MODULE 5:

Digital Input-Output and Serial Communication:

Parallel Ports, Lighting LEDs, Flashing LEDs, Read Input from a Switch, Toggle the LED state by pressing the push button, LCD interfacing.

Asynchronous Serial Communication, Asynchronous Communication with USCI_A, Communications Peripherals in MSP430, Serial Peripheral Interface.

(Text: Selected topics from Ch4 & Ch7 and Ch7- 7.1, Ch10 – 10.1, 10.2, and 10.12)

Parallel Ports

Eight registers associated with configuration of ports:

Port P1 input, P1IN: reading returns the logical values on the inputs if they are configured for digital input/output. This register is read-only and volatile. It does not need to be initialized because its contents are determined by the external signals.

Port P1 output, P1OUT: writing sends the value to be driven to each pin if it is configured as a digital output. If the pin is not currently an output, the value is stored in a buffer and appears on the pin if it is later switched to be an output. This register is not initialized and you should therefore write to P1OUT *before* configuring the pin for output.

Port P1 direction, P1DIR: clearing a bit to 0 configures a pin as an input, which is the default in most cases. Writing a 1 switches the pin to become an output. This is for digital input and output; the register works differently if other functions are selected using P1SEL.

Port P1 resistor enable, P1REN: setting a bit to 1 activates a pull-up or pull-down resistor on a pin. Pull-ups are often used to connect a switch to an input. The resistors are inactive by default (0). When the resistor is enabled (1), the corresponding bit of the P1OUT register selects whether the resistor pulls the input up to *V*CC (1) or down to *V*SS (0).

Port P1 selection, P1SEL: selects either digital input/output (0, default) or an alternative function (1). Further registers may be needed to choose the particular function.

Port P1 interrupt enable, P1IE: enables interrupts when the value on an input pin changes. This feature is activated by setting appropriate bits of P1IE to 1. Interrupts are off (0) by default. The whole port shares a single interrupt vector although pins can be enabled individually.

Port P1 interrupt edge select, P1IES: can generate interrupts either on a positive edge (0), when the input goes from low to high, or on a negative edge from high to low (1). It is not possible to select interrupts on both edges simultaneously but this is not a problem because the direction can be reversed after each transition. Care is needed if the direction is changed while interrupts are enabled because a spurious interrupt may be generated. This register is not initialized and should therefore be set up before interrupts are enabled.

Port P1 interrupt flag, P1IFG: a bit is set when the selected transition has been detected on the input. In addition, an interrupt is requested if it has been enabled. These bits can also be set by software, which provides a mechanism for generating a software interrupt (SWI).

Interfacing LEDs to microcontroller

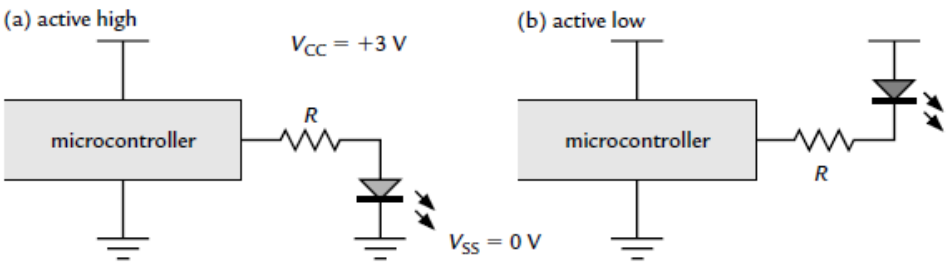


Figure Connection of an LED to a pin of a microcontroller: (a) active high and (b) active low.

Lighting LEDs

1) Program in C to light LEDs with a constant pattern. *(Olimex 1121 STK board with LEDs active low on P2.3,4)*

```
#include <msp430x11x1.h>                // Specific device
void main (void)
{
WDTCTL = WDTPW | WDTHOLD;               // Stop watchdog timer
P2DIR = 0x18;                           // Set pins with LEDs to output , 0b00011000
P2OUT = 0x08;                           // LED2 (P2.4) on , LED1 (P2.3) off (active low!)
for (;;)
{ }                                     // ...loop forever( doing nothing)
}
```

2) Program in assembly language to light LEDs with a constant pattern. *(Olimex 1121STK with LEDs active low onP2.3,4)*

```
#include <msp430x11x1.h>                ; Header file for this device
ORG 0xF000                              ; Start of 4KB flash memory
Reset:                                  ; Execution starts here
mov.w  #WDTPW|WDTHOLD ,& WDTCTL    ; Stop watchdog timer
mov.b  #00001000b,& P2OUT               ; LED2 (P2.4) on , LED1 (P2.3) off (active low!)
mov.b  #00011000b,& P2DIR               ; Set pins with LEDs to output
InfLoop: jmp loop                       ; Loop forever (doing nothing)...
;-----------------------------------------------------------------------
ORG 0xFFFE                              ; Address of MSP430 RESET Vector
DW Reset                                ; Address to start execution
END
```

It is possible to use the linker with assembly language so that addresses need not be written into the program. This is called relocatable assembly. The directive RSEG CODE tells the assembler that the following instructions should be put in the CODE segment, which the linker then puts at the correct address in flash memory. RSEG stands for "relocatable segment," meaning that the address is assigned by the linker

