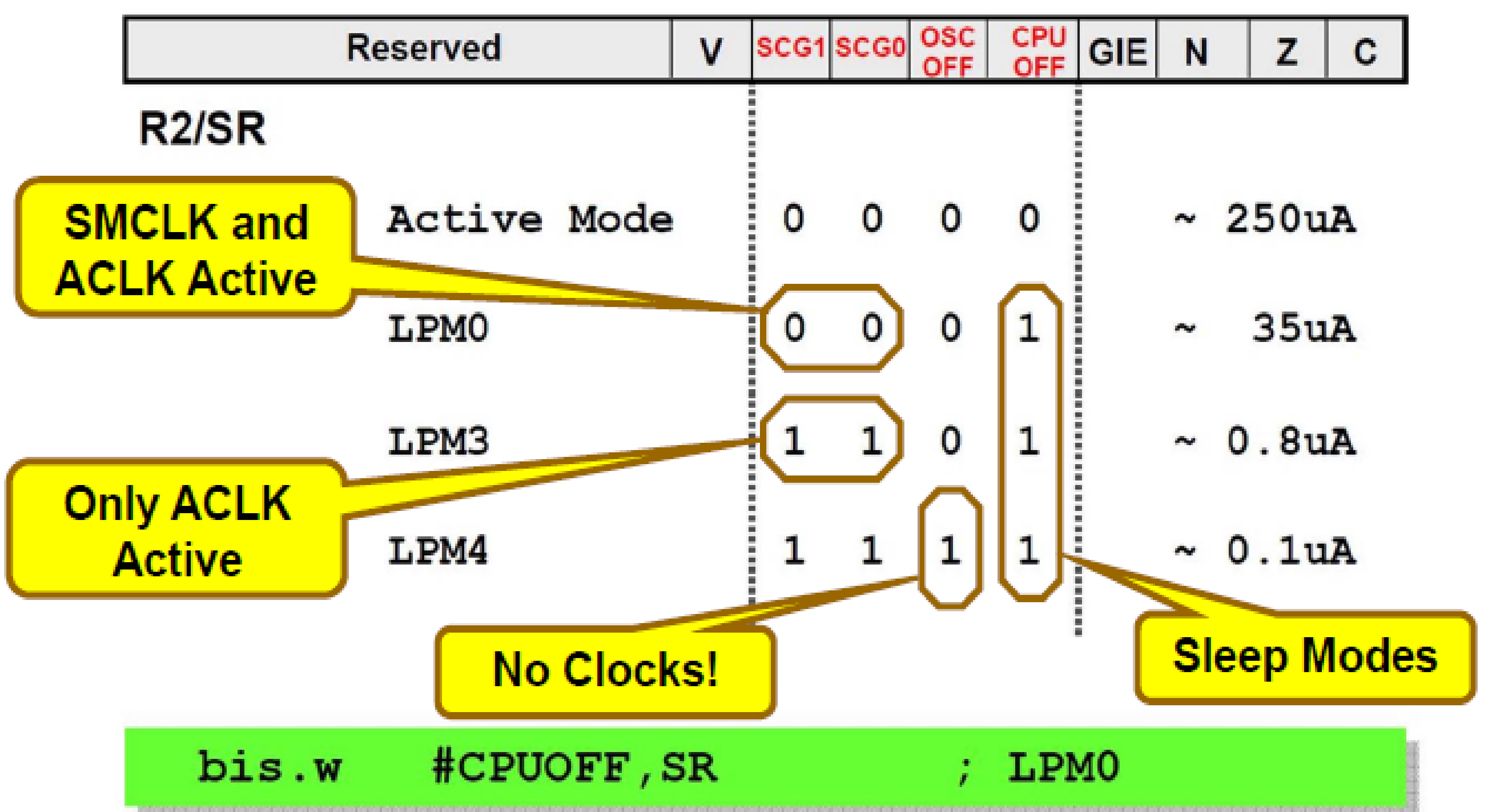
```
;      MSP430 Clock - Set DCO to 8 MHz:
       mov.b   #CALBC1_8MHZ,&BCSCTL1    ; Set range
       mov.b   #CALDCO_8MHZ,&DCOCTL     ; Set DCO step + modulation
```

- Using C code:

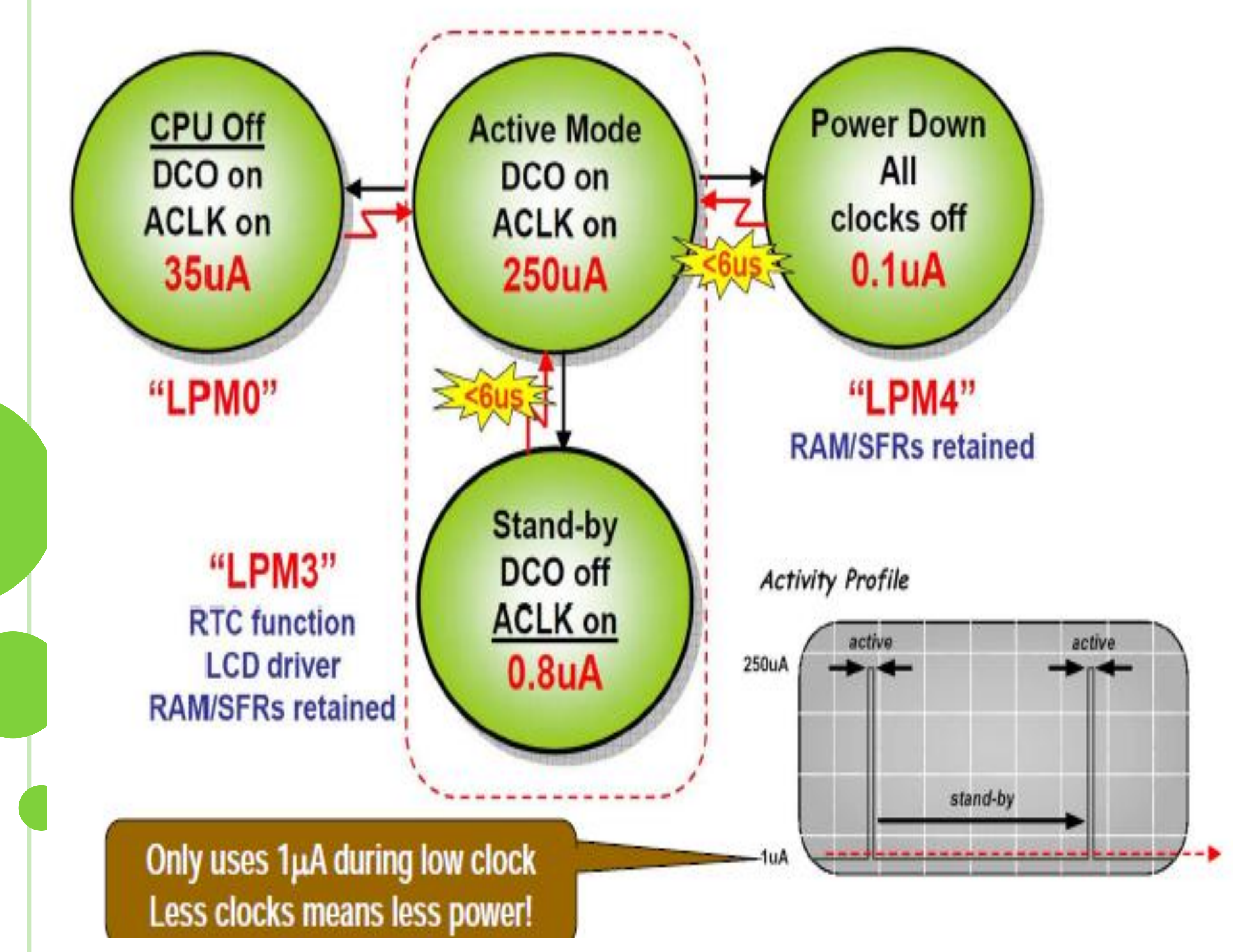
```
//     MSP430 Clock - Set DCO to 8 MHz:
       BCSCTL1 = CALBC1_8MHZ;          // Set range 8MHz
       DCOCTL = CALDCO_8MHZ;           // Set DCO step + modulation
```

Controlling Low Power Modes

Status bits and low-power modes



Msp 430 Clock Modes



STATUS BITS AND LOW-POWER MODES

SCG1	SCG0	OSC OFF	CPU OFF	MODE	CPU & 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 are 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 remains disabled, ACLK is active
1	1	1	1	LPM4	CPU and all clocks disabled

Entering/Exiting Low-Power Modes

Interrupt wakes MSP430 from low-power modes:

- Enter ISR:
  - PC and SR are stored on the stack
  - CPUOFF, SCG1, OSCOFF bits are automatically reset - entering active mode
  - MCLK must be started so CPU can handle interrupt
- Options for returning from ISR:
  - Original SR is popped from the stack, restoring the previous operating mode
  - SR bits stored on stack can be modified within ISR to return to a different mode when RETI is executed

Setting Low-Power Modes

- Setting low-power mode puts the microcontroller "to sleep" – so usually, interrupts would need to be enabled as well.
- Enter LPM3 and enable interrupts using assembly code:

```
;      enable interrupts / enter low-power mode 3
       bis.b   #LPM3+GIE,SR          ; LPM3 w/interrupts
```

- Enter LPM3 and enable interrupts using C code:

```
//  enable interrupts / enter low-power mode 3
    __bis_SR_register(LPM3_bits + GIE);
```

Sample Code (MSP430G2xx1 _ta_01)

```
void main(void) { //Toggle P1.0 every 50000 cycles
  WDTCTL = WDTPW + WDTHOLD; // Stop WDT
  P1DIR |= 0x01;            // P1.0 output
  CCTL0 = CCIE;             // CCR0 interrupt enabled
  CCR0 = 50000;
  TACTL = TASSEL_2 + MC_2;  // SMCLK, contmode
  _BIS_SR(LPM0_bits + GIE); // LPM0 w/ interrupt
}
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void) {
  P1OUT ^= 0x01;            // Toggle P1.0
  CCR0 += 50000;            // Add Offset to CCR0
}
```

Use _BIC_SR_IRQ(LPM0_bits) to exit LPM0

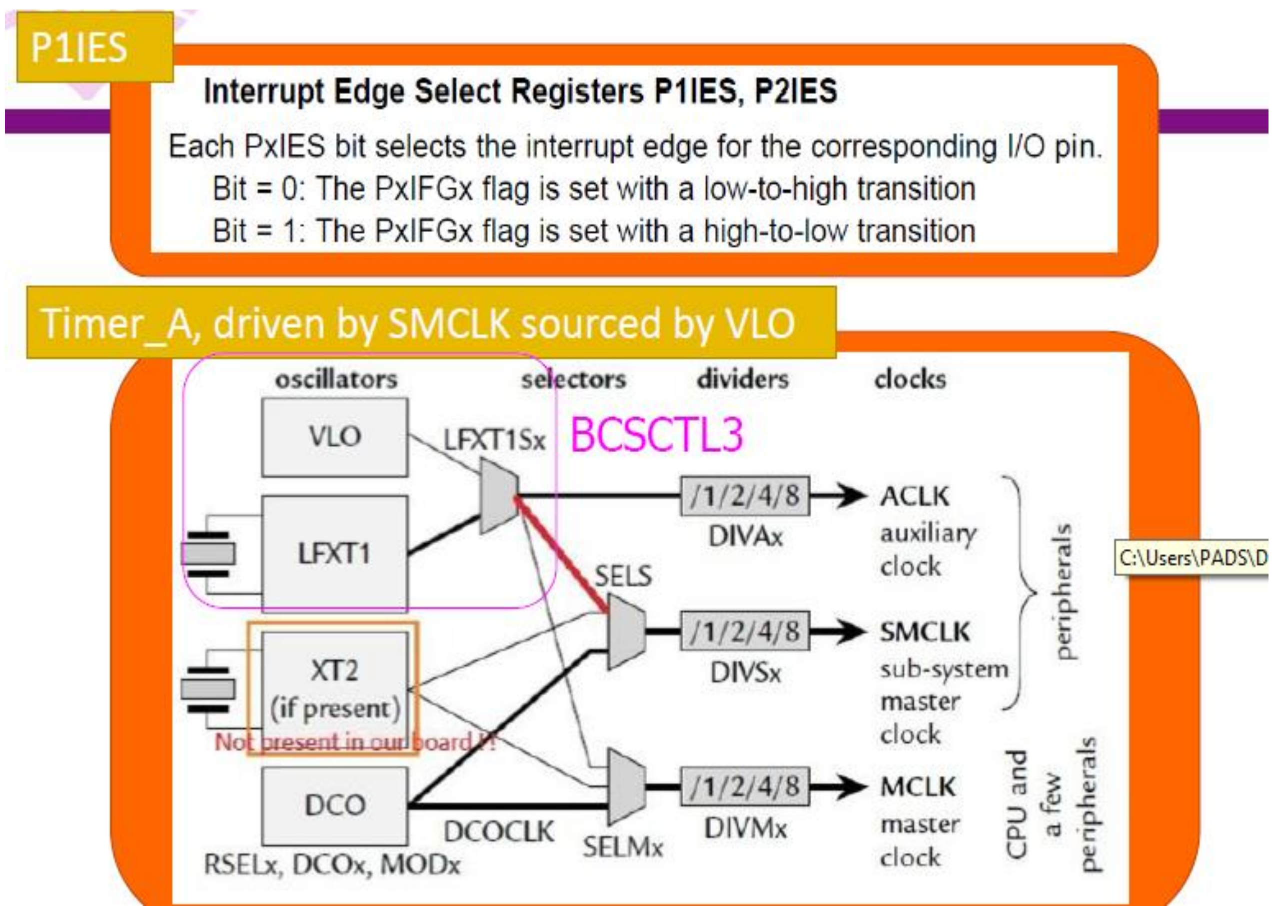
Flash green LED at 0.5 Hz using interrupt from Timer_A, driven by ACLK sourced by VLO in LPM3.

•While pushing the button:

- Change from LPM3 to LPM0

- Wake up every 1 sec using interrupt from Timer_A driven by SMCLK sourced by VLO.

- On wake up, measure the temperature. If the temperature is higher than 737, flash the red LED at 5 Hz; otherwise flash the green LED at 1 Hz.



• When the button is released, returns the system to LPM3 and flash green LED at 0.5 Hz again.

TIMERS

1. Watchdog timer:

Its main function is to protect the system against malfunctions but it can instead be used as an interval timer if this protection is not needed. (*petting, feeding, or kicking the dog)*

2. Timer_A: It typically has three channels and is much more versatile than the simpler timers. Timer_A can handle external inputs and outputs directly to measure frequency, time-stamp inputs, and drive outputs at precisely specified times, either once or periodically.

There are internal connections to other modules so that it can measure the duration of a signal from the comparator, for instance. ***It can also generate interrupts***.

3. Timer_B: Included in larger devices of all families. It is similar to Timer_A with some extensions that make it more suitable for driving outputs such as pulse-width modulation. Against this, it lacks a feature of sampling inputs in Timer_A that is useful in communication.

TIMERS

4. *Basic timer*: Present in the MSP430x4xx family only.

It provides the clock for the LCD and acts as an interval timer. Newer devices have the LCD_A controller, which contains its own clock generator and frees the basic timer from this task.

5. *Real-time clock*: In which the basic timer has been extended to provide a real-time clock in the most recent MSP430x4xx devices.

- The main applications of timers are to:
  - generate events of fixed time-period
  - allow periodic wakeup from sleep of the device
  - count transitional signal edges
  - replace delay loops allowing the CPU to sleep between operations, consuming less power
  - maintain synchronization clocks

Watchdog Timer

- **- To protect the system against failure of the software, such as the program becoming trapped in an *unintended, infinite loop*. Left to itself,**
- **The watchdog counts up and resets the MSP430 when it reaches its limit.**
- **Keep clearing the counter before the limit is reached to prevent a reset.**
- **Operation of WDT is controlled by the 16-bit register *WDTCTL.***
- **Password *WDTPW = 0x5A* in the upper byte.**
- **A reset will occur if a value with an incorrect password is written to WDTCTL.**
- **This can be done deliberately if you need to reset the chip from software.**
- **Reading WDTCTL returns 0x69 in the upper byte, so reading WDTCTL & writing the value back violates the password and causes a reset.**

The lower byte of the watchdog timer control register WDTCTL

7	6	5	4	3	2	1	0
WDT-HOLD	WDT-NMIES	WDTNMI	WDT-TMSEL	WDT-CNTCL	WDTSSEL	WDTISx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r0(w)	rw-(0)	rw-(0)	rw-(0)

- **The lower byte of WDTCTL contains the bits that control the operation of the watchdog timer.**
- **The RST/*NMI pin is also configured using this register (which* not forget when servicing the watchdog)**
- **Most bits are reset to 0 after a power-on reset (POR) but are unaffected by a power-up clear (PUC).**
- **This distinction is important in handling resets caused by the watchdog.**
- **The exception is the WDTCNTCL bit, labeled r0(w). This means that the bit always reads as 0 but a 1 can be written to stimulate some action, clearing the counter in this case.**

Watchdog counter

- The watchdog counter is a 16-bit register ***WDTCNT***, which is not visible to the user.
- It is clocked from either SMCLK or ACLK, according to the WDTSSEL bit.
- The reset output can be selected from bits 6, 9, 13, or 15 of the counter. Thus the period is $2^6 = 64, 512, 8192, \text{or } 32{,}768$ times the period of the clock.
- This is controlled by the WDTISx bits in WDTCTL. The intervals are roughly 2, 16, 250, and 1000 ms if the watchdog runs from ACLK at 32 KHz.
- The watchdog is always active after the MSP430 has been reset. By default the clock is SMCLK, which is in turn derived from the DCO at about 1 MHz.
- The default period of the watchdog is the maximum value of 32,768 counts, which is therefore around 32 ms.
- You must clear, stop, or reconfigure the watchdog before this time has elapsed.

Program wdtest1.c to demonstrate the watchdog timer.

- selected the clock from ACLK (WDTSSEL = 1) and the longest period (WDTISx = 00), which gives 1s with a 32 KHz crystal for ACLK. (restart)
- LED1 shows the state of button B1 and LED2 shows WDTIFG.
- The watchdog is serviced by rewriting the configuration value in a loop while button B1 is held down.
- If the button is left up for more than 1s the watchdog times out, raises the flag WDTIFG, and resets the device with a PUC. This is shown by LED2 lighting.

```
//----------------------------------------------------------------------
#include <io430x11x1.h>                    // Specific device
//----------------------------------------------------------------------
// Pins for LEDs and button
#define LED1      P2OUT_bit.P2OUT_3
#define LED2      P2OUT_bit.P2OUT_4
#define B1        P2IN_bit.P2IN_1
// Watchdog config: active, ACLK/32768 -> 1s interval; clear counter
#define WDTCONFIG     (WDTCNTCL|WDTSSEL)
// Include settings for _RST/NMI pin here as well
//----------------------------------------------------------------------
void main (void)
{
    WDTCTL = WDTPW | WDTCONFIG;            // Configure and clear watchdog
    P2DIR = BIT3 | BIT4;                   // Set pins with LEDs to output
    P2OUT = BIT3 | BIT4;                   // LEDs off (active low)
    for (;;) {                             // Loop forever
        LED2 = ~IFG1_bit.WDTIFG;           // LED2 shows state of WDTIFG
        if (B1 == 1) {                     // Button up
            LED1 = 1;                      // LED1 off
        } else {                           // Button down
            WDTCTL = WDTPW | WDTCONFIG;    // Feed/pet/kick/clear watchdog
            LED1 = 0;                      // LED1 on
        }
    }
}
```

WATCHDOG TIMER…

a. Fail - safe Clock Source for Watchdog Timer+

- This includes fail-safe logic to preserve the watchdog's clock.

- Suppose that the watchdog is configured to use ACLK and the program enters low-power mode 4 to wait for an external interrupt,
  - i. The old watchdog (WDT) stops during LPM4 and resumes counting when the device is awakened.
  - ii. WDT+ does not let the device enter LPM4 because that would disable its clock. Therefore it is not possible to use LPM4 with WDT+ active; the watchdog must first be stopped by setting WDTHOLD.
  - iii. Similarly, it is not possible to use LPM3 if WDT+ is active and gets its clock from SMCLK.
  - iv. If its clock fails, WDT+ switches from ACLK or SMCLK to MCLK and takes this from the DCO if an external crystal fails.
- The watchdog interval may change dramatically but there must be serious problems elsewhere if this happens.

- Newer devices, including the MSP430F2xx family and recent members of the MSP430x4xx, have the enhanced watchdog timer+ (WDT+).

WATCHDOG TIMER…

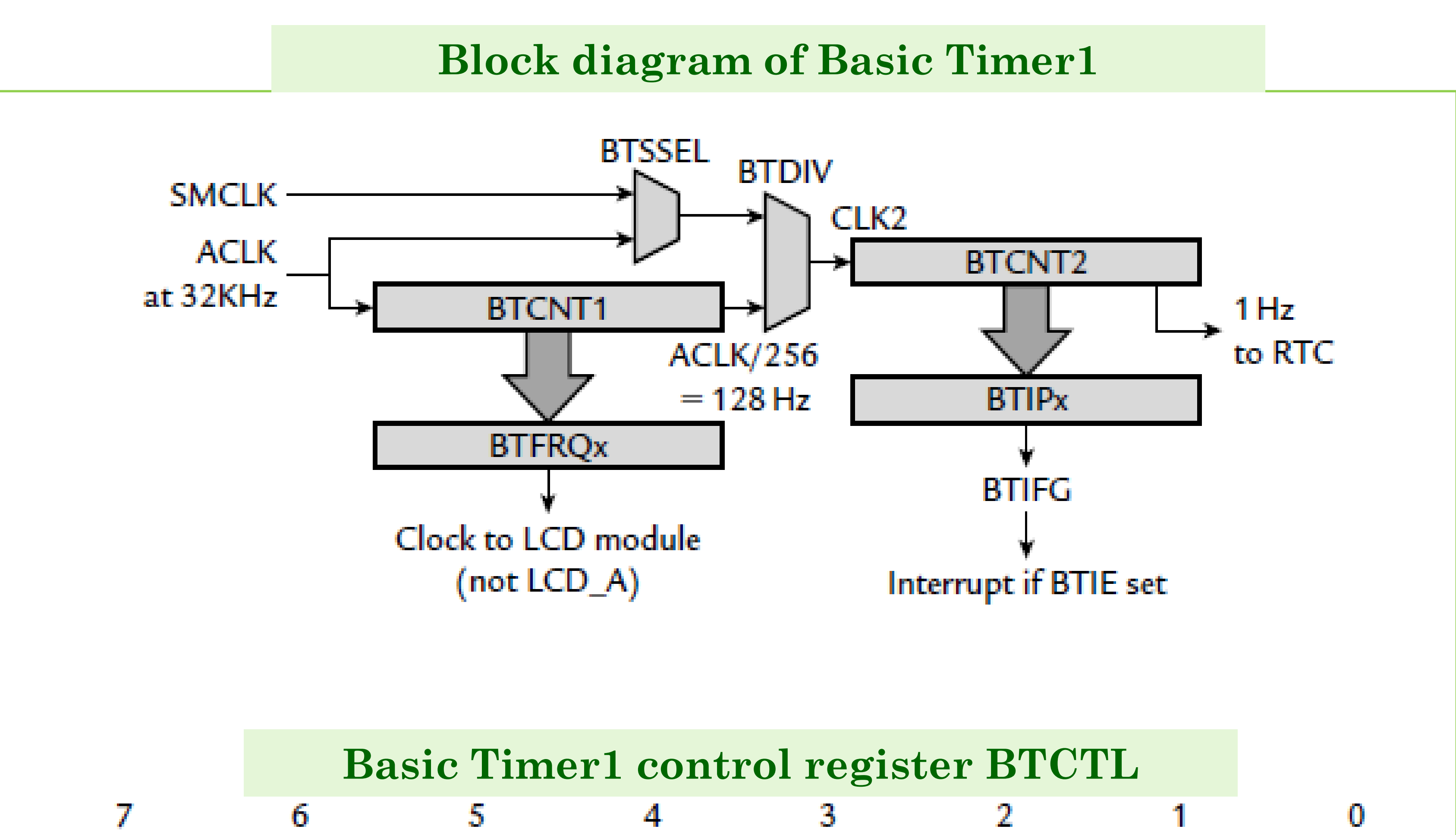
2. Watchdog as an Interval Timer

- **The watchdog can be used as an interval timer if its protective function is not desired.**
- **Set the WDTTMSEL bit in WDTCTL for interval timer mode.**
  - **The periods are the same as before and again WDTIFG is set when the timer reaches its limit, but no reset occurs.**
  - **The counter rolls over and restarts from 0.**
  - **An interrupt is requested if the WDTIE bit in the special function register IE1 is set.**
  - **This interrupt is maskable and as usual takes effect only if GIE is also set.**
  - **The watchdog timer has its own interrupt vector, which is fairly high in priority but not at the top.**
  - **It is not the same as the reset vector, which is taken if the counter times out in watchdog mode.**
- **The WDTIFG flag is automatically cleared when the interrupt is serviced. It can be polled if interrupts are not used.**

Basic Timer1

It provides the clock for the LCD module and generates periodic interrupts. (a real-time clock driven by a signal at 1Hz)

Block diagram of Basic Timer1



Basic Timer1 control register BTCTL

7	6	5	4	3	2	1	0
BTSSEL	BTHOLD	BTDIV	BTFRFQx		BTIPx		

Real-Time Clock

- It counts seconds, minutes, hours, days, months, and years.
- Alternatively it can be used as a straight forward
- It is configured in calendar mode by setting RTCMODEx=11 in the control register RTCCTL.

