Getting Started Kits

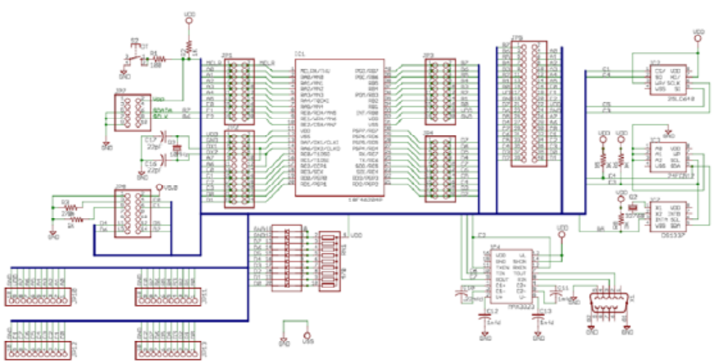
Kit Contents

Each kit has the following items:

- **Board with microcontroller (18(L)F4620)**
- **Power brick for the board.**
- **Programmer, and power brick for programmer.**
- **USB logic analyzer**
- **Digital Volt Meter.**
- **Serial cable**
- **Needle nose and cutting pliers.**

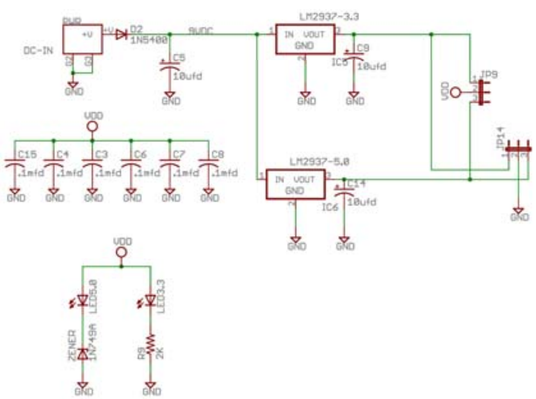
Board

The schematic of the board is shown below:

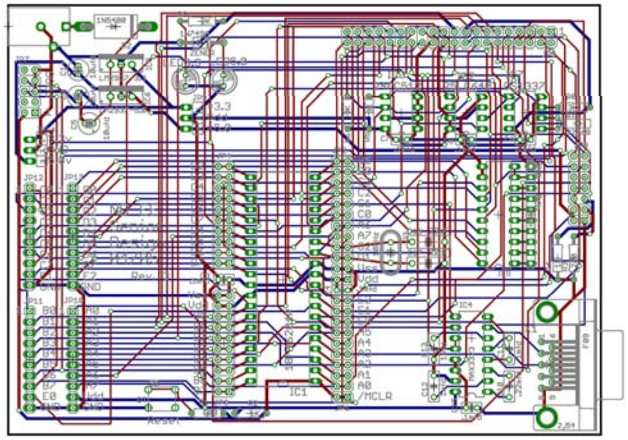


Board

Power Section:



Board Layout



EE-41430 Fall 2007

Microcontroller Functions -5

Board Features

This is a general purpose board, that will accept most of the 40 pin Microchip microcontrollers.

It has built in:

- Serial port interface
- LED's on port D
- LCD interface (also on port D)
- Programmer interface
- Locations where an RTC chip and serial EEPROMS may be added.

EE-41430 Fall 2007

Microcontroller Functions -6

Board Features

The board has 2 voltage regulators on it, and you can select to run the processor at either 3.3 or 5.0 volts. (LCD gets 5 volts in any case.)

Your board might have either an 18LF4620 or an 18F4620.

Note: Only the 18LF4620 will run at 3.3 volts.

The processor has a simple program in it, which writes to the LCD display, blinks the lights, and then exercises the serial port.

EE-41430 Fall 2007

Microcontroller Functions -7

Notes

Some of these boards were built by students in previous years. No guarantees.

The programmer connection to the board may or may not be keyed to prevent incorrect connections. The edge of the connector towards you is painted gold or silver. Incorrectly connecting the programmer has the nasty habit of blowing very small transistors on the board. I have replacements, but you will be doing the repairs.

You should plan on ordering free sample microcontrollers from Microchip, before you do something that destroys the one you have.