3) Relocatable assembly language to light LEDs. *(Olimex 1121 STK board with LEDs active low on P2.3,4)*

```
#include <msp430x11x1.h>                ; Header file for this device
RSEG CODE                               ; Program goes in code memory
Reset:                                  ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL          ; Stop watchdog timer
mov.b #00001000b,& P2OUT                ; LED2 (P2.4) on , LED1 (P2.3) off (active low!)
mov.b #00011000b,& P2DIR                ; Set pins with LEDs to output
Loop: jmp Loop ;                        ; Loop forever ... (doing nothing)
;-----------------------------------------------------------------------
RSEG RESET ; Segment for reset vector
DW Reset ; Address to start execution
END
```

Read Input from a Switch

1) Single Loop with a Decision

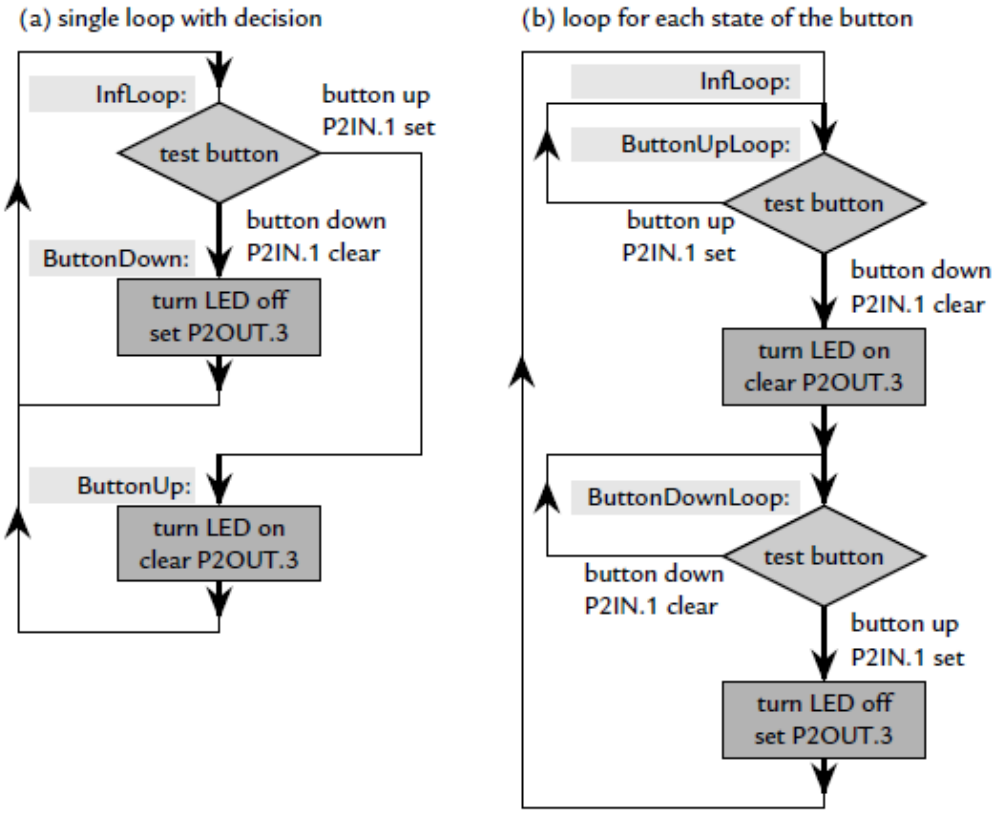


Figure : Two flow diagrams for lighting LED1 when button B1 is pressed: (a) single loop containing a decision, (b) loop for each state of the button.

4) Program in C to light LED1 when button B1 is pressed. This version has a single loop containing a decision statement. *(Olimex 1121 STK board , LED1 active low on P2.3, button B1 active low on P2.1)*

```
include <msp430x11x1.h>                    // Specific device
// Pins for LED and button on port 2
#define LED1 BIT3
#define B1 BIT1
void main (void)
{
WDTCTL = WDTPW | WDTHOLD;                  // Stop watchdog timer
P2OUT |= LED1;                             // Preload LED1 off (active low!)
P2DIR = 0x08;                              // Set pin with LED1 to output(P2.3 is made output)
for (;;)                                   // Loop forever
{
if ((P2IN & B1) == 0)                      // Is button down? (active low)
P2OUT &= ~LED1;                            // Yes: Turn LED1 on (active low!)
else
P2OUT |= LED1;                             // No: Turn LED1 off (active low!)
}
}
```

5) Program with single loop in assembly language to light LED1 when button B1 is pressed.

```
; Olimex 1121STK , LED1 active low on P2.3, B1 active low on P2.1
#include <msp430x11x1.h>                    ; Header file for this device
                                            ; Pins for LED and button on port 2
LED1 EQU BIT3
B1 EQU BIT1
RSEG CODE                                   ; Program goes in code memory
Reset:                                      ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL              ; Stop watchdog timer
mov.b  #00001000b, &P2OUT                   ; Preload LED1 off (active low!)
mov.b #00001000b, &P2DIR                    ; Set pin with LED1 to output
Loop: bit.b #B1 ,&P2IN                      ; Test bit B1 of P2IN ; Loop forever
        jnz open                            ; Jump if not zero , button up
        bic.b #LED1 ,& P2OUT                ; Turn LED1 on (active low!)
        jmp Loop                            ; Back around infinite loop
        bis.b #LED1 ,& P2OUT                ; Turn LED1 off (active low!)
        jmp Loop                            ; Back around infinite loop
;-------------------------------------------------------------------
RSEG RESET ; Segment for reset vector
DW Reset ; Address to start execution
END
```

[The above program can be repeated with two Loops, One for Each State of the Button]