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

The Real-Time Clock control register RTCCTL.

a power-on reset, unlike BTCTL, and the RTCHOLD bit is set so that the clock does not run by default.

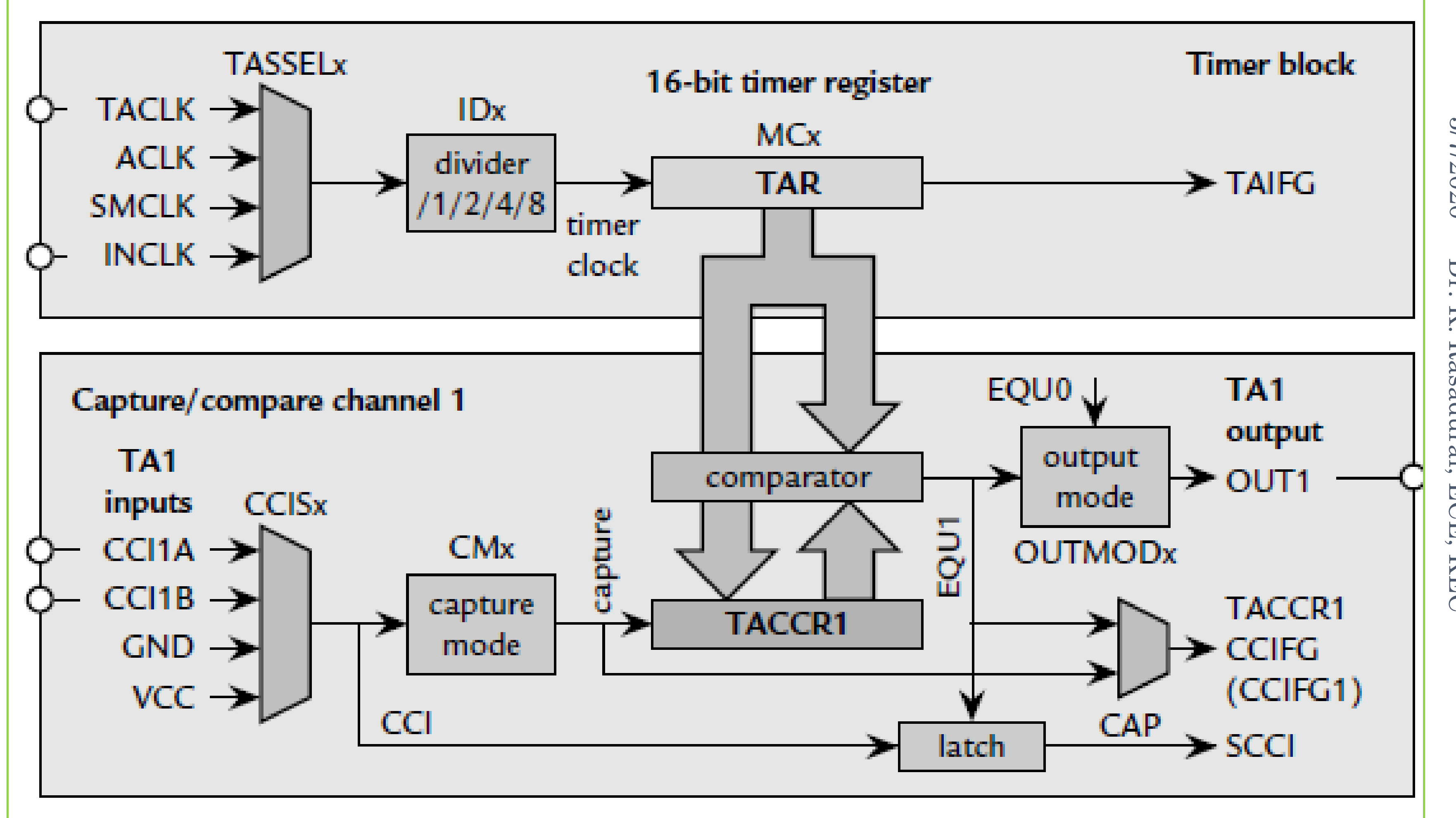
The current time and date are held in a set of registers that contain the following bytes:

- 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.

TIMER_A

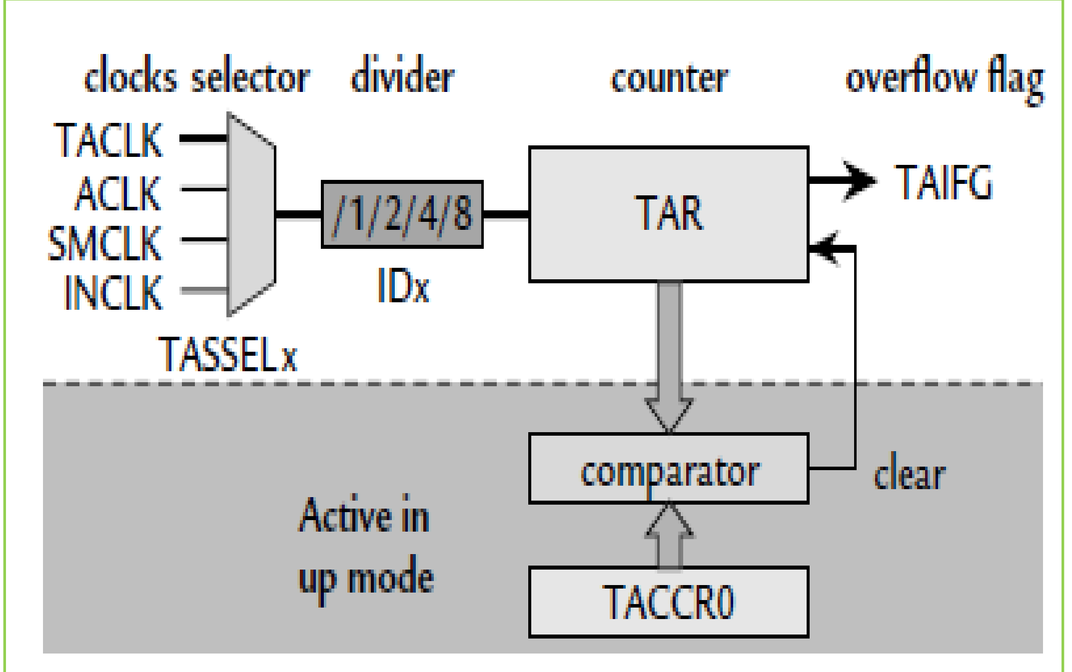
- Two main parts to the hardware:
  - **a. Timer block: The core, based on the 16-bit register TAR. There is a choice of sources** for the clock, whose frequency can be divided down (prescaled). The timer block has no output but a flag TAIFG is raised when the counter returns to 0.
  - **b. Capture/compare 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**
    - **Compare**
    - **Request an interrupt**
    - **Sample**

Block diagram of Timer_A : the timer block and capture / compare channel 1



The circles show external signals that may be brought out to pins of the device.

Timer Block



a 16-bit timer register TAR, which is central to the operation of the timer.

It can be chosen from four sources by using the TASSELx bits:

- **SMCLK**
- **ACLK**
- **TACLK**
- **INCLK**.

Input clock		Timer clock		Range of timer	
Source	Frequency	Divider	Resolution	Frequency	Period
SMCLK	16 MHz	1	$\frac{1}{16}$ µs	240 Hz	4 ms
SMCLK	1 MHz	1	1 µs	15 Hz	66 ms
SMCLK	1 MHz	8	8 µs	2 Hz	0.5 s
ACLK	32 KHz	1	31 µs	$\frac{1}{2}$ Hz	2 s
ACLK	32 KHz	8	240 µs	$\frac{1}{16}$ Hz	16 s

TxCTL Control Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(Used by Timer_B)						TxSSELx		IDx		MCx		-	TxCLR	TxIE	TxIFG

Bit		Description	
9-8	TxSSELx	Timer_x clock source:	0 0 ⇒ TxCLK 0 1 ⇒ ACLK 1 0 ⇒ SMCLK 1 1 ⇒ INCLK
7-6	IDx	Clock signal divider:	0 0 ⇒ / 1 0 1 ⇒ / 2 1 0 ⇒ / 4 1 1 ⇒ / 8
5-4	MCx	Clock timer operating mode:	0 0 ⇒ Stop mode 0 1 ⇒ Up mode 1 0 ⇒ Continuous mode 1 1 ⇒ Up/down mode
2	TxCLR	Timer_x clear when TxCLR = 1	
1	TxIE	Timer_x interrupt enable when TxIE = 1	
0	TxIFG	Timer_x interrupt pending when TxIFG = 1	

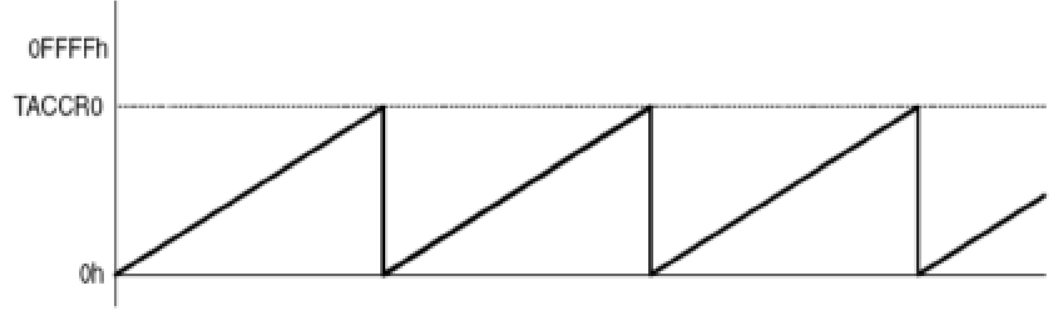
4 Modes of Operation

- Timer reset by writing a 0 to TxR
- Clock timer operating modes:

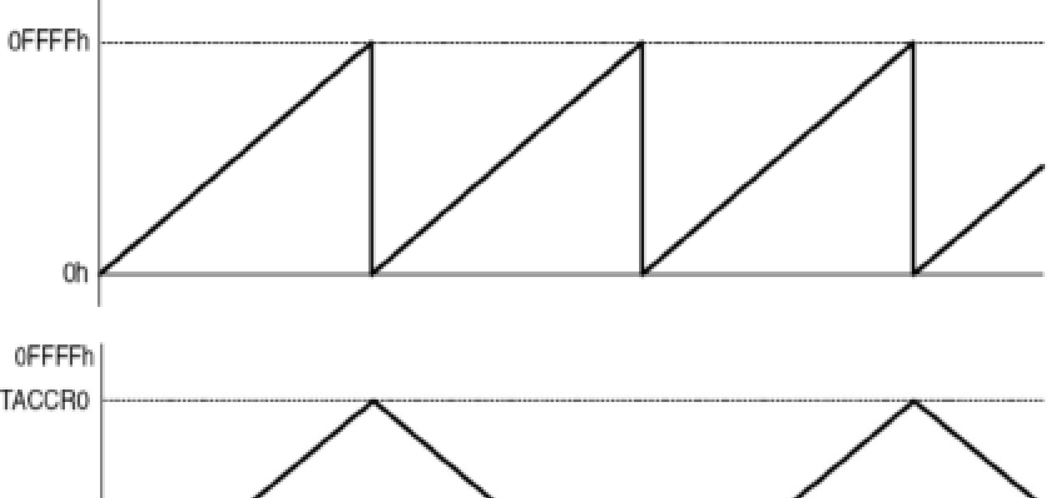
MCx	Mode	Description
0 0	Stop	The timer is halted.
0 1	Up	The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register.
1 0	Continuous	The timer repeatedly counts from 0x0000 to 0xFFFF.
1 1	Up/down	The timer repeatedly counts from 0x0000 to the value in the TxCCR0 register and back down to zero.

Timer Modes

- Up Mode



- Continuous Mode
- Up/Down Mode



12.3.1 TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Field	Bits	Description
Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted. 01 Up mode: the timer counts up to TACCR0. 10 Continuous mode: the timer counts up to 0FFFFh. 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0 Interrupt disabled 1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag 0 No interrupt pending 1 Interrupt pending

12.3.2 TAR, Timer_A Register

15	14	13	12	11	10	9	8
TARx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TARx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

TARx Bits 15-0 Timer_A register. The TAR register is the count of Timer_A.

12.3.3 TACCRx, Timer_A Capture/Compare Register x

15	14	13	12	11	10	9	8
TACCRx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TACCRx							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

TACCRx Bits 15-0 Timer_A capture/compare register.

Compare mode: TACCRx holds the data for the comparison to the timer value in the Timer_A Register, TAR.

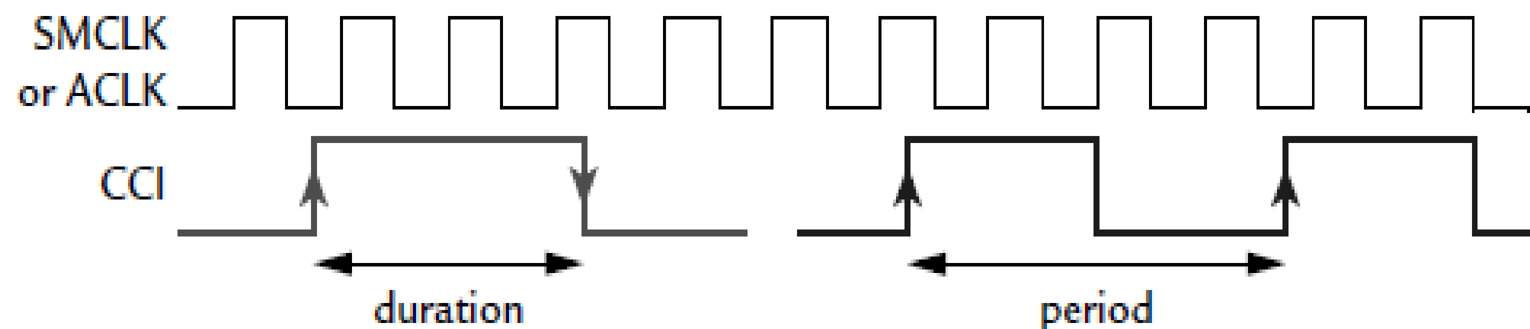
Capture mode: The Timer_A Register, TAR, is copied into the TACCRx register when a capture is performed.

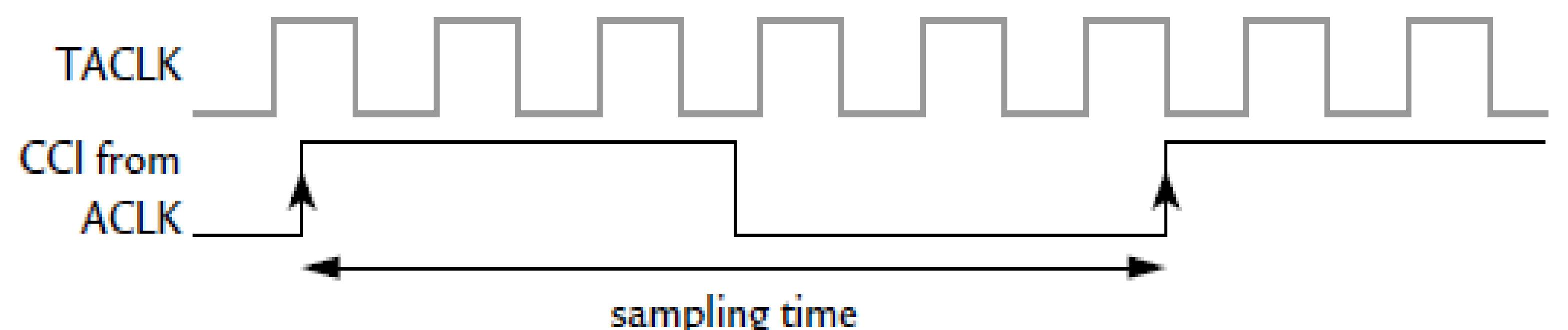
Table 12-2. Output Modes

OUTMODx	Mode	Description
000	Output	The output signal OUTx is defined by the OUTx bit. The OUTx signal updates immediately when OUTx is updated.
001	Set	The output is set when the timer *counts* to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected and affects the output.
010	Toggle/Reset	The output is toggled when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
011	Set/Reset	The output is set when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
100	Toggle	The output is toggled when the timer *counts* to the TACCRx value. The output period is double the timer period.
101	Reset	The output is reset when the timer *counts* to the TACCRx value. It remains reset until another output mode is selected and affects the output.
110	Toggle/Set	The output is toggled when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.
111	Reset/Set	The output is reset when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.

Measurement in the Capture Mode

- The Capture mode is used to take a time stamp of an event:
  - to note the time at which it occurred.
- A measurement typically requires two or more captures,
  - The timer can be used in two opposite ways illustrated in Figure:
  - Two ways in which the Capture mode is used to time a signal.





Measurement of Time: Press and Release of a Button
Measurement of Time: Reaction Timer
Measurement of Frequency: Comparison of SMCLK and ACLK

The formula is

$$\text{time (ms)} = \frac{1000 \times \text{time (counts)}}{f_{\text{ACLK}}}$$

with $f_{\text{ACLK}} = 32\,\text{KHz} = 32{,}768\,\text{Hz}$. It is tempting to "simplify" the numbers to give

$$\text{time (ms)} = \frac{\text{time (counts)}}{32{,}768}$$

Output in the Continuous Mode

The Continuous mode is typically used in the following circumstances:

- All channels are needed for output, including channel 0.
- Outputs must be driven at different, unrelated frequencies.
- Single delays are required rather than periodic signals.
- Some channels are used for capture and some for compare events.

- **Generation of Independent, Periodic Signals**
- **A Single Pulse or Delay with a Precise Duration**
- **Generation of a Precise Frequency**

PWM

- **Output in the Up Mode: Edge-Aligned Pulse-Width Modulation**
  - **Uses of Channel 0 in the Up Mode**
  - **Edge-Aligned PWM**
  - **Simple PWM**
  - **Design of PWM**
  - **Software-Assisted PWM**
- **Output in the Up/Down Mode: Centered Pulse-Width Modulation**

WHAT TIMER WHERE?

- Five types of timer in the MSP430
- Pulse-width modulation: Use Timer_B if available on your device, otherwise Timer_A. Connect the load directly to an output of the timer so that it can be driven directly by hardware.
- Less regular outputs: Connect directly to an output of Timer_A or B. Use the Up mode if the intervals between changes are always the same, as in many forms of communication. The Continuous mode is easier if the intervals vary.
- Inputs to be sampled at regular intervals: Connect directly to an input of Timer_A and use the Sampling mode (the Compare mode with the SCCI bit). This applies mainly to communications.
- Inputs to be timed: Connect slow inputs directly to a Capture input of Timer_A or B. Fast signals should be connected to one of the timer clock inputs, such as TACLK or INCLK.
- Interaction with other peripherals: Use the internal connections to other peripherals wherever possible, both for capture and compare events. This gives precise timing and saves power if the CPU need not be restarted.

WHAT TIMER WHERE?

- **Periodic software interrupts:** A wide range of options are available and the selection is less clear:
  - Try the watchdog timer if it is not needed as a watchdog and if the interval is appropriate; there is a choice of only four intervals for a given clock frequency.These are roughly 2, 16, 250, and 1000 ms from ACLK at 32 KHz, slower if VLO is used instead. Shorter intervals can be obtained by using SMCLK instead of ACLK.
  - The obvious choice in a MSP430x4xx device is Basic Timer1, again provided that the interval is convenient. The typical range is from about 16 ms to 2s. The real-time clock gives further options if available.
  - If neither of these is suitable you use Timer_A or B, which can produce almost any interval esired. The snag is that this may interfere with the use of their more advanced features.
- **Less regular software interrupts**: Use Timer_A or B, preferably in the Continuous mode.
- **The last resort:** Use software loops. Avoid these whenever possible except for trivial cases, such as delays while a clock stabilizes.

MIXED-SIGNAL SYSTEMS: ANALOG INPUT AND OUTPUT

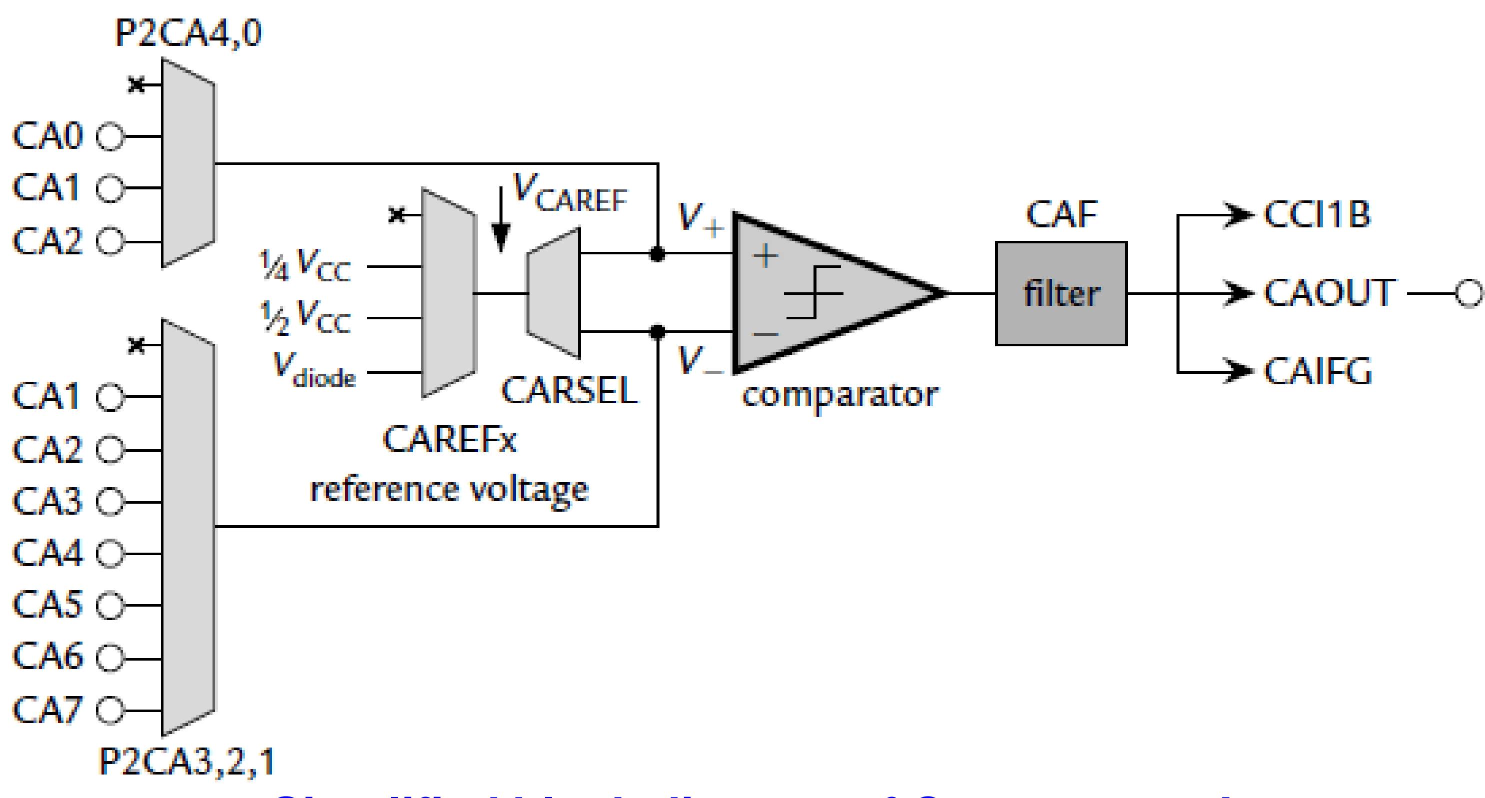
- **Comparator:** Simple and cheap module that cannot perform a conversion by itself but is usually used with Timer_A to measure the time-constant of an external *RC circuit*. There are two versions, Comparator_A and Comparator_A+.