EE-41430 Fall 2007

Microcontroller Functions -8

Notes

The power brick used for the microcontroller board is cheap and unregulated. Don’t expect the voltage you get to be the voltage on the slide switch.

You will want about 7 volts out of the power brick. (In one of the tasks, you will determine why.) You should use the meter to find the lowest setting of the slide switch that gives you about 7 volts.

There is a polarity switch on the power brick. It should be to the left.

Notes

The USB logic analyzer software should be on the machines in the ELC, (as should the rest of the software used in this course.)

When connecting the logic analyzer, you can place the leads directly over the header pins that are on the board. This is far easier then the little clips.

Do not remove the leads from the logic analyzer to use for jumpers. They are expensive ($60 to replace the set), and I lost too many in the past. (If I see you doing this, you will loose your logic analyzer and it is a very useful tool.)

18F4620 Microcontroller

R. M. Schafer
EE – Senior Design

Why this processor?

This processor is overkill for most projects. (More memory and features than necessary.)

It makes a lot of sense to use a processor with excess capacity to develop your prototype. (Why?)

Device Features

Power Management:

Power Managed Modes:

- Run: CPU on, peripherals on
- Idle: CPU off, peripherals on
- Sleep: CPU off, peripherals off
- Idle mode currents down to 2.5 µA typical
- Sleep mode current down to 100 nA typical
- Timer1 Oscillator: 1.8 µA, 32 kHz, 2V
- Watchdog Timer: 1.4 µA, 2V typical
- Two-Speed Oscillator Start-up

Device Features

Clock

Flexible Oscillator Structure:

- Four Crystal modes, up to 40 MHz
- 4x Phase Lock Loop (PLL) – available for crystal and internal oscillators)
- Two External RC modes, up to 4 MHz
- Two External Clock modes, up to 40 MHz
- Internal oscillator block:
  - 8 user selectable frequencies, from 31 kHz to 8 MHz
  - Provides a complete range of clock speeds from 31 kHz to 32 MHz when used with PLL
  - User tunable to compensate for frequency drift
- Secondary oscillator using Timer1 @ 32 kHz
- Fail-Safe Clock Monitor
  - Allows for safe shutdown if peripheral clock stops

Device Features

Peripherals:

- High-current sink/source 25 mA/25 mA
- Three programmable external interrupts
- Four input change interrupts
- Up to 2 Capture/Compare/PWM (CCP) modules, one with Auto-Shutdown (28-pin devices)
- Enhanced Capture/Compare/PWM (ECCP) module (40/44-pin devices only):
  - One, two or four PWM outputs
  - Selectable polarity
  - Programmable dead time
  - Auto-Shutdown and Auto-Restart

Device Features

Peripherals:

- Master Synchronous Serial Port (MSSP) module supporting 3-wire SPI™ (all 4 modes) and I^2C™ Master and Slave modes
- Enhanced Addressable USART module:
  - Supports RS-485, RS-232 and LIN 1.2
  - RS-232 operation using internal oscillator block (no external crystal required)
  - Auto-Wake-up on Start bit
  - Auto-Baud Detect
- 10-bit, up to 13-channel Analog-to-Digital Converter module (A/D):
  - Auto-acquisition capability
  - Conversion available during Sleep
- Dual analog comparators with input multiplexing
- Programmable 16-level High/Low-Voltage Detection (HLVD) module:
  - Supports interrupt on High/Low-Voltage Detection

Configuration Bits

There are a number of what are called **configuration bits** or **fuses** that are associated with the 18F4620.

These are bits that cannot be changed during the running of the program, but can only be set when the device is programmed.

These bits can either be set within the programmer software, or using compiler directives.

The latter is a better approach.

Configuration Bits

Configuration bits control a number of things and are discussed under "Special Features of the CPU" in the documentation.