6) Program in C to light LED1 when button B1 is pressed. This version has a loop for each state of the button.
(Olimex 1121 STK board , LED1 active low on P2.3, button B1 active low on P2.1)

```
#include <msp430x11x1.h> // Specific device
// Pins for LED and button on port 2
#define LED1 BIT3
#define B1 BIT1
void main (void)
{
WDTCTL = WDTPW | WDTHOLD;                   // Stop watchdog timer
P2OUT = LED1;                               // Preload LED1 off (active low!)
P2DIR = LED1;                               // LED1 pin output , others input
for (;;)
 {                                          // Loop forever
while ((P2IN & B1) != 0)
 { }                                        // Loop while button up (active low) doing nothing
 P2OUT &= ~LED1;                            // Turn LED1 on (active low!)
while ((P2IN & B1) == 0)
{ }                                         // Loop while button down(active low) doing nothing
P2OUT |= LED1;                              // Turn LED1 off (active low!)
}
}
```

7) Program in assembly to light LED1 when button B1 is pressed.(two loops)

(Olimex 1121 STK board , LED1 active low on P2.3, button B1 active low on P2.1)

```
#include <msp430x11x1.h>                    ; Header file for this device
; Pins for LED and button on port 2
LED1 EQU BIT3
B1 EQU BIT1
RSEG CODE                                   ; Program goes in code memory
Reset:                                      ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL              ; Stop watchdog timer
bis.b #LED1 ,& P2OUT                        ;Preload LED1 off (active low!) or mov.b #00001000b,&P2OUT
bis.b #LED1 ,& P2DIR                        ; Set pins with LED1 to output or mov.b #00001000b,&P2DIR
open: bit.b #B1 ,&P2IN                      ; Test bit B1 of P2IN
        Jnz open                            ; Jump if not zero , open
        bic.b #LED1,&P2OUT                  ; Turn LED1 on (active low!)
closed:  bit.b #B1, &P2IN                   ; Test bit B1 of P2IN
      jz closed                             ; Jump if zero , closed
      bis #LED1,&P2OUT                      ;Turn LED1 off (active low!)
      jmp open                              ; Test bit B1 of P2IN

;-------------------------------------------------------------------
RSEG RESET                                  ; Segment for reset vector
DW Reset                                    ; Address to start execution
END
```

Addressing Bits Individually in C

Program in C to light LED1 when button B1 is pressed. *(Olimex 1121 STK board , LED1 active low on P2.3, button B1 active low on P2.1)*

```
#include <io430x11x1.h>                 // Specific device , new format header
// Pins for LED and button
#define  LED1  P2OUT_bit.P2OUT_3
#define  B1    P2IN_bit.P2IN_1
void main (void)
{
WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer
LED1 = 1;                           // Preload LED1 off (active low!)
P2DIR_bit.P2DIR_3 = 1;              // Set pin with LED1 to output
for (;;)                              // Loop forever
{
while (B1 != 0)  { }                // wait until button is open; doing nothing
LED1 = 0;                           // Turn LED1 on (active low!)
while (B1 == 0)  { }                // wait until button is open; doing nothing
LED1 = 1;                           // Turn LED1 off (active low!)
}
}
```

Flashing Light by Software Delay

Program to flash LEDs with a frequency of roughly 1Hz using a software delay.

```
// Olimex 1121STK , LED1 ,2 active low on P2.3,4
#include <msp430x11x1.h>          // Specific device
// Pins for LEDs
#define LED1 BIT3
#define LED2 BIT4
#define DELAYLOOPS 50000          // Iterations of delay loop;
void main (void)
{
volatile unsigned i;              // Loop counter: volatile!
WDTCTL = WDTPW | WDTHOLD;         // Stop watchdog timer
P2OUT = ˜LED1;                    // Preload LED1 on , LED2 off
P2DIR = LED1|LED2;                // Set pins with LED1 ,2 to output or (P2DIR = 0x18;)
for (;;) {                        // Loop forever
for (i = 0; i < DELAYLOOPS; ++ i)
{ }                               // Empty delay loop
P2OUT ˆ= LED1|LED2;               // Toggle LEDs
}
}
```

Delay Loop in Assembly Language

Program in assembly language to flash LEDs with a frequency of roughly 1Hz using a software delay.

```
; Olimex 1121STK , LED1 ,2 active low on P2.3,4
#include <msp430x11x1.h>                    ; Header file for this device
; Pins for LED on port 2
LED1 EQU BIT3
LED2 EQU BIT4
DELAYLOOPS EQU 50000                        ; Iterations of delay loop; reduce for simulation
RSEG CODE                                   ; Program goes in code memory
Reset:                                      ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL              ; Stop watchdog timer
mov.b #LED2 ,& P2OUT                        ; Preload LED1 on , LED2 off
bis.b #LED1|LED2 ,& P2DIR                   ; Set pins with LED1 ,2 to output(mov.b #
back:  clr.w R4                             ; Loop forever  ; Initialize loop counter
Loop: inc.w R4                              ; Increment loop counter
      cmp.w #DELAYLOOPS ,R4                 ; Compare with maximum value
     jne Loop                               ; Repeat loop if not equal [2]
     xor.b #LED1|LED2 ,& P2OUT              ; Toggle LEDs
     jmp back                               ; Back around infinite loop
RSEG RESET ; Segment for reset vector
DW Reset ; Address to start execution
END
```

Asynchronous serial communication

Asynchronous serial communication usually requires only a single wire for each direction plus a common ground. Asynchronous serial communication can be managed in hardware by a peripheral called a universal asynchronous receiver/transmitter (UART). Most general-purpose connections are full duplex, meaning that data can be sent simultaneously in both directions.