- **Successive - approximation ADC:** The general-purpose type of ADC for many years. It is fast and relatively straightforward to understand. There are two versions, ADC10 and ADC12, which give 10 and 12 bits of output.

- **Sigma–delta ADC**: A more complicated ADC that works in a quite different way to give higher resolution (more bits) but at a slower speed. There are two versions, SD16 and SD16_A, both of which give a 16-bit output.

COMPARATOR_A

- **Architecture of Comparator_A+**



Simplified block diagram of Comparator_A+

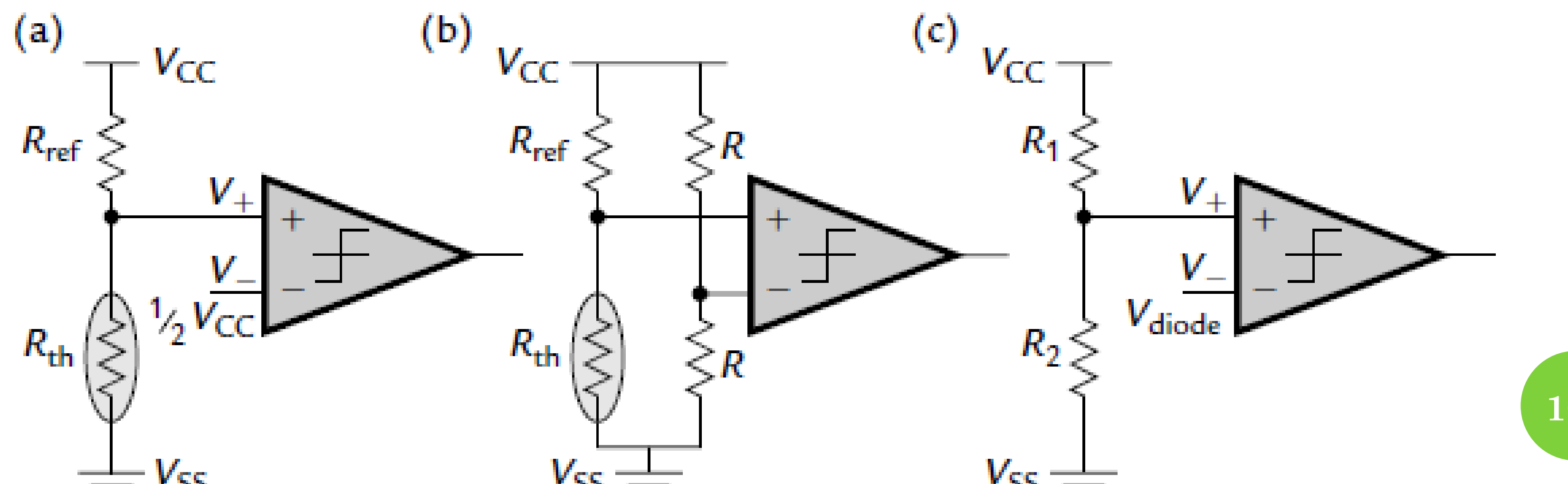
COMPARATOR_A

- **Operation of Comparator_A+**

(a) Thermistor connected in a potential divider with the internal reference 0.5*VCC*.

(b) Wheatstone bridge with two* resistors *R to represent the internal reference.

(c) Battery voltage monitor against* the fixed internal reference voltage *Vdiode.



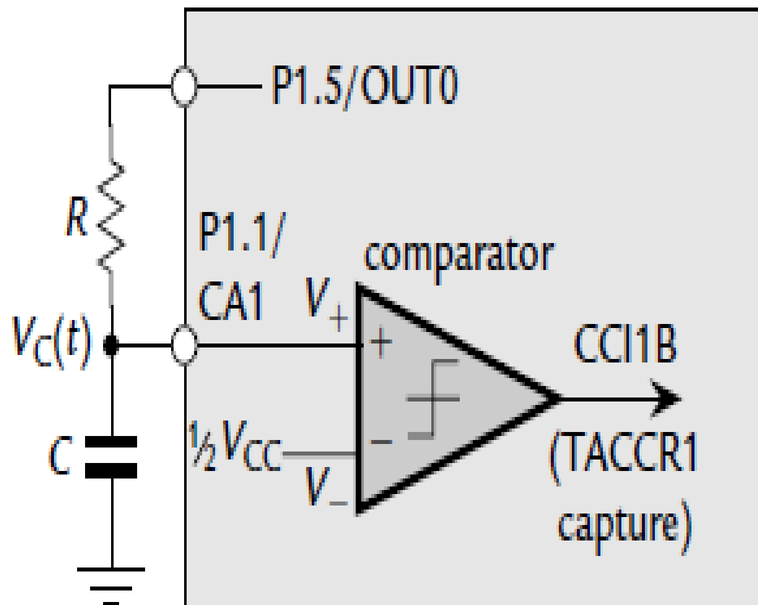
COMPARATOR_A

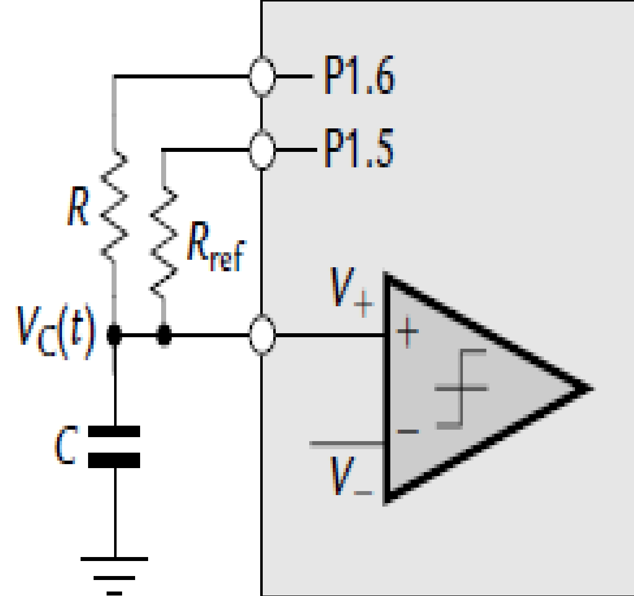
Slope Conversion of a Resistance

- Connection of a resistor and capacitor to the comparator to form a slope ADC.

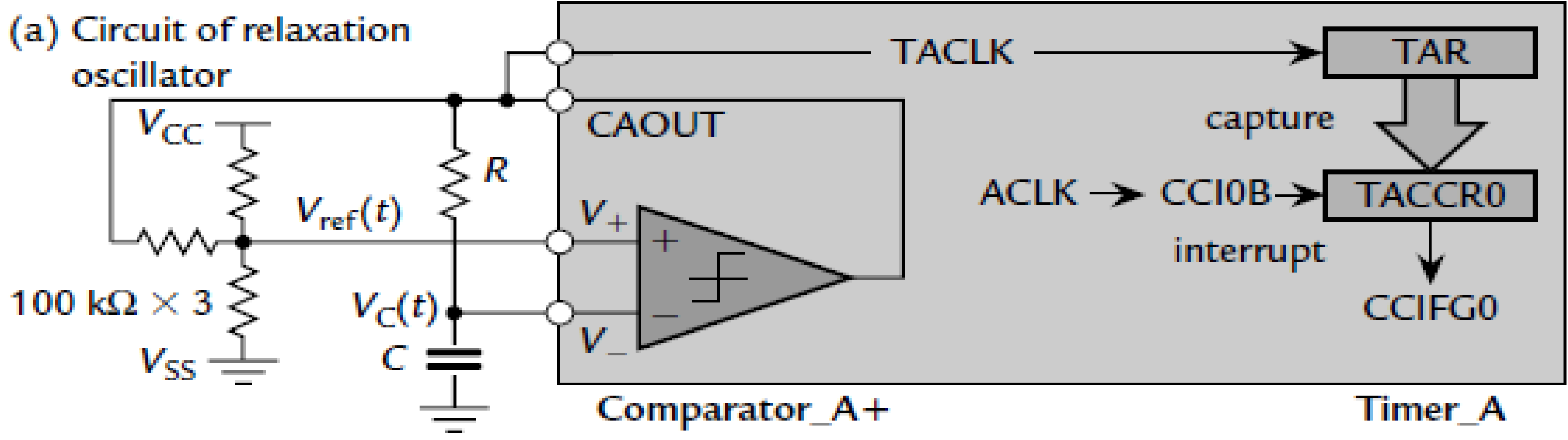
  (a) Single resistor and

  (b) reference resistor for comparison with the unknown resistor.



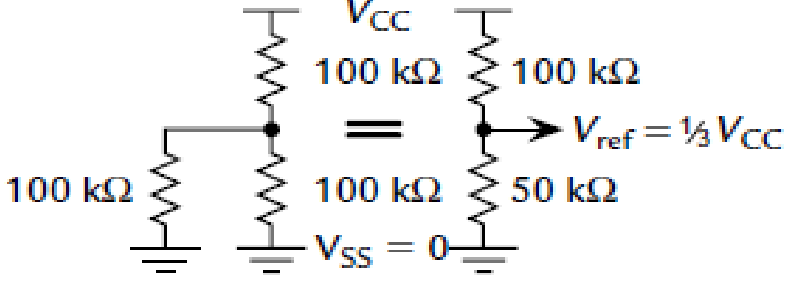


Relaxation Oscillator with Comparator_A



(b) Reference chain when CAOUT = 1

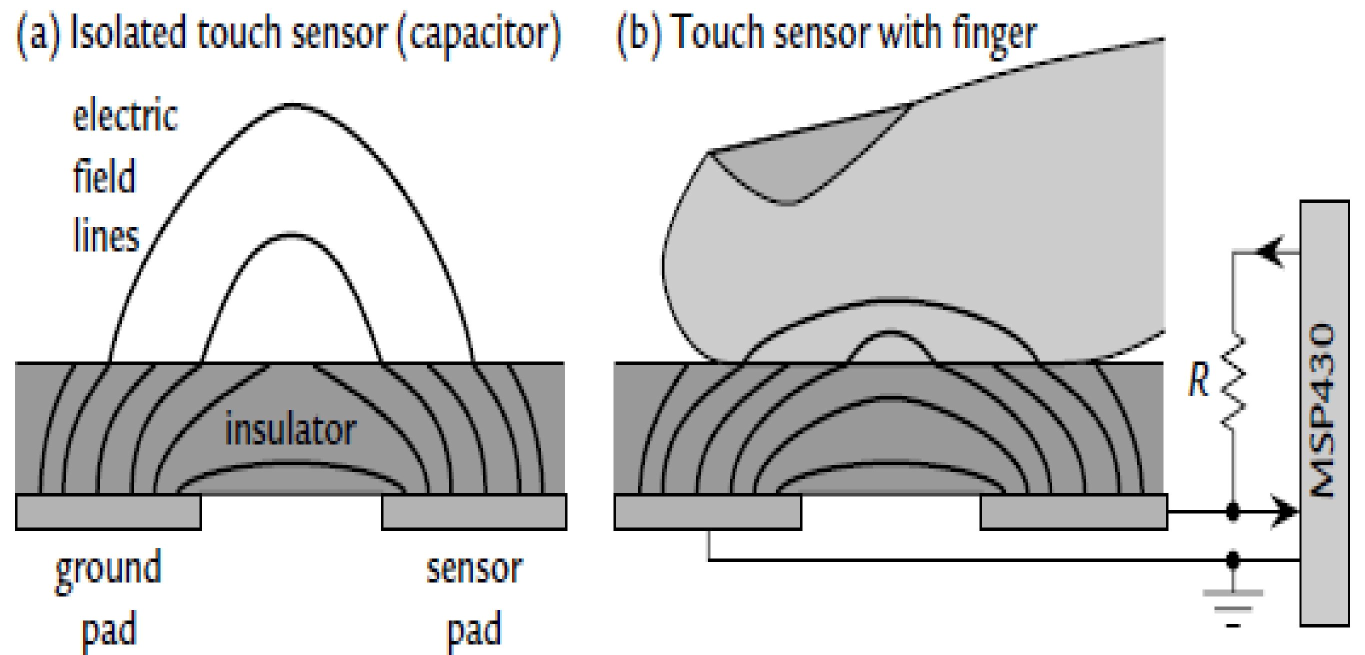
V_{CC}
100 kΩ
100 kΩ
50 kΩ
$V_{ref} = ⅔ V_{CC}$
100 kΩ
100 kΩ
$V_{SS} = 0$



(a) Circuit with the capacitor connected to the inverting input of the comparator so that CAOUT provides the correct drive for the resistor. The circuit for the threshold voltage has three equal resistors so that its output voltage is (b) 1/3 *VCC when CAOUT = 1 and* (c) 2/3 *VCC when CAOUT = 0.*

Comparator_A

- **Capacitative Touch Sensing with Comparator_A**



Operation of a simple capacitative touch sensor.

(a) Two conducting pads on the bottom of an insulating sheet form a capacitor, whose electric field extends outside the top of the sheet.

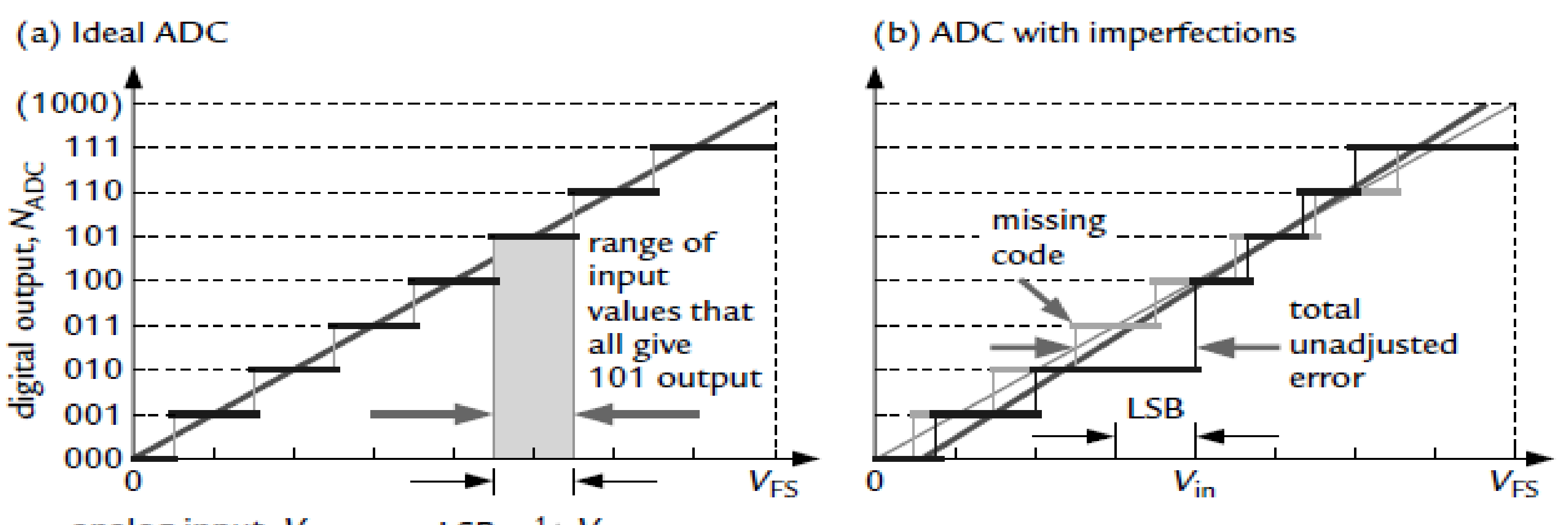
(b) A finger on top of the insulator distorts the electric field and increases the capacitance between the pads.

Analog-to-Digital Conversion

1. **Resolution, Precision, and Accuracy**
2. **Signal-to-Noise Ratios**
3. **Jitter in Timing**
4. **Sampling in Time and Aliasing**
5. **Practical Issues with ADCs**
    - **Input range**
    - **Voltage reference**
    - **Noise and filtering**
    - **Decoupling and layout**

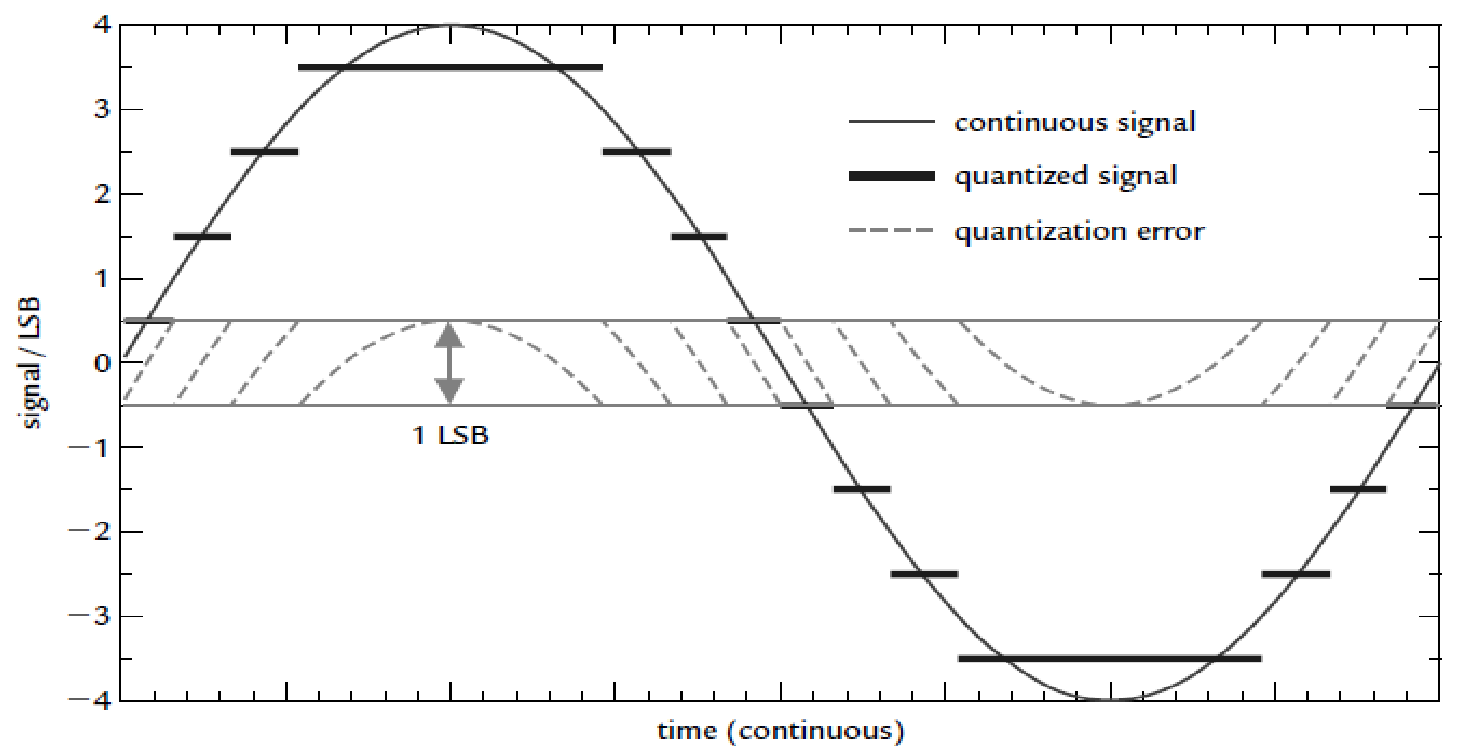
Analog-to-Digital Conversion

- **General Issues**
- **1. Resolution, Precision, and Accuracy**
- **Accuracy:** How close a measurement is to its "true" value, which would be produced by an ideal system. This is easy to define but ha rd to measure.
- **Resolution or precision:** The number of distinct output values that a measurement can provide. Alternatively, it can be specified as the change in input that corresponds to the minimum change in output, 1 bit.