TABLE 23-1: CONFIGURATION BITS AND DEVICE IDs

File Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300001h	CONFIG1H	IESO	FCMEN	—	—	FOSC3	FOSC2	FOSC1	FOSC0	00-- 0111
300002h	CONFIG2L	—	—	—	BORV1	BORV0	BOREN1	BOREN0	PWRTEN	---1 1111
300003h	CONFIG2H	—	—	—	WDTPS3	WDTPS2	WDTPS1	WDTPS0	WDTEN	---1 1111
300005h	CONFIG3H	MCLRE	—	—	—	—	LPT1OSC	PBADEN	CCP2MX	1--- -011
300006h	CONFIG4L	DEBUG	XINST	—	—	—	LVP	—	STVREN	10-- -1-1
300008h	CONFIG5L	—	—	—	—	CP3(1)	CP2	CP1	CP0	---- 1111
300009h	CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
30000Ah	CONFIG6L	—	—	—	—	WRT3(1)	WRT2	WRT1	WRT0	---- 1111
30000Bh	CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111- ----
30000Ch	CONFIG7L	—	—	—	—	EBTR3(1)	EBTR2	EBTR1	EBTR0	---- 1111
30000Dh	CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
3FFFFEh	DEVID1(1)	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	xxxx xxxx(2)
3FFFFFh	DEVID2(1)	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	0000 1100

Configuration Bits

These bits are set in pragma directives:

```
#pragma DATA 0x300001, _OSC_HS_1H  // HS osc
#pragma DATA 0x300003, _WDT_OFF_2H // wdt off
#pragma DATA 0x300006, _LVP_OFF_4L // lvp off
#pragma DATA _CONFIG3H, _MCLRE_ON_3H  //enable mclr
```

These are using definitions found in the system.h include file.

Configuration Bits

The location of the config register is defines:

```
#define _CONFIG1H            0x00300001
#define _CONFIG2L            0x00300002
#define _CONFIG2H            0x00300003
#define _CONFIG3H            0x00300005
#define _CONFIG4L            0x00300006
```

Various bit in the resisters are defined also:

```
#define _OSC_HS_1H           0x000000F2 // HS
#define _OSC_RC_1H           0x000000F3 // RC
#define _OSC_EC_1H           0x000000F4 // EC-OSC2 as Clock Out
#define _OSC_ECIO6_1H        0x000000F5 // EC-OSC2 as RA6
#define _OSC_HSPLL_1H        0x000000F6 // HS-PLL Enabled
```

Configuration Bits

To turn off the watch dog timer:

```
#pragma DATA 0x300003, _WDT_OFF_2H
```

or

```
#pragma DATA _CONFIG2H, _WDT_OFF_2H
```

To set the oscillator to high speed and multiply the crystal speed by 4:

```
#pragma DATA 0x300001, _OSC_HSPLL_1H //40 mhz
```

Things like the latter should be done advisedly, because the 18LF4620 will not operate at 40 MHz at lower voltages.

Configuration Bits

I recommend that you:

- Turn off the watch dog timer, until you are sure that you want to use it.
- Turn off low voltage programming
- The boards that you will be using have 10MHz crystals on them. You should set the oscillator to HS or HSPLL, depending on the desired speed.

In addition, there is a pragma for the clock frequency:

```
#pragma CLOCK_FREQ 10000000
```

This directive is necessary if you use any of the built in delay routines, as the program needs to know how fast the clock is.

18F4620 Ports

In the following slides, we will look at the ports (I/O) available on the microcontroller form both an electrical and software point of view.

Port Electrical Characteristics

When connecting things to ports, we are concerned with the current and voltage at the pins.

The electrical spec specifies:

- V_{IH} – Voltage that will be interpreted as high on an input.
- V_{IL} – Voltage that will be interpreted as low on an input
- V_{OH} – Voltage on the pin in the output high state.
- V_{OL} – Voltage on the pin in the output low state.
- I_{OH} – Current a pin will source in the high output state.
- I_{OL} – Current a pin will sink in the low output state.

Note that as inputs, the impedance is very high, and thus there is very little current into the device. This is called “leakage” current and is on the order of 1μA.

Port Electrical Characteristics

The input voltages are the easiest, since the input impedance is so high.

For the 4620 running at 5.0 volts, these are the values:

V_{IH}	**minimum**	**2.0 volts**
V_{IL}	**maximum**	**0.8 volts**

This means that the minimum voltage on an input pin that is guaranteed to be read as "high" is 2.0 volts, and the maximum voltage on an input pin that is guaranteed to be read as "low" is 0.8 volts.