Format of Data for Asynchronous Transmission

The line idles high when there is no data transmission 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. A parity bit may also be added to 8-bit data and the MSP430 bootstrap loader uses this format.

-A minimum of 1 stop bit is needed to separate each frame and provide a high level before the falling edge of the next stop bit. Slower systems may specify more stop bits, often 1 or 2, so that they can keep up with the flow of data but this is now rare.

-The *baud rate* gives the frequency at which bits are transmitted on the line. It is the inverse of the bit period and the name is used to distinguish it from the rate at which useful data are communicated. Each 8 bits of data are accompanied by a start and stop bit so the maximum data rate is only 8/10 of the baud rate. A common speed for embedded systems is 9600 baud but both higher and lower rates are also used.

-No clock is transmitted in asynchronous communication so the transmitter and receiver must run independently at nearly the same baud rates.

Example -Two asynchronous bytes that carry the data 0x55 and 0xFF. Each frame is delimited by a start (ST) bit and a stop (SP) bit is as shown.

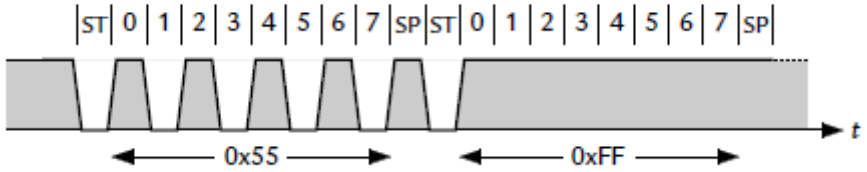


Figure shows 2 bytes. The first carries the value 0x55 (lsb first), which gives the maximum number of transitions within the frame. An advanced receiver can resynchronize to these transitions, which makes it easy to match the frequency of the transmitter. In fact a byte of 0x55 is used in a local interconnect network so that a receiver can synchronize its clock to that of the transmitter.

Interface Standards and RS-232

RS232 is an ancient standard, originally intended for connecting equipment such as a teletype (data terminal equipment) to a modem (data communication equipment). RS-232 specifies the following voltages for the signals,

- Logical 1 is represented by a voltage between −15 V and −3 V. This is also called mark.
- Logical 0 is represented by a voltage between +3 V and +15 V, also called space.
- The crossover region between ±3 V does not correspond to valid data.
- Connections must be resistant to short circuits and to voltages of ±25 V.

This presents problems for the MSP430, whose natural range of voltages lies wholly within the crossover region. The original recommendation was for signals to be at ±12 V but many systems now use lower voltages, often ±5 V. This is closer to the voltages used for digital electronics but the negative voltage is problematic. External circuits for a transmitter and receiver or transceiver are therefore needed to connect the MSP430 or any other microcontroller to an RS-232 port. There are several ways in which they can be implemented:

• Special transceiver ICs are available that generate the voltages needed for RS-232 transmission and provide interfaces between the external and internal signals, including isolation and protection. Perhaps the best known family is the MAX232 and descendants. The original MAX232 was intended for 5 V systems but newer devices work from lower voltages. Typically they include charge pumps to produce voltages of $\pm 2V_{CC}$, which provide the RS-232 levels. Three or four capacitors are needed for the charge pumps

• If this is too expensive, numerous simple circuits use a couple of transistors and a few passive components for the receiver and transmitter. They “steal” the voltage needed for RS-232 from the line driven by the other transmitter, which must therefore come from a full-strength port. This should be true for a computer but not necessarily for another embedded system. This approach is used in the Olimex 1121STK.

• An opto-isolator can be used to provide greater separation between the microcontroller and the RS-232 circuits. The input to the opto-isolator drives an LED, which shines onto a phototransistor that provides the output. Thus there is no electrical connection between the two sides.

Communication Peripherals in the MSP430

Currently three types of communication peripherals are offered in different variants of the MSP430.

1) Universal Serial Interface

The universal serial interface (USI) is a lightweight module, which is included in the small F20x2 and F20x3 devices. It handles only synchronous communication—SPI and I²C. It includes a shift register, clock generator, bit counter, and a few extra items used to assist I²C. This is sufficient to run SPI easily provided that a SS signal is not needed. I²C is inevitably more complicated and needs considerable help from software. The USI is a great advance on bit-banging, particularly for a slave, despite its limitations. The I²C mode is a little buggy at the time of writing and needs considerable care.

2) Universal Serial Communication Interface

Recent, larger devices in the MSP430F2xx and MSP430F4xx families contain one or more *universal serial communication interface* (USCI) modules. The hardware handles almost all aspects of the communication, unlike the USI, so the software needs only to provide the data to transmit and store the received data in normal operation. Typically this requires only a couple of small interrupt service routines. The USCI can use direct memory. Each USCI contains two channels, A and B. These are largely independent but share a few registers and interrupt vectors

--Asynchronous channel, USCI_A: Acts as a universal asynchronous receiver/transmitter (UART) to support the usual RS-232 communication. It can detect the baud rate of an incoming signal, which enables its use on a local interconnect network (LIN). The output signal can be modulated for an infrared diode to work with IrDA and the incoming signal can be decoded to match. Finally, it can also handle SPI.

--Synchronous channel, USCI_B: Handles both SPI and I^2C as either master or slave. It contains a full state machine to run I^2C communications in compliance with the specification from NXP (formerly Philips). This includes multiple masters and clock stretching.