ADC Signal-to-Noise Ratios

- **The total unadjusted error is called a *static parameter because in principle it is measured* by sweeping the input slowly through its range. The other type of parameter is *dynamic,* where we look at the performance as a function of time.**
- **The quantization of the input is a type of error.**
  - **by plotting the continuous signal, its quantized version, and the difference, called the *quantization error***



SIGNAL-TO-NOISE RATIOS…

- **Typically the quantization noise is equally likely to take any value between ±½ LSB, in which case its rms value can be shown to be LSB/ √12. The sine wave has a peak amplitude of 4LSB so its rms value is 4LSB/√2. The ratio of these two is called the *signal-to-noise ratio, or SNR. It is always quoted in decibels (dB) and is defined by***

$$\mathrm{SNR} = 20\log_{10}\left(\frac{\text{rms amplitude of signal}}{\text{rms amplitude of noise}}\right) \quad \mathrm{dB}$$

$$= 20\log_{10}\left[\frac{(4/\sqrt{2})\,\mathrm{LSB}}{(1/\sqrt{12})\,\mathrm{LSB}}\right] = 20\log_{10}(4\sqrt{6}) = 20\mathrm{dB}.$$

$$\mathrm{SNR} = 20\log_{10}\left[\frac{(½\times 2^N/\sqrt{2})\,\mathrm{LSB}}{(1/\sqrt{12})\,\mathrm{LSB}}\right] = 20\log_{10}\left(\sqrt{\tfrac{3}{2}}\,2^N\right)$$

$$= 20\log_{10}\left(\sqrt{\tfrac{3}{2}}\right) + 20\log_{10}(2^N)$$

$$= 10\log_{10}(\tfrac{3}{2}) + 20N\log_{10}(2) \quad \mathrm{dB}.$$

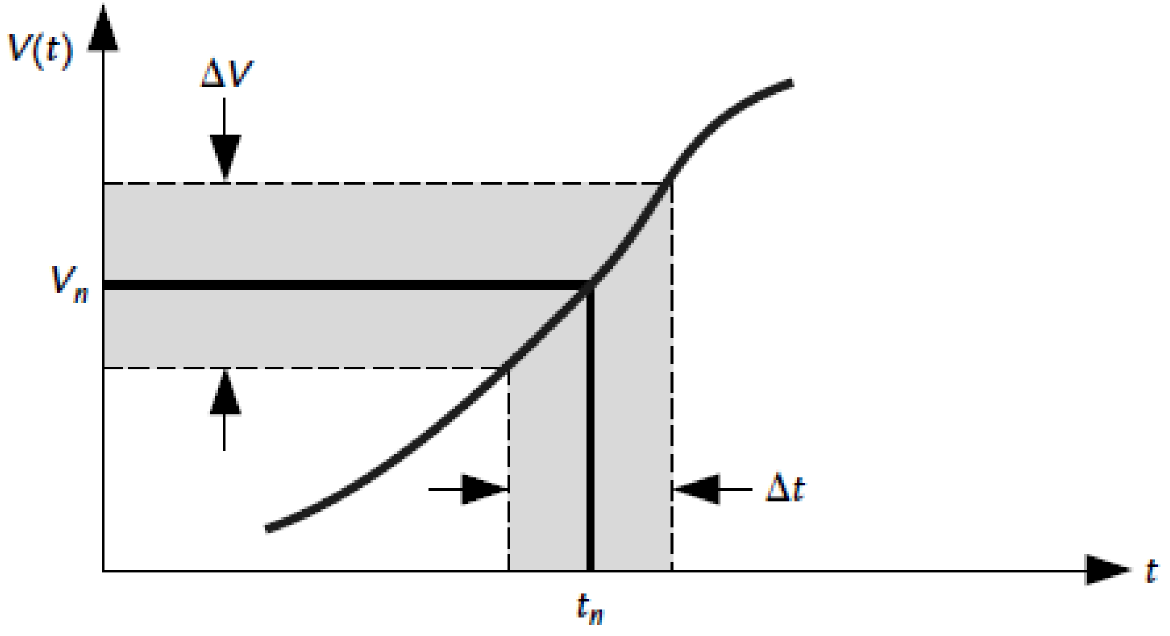
JITTER IN TIMING

- **Accurate samples depend on accurate timing as well as the transfer characteristic. Ideally the input *V(t) is sampled at time t_n when the voltage is Vn*. However, errors in the timing with a spread of (*del*)*t lead to errors in the voltage of V. They* are related by**

$$\Delta V = \left| \frac{dV}{dt} \right| \Delta t.$$

- **Clearly this is more important for rapidly varying signals. Accurate timing requires samples to be triggered by a timer through hardware rather than by software. The successive-approximation ADC10 and ADC12 modules in the MSP430 therefore have internal connections to Timer_A and Timer_B. This is not necessary for the sigma–delta ADCs because they are much slower.**

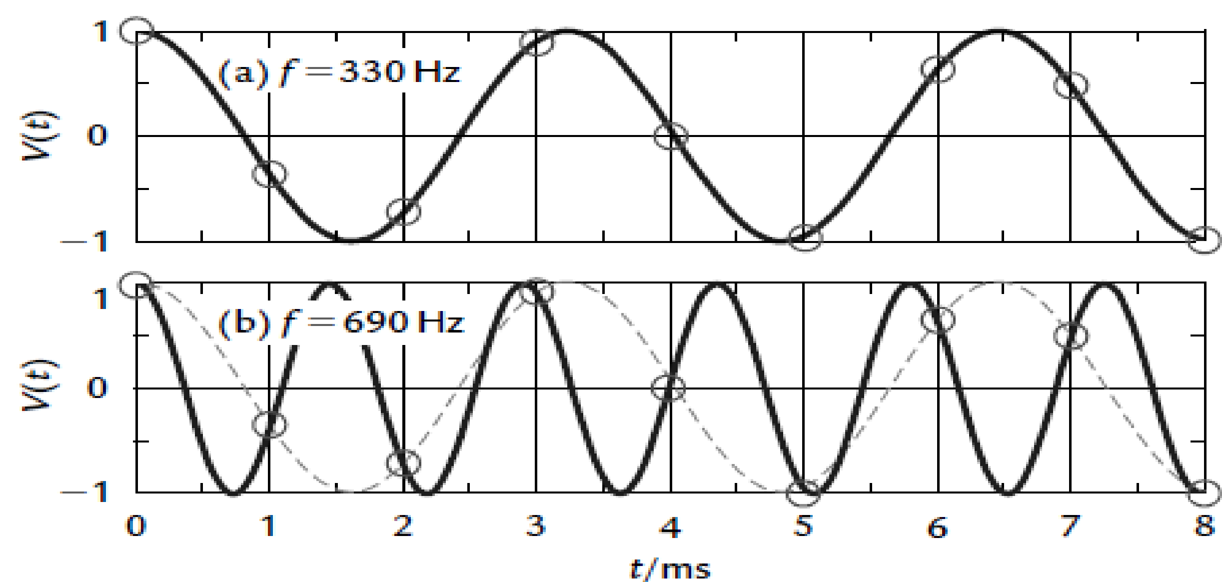
Jitter in Timing



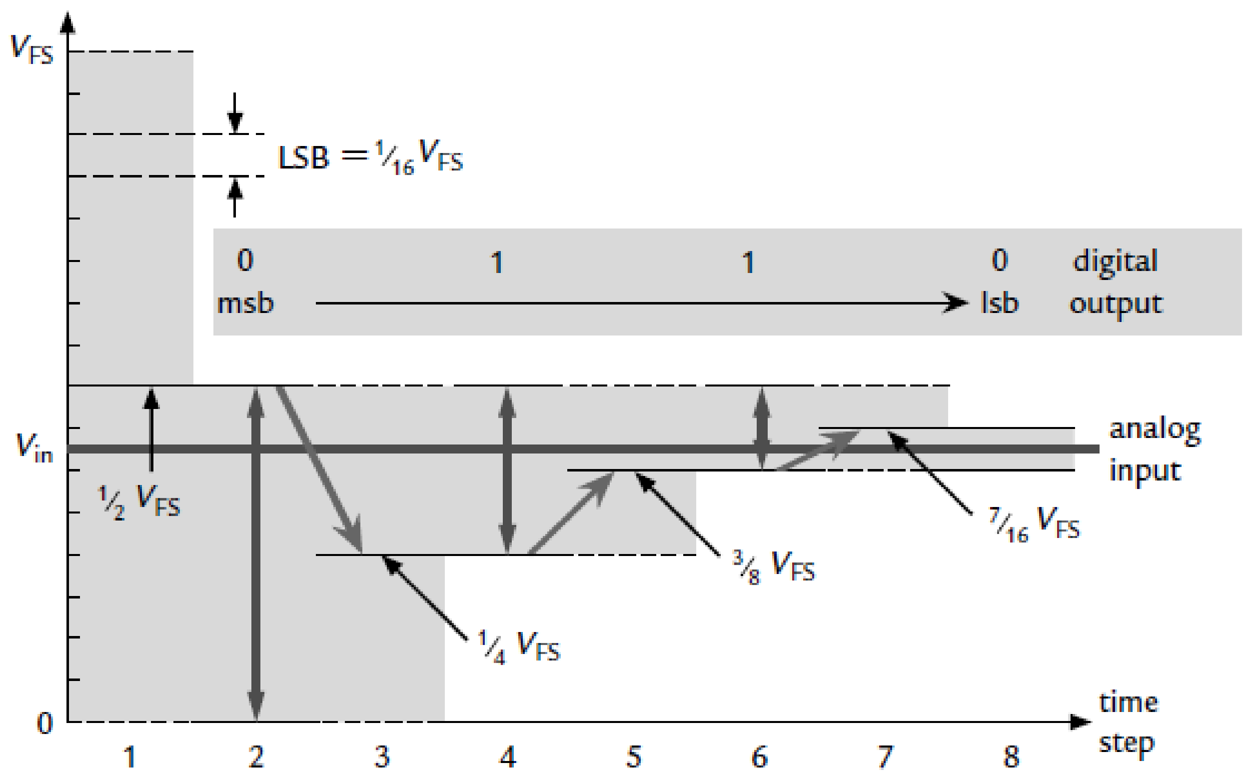
Effect of jitter in timing of sample on voltage recorded.

SAMPLING IN TIME AND ALIASING

- **We saw how finite resolution in voltage—quantization—affects the signal. Now we look at the equivalent problem in time. Let**
  - ***f be the frequency of the signal, assumed to be a simple sine wave.***
  - ***fs be the rate at which it is sampled, with Ts = 1/f_s the interval between samples.***
- **The sampling frequency f_s *is often quoted with units of "samples per second" (sps) rather* than hertz (Hz) but the meaning is the same. Suppose that f_s *= 1ksps to keep the numbers* simple and consider a signal with frequency *f = 310Hz.***
- ***a plot of* the continuous signal and the discrete samples every 1ms. There are no obvious problems.**

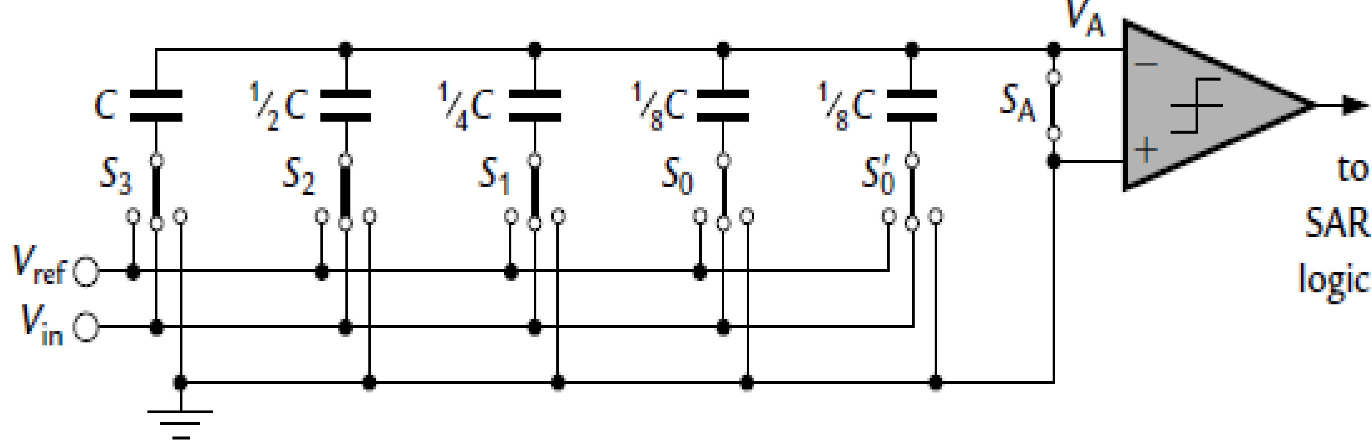


ANALOG-TO-DIGITAL CONVERSION: SUCCESSIVE APPROXIMATION



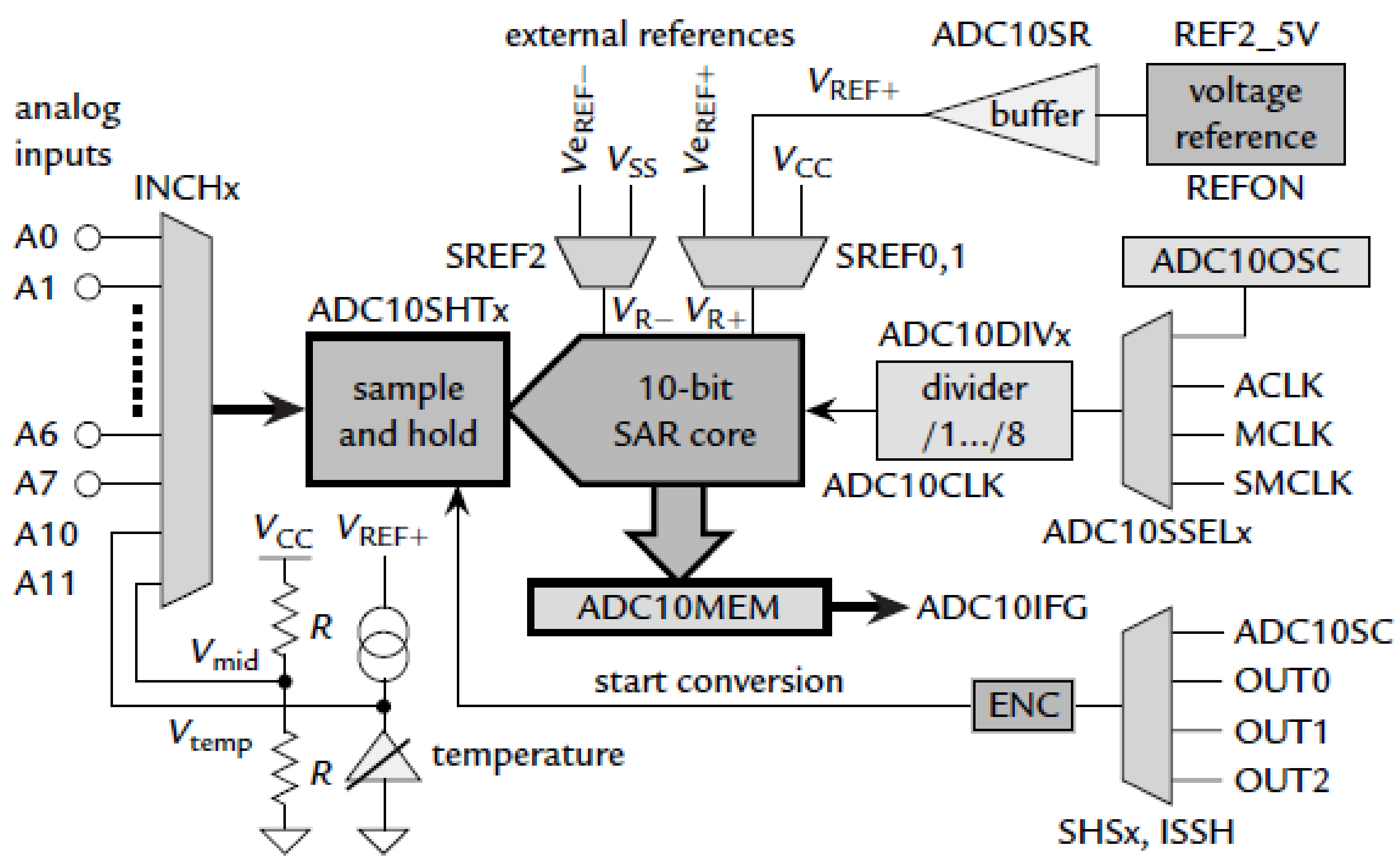
Operation of a 4-bit successive-operation ADC with an input of *Vin = 0.4 V_{FS}*.

Operation of a Switched-Capacitor SAR ADC



Circuit of a 4-bit, charge-redistribution SAR ADC. The switches are in the positions to sample the input.

THE ADC10 SUCCESSIVE-APPROXIMATION ADC



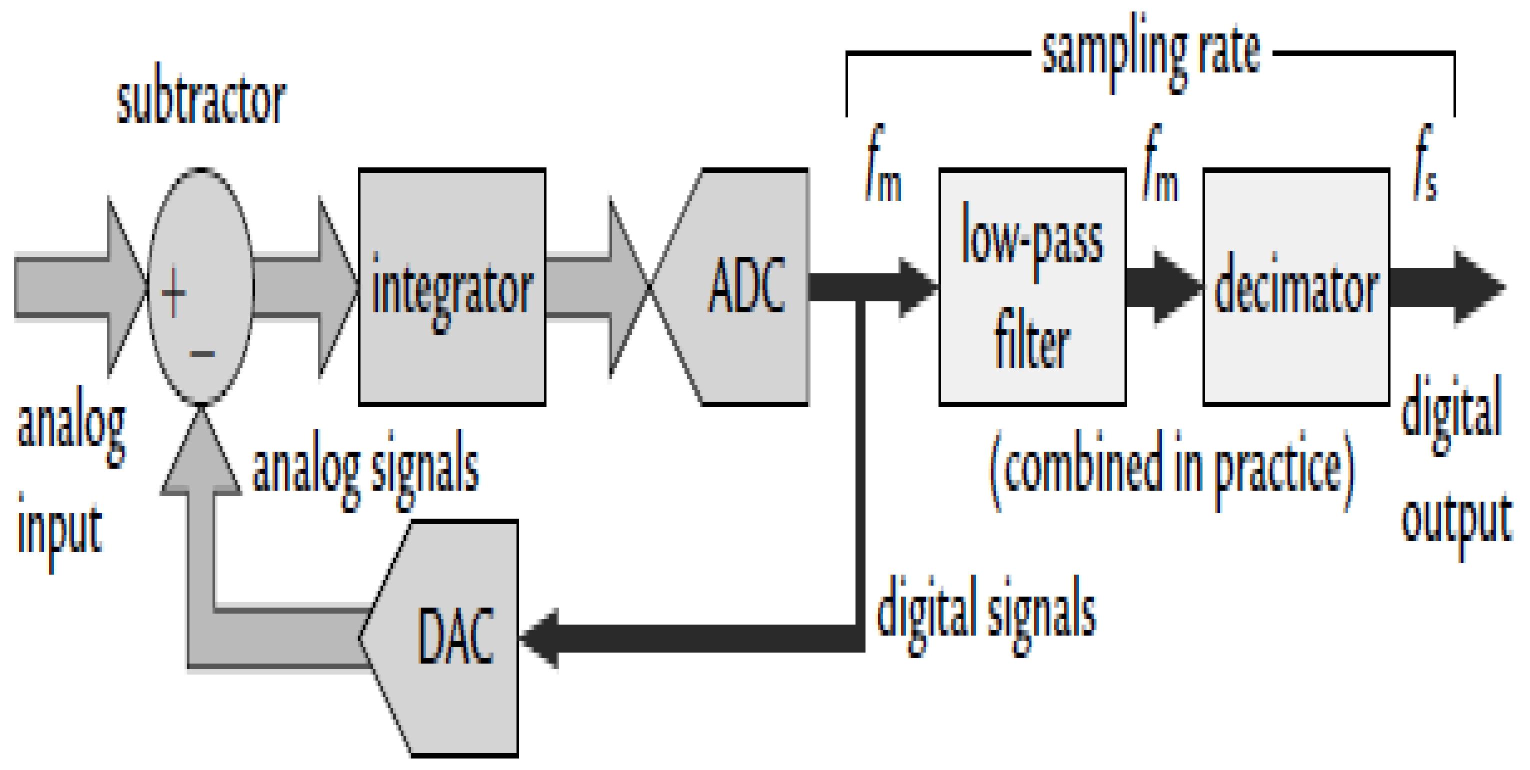
Simplified block diagram of the ADC10. The connections for external references, automatic sequences of conversions, and the data transfer controller are omitted for clarity.

Analog-to-Digital Conversion: Sigma–Delta

Architecture of a Sigma–Delta ADC

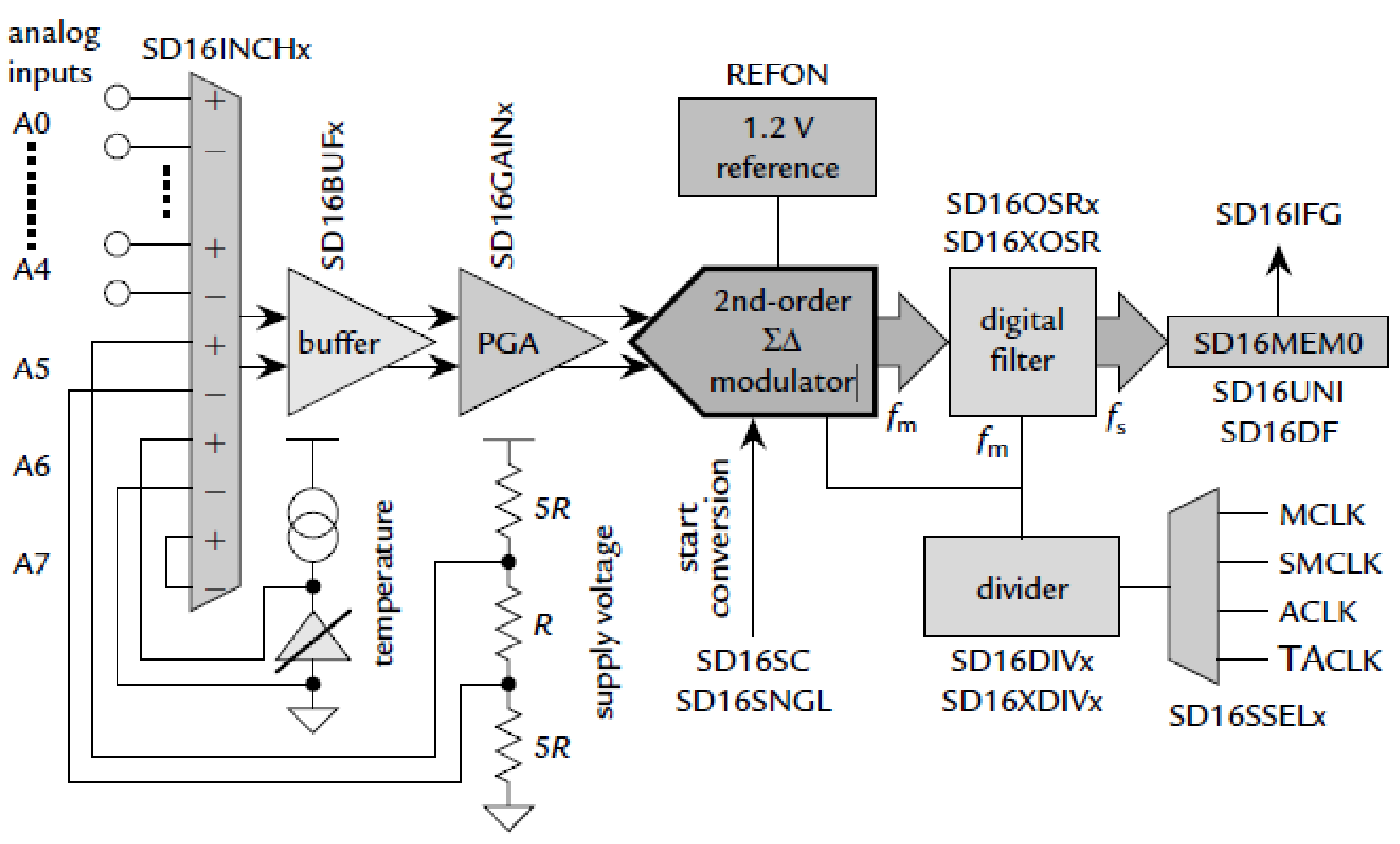
Digital Filters in Sigma–Delta ADCs

The Final Result from a Sigma–Delta ADC

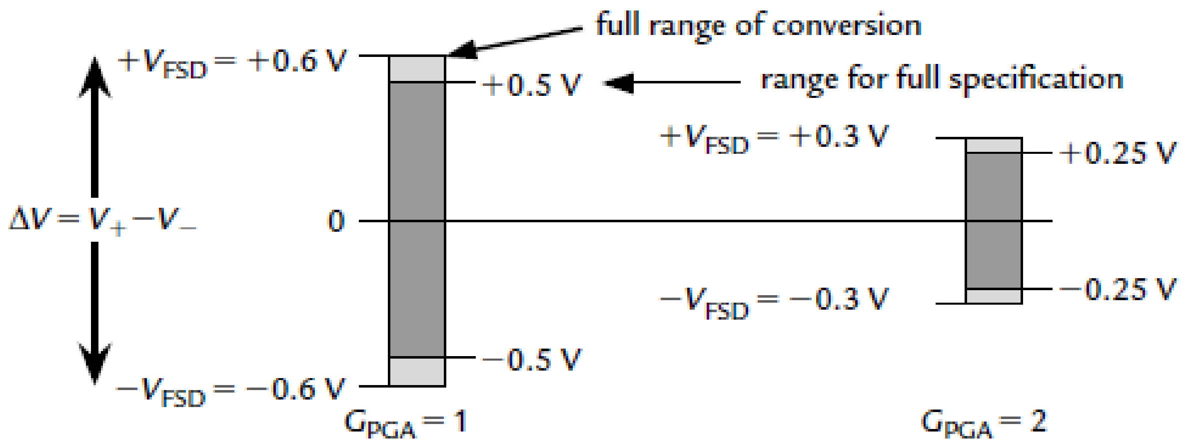


Block diagram of a sigma–delta ADC. The loop forms the sigma–delta modulator, which is followed by a digital filter.