Port Electrical Characteristics

26.3 DC Characteristics: PIC18F2525/2620/4525/4620 (Industrial)
PIC18LF2525/2620/4525/4620 (Industrial)

DC CHARACTERISTICS			**Standard Operating Conditions (unless otherwise stated)** Operating temperature -40°C ≤ TA ≤ +85°C for industrial			
Param No.	**Symbol**	**Characteristic**	**Min**	**Max**	**Units**	**Conditions**
	V_{IL}	**Input Low Voltage**				
		I/O ports:				
D030		with TTL buffer	V_{SS}	0.15 V_{DD}	V	V_{DD} < 4.5V
D030A			—	0.8	V	4.5V ≤ V_{DD} ≤ 5.5V
D031		with Schmitt Trigger buffer	V_{SS}	0.2 V_{DD}	V	
D032		$\overline{\text{MCLR}}$	V_{SS}	0.2 V_{DD}	V	
D033		OSC1	V_{SS}	0.3 V_{DD}	V	HS, HSPLL modes
D033A		OSC1	V_{SS}	0.2 V_{DD}	V	RC, EC modes(1)
D033B		OSC1	V_{SS}	0.3 V_{DD}	V	XT, LP modes
D034		T13CKI	V_{SS}	0.3 V_{DD}	V	
	V_{IH}	**Input High Voltage**				
		I/O ports:				
D040		with TTL buffer	0.25 V_{DD} + 0.8V	V_{DD}	V	V_{DD} < 4.5V
D040A			2.0	V_{DD}	V	4.5V ≤ V_{DD} ≤ 5.5V
D041		with Schmitt Trigger buffer	0.8 V_{DD}	V_{DD}	V	
D042		$\overline{\text{MCLR}}$	0.8 V_{DD}	V_{DD}	V	
D043		OSC1	0.7 V_{DD}	V_{DD}	V	HS, HSPLL modes
D043A		OSC1	0.8 V_{DD}	V_{DD}	V	EC mode
D043B		OSC1	0.9 V_{DD}	V_{DD}	V	RC mode(1)
D043C		OSC1	1.6	V_{DD}	V	XT, LP modes
D044		T13CKI	1.6	V_{DD}	V	

Port Electrical Characteristics

Outputs aren't as easy, since we must consider both the voltage and the current. For example, if we short an output pin to ground, we shouldn't expect it to produce a "high" output voltage.

V_{OH}	**minimum**	**4.3 volts**	**I_{OH} < -3 mA**
V_{OL}	**maximum**	**0.6 volts**	**I_{OL} < 8.5 mA**

The V_{OH} spec says a high output will be a minimum of 4.3 volts as long as the current out of the pin is less than 3 mA.

The V_{OL} spec says a low output will be a maximum of 0.6 volts as long as the current into the pin is less than 8.5 mA.

Note that current into the device is defined as positive.

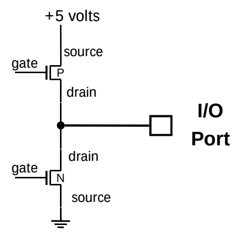
Port Electrical Characteristics

26.3 DC Characteristics: PIC18F2525/2620/4525/4620 (Industrial)
PIC18LF2525/2620/4525/4620 (Industrial) (Continued)

DC CHARACTERISTICS			***Standard Operating Conditions (unless otherwise stated)*** Operating temperature -40°C ≤ TA ≤ +85°C for industrial			
Param No.	**Symbol**	**Characteristic**	**Min**	**Max**	**Units**	**Conditions**
	V_{OL}	**Output Low Voltage**				
D080		I/O ports	—	0.6	V	I_{OL} = 8.5 mA, V_{DD} = 4.5V, -40°C to +85°C
D083		OSC2/CLKO (RC, RCIO, EC, ECIO modes)	—	0.6	V	I_{OL} = 1.6 mA, V_{DD} = 4.5V, -40°C to +85°C
	V_{OH}	**Output High Voltage(3)**				
D090		I/O ports	V_{DD} – 0.7	—	V	I_{OH} = -3.0 mA, V_{DD} = 4.5V, -40°C to +85°C
D092		OSC2/CLKO (RC, RCIO, EC, ECIO modes)	V_{DD} – 0.7	—	V	I_{OH} = -1.3 mA, V_{DD} = 4.5V, -40°C to +85°C

Port Electrical Characteristics

Remember that the output stage is a couple of transistors. A MOSFET (in the triode operating region) looks like a small resistor. We should expect a resistor like relationship between the voltage and current at the pin.