Some devices have more than one USCI, in which case the modules are called USCI_A0, USCI_A1, and so on. There is a small difference because the interrupt flags and enable bits for the "0" modules are in the special function registers IFG2 and IE2, while those for the "1" modules are in their own registers, UC1IFG and UC1IE. Although the module is called USCI, the names of its registers and bits start with only UC.

3) Universal Synchronous/Asynchronous Receiver/Transmitter

The universal synchronous/asynchronous receiver/transmitter (USART) is an older module, which has been superseded by the USCI. In most cases it provides only asynchronous communication and SPI but it also handles I^2C in a few devices.

4) Bit-Banging

If no peripheral is available for communication then bit banging has to be used. This means that every aspect of the signals is handled in software, assisted where appropriate by Timer_A or B to ensure precise timing.

Synchronous masters: Easy to bit-bang because the master generates the clock and therefore has full control over the timing. A timer can be used for slow communication but often the software requires so many clock cycles that the MSP430 can simply run at full speed, particularly for SPI. Actions are triggered by edges of the clock, which does not need a precise frequency.

Synchronous slaves: More difficult. The problem is that the slave must react quickly when a clock transition arrives from the master. It may be possible to use interrupts on the input ports but the latency may be too long, in which case a polling loop with the MSP430 running at full power is unavoidable. It can be done, at least for fairly slow rates. Some extra hardware almost always is needed to detect start and stop conditions for an I^2C slave.

Serial Peripheral Interface

The serial peripheral interface was introduced by Motorola and is the simplest synchronous communication protocol in general use. It is not a fixed standard like I2C.

The concept of SPI is shown in Figure below for the minimal system of two devices. One device is the master and the other the slave. The master provides the clock for both devices and a signal to select (enable) the slave, but the path followed by the data is identical in each. In its full form SPI requires four wires (plus ground, which is essential but never counted) and transmits data simultaneously in both directions (full duplex) between two devices.

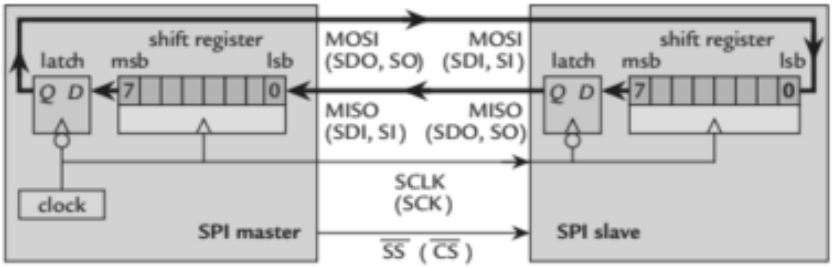


Figure : Serial peripheral interface between a master and a single slave.

Motorola's nomenclature for the two data connections is "master in, slave out" (MISO) and "master out, slave in" (MOSI). The two MISO pins should be connected together and likewise the two MOSI pins. Other terms are widely used, such as SDI, SI, or DIN for serial data in and SDO, SO, or DOUT for serial data out. The final signal selects the slave. This is usually active low and labelled SS for slave select (Motorola), 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.

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: One byte is the most common length of a transfer but any number of bits can be transmitted and a word of 16 bits is natural with the MSP430. Many data converters also use SPI with 16-bit transfers.

Suppose that we want to send only a single bit. This requires three steps:

1. Put data on output.

2. Read data from input.

3. Remove data from output.

A single cycle of the clock provides stimuli for only two of these so the third is taken from SS. It can either start or end the transmission and this option is specified by the clock phase bit, CPHA in Motorola's notation:

CPHA = 0: A transition on SS starts the transaction and causes the first bit of data to be placed on the outputs. The inputs are read and clocked into the shift register on the leading (first) edge of the first clock

pulse. The second bit is put on the output at the trailing (second) edge of the first clock pulse and this continues until the last bit is read on the leading edge of the last clock pulse. The outputs are removed after the trailing edge of the last clock pulse or when SS becomes inactive. In summary, data are

- Read on the leading edge of each clock pulse.
- Written on the trailing edge of each clock pulse.

It is essential that SS goes inactive between transfers because the first output is stimulated by SS becoming active.

CPHA = 1: The first bit of data is placed on the output following the leading edge of the first clock pulse. It is read on the trailing edge of the pulse. This continues until the last bit has been read on the trailing edge of the last clock pulse. Thus data are

- Written on the leading edge of each clock pulse.
- Read on the trailing edge of each clock pulse.

The outputs are removed when SS becomes inactive. Thus SS is needed only to control when the slave should drive its output, not to provide any timing. If there is only one slave, whose output can remain active at all times, the SS signal need not be used. Thus CPHA controls whether writing and reading take place on the leading and trailing edges of the clock pulses or vice versa. This is the first of many options that control the configuration of SPI. A related option is the clock polarity, selected with the CPOL bit:

CPOL = 0: Clock idles low between transfers.

CPOL = 1: Clock idles high between transfers.

There is no fundamental difference between these two polarities: One is just the complement of the other. Combinations of the two bits CPOL and CPHA give four standard modes for the clock in SPI, which are listed in Table below.

Table The four standard modes for the clock in SPI.

Mode	CPOL/CKPL	CPHA/$\overline{\text{CKPH}}$
0	0	0
1	0	1
2	1	0
3	1	1

In modes 0 and 3 the data are read on rising edges and written on falling edges.