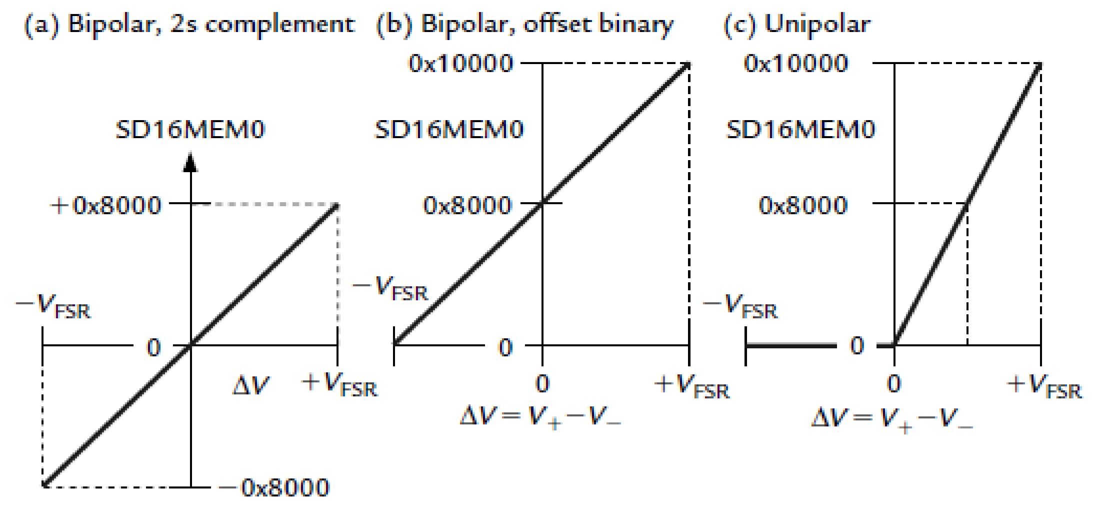
THE SD16_A SIGMA–DELTA ADC



Block diagram of the SD16_A sigma–delta ADC. The external connections for the reference voltage and some aspects of the interrupts are omitted for clarity.



RANGE OF INPUTS TO THE SD16 USING ITS INTERNAL 1.2V REFERENCE. THE PROGRAMMABLE GAIN AMPLIFIER IS SET TO *GPGA = 1* AND *2*.



Output formats from the SD16: (a) bipolar twos complement, (b) bipolar offset binary, and (c) unipolar.

THANK YOU

KUPPAM ENGINEERING COLLEGE

ONLINE COURSE : COVID-19 ZOOM VIDEO CLASSES

SUBJECT : MICROPROCESSOR & MICROCONTROLLER

BY

Dr. K. RASADURAI
Department of Electronics and Communications Engineering
KUPPAM ENGINEERING COLLEGE,
Kuppam – 517425, Chittoor Dist., Andhra Pradesh

Content

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

SERIAL COMUNICATION INTERFACE

Interfacing for Data Communication between Processors & Digital Peripherals:

- Communication Port
  - Collection of signal wires: data, handshake/control/status, clock
- Data Communication Modes
  - Parallel: Several data bits at a time
  - Serial: Single data bit at a time
- Serial Communication Modes
  - Asynchronous communication (UART, ACIA, SCI, etc.): Clock generated at Tx and Rx with the same nominal value. Clock not transmitted.
  - Synchronous serial communication (SRT, SPI, I2C, etc.): Clock generated by the master; used by Tx & Rx; transmitted using a separate line or by combining it with data (Manchester coding).

SERIAL COMUNICATION INTERFACE..

- Serial Communication Standards
    - Interface Logic Levels
    - Physical Link (cables & connectors)
    - Data Transfer Protocol
    - Bandwidth, Noise, Range
- Communication Devices
    - Data terminal equipment (DTE): computer, terminal, etc.
    - Data communication equipment (DCE): modem, printer, etc.
- Data Frame
  - Non-divisible packet of bits
    - Bit Time: basic time interval, Bit Rate: no. of bits / s
    - Baud Rate: no.of pulses / s
    - Data: information data bits
    - Overhead: start / stop / parity, synchronization messages, etc.
    - Data Bandwidth / Throughput: no. of information bits (excluding overhead) / s

SERIAL COMUNICATION INTERFACE…

Simplex/Duplex Communication

- Simplex: Information transfer in one direction only (excluding status / handshakes, etc.).
- Half-duplex: Information transfer in one direction at a time.
- Full-duplex: Simultaneous bi-directional information transfer.

Communication Logic Levels

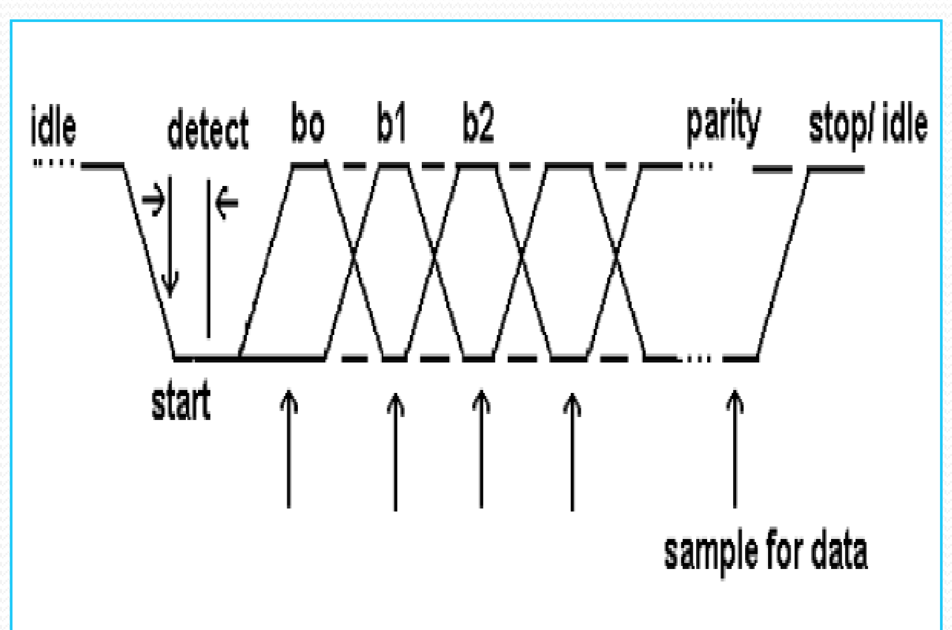
- CMOS (processor port pins): true/mark: ≈ 5V, false/space: <0.1 V.
- RS 232 (drivers): Negative logic, Non-return-to-zero (NRZ), true/mark: –12 V, false/space: +12 V, idle state: true (–12 V).
- Differential voltage (drivers): To reduce the effect of electrostatic interference and ground noise. RS 485: true/mark: –3 V, false/space: +3 V.
- Open collector (processor port pins / drivers): Low & high Z, with passive pull-up.
- Tri-stated (processor port pins / drivers): Low, high, & high Z (idle).
- Current loop (drivers, 4/20 mA): To reduce the effect of inductive interference.
- Opto-coupler: For electrical isolation.

SERIAL COMUNICATION INTERFACE….

- Serial Bus with Multi-drop Network
  - Tri-state logic: Disable the driver after transmission. RTS = 0 for transmission, RTS = 1 after completion. Data: 0-127, Address: 128-255.
  - Collision detection & avoidance: Transmit a frame. Receive it & check for integrity. If collision is detected, wait for a random delay & retransmit.
- Data Transfer Protocols
  - Fixed length messages
  - Message length after the address
  - Special character as terminator
- Interconnection Topology
  - Point to Point , Star, Bus, Ring

SERIAL COMUNICATION INTERFACE…..

- **Asynchronous Communication**
  - No clock transmission. Only data & handshake lines. Tx & Rx use local clocks with same nominal value, not synchronized.
  - Transmission: Idle state, start bit, data bits, (parity, error correction bits), stop bit(s) / idle state. Reception: Detect start (1 → 0) transition,wait 1/2 bit time, sample the input at bit time intervals.



- **Clock tolerance**

  Tx bit time = Tb

  Rx bit time = Tb+Δ

  Cumulative error = NΔ < 0.5Tb

(N = No. of bits (including start, excluding stop/idle)

⇒ For N=10, Δ/Tb < 5%.

- **Baud rate:**

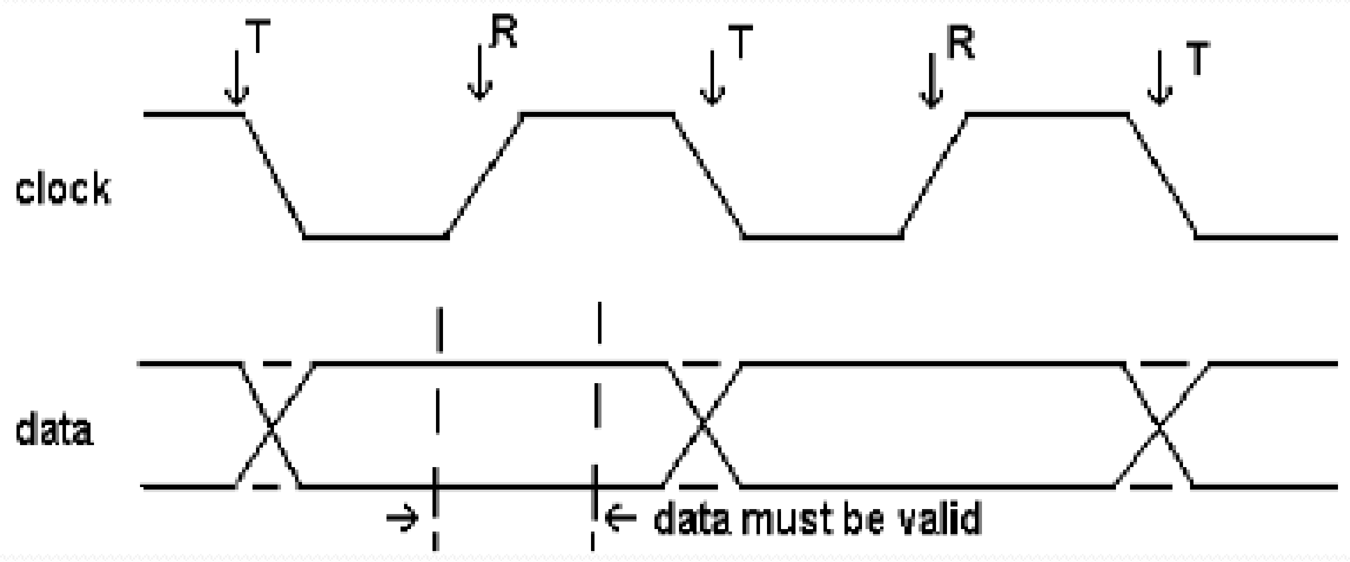
  Limited by clock tolerance.

- **Throughput (data bandwidth) for a given baud rate:**

  Low due to overheads per frame and small frame size.

SERIAL COMUNICATION INTERFACE……

- **Synchronous Communication**
  - **Tx & Rx use clock generated by the master.**
    - Output at one clock edge (falling) & input at the other edge (rising).
    - Signal lines: Data, Clock, [Select] (Clock & data may be combined)
    - Clock [& select] generated by the master
    - Drivers may be needed
    - No basic restriction on frame length



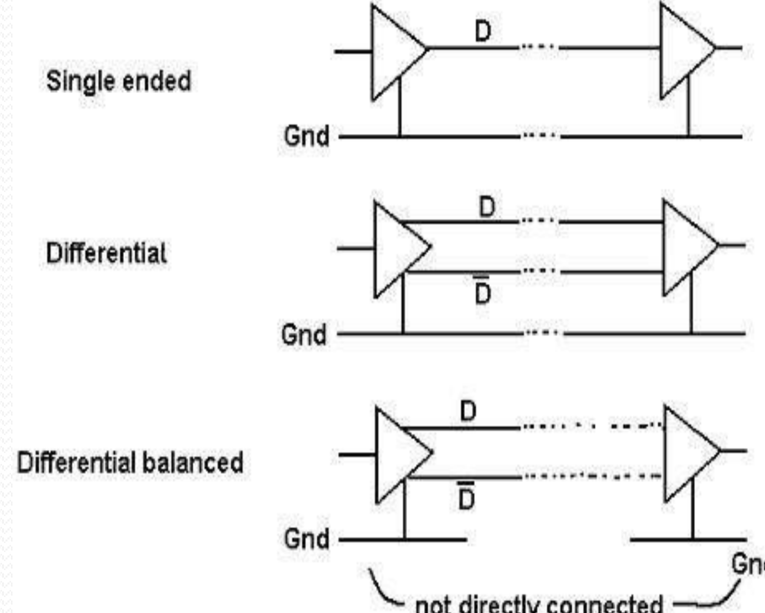
SERIAL COMUNICATION INTERFACE…….

- **Interface Cables & Connectors**
  - **Cables**
    - **Parallel wires**
      - Higher possibility of interference between lines carrying signal in opposite directions. Ground between critical lines.
      - Suited for short distance, high throughput.
    - **Shielded cable**
      - Shield connected to frame ground at one end (signal ground → power supply ground)
      - Reduced RF & electrostatic interference
    - **Twisted pair**
      - Reduced inductive pick-up
      - Baud rate limited due to increased capacitive loading
- **Connectors**
  - DB25 / RS232: 1-13,14-25; DB9 / E1A-574: 1-5, 6-9; RJ45: 1-8.

SERIAL COMUNICATION INTERFACE.......

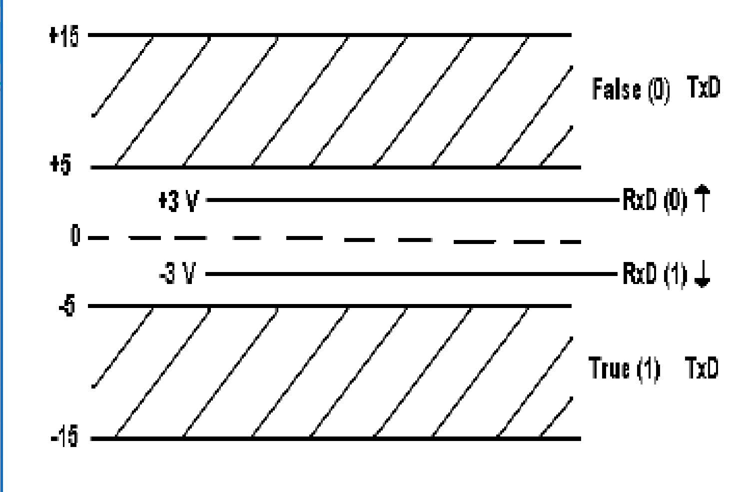
- **Serial Interface Standards**
- **Problem to be tackled**
  - **Signal attenuation**
  - **Pulse transition delay / double pulsing due to reflection**
  - **Interference between signal lines**
  - **External pick up Difference in ground potential**
  - **Over voltage & Over current**

- **Some solutions**
  - **Large voltage or current levels**
  - **Trapezoidal pulses**
  - **Matched termination**
  - **Use of current loop**
  - **Differential voltage transmission**
  - **Special cables: Shielded, Grounded shielded, Twisted pair**



SERIAL COMUNICATION INTERFACE……..

- RS 232 Serial Link
  - Negative, Non-return to zero (NRZ) logic
  - Tx: — 5V to —

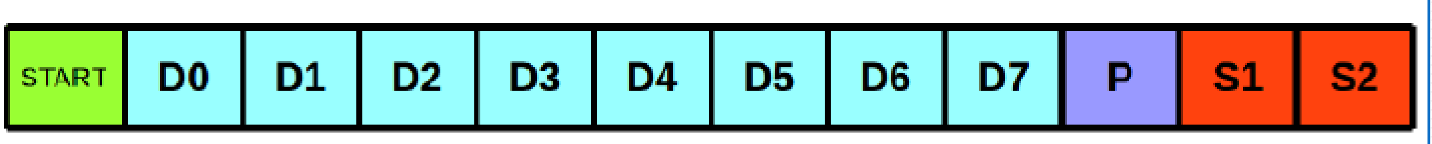


Name
Data
1
Signal Level
Mark
Start Bit
LSB
MSB
Parity Bit
Stop Bits
V_{CC}
Space
0V
0

RS232 Format (Levels at the MSP430)

Name
Data
0
Signal Level
Space
>+3V
Start Bit
LSB
MSB
Parity Bit
Stop Bits
Mark
<–3V
1

RS232 Format (Levels on the Transmission Line)

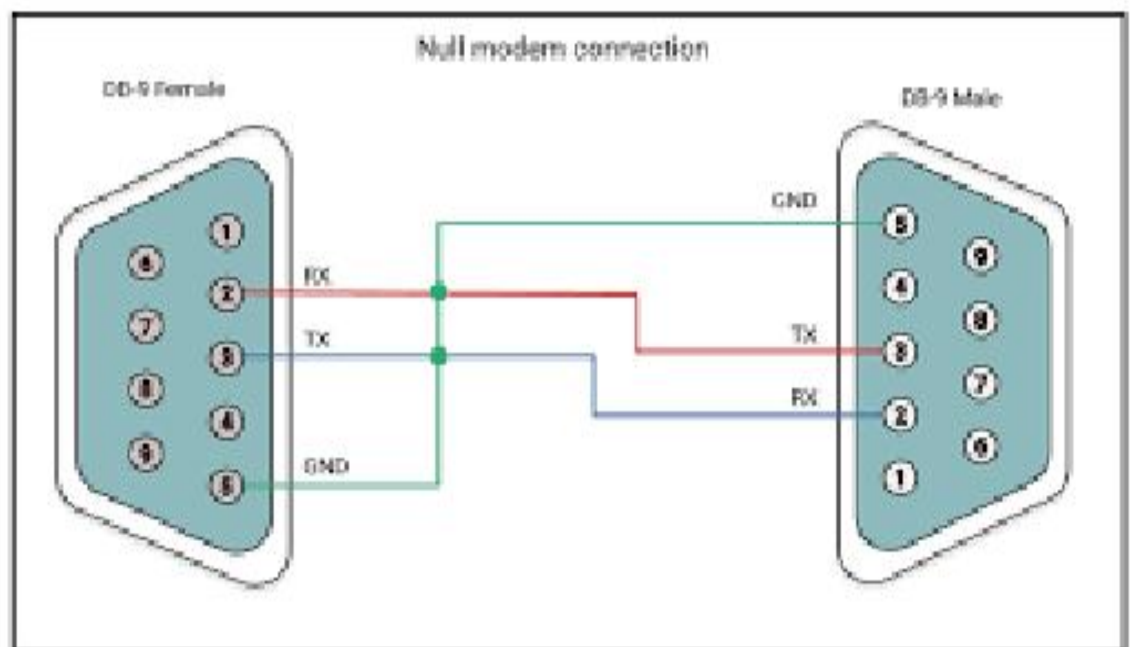


SERIAL COMUNICATION INTERFACE……..

- Signaling
    - Single ended link
    - Shielded cable with shield connected to frame ground at DTE.
    - Signal ground connected to power supply ground at both ends.

Pin #	Signal
1	DCD
2	RX
3	TX
4	DTR
5	GND
6	DSR
7	RTS
8	CTS
9	RI

- Connectors
- DB25 (RS 232): 25 pins, 21 signals
- DB9 (EIA 574): 9 pins, 9 signals
- RJ45 (EIA 561): 8 pins, 8 signals



SERIAL COMUNICATION INTERFACE……..

USCI Overview

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.

The ***USCI_Ax*** modules support:

- UART mode
- Pulse shaping for IrDA communications
- Automatic baud rate detection for LIN communications
- SPI mode

The ***USCI_Bx*** modules support:

- I2C mode
- SPI mode

SERIAL COMUNICATION INTERFACE……..
COMMUNICATION MODULE COMPARISON

USART	USCI NEW	USI NEW
UART: - Only one modulator - n/a - n/a - n/a	**UART:** - Two modulators support n/16 timings - Auto baud rate detection - IrDA encoder & decoder - Simultaneous USCI_A and USCI_B (2 channels)	- - -
SPI: - Only one SPI available - Master and Slave Modes - 3 and 4 Wire Modes	**SPI:** - Two SPI (one on each USCI_A and USCI_B) - Master and Slave Modes - 3 and 4 Wire Modes	**SPI:** - Only one SPI available - Master and Slave Modes
I2C: *(on '15x/'16x only)* - Master and Slave Modes - up to 400kbps	**I2C:** - Simplified interrupt usage - Master and Slave Modes - up to 400kbps	**I2C:** - SW state machine needed - Master and Slave Modes

SERIAL COMUNICATION INTERFACE……….

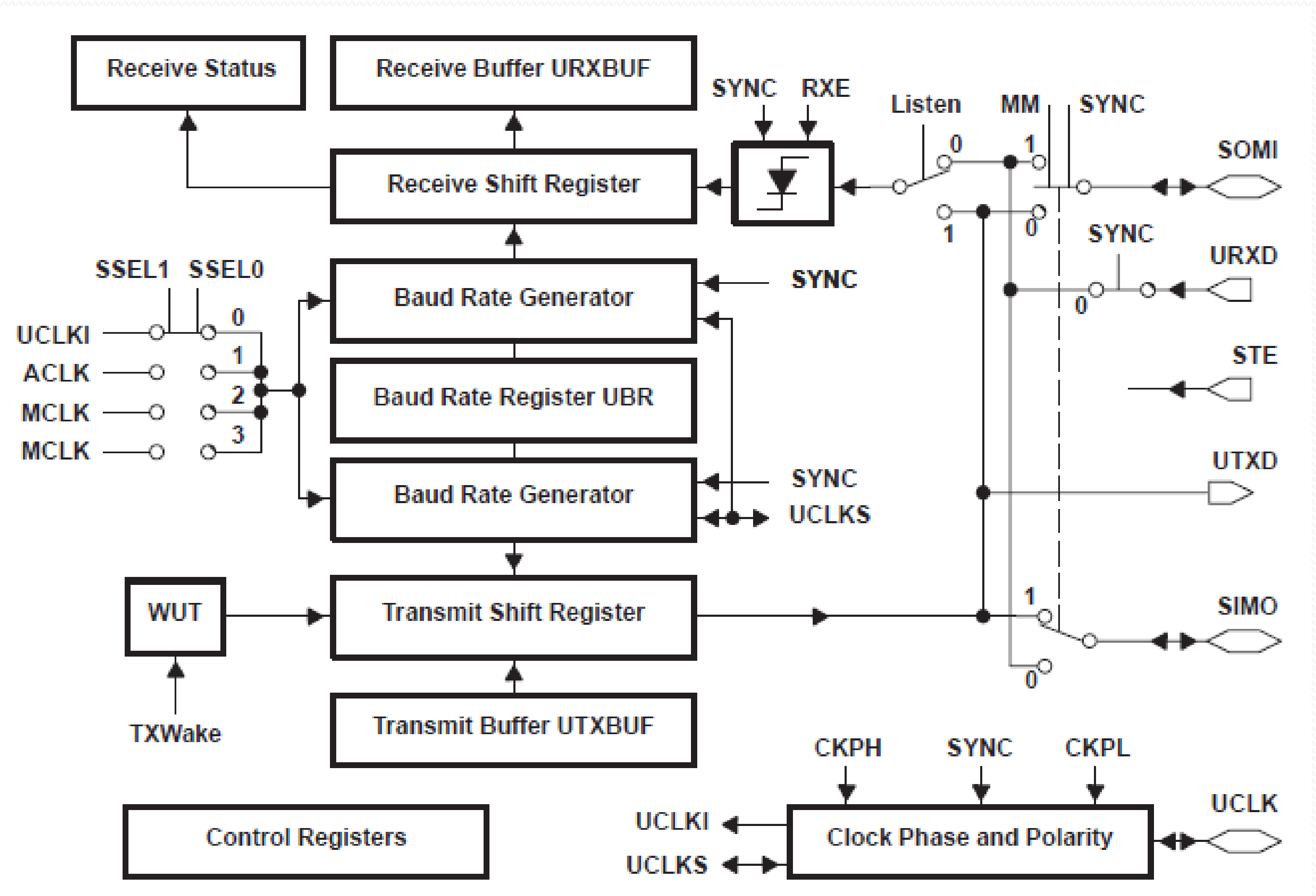
- **USCI Introduction: SPI Mode**
- **In synchronous mode, the USCI connects the MSP430 to an external system via three or four pins: UCxSIMO, UCxSOMI, UCxCLK, and UCxSTE. SPI mode is selected when the UCSYNC bit is set and SPI mode (3-pin or 4-pin) is selected with the UCMODEx bits.**