Port Electrical Characteristics

In addition to sourcing 3.5 mA and sinking 8mA, there is also a restriction on the total current sourced or sunk by groups of I/O ports.

Added together, the current sourced or sunk by all ports combined can't exceed 200 mA.

The absolute maximum information is found in the device document.

Port Electrical Characteristics

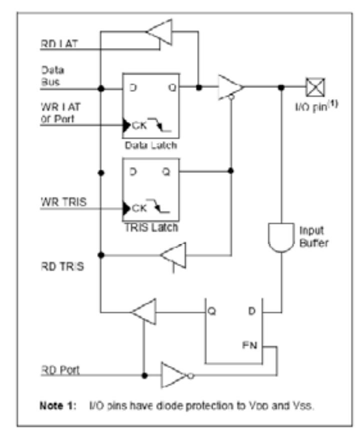
26.0 ELECTRICAL CHARACTERISTICS

Absolute Maximum Ratings(†)

Ambient temperature under bias	-40°C to +125°C
Storage temperature	-65°C to +150°C
Voltage on any pin with respect to VSS (except VDD and $\overline{MCLR}$)	-0.3V to (VDD + 0.3V)
Voltage on VDD with respect to VSS	-0.3V to +7.5V
Voltage on $\overline{MCLR}$ with respect to VSS (Note 2)	0V to +13.25V
Total power dissipation (Note 1)	1.0W
Maximum current out of VSS pin	300 mA
Maximum current into VDD pin	250 mA
Input clamp current, IIK (VI < 0 or VI > VDD)	±20 mA
Output clamp current, IOK (VO < 0 or VO > VDD)	±20 mA
Maximum output current sunk by any I/O pin	25 mA
Maximum output current sourced by any I/O pin	25 mA
Maximum current sunk by all ports	200 mA
Maximum current sourced by all ports	200 mA

18F4620 Ports

Many of the ports on the 4620 have special functions, and therefore have different electronics, but the basic port looks like this:

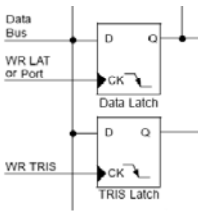


18F4620 Ports

There are two latches (flip-flops) associated with each bit of the port.

The top one holds the data that is being output to the port.

The bottom one controls whether this bit is an input or an output.

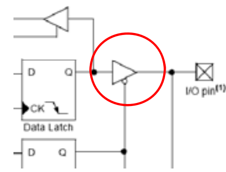


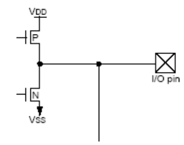
Holds the data being output

Determines if the port is an input port or an output port.

18F4620 Ports

The tri-state gate lets ports be either outputs or inputs.

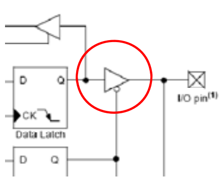


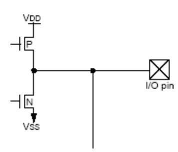


18F4620 Ports

To be used as an output, one or the other of the transistors is on, allowing data to be sent to the I/O pin.

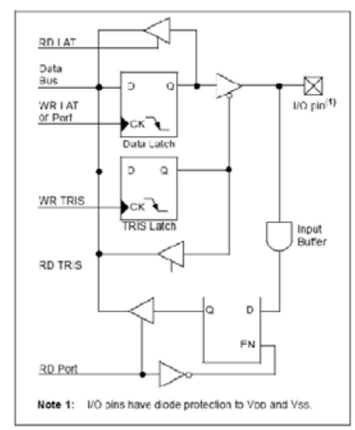
When both transistors are off, the "output" is disconnected and the input can be read.





18F4620 Ports

Note: There are a variety of read and write signals that do different thing.



Basic I/O

So how do I uses these ports in a C program?

There are three registers associated with each I/O port. They are called:

- **port**
- **lat**
- **tris**

Tris is the register associated with the direction (input or output of the pin)

Port is where your read from to see what the inputs are.