Figures below illustrate waveforms for a 4-bit transfer in modes 0 and 3. These show the different relations between the clock and SS. In both cases SS becomes active one half cycle before the first edge on the clock and is released to the inactive level one half cycle after the last edge on the clock. The slave may show spurious data on its output at either the start or end of the transmission, shown by the “?”, but there are no rising edges that would cause these values to be read so they have no practical impact.
The output from the slave (MISO) goes to a high-impedance state when SS is inactive. This is shown by the “floating” level on the diagrams. It is not important if only two devices are connected but becomes vital if more than one slave shares a bus.

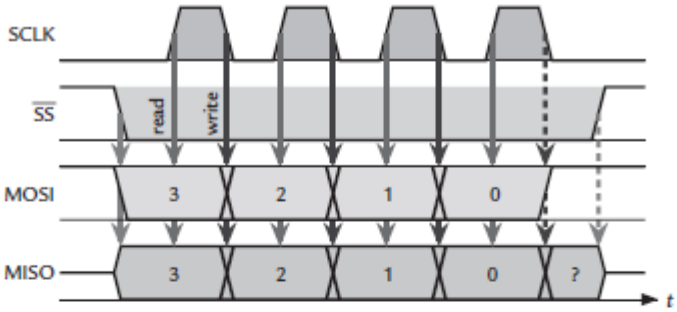


Figure A complete transfer of 4 bits using SPI in mode 0 (CPHA = 0, CPOL = 0). The first bit is placed on the outputs when SS becomes active (low). Inputs are read on rising edges of the clock and subsequent bits are placed on the outputs on falling edges of the clock.

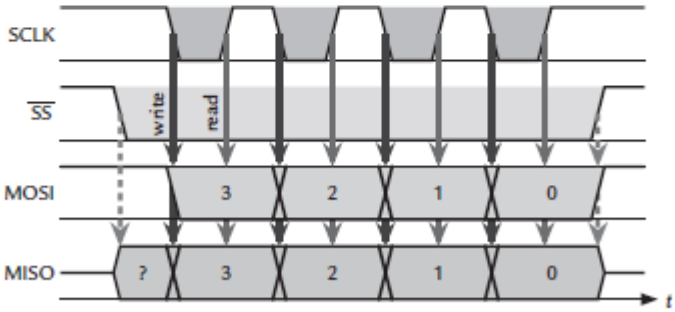


Figure A complete transfer of 4 bits using SPI in mode 3 (CPHA = 1, CPOL = 1). Bits are placed on the outputs on falling edges of the clock and read on rising edges of the clock.

• The data in SPI are usually shifted with the most significant bit (msb) first but it can be the other way; the bidirectional three-wire variant uses lsb first.

• The slave select signal (or whatever it is called for the particular device) is usually active low but occasionally active high.

SPI is used to connect a single slave to a master. It is not really a bus because it lacks protocols to control and acknowledge transactions but can be extended to handle more than one slave. There are two ways of doing this, shown in figure below.

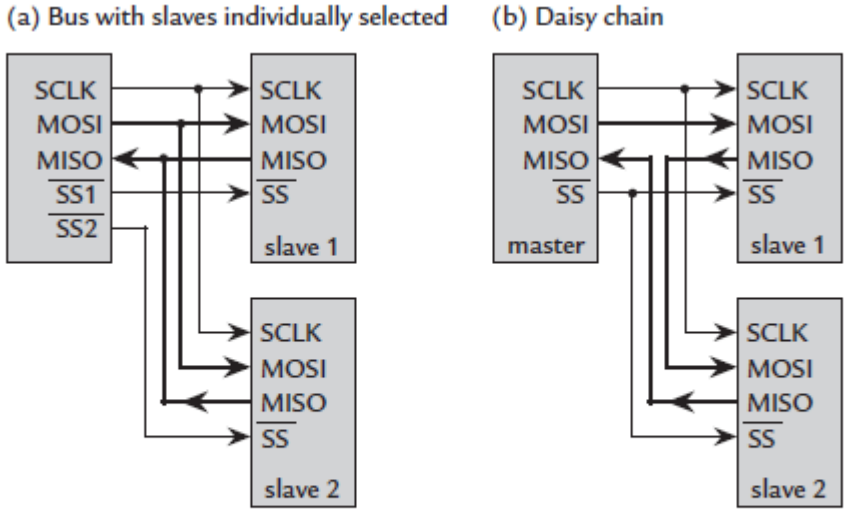


Figure Two ways of connecting two slaves to a single master using SPI. (a) A slave can be selected individually by providing separate SS lines. (b) All slaves can be connected in a "daisy chain," in which case they must all be updated together.

In both cases the master provides the clock to all the slaves. Slaves can be addressed individually in the first configuration. All the MOSI pins are connected together, as are the MISO pins, but each slave's SS pin is connected to a separate pin on the master. Slaves must ignore data on MOSI when their SS pin is idle,

despite the activity on the clock, and must leave their MISO pins in a high-impedance state. In other words, the MISO pins must
have three-state outputs.