- **SPI mode *features* include:**
  - **7- 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**

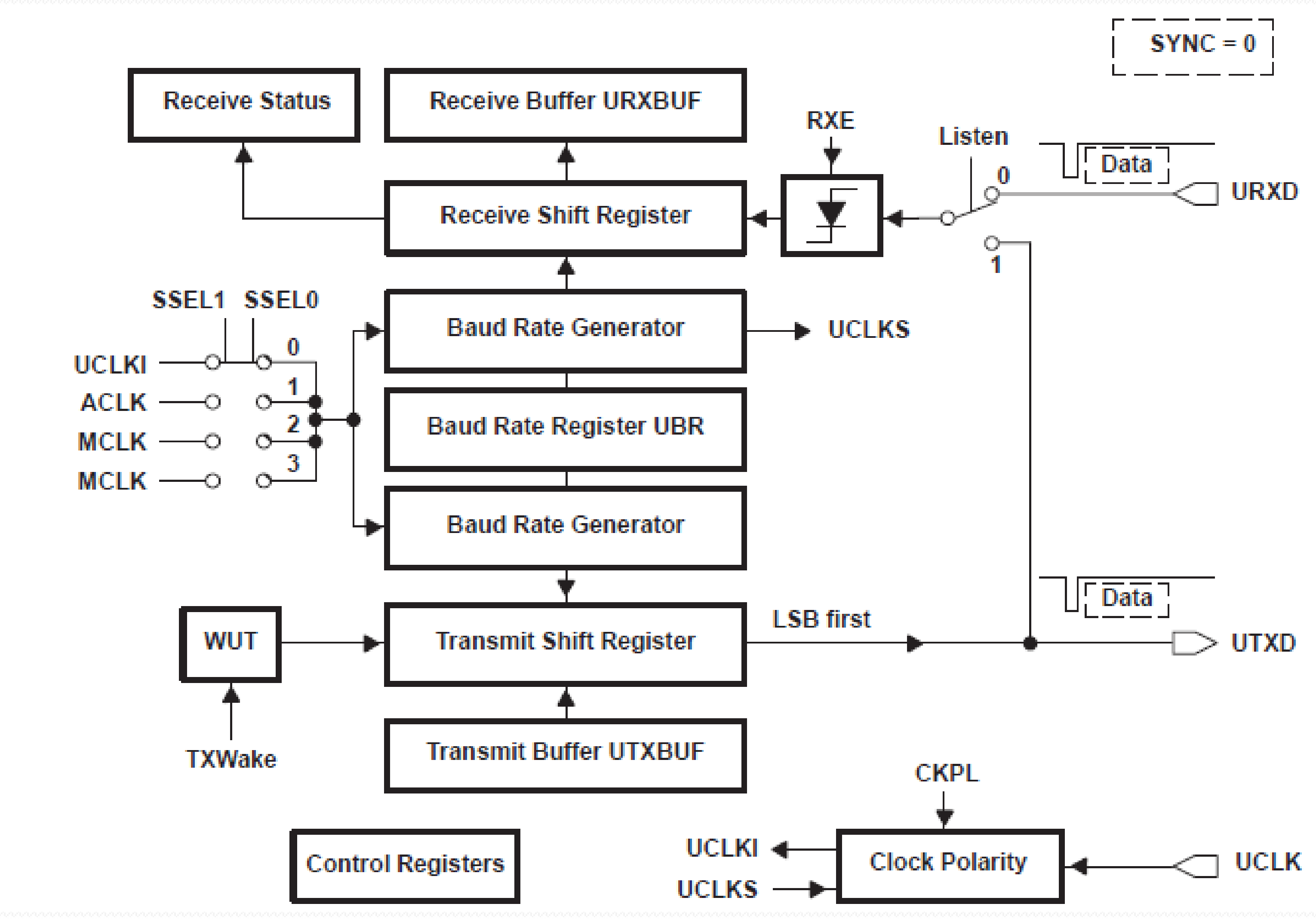
Communication Peripherals in the MSP430

- Universal Serial Interface
- Universal Serial Communication Interface
  1. Asynchronous channel, USCI_A
  2. Synchronous channel, USCI_B
- Universal Synchronous/Asynchronous Receiver/Transmitter
- Bit-Banging
  a. Synchronous masters
  b. Synchronous slaves
  c. Asynchronous communication

MSP430 USART Module



USART Switched to the UART Mode



USART Control Registers Used in the UART Mode

Register Name	Mnemonic	Register Access	Address	Initial State
USART Control Register	UCTL	Read/Write	070h	See below
Transmit Control Register	UTCTL	Read/Write	071h	See below
Receive Control Register	URCTL	Read/Write	072h	See below
Modulation Control Register	UMCTL	Read/Write	073h	unchanged
Baud-Rate Register 0	UBR0	Read/Write	074h	unchanged
Baud-Rate Register 1	UBR1	Read/Write	075h	unchanged
Receive Buffer	URXBUF	Read Only	076h	unchanged
Transmit Buffer	UTXBUF	Read/Write	077h	unchanged

UCTL 070h	7							0
	PENA	PEV	SP	CHAR	Listen	SYNC	MM	SWRST
	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

UBR0 074h	7							0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^1
	rw	rw	rw	rw	rw	rw	rw	rw

UTCTL 071	7							0
	un-used	CKPL	SSEL 1	SSEL0	URXSE	TXWake	unused	TXEPT
	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

UBR1 075h	7							0
	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8
	rw	rw	rw	rw	rw	rw	rw	rw

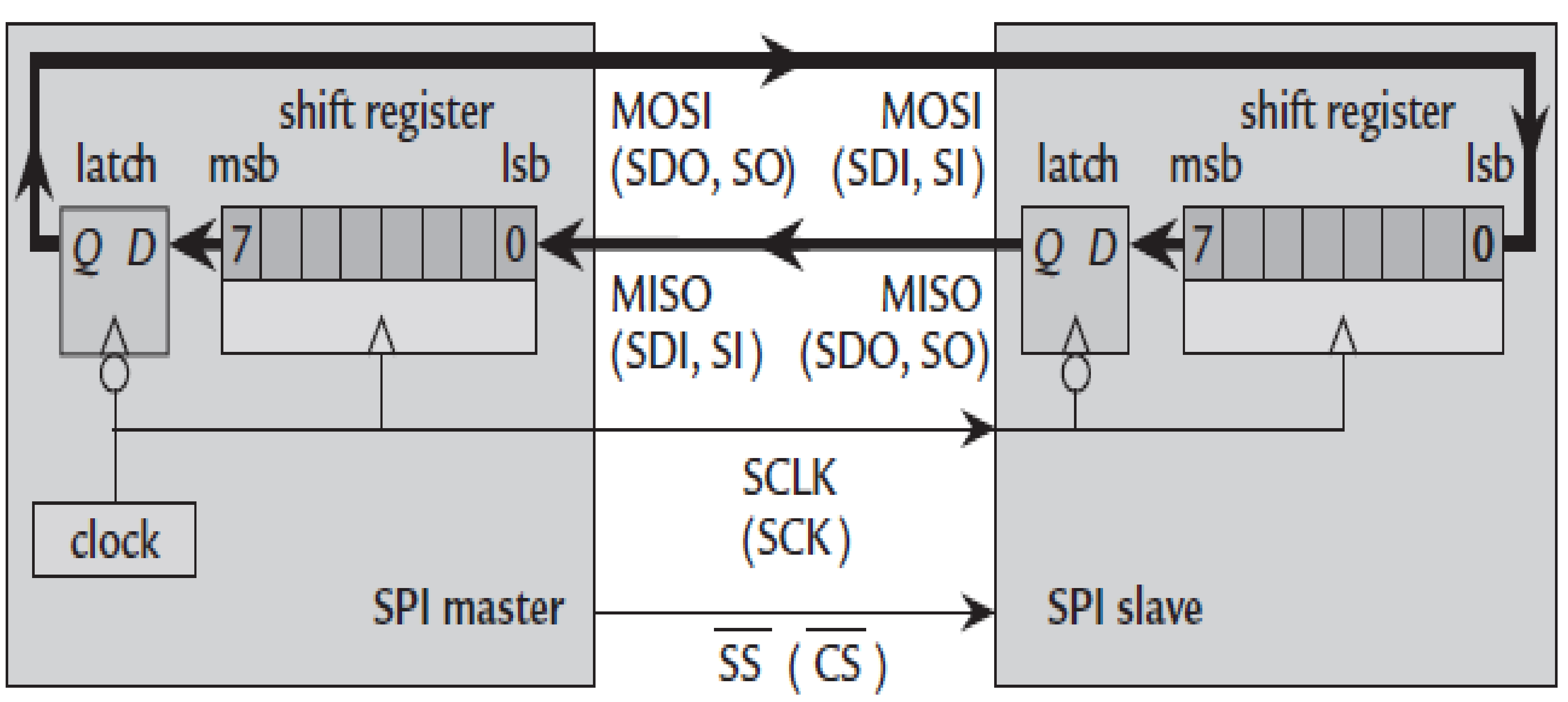
URCTL 072h	7							0
	FE	PE	OE	BRK	URXEIE	URXWIE	RXWake	RXERR
	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

UMCTL 073h	7							0
	m7	m6	m5	m4	m3	m2	m1	m0
	rw	rw	rw	rw	rw	rw	rw	rw

URXBUF 076h	7							0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	r	r	r	r	r	r	r	r

UTXBUF 077h	7							0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	rw	rw	rw	rw	rw	rw	rw	rw

SERIAL PERIPHERAL INTERFACE (SPI)



Serial peripheral interface between a master and a single slave

SERIAL PERIPHERAL INTERFACE (SPI)

SPI requires :

- four wires (plus ground) and

- transmits data simultaneously in both directions (full duplex) between two devices.

- "master in, slave out" (MISO) and

- "master out, slave in" (MOSI).

- Other terms are widely used, such as SDI, SI, or DIN for serial data in and SDO, SO, or DOUT for serial data out.

- Clock signal including SCLK, SPSCK, and SCK.

- The final signal selects the slave.

- This is usually active low and labeled SS for slave select, CS for chip select, or CE for chip enable.

- A slave should drive its output only when SS is active; the output should float at other times in case another slave is selected.

- In some modes of SPI, the first bit should be placed on the output when SS becomes active to start a new transfer.

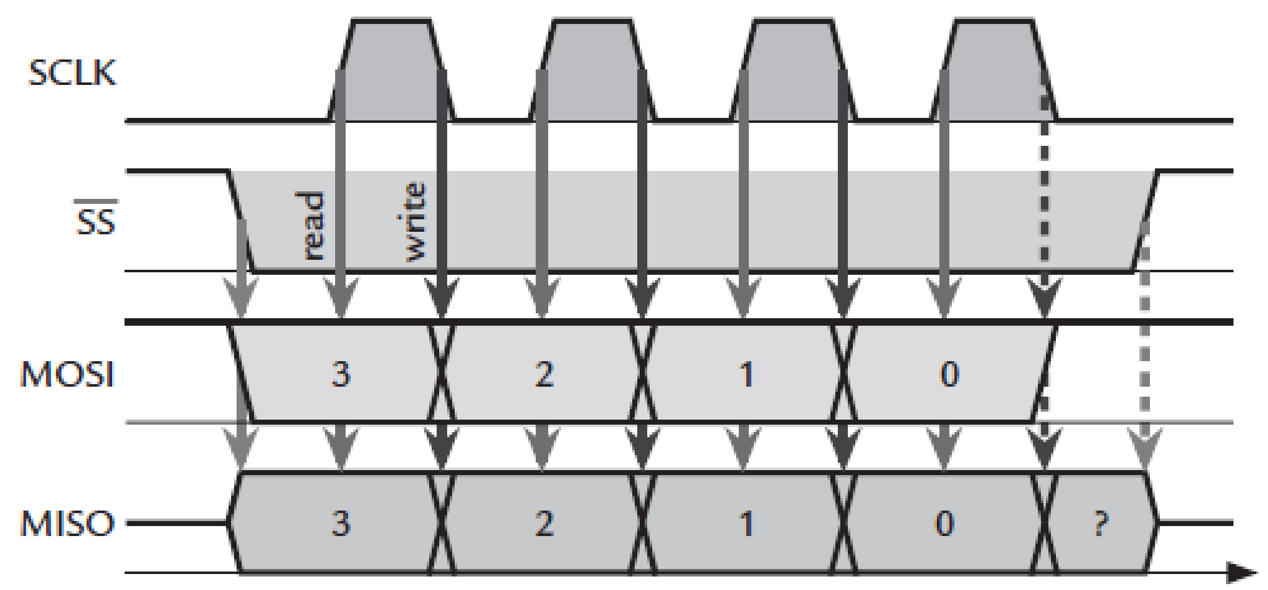
SERIAL PERIPHERAL INTERFACE (SPI)

- The concept of SPI is based on two shift registers, one in each device, which are connected to form a loop.
- The registers usually hold 8 bits.
- Each device places a new bit on its output from the most significant bit (msb) of the shift register when the clock has ***a negative edge*** and reads its input into the lsb of the shift register on ***a positive edge*** of the clock.
- Thus a bit is transferred in each direction during each clock cycle.
- After eight cycles the contents of the shift registers have been exchanged and the transfer is complete.
- Transmission and reception are clearly inseparable: Thus a byte must be transmitted in order to receive a byte.

SERIAL PERIPHERAL INTERFACE (SPI)

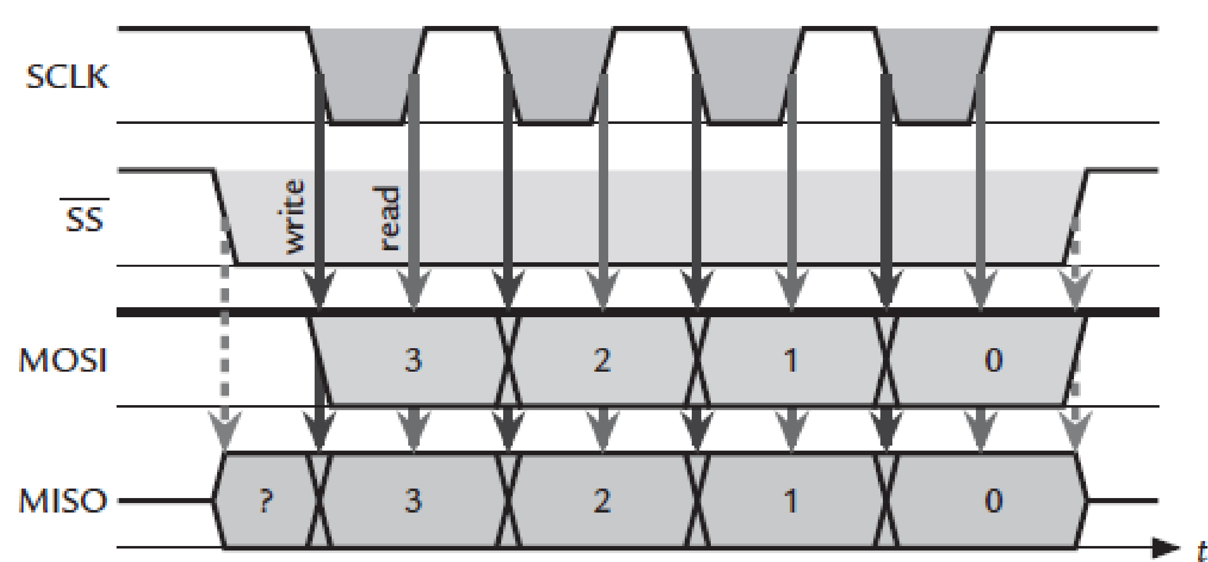
- For example, an external DAC configured as an SPI slave may never need to return digital data to a microcontroller so there is no need for the MISO connection.
- However, the same steps always take place internally: The only difference is that the output of the DACs shift register never leaves the chip.
- This contrasts with asynchronous communication, where transmission and reception are independent.
- The SS line can sometimes be omitted if only two devices are connected, in which case the slave's SS pin should be tied to ground so that it is enabled permanently.

SERIAL PERIPHERAL INTERFACE (SPI)



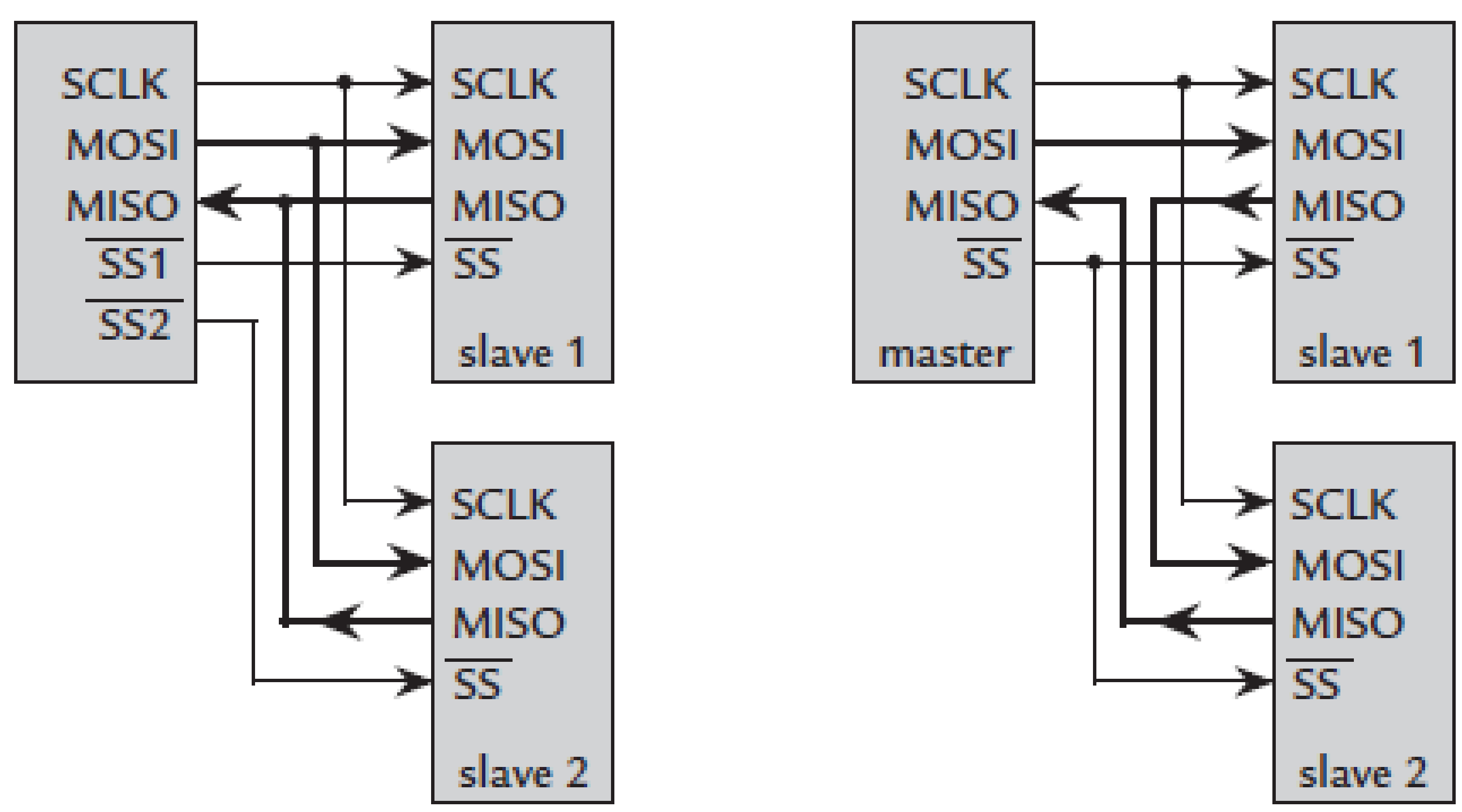
A complete transfer of 4 bits using SPI in mode 0 (CPHA = 0, CPOL = 0).

A complete transfer of 4 bits using SPI in mode 3 (CPHA = 1, CPOL = 1)



CPHA - *clock phase* bit
CPOL - clock polarity bit
CKPL = CPOL for the polarity
CKPH = CPHA for the phase

SERIAL PERIPHERAL INTERFACE (SPI)

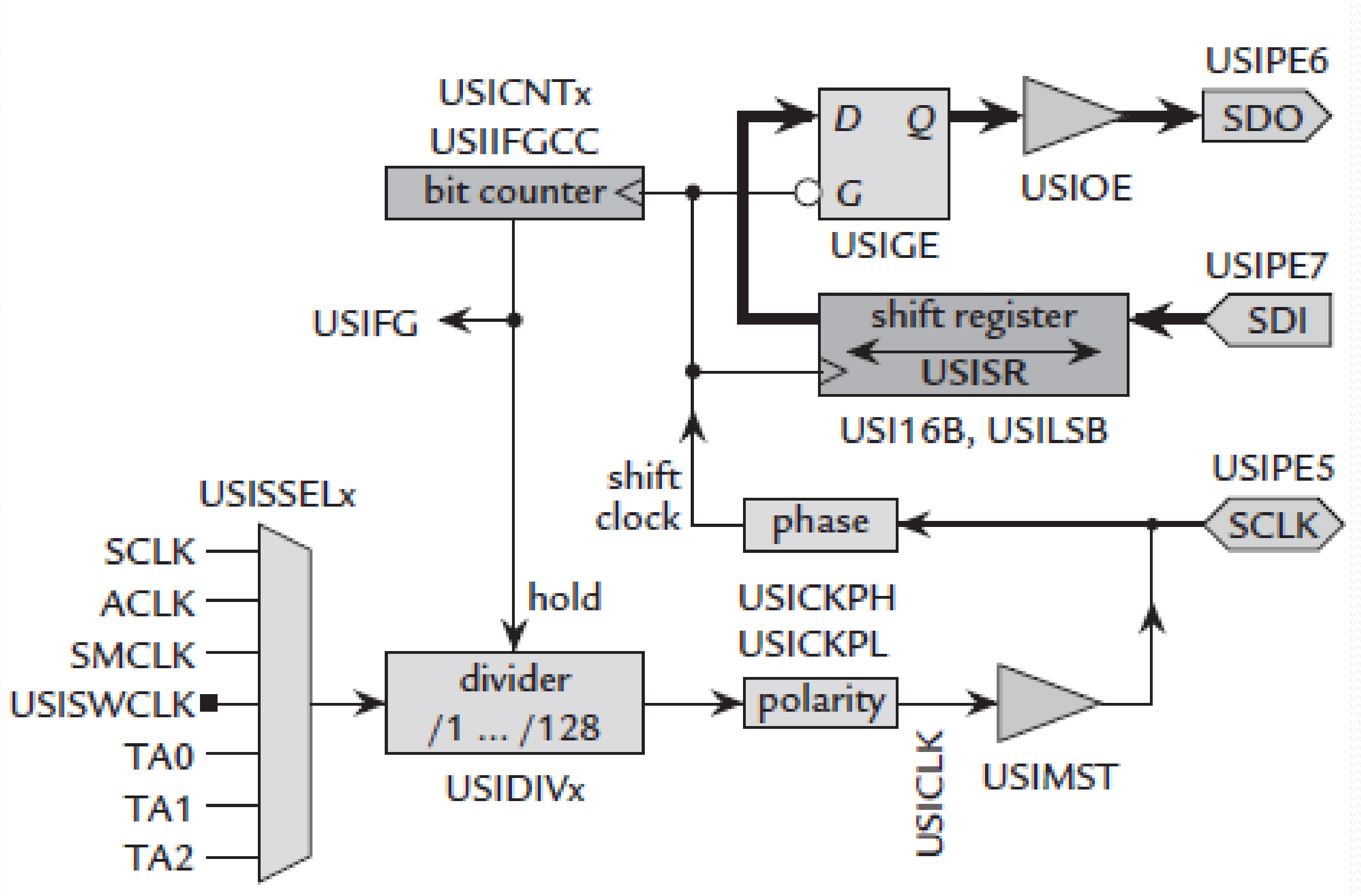


Two ways of connecting two slaves to a single master using SPI.