The alternative is to connect all the devices in a "daisy chain," as shown in Figure (b). In this configuration the MOSI pins are *not* all connected together, nor the MISO pins, but rather the MISO pin of a slave is connected to the MOSI pin of the next slave in the chain. The MOSI pin of of the final slave is connected to MOSI on the master. Effectively all the shift registers inside each device are connected into a single, long loop. In principle the master needs to contain a shift register whose length is the sum of the lengths of the registers in all the slaves. The slaves must allow data to be clocked through them transparently while SS is held active and react to the new data only when SS is released

SPI is sometimes called a *four-wire* bus. It is also based on the concept of a shift register but only one data line is active at a time—it is not full-duplex. A disadvantage of SPI is that it uses four pins and a *three-wire* bus is sometimes used to save a pin. Usually this means that the two lines for data, MOSI and MISO, are multiplexed onto a single, bidirectional line for serial data (SDA). This abandons the full-duplex nature of SPI and a protocol is needed to ensure that the master and slave do not attempt to transmit simultaneously. The clock usually uses mode 0 and data are transmitted with the lsb first.
It is fast because the data and clock lines are always driven actively, unlike I^2C, and can run at tens of megahertz (but not on a MSP430). The clock does not need a precise or stable frequency because everything is triggered by edges. It rapidly becomes clumsy for multiple slaves but is the simplest serial interface for connecting a single device to a microcontroller.

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 further option is to encode the output for IrDA, which is a format for infrared transmission and is chosen with the UCIREN bit. It is described in the application note

The USCI_A offers sophisticated options for setting the baud rate, which merit a separate section.

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, fBITCLK = $f$baud.

• **BITCLK16** is the sampling clock in oversampling mode, with a frequency fBITCLK16 = 16fBITCLK.

The periods of these clocks are TBITCLK = 1/fBITCLK and so on. There are two modes for setting the baud rate, illustrated in Figure below. These are selected with the UCOS16 bit in the modulation control register UCA0MCTL.

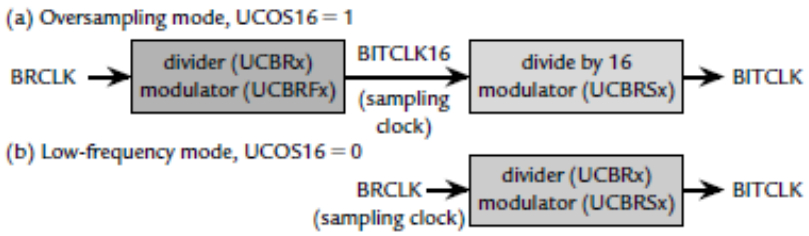


Figure : Clock generator in USCI_A showing (a) oversampling mode with two dividers and intermediate clock BITCLK16, and (b) low-frequency mode with a single divider.

Oversampling mode, UCOS16 = 1: BRCLK is first divided to give BITCLK16, which is further divided by a factor of 16 to give BITCLK. BITCLK16 is used tocontrol the sampling of received bits.

Low-frequency mode, UCOS16 = 0: BRCLK is used directly as the sampling clock and is divided to give BITCLK. In general *f*BRCLK _= 16*f*BITCLK . Simple division of BRCLK by an integer does not give a sufficiently accurate baud rate in many cases. Each divider therefore has a modulator to permit finer control over the average frequency of BITCLK.

UCA0BR0 and **UCA0BR1:** Together act as UCBRx, which provides the main divider of BRCLK as in SPI mode.
UCBRFx: Modulates the divider that gives BITCLK16 in oversampling mode. This allows the ratio *f*BRCLK/*f*BITCLK16 to be set more accurately than an exact integer would permit (it can be specified to the nearest 1/16).
UCBRSx: Modulates the divider that gives BITCLK in a similar way. All this complication is needed because the baud rate is rarely close to being a perfect factor of *f*BRCLK, the frequency of the clock supply to the module.

Liquid Crystal Displays

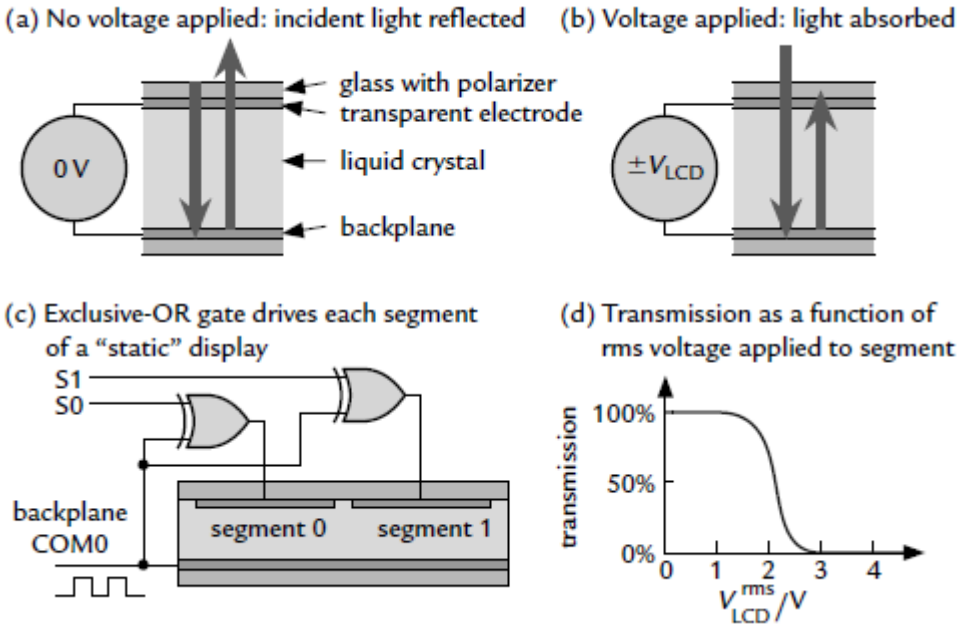
A liquid crystal display (LCD) uses much less power than LEDs. An LCD does not emit light itself but controls the intensity of reflected or transmitted light. It therefore works well in strong ambient lighting but a backlight must be provided for a display to be used in dark surroundings.