(a) A slave can be selected individually by providing separate SS lines.

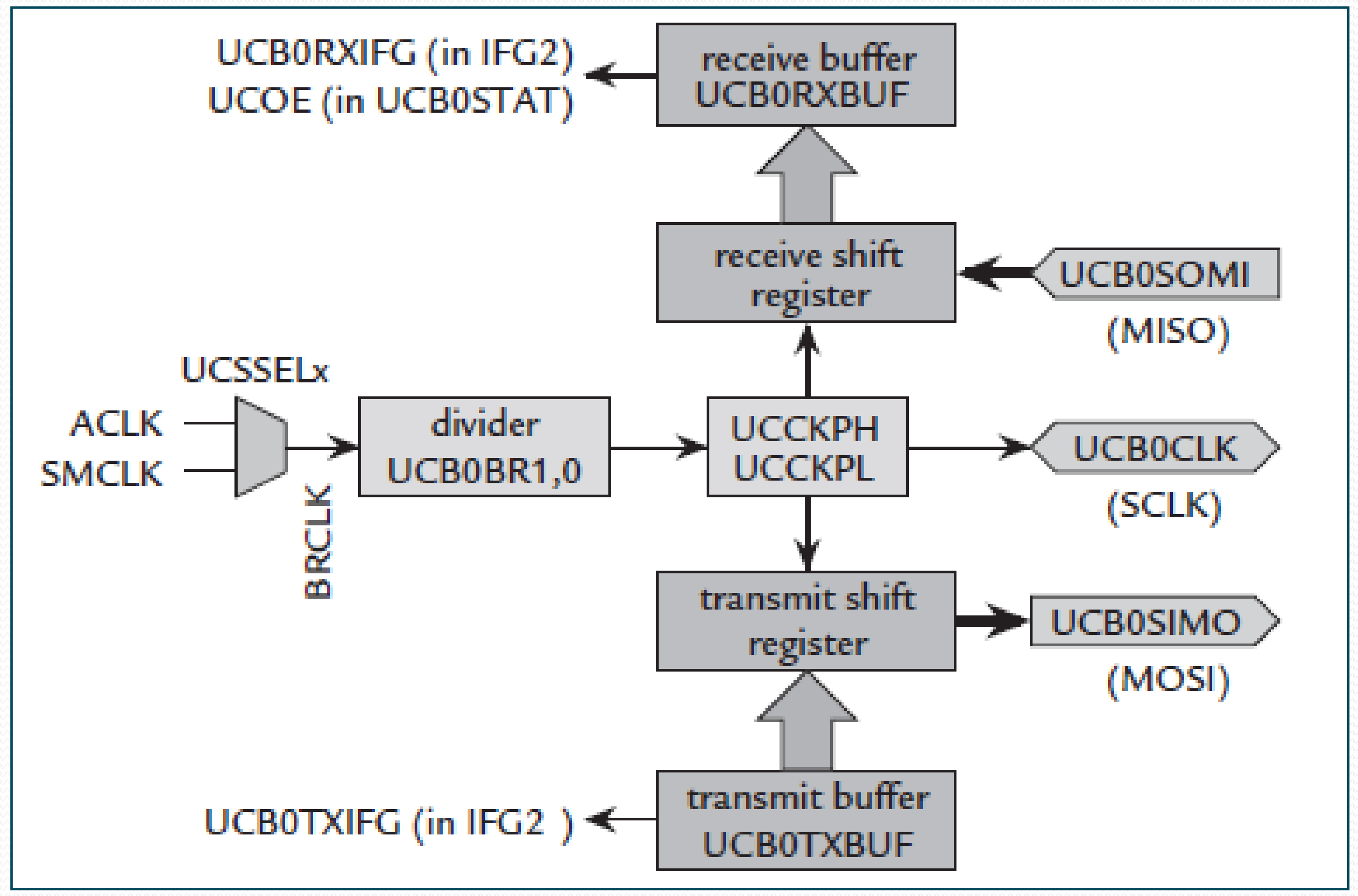
(b) All slaves can be connected in a "daisy chain," in which case they must all be updated together.

SERIAL PERIPHERAL INTERFACE (SPI) with USI



Simplified block diagram of the USI in SPI mode (USII2C = 0) with the principal bits that configure it. The path for data through the shift register is emphasized with heavy lines.

SERIAL PERIPHERAL INTERFACE (SPI) with USCI



Simplified block diagram of the USCI_Bo module in SPI master mode

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 Operation: SPI Mode

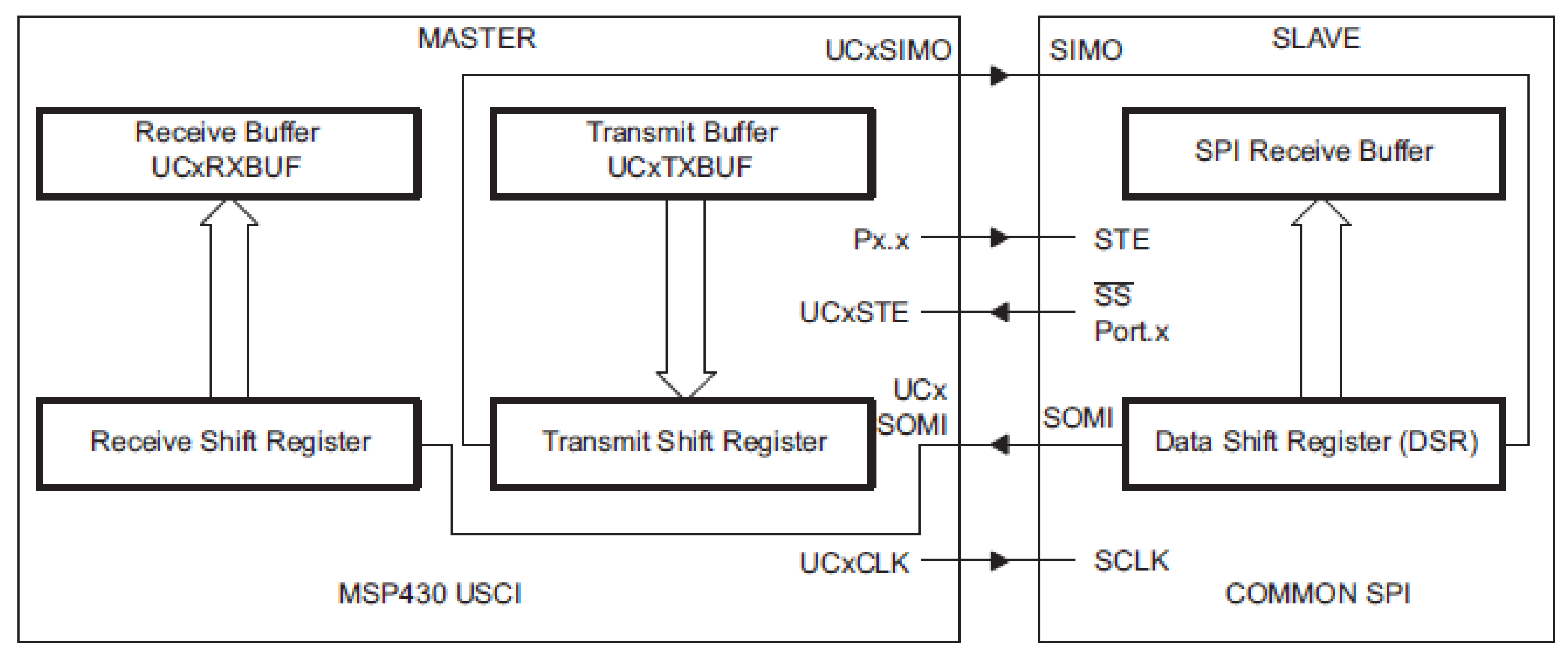
UCMODEx	UCxSTE Active State	UCxSTE	Slave	Master
01	High	0	Inactive	Active
		1	Active	Inactive
10	Low	0	Active	Inactive
		1	Inactive	Active

USCI Operation: SPI Mode

- ***USCI Initialization and Reset***
  - The USCI is reset by a PUC (Power up clear)or by the UCSWRST bit.
- **Initializing or Re-Configuring the USCI Module**
  - The recommended USCI initialization/re-configuration process is:
    1. Set UCSWRST (BIS.B #UCSWRST,&UCxCTL1)
    2. Initialize all USCI registers with UCSWRST=1 (including UCxCTL1)
    3. Configure ports
    4. Clear UCSWRST via software (BIC.B #UCSWRST,&UCxCTL1)
    5. Enable interrupts (optional) via UCxRXIE and/or UCxTXIE
- ***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.

USCI Operation: SPI Mode

- *Master Mode*



USCI Master and External Slave

- 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 TX shift register when the TX shift register is empty, initiating data transfer on UCxSIMO starting with either the most-significant or least-significant bit depending on the UCMSB setting.

USCI Operation: SPI Mode

- 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 RX shift register to the received data buffer UCxRXBUF and the receive interrupt flag, UCxRXIFG, is set, indicating the RX/TX operation is complete.

- A set transmit interrupt flag, UCxTXIFG, 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.

USCI Operation: SPI Mode

- ***Master Mode***

- In 4-pin master mode, UCxSTE is used to prevent conflicts with another master and controls the master as described in Table 16-1. 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 will be 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 re-written 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.

USCI Operation: SPI Mode

- ***Slave Mode***

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.

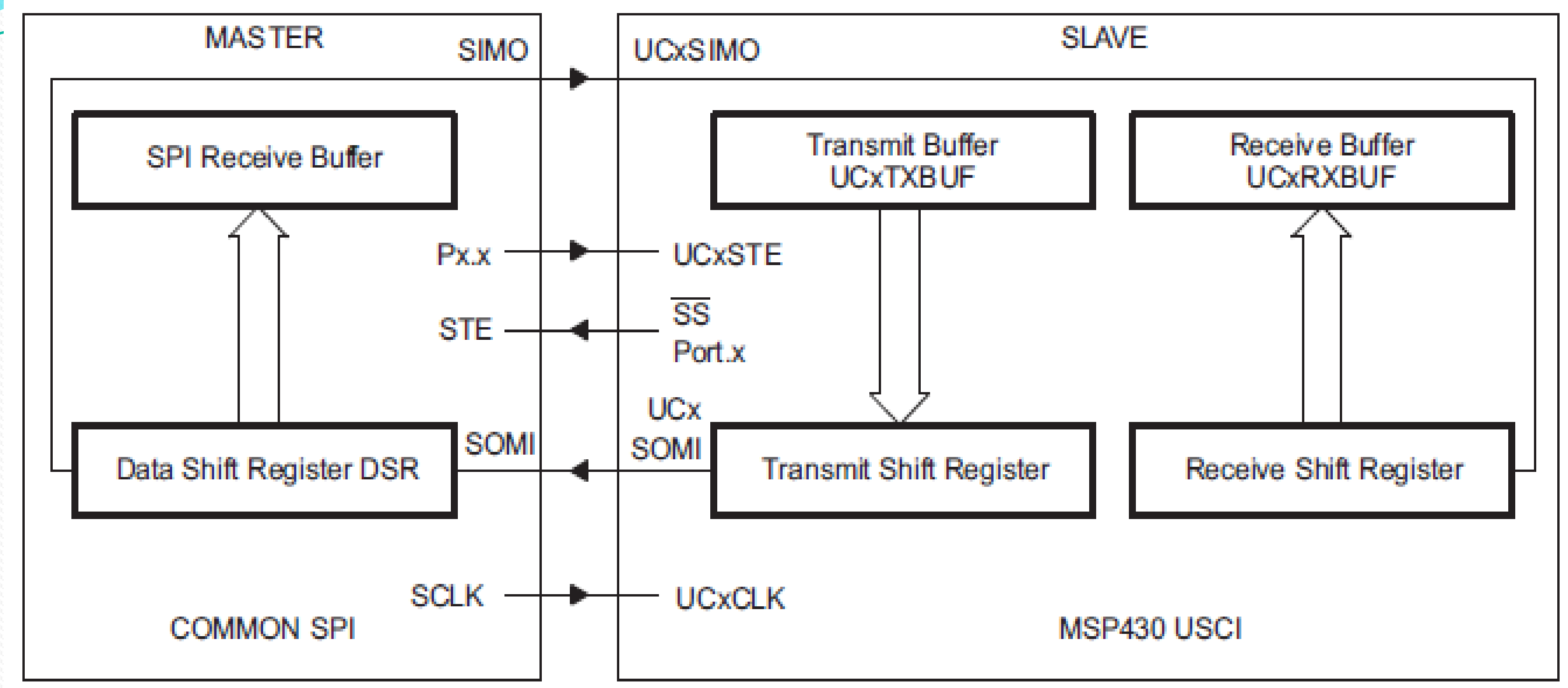
Data written to UCxTXBUF and moved to the TX shift register before the start of UCxCLK is transmitted on UCxSOMI.

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 UCxRXIFG 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.

Slave Mode



USCI Slave and External Master

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

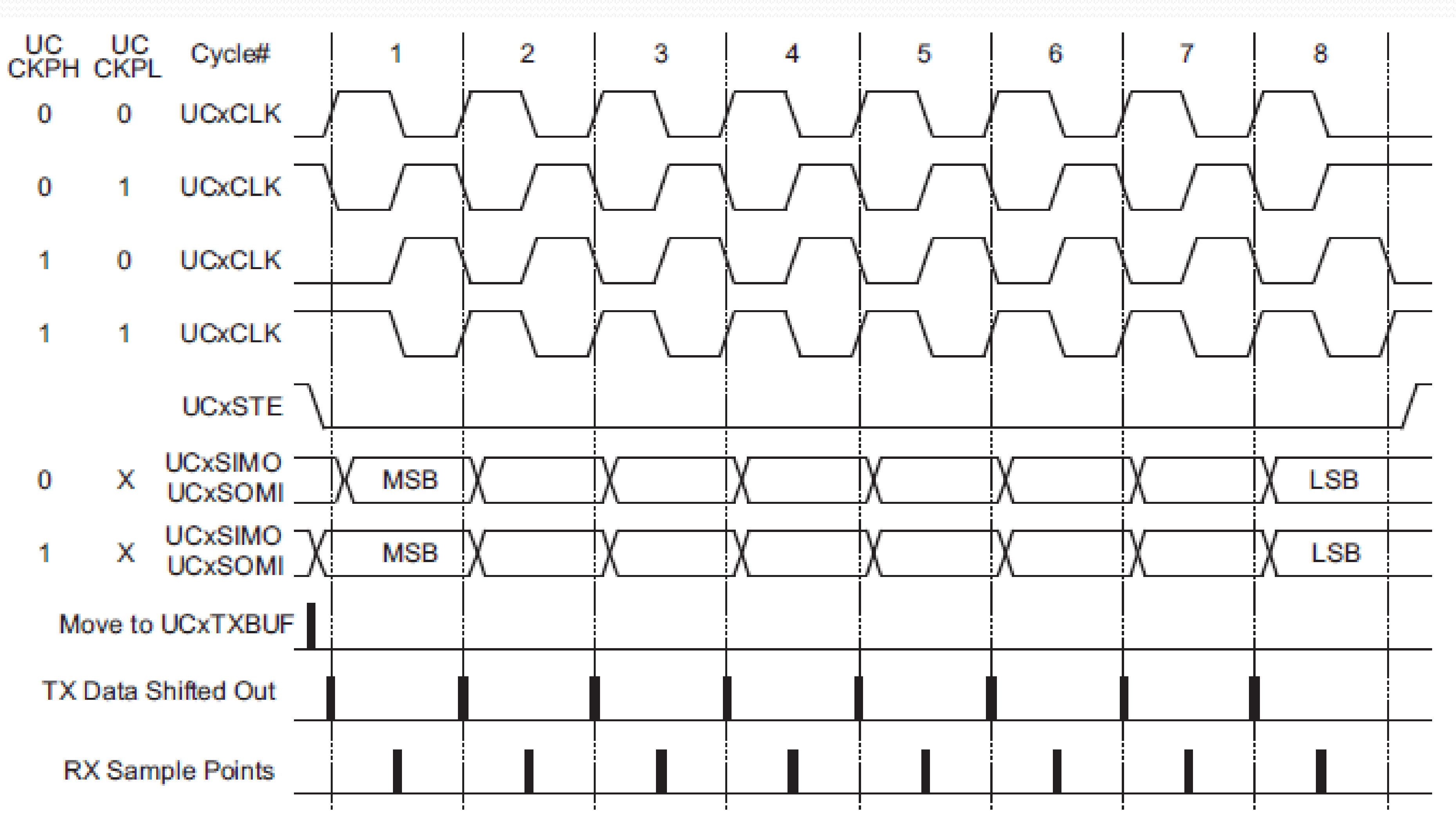
- When the USCI module is enabled by clearing the UCSWRST bit it is ready to receive and transmit. In master mode the bit clock generator is ready, but is not clocked nor producing any clocks. In slave mode the bit clock generator is disabled and the clock is provided by the master.
- A transmit or receive 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 will begin 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

Serial Clock Polarity and Phase



USCI SPI Timing with UCMSB = 1

SPI Interrupts

- SPI Transmit Interrupt Operation
- SPI Receive Interrupt Operation
- USCI Interrupt Usage

Shared Receive Interrupt Vectors Software Example

```
USCIA0_RX_USCIB0_RX_ISR
   BIT.B    #UCA0RXIFG, &IFG2    ; USCI_A0 Receive Interrupt?
   JNZ      USCIA0_RX_ISR
USCIB0_RX_ISR?
   ; Read UCB0RXBUF (clears UCB0RXIFG)
   ...
   RETI
USCIA0_RX_ISR
   ; Read UCA0RXBUF (clears UCA0RXIFG)
   ...
   RETI
```

Shared Transmit Interrupt Vectors Software Example

```
USCIA0_TX_USCIB0_TX_ISR
   BIT.B    #UCA0TXIFG, &IFG2    ; USCI_A0 Transmit Interrupt?
   JNZ      USCIA0_TX_ISR
USCIB0_TX_ISR
   ; Write UCB0TXBUF (clears UCB0TXIFG)
   ...
   RETI
USCIA0_TX_ISR
   ; Write UCA0TXBUF  (clears UCA0TXIFG)
   ...
   RETI
```

Inter-integrated Circuit Bus

- The I^2C bus uses only two bidirectional lines:
  - Serial data (SDA)
  - Serial clock (SCL)

  -of course there must be a connection for ground as well.

  -It is often called the *two-wire interface*.

Thus I^2C provides the full functionality of a bus while using fewer lines than SPI.

The first is that it is slow, only 100 kbit/sec in standard mode, because of the electrical arrangements needed to avoid damage if two nodes attempt to transmit simultaneously.

Second, a protocol must be observed: You cannot merely transmit the data and nothing more, as in SPI. More hardware is needed than a simple shift register and transmissions must be controlled by logic such as a state machine.

This may be implemented either in hardware, as in the USCI_B, or in software for the USI.

Inter-integrated Circuit Bus

- Transfers on the bus take place between a master and a slave.
- Each slave has a unique address, which is usually 7 bits long.
- The master starts the transfer, provides the clock, addresses a particular slave, manages the transfer, and finally terminates it.
- There may be more than one master on the bus although only one can be in control at a time.
- I^2C have only a single master and a few slaves—sometimes just one.
- SPI would be simpler in this case but I^2C saves pins.

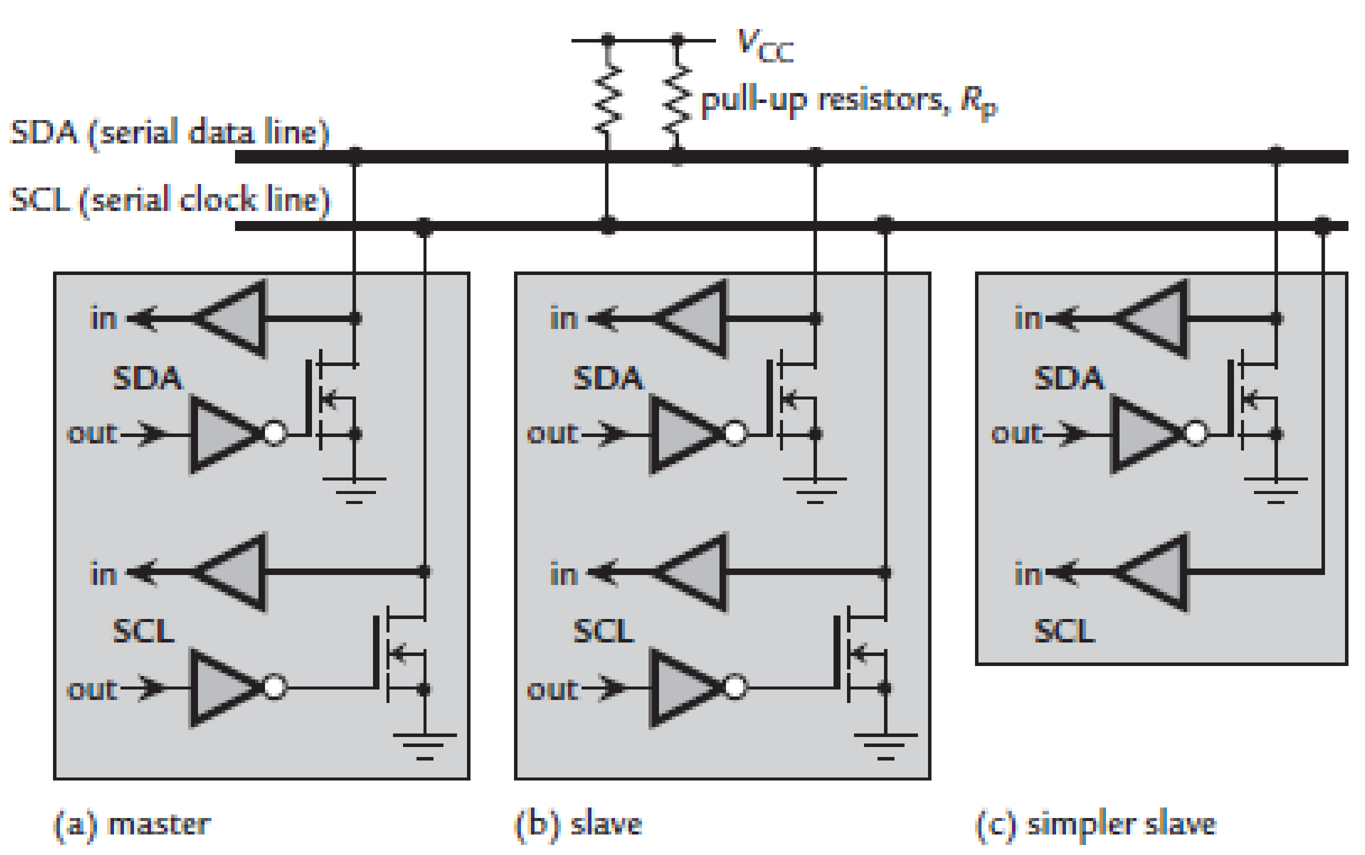
- **Two-wire multi-master multi-slave synchronous half-duplex bus**
  - **Signals: Single-ended voltage, 100 kHz to 5 MHz, short-range.**
  - **open-drain lines with passive pull-up to +5 V or +3.3 V:**
  - **Serial Data (SDA), Serial Clock.**
  - **Bit rate: 10 kbps (low-speed), 100 kbps (standard), 400 kbps (fast mode, Fm), 1 Mbps (Fm+), 3.4 Mbps (high-speed).**
  - **Devices with unique addresses (7, 10, or 16 bit address space).**
  - **No. of nodes: Limited by the address space & the total bus capacitance of 400 pF.**

Inter-integrated Circuit Bus

Basic Features

- Lines: Serial Clock (SCL), Serial Data (SDA); Device address: 7-bit
- Node types
  - Master: Generates clock; Initiates communication with slaves.
  - Slave: Receives clock; Responds when addressed by the master.
- Multi-master bus: Master and slave roles may be changed between messages, after a STOP bit.
- Hardware overhead: Clock stretching by slave.
- Protocol overheads: Slave address, [Register address within the slave device], Per-byte ACK/NACK bits.
- Throughput: limited by overheads & clock stretching by slave.