LCDs fall into three classes.
Segmented LCDs: the simplest and can be driven directly by the MSP430x4xx family. These displays include the familiar seven-segment numerical displays found in watches, meters, and many other applications. Further segments can be added to each character to allow alphanumeric display.

Character-based LCDs: have a dot-matrix display, often with 1–4 rows of 8–72 characters. They can typically display a set of around 256 characters drawn from the ASCII characters, arrows, and a selection of other symbols. They are addressed by sending a byte to define each character. Further control bytes are used to clear the display, return the cursor to home and so on.

Fully graphical LCDs: found on every mobile (cell) phone,
The interface to the microcontroller needs only a few pins but the MSP430 has to send the value of each pixel to the display and must therefore generate the shape of every character.

Figure (a) shows the basic construction of a reflective LCD.



Two glass plates carry transparent electrodes on their opposing faces and there is a mirror below the lower plate. The gap between is filled with a liquid crystal. Incident light is reflected and the display appears clear when no bias is applied to the electrodes. A sufficiently large bias changes the optical properties of the liquid crystal so that reflected light is no longer transmitted through the upper glass.

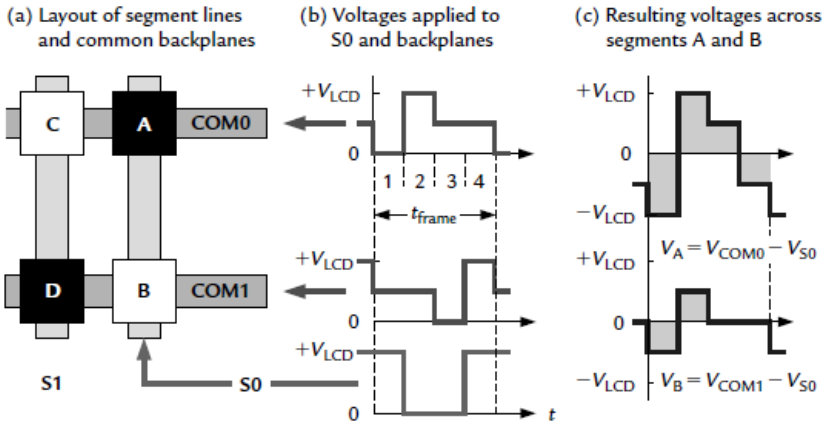
The segment now appears dark, as in Figure (b). Electrically the display is similar to a capacitor, LCDs must be driven with AC, not DC. A steady voltage of only a few tens of millivolts leads to electrolysis of the liquid crystal, which eventually destroys the display. The two electrodes of a segment are therefore driven with square waves in antiphase to produce an alternating voltage with zero mean. The frequency is low, typically around 100 Hz, but must not be close to multiples of the AC mains (line) frequency (50 or 60 Hz).

Almost all displays have more than one segment. The simplest approach is to drive these individually as shown in Figure (c). There is a common backplane for all segments, called COM0 here, and each segment on the front has a separate connection. A square wave provides a clock to bias the display. This signal is applied directly to the backplane and through an exclusive-OR gate with a control signal to each segment:

- If the control signal is low the exclusive-OR gate transmits the clock unchanged, the same voltage is applied to the front and back electrodes, there is no potential difference, and the segment remains clear.
- A high control signal causes the exclusive-OR to invert the clock so that an alternating bias is applied to the segment, which turns dark.

This type of display is called *static* despite the requirement for an AC drive. It is simple but suffers from the obvious disadvantage of needing a large number of pins, one per segment plus the backplane. Most displays are therefore multiplexed to fewer pins, like displays with LEDs.

The method of two-way multiplexing is shown in Figure below. Each segment line Sn now controls two individual segments, one on each of the two backplanes COM0 and COM1.



The sketches in Figure (b) show the waveforms applied to the backplanes and S0. Segment A is driven by S0 and COM0 and is on (dark), while B is driven by the same segment line S0 but the other backplane COM1 and is off (clear). Each period of the waveforms, called a *frame*, is divided into four phases:

1. The segments on $_{COM0}$ are addressed in the first phase by pulling COM0 to ground (0 V). Segment A should be on and S0 is therefore driven to its maximum value, V_{LCD}. The bias across the segment is $V_A = V_{COM0} - V_{S0} = -V_{LCD}$. The segments on COM1 should be inactive during this phase and it is therefore put at a "neutral" voltage of $12V_{LCD}$. This gives VB = $V_{COM1} - V_{S0} = -1/2V_{LCD}$.
2. The voltages in the second phase are the opposite of those in the first to ensure a pure AC signal with zero mean. This time COM0 is driven to V_{LCD} and S0 is pulled to ground to give $V_A = +V_{LCD}$. The backplane that is not being addressed, COM1, remains at its neutral voltage of 1 /$2V_{LCD}$ so that V_B =+1/ $2V_{LCD}$.
3. Now it is the turn of COM1 to be addressed so it is pulled to ground and COM0 is set to neutral, 1/2VLCD. Segment B should be off and S0 is therefore pulled to ground as well.
4. This is the opposite of phase 3 to ensure that the mean voltage remains 0.

The resulting bias on the two segments A and B is shown in Figure 7.16(c). It is not possible to apply either the maximum voltage $\pm V$LCD at all times to segments that should be on nor a constant value of 0 to those that should be off. The response of a segment depends on the root mean square (rms) value of the bias across it