Hardware for I²C



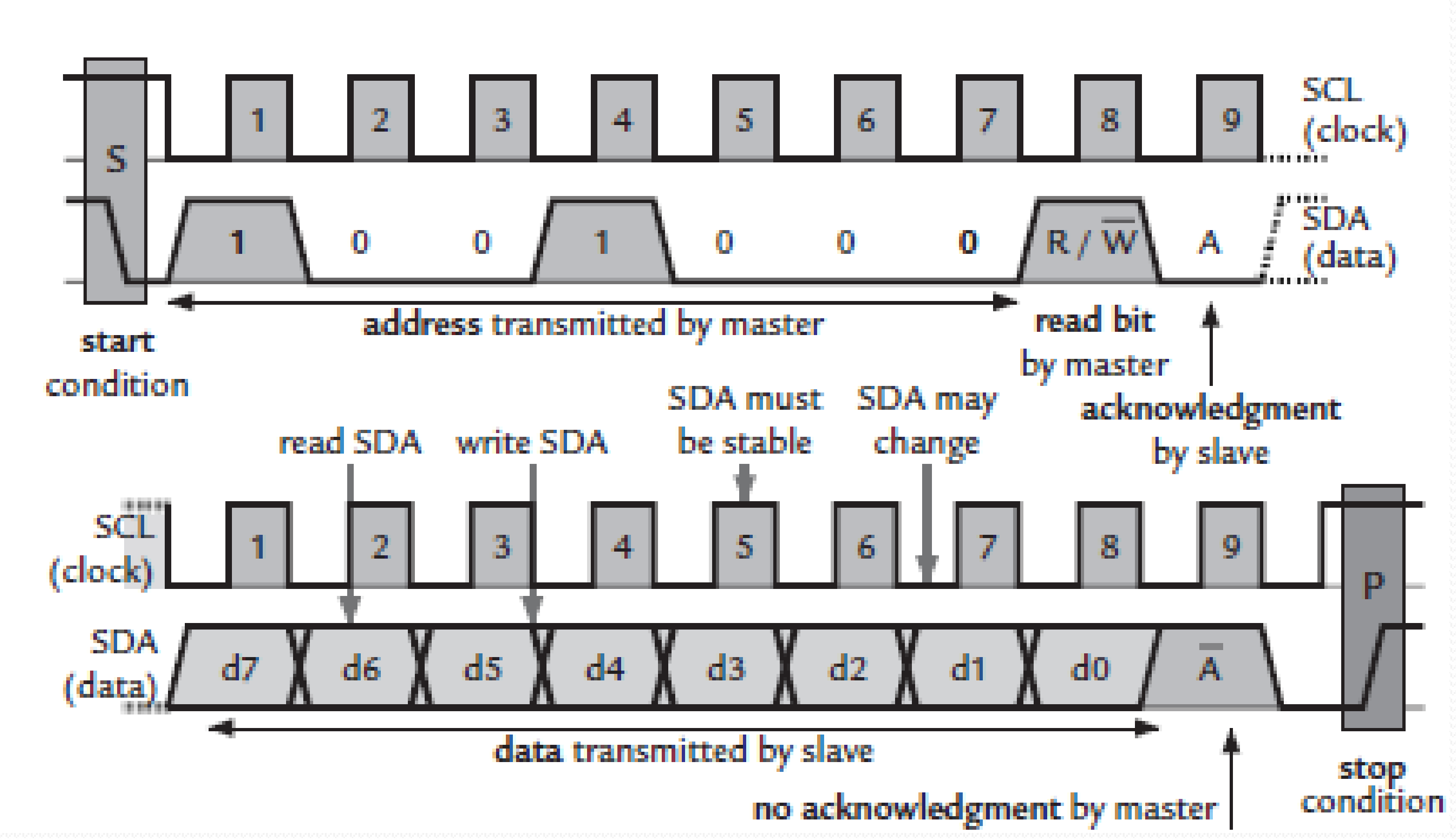
Electronic interface to the I²C bus. The two lines of the bus, SCL and SDA, are bidirectional and pulled up to *VCC with resistors Rp.*

***(a) A master device can* read and write both SCL and SDA independently.**

(b) (b) A slave may have an identical interface but

(c) (c) most slaves cannot drive SCL.

I²C Protocol



Simple transfer over I²C. The master writes an address, which is acknowledged by the slave, and reads a single byte from the slave.

From the idle state with both SCL and SDA high:

- 1. The master sends a start condition (S) by pulling SDA low while SCL is high.
- 2. The master starts the clock and puts the first bit of the address on SDA after SCL has gone low.
- 3. The value on SDA is valid after SCL has gone high and is read by all slaves on the bus.
- 4. The last two steps are repeated until all 7 bits of the address have been sent.
- 5. The final bit of the first byte specifies the direction for the rest of the transfer. Here it is R/*W= 1, which shows that the master wishes to read data from the slave.*
- 6. The ninth bit is the acknowledgment (A or Ack), which is low and is sent by the slave that recognizes its address.
- 7. The master must check that a slave acknowledges the address and abort the transfer if the low bit is missing.
- 8. The next 8 clock cycles are used to transmit 1 byte of data from the slave to the master. The master continues to provide the clock.
- 9. The ninth bit would normally be an acknowledgment but this is the exceptional case: The master does *not acknowledge the final byte that it wishes to read in a* transfer. This signals to the slave that the master has received sufficient data. Here the master expects only a single byte so it does not pull SDA low. This is a “not acknowledgment” signal (A or Nack).
- 10. There is a final cycle of the clock to set up the stop signal. The master pulls SDA low after the falling edge of the clock, which is the normal time for changing SDA. It releases it again after the final rising edge of the clock to give a rising edge on SDA while SCL is high, which provides the stop signal (P).

Inter-Integrated Circuit Bus (I^2C or I2C)

Two-wire multi-master multi-slave synchronous half-duplex bus

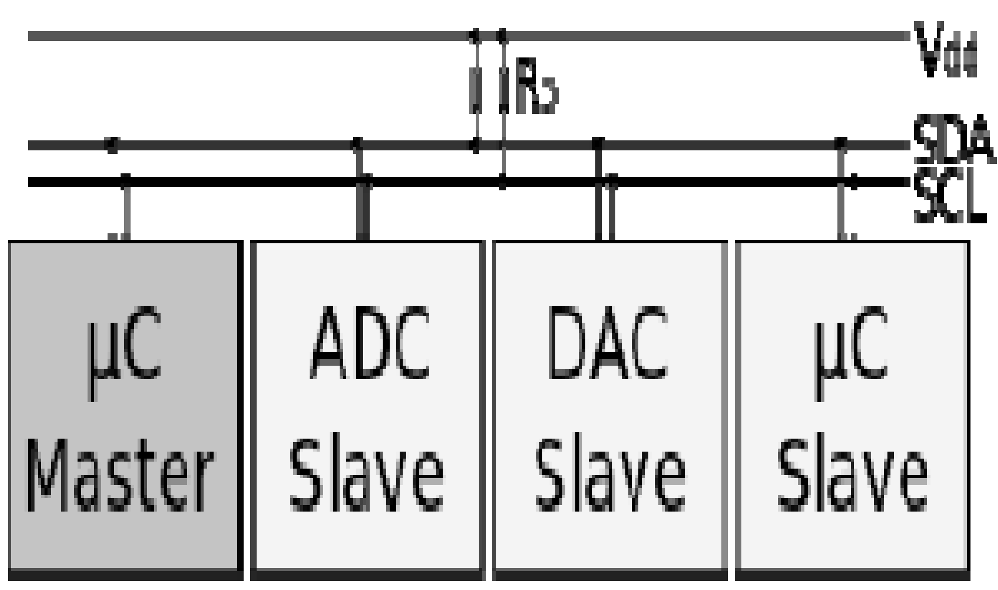
- **Signals: Single-ended voltage, 100 kHz to 5 MHz, short-range.**
- **2 open-drain lines with passive pull-up to +5 V or +3.3 V: Serial Data (SDA), Serial Clock.**
- **Bit rate: 10 kbps (low-speed), 100 kbps (standard), 400 kbps (fast mode, Fm), 1 Mbps (Fm+), 3.4 Mbps (high-speed).**
- **Devices with unique addresses (7, 10, or 16 bit address space).**
- **No. of nodes: Limited by the address space & the total bus capacitance of 400 pF.**

Inter-Integrated Circuit Bus (I^2C or I2C)

- Features
  - Lines: Serial Clock (SCL), Serial Data (SDA); Device address: 7-bit
  - Node types
    - Master: Generates clock; Initiates communication with slaves.
    - Slave: Receives clock; Responds when addressed by the master.
  - Multi-master bus: Master and slave roles may be changed between messages, after a STOP bit.
  - Hardware overhead: Clock stretching by slave.
  - Protocol overheads: Slave address, [Register address within the slave device], Per-byte ACK/NACK bits.
  - Throughput: limited by overheads & clock stretching by slave.

Inter-Integrated Circuit Bus (I^2C or I2C)

One master (microcontroller) & 3 slaves (ADC, DAC, microcontroller)



Inter-Integrated Circuit Bus (I^2C or I2C)

Data Transfer

- Operation sequence
  - Master: Sends START bit , slave address (7-bit), read/write bit (write = 0, read =1).
  - Slave: Responds (after receiving the address and read/write bit) with ACK bit (active low).
  - Master: Continues in Tx/Rx mode.
  - Slave: Continues in complementary (Rx/Tx) mode.
- Bit sequence
  - Address & data bits: SDA transitions with SCL low; MSB first.
  - Start bit: SDA high-to-low transition with SCL high.
  - Stop bit: SDA low-to-high transition with SCL high.

Inter-Integrated Circuit Bus (I^2C or I2C)

Write-to-Slave: Master repeatedly sends a byte with the slave sending an ACK bit.

- Read-from-Slave: Master repeatedly receives a byte from the slave & sends an ACK bit after every byte but the last one.
- End of transfer: Master sends STOP to release the bus or another START bit to retain bus control for another transfer.
- Logic: Pulled low (any device) = 0, Floating (all devices) = 1.
- Clock stretching using SCL: Addressed slave holds SCL low after receiving (or sending) a byte, if not ready for more data. The master waits for SCL to go high. Waits for an additional minimum time (standard: 4 μs) before pulling it low.

Inter-Integrated Circuit Bus (I^2C or I2C)

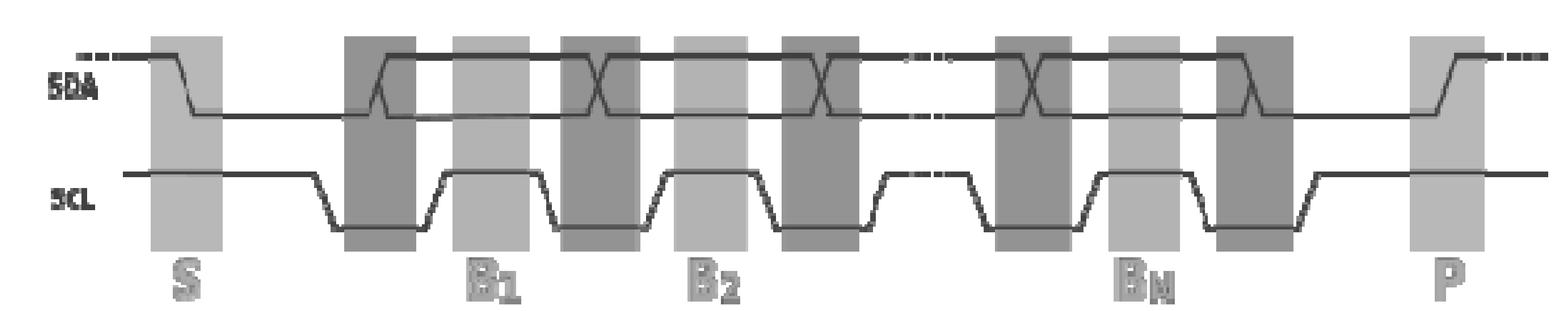
Bidirectional Buffering & Multiplexing

- **Buffering: Splitting large bus segments into smaller ones to limit the capacitance of a bus segment .**

- **Multiplexing: Separating multiple devices with the same address.**

Inter-Integrated Circuit Bus (I^2C or I2C)

- Timing Diagram
  - SDA changed after the SCL falling edge & sampled on the SCL rising edge (avoids false marker detection)
  - START bit (S): SDA pulled low while SCL high.
  - First bit (B_1) written on SDA by Tx while SCL low. SDA read by Rx when SCL rises.
  - Write & read repeated (B_2, ..): SDA transitioning while SCL low; SDA read while SCL rises.
  - STOP bit (P): SDA high while SCL is high.



Inter-Integrated Circuit Bus (I^2C or I2C)

Applications

- Low pin count, Low cost, Low to moderate speed
  - EEPROM for configuration data; NVRAM for user settings.
  - Real-time clock; Low speed DACs and ADCs; Sensors with digital readout; Power supplies with digital control.

Limitations

- Conflict of slave addresses. May be solved by having device pins for user settable address.
- Spurious address detection due to speed mismatch.
- Throughput degradation due to clock stretching. Separate segments for low and high latency devices.
- Problems due to shared bus.

Asynchronous Serial Communication

- The main reason is simplicity.
- Asynchronous serial communication can be managed in hardware by a peripheral called a universal asynchronous receiver/transmitter, which is not complicated and is therefore built into many microcontrollers.
- Even if this is not available, it can be emulated with a timer assisted by software.
- USB is *much more difficult to handle.*

- In practice it is not a big problem to use an asynchronous serial link to a personal computer because USB–serial converters are readily available.
- They provide "virtual COM ports" under Windows, which appear much like the real hardware on older machines.

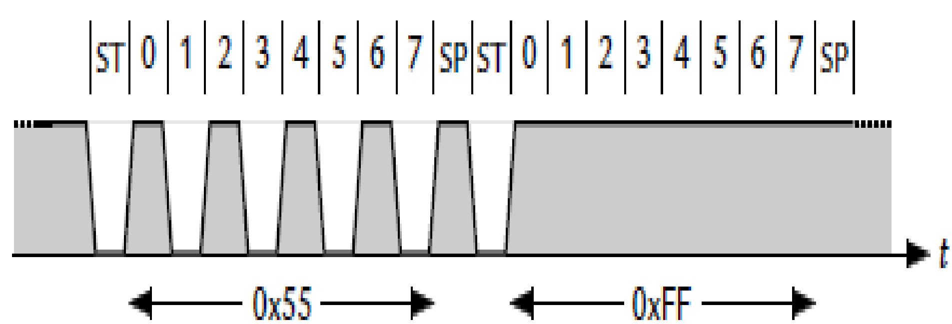
Asynchronous Serial Communication

- Asynchronous serial communication usually requires only a single wire for each direction plus a common ground.
- Most general-purpose connections are full duplex, meaning that data can be sent simultaneously in both directions. These act independently, unlike SPI.
  - (There are usually no further control lines, nor is there anything like the protocol required to run a I^2C bus; characters are simply sent when required.)
- It really is straightforward, which explains its continuing popularity.
- Issues such as the detection and correction of errors are usually handled by the application that supervises the communication.
  - For example, a block of data may be followed by a checksum to confirm that it has been received correctly.
  - Asynchronous links usually connect only two pieces of equipment but a few buses use asynchronous communication.

Asynchronous Serial Communication

Format of Data for Asynchronous Transmission

- The usual format of asynchronous data in the section "Operation of Timer_A in the Sampling Mode".
- Data are sent in short *frames, each of which typically* contains a single byte.
  - Two examples are shown in Figure .

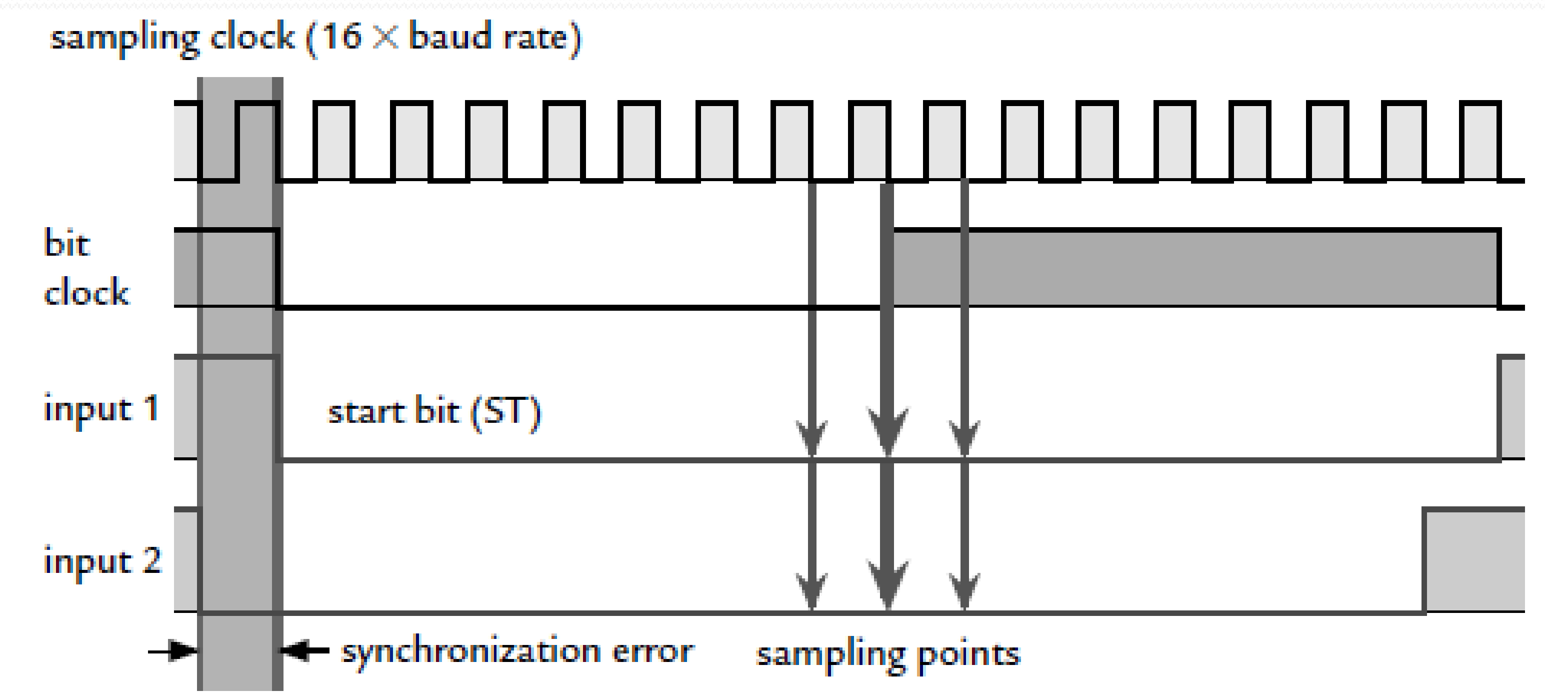


The line idles high and each frame contains

- One low start bit (ST).
- Eight data bits, usually lsb first.
- One high stop bit (SP).

- The bits are either high or low and have no gaps between them, a format known as *non-return to zero (NRZ).*
- *They are usually sent with lsb first, which is the reverse order* compared with I^2C or the usual sequence on SPI.
- The format of the frame is called 8-N-1 because there are 8 bits of data, no parity bit, and 1 stop bit. You may occasionally encounter other formats.
  - For example, the basic ASCII code has only 128 values so 7 bits of data were often sent in the past. The eighth bit was sometimes used for parity as a simple check for errors in transmission.
  - Aparity bit may also be added to 8-bit data and the MSP430 bootstrap loader uses this format.

Asynchronous Serial Communication



Clocks and received data in a UART with the sampling clock running at 16 times the baud rate. Reception starts when a falling edge at the beginning of a start bit is detected. The two input signals correspond to the latest and earliest edges that would trigger reception to start at the same time. The low start bit is followed by a high lsb of the data.

Asynchronous Communication with the USCI_A

- The USCI_A has several modes of operation.
- First, it can be used for SPI in the same way as USCI_B by setting the UCSYNC bit in UCA0CTL0.

- If this bit is clear, there are four asynchronous modes. These are selected with the UCMODExx bits, also in UCA0CTL0.
  - Standard UART mode, UCMODExx= 00.
  - Multiprocessor modes, UCMODExx=01 or 10.
    - These are used to detect addresses when more than two devices are used on a bus, such as RS-485.
  - Automatic baud rate detection, UCMODExx=11. This is particularly intended for LIN.

a. Setting the Baud Rate with the USCI_A

b. Operation of the USCI_A

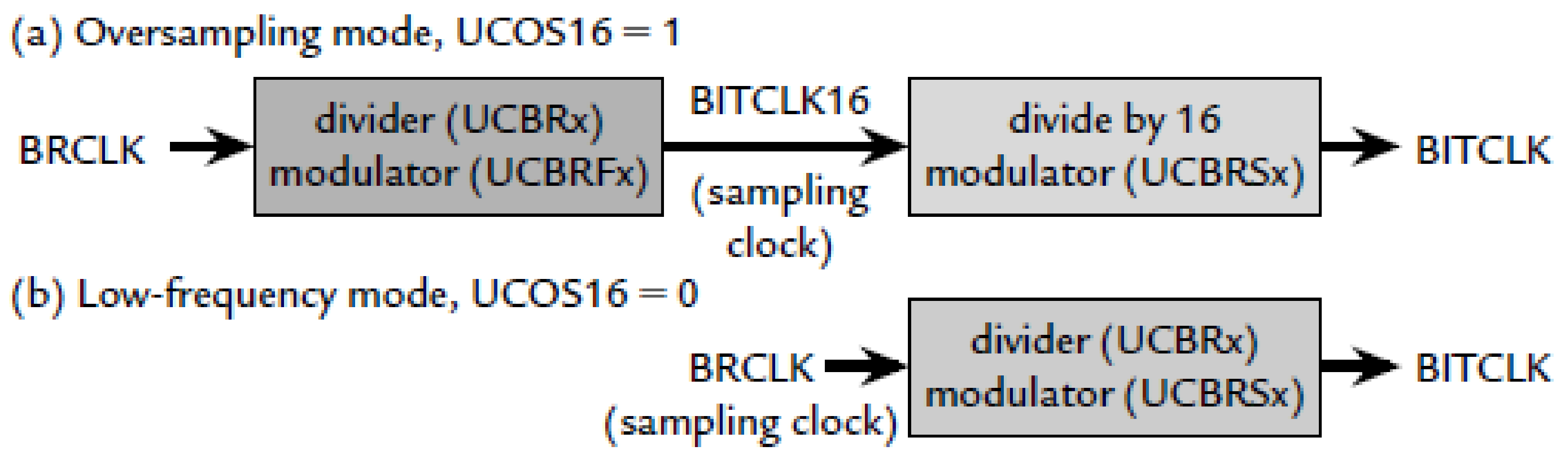
Asynchronous Communication with the USCI_A

Setting the Baud Rate with the USCI_A

The most complicated aspect of configuring the USCI_A is setting the baud rate. For a start, there are three clocks in the USCI_A:

- **BRCLK is the input to the module (SMCLK, ACLK, or UCA0CLK).**
- **BITCLK controls the rate at which bits are received and transmitted. Ideally its** frequency should be the same as the baud rate, $f_{BITCLK} = f_{baud}$.
- **BITCLK16 is the sampling clock in oversampling mode, with a frequency** $f_{BITCLK16} = 16 f_{BITCLK}$.

The periods of these clocks are $T_{BITCLK} = 1/f_{BITCLK}$ *and so on. There are two modes for* setting the baud rate. These are selected with the UCOS16 bit in the modulation control register UCA0MCTL.



Asynchronous Serial Communication

Asynchronous Serial Communication