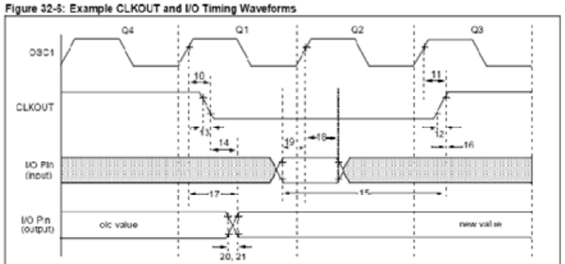
Lat is where you write to (does the same thing as writing to port)

Basic I/O

Note that the device does a read-modify-write. That means that the state of the port is first read, then it is modified, then it is reloaded into the register.

In some circumstances, this can cause issues if you are writing and immediately reading and the signal hasn't had a chance to settle.

Figure 32-5: Example CLKOUT and I/O Timing Waveforms



Note: Refer to Figure 32-4 for load conditions.

4620 Ports

The 4620 has:

- **4 8-bit ports named a, b, c, and d.**
- **One 3-bit port named e.**

Thus there is, associated with port a, three 8-bit registers:

- **porta**
- **trisa**
- **lata**

The same is true, mutatis mutandis, for the other ports.

4620 Ports

The registers associated with each port (and all of the rest of the registers that we will discuss) are memory mapped into particular locations.

Fortunately, the header file <system.h> makes it easy to use these memory mapped registers.

Address	Name
F9Fh	IPR1
F9Eh	PIR1
F9Dh	PIE1
F9Ch	—[2]
F9Bh	OSCTUNE
F9Ah	—[2]
F99h	—[2]
F98h	—[2]
F97h	—[2]
F96h	TRISE[3]
F95h	TRISD[3]
F94h	TRISC
F93h	TRISB
F92h	TRISA
F91h	—[2]
F90h	—[2]
F8Fh	—[2]
F8Eh	—[2]
F8Dh	LATE[3]
F8Ch	LATD[3]
F8Bh	LATC
F8Ah	LATB
F89h	LATA
F88h	—[2]
F87h	—[2]
F86h	—[2]
F85h	—[2]
F84h	PORTE[3]
F83h	PORTD[3]
F82h	PORTC
F81h	PORTB
F80h	PORTA

system.h

The system.h file defines register names (in all caps) as:

```
#define PORTA          0x00000F80
#define PORTB          0x00000F81
#define PORTC          0x00000F82
#define PORTD          0x00000F83
#define PORTE          0x00000F84
#define LATA           0x00000F89
#define LATB           0x00000F8A
#define LATC           0x00000F8B
#define LATD           0x00000F8C
#define LATE           0x00000F8D
#define DDRA           0x00000F92
#define TRISA          0x00000F92
#define DDRB           0x00000F93
#define TRISB          0x00000F93
#define DDRC           0x00000F94
#define TRISC          0x00000F94
#define DDRD           0x00000F95
#define TRISD          0x00000F95
#define DDRE           0x00000F96
```


system.h

The system.h file defines registers in lower case using

```
volatile char porta      @PORTA;
volatile char portb      @PORTB;
volatile char portc      @PORTC;
volatile char portd      @PORTD;
volatile char porte      @PORTE;
volatile char lata       @LATA;
volatile char latb       @LATB;
volatile char latc       @LATC;
volatile char latd       @LATD;
volatile char late       @LATE;
volatile char ddra       @DDRA;
volatile char trisa      @DDRA;
volatile char ddrb       @DDRB;
volatile char trisb      @DDRB;
volatile char ddrc       @DDRC;
volatile char trisc      @DDRC;
volatile char ddrd       @DDRD;
volatile char trisd      @DDRD;
volatile char ddre       @DDRE;
volatile char trise      @DDRE;
```


system.h

The net effect is that registers names (in lower case) can be used as ordinary variables.

This is true of all of the registers, not just the registers associated with the ports.

Parallel I/O

Suppose we have LED's connected to port a on the microcontroller. If we wish to send the value of a variable data to the LED's, we need to do two things:

- **Make port a into an output port**
- **Send the desired data to port a**

In C, this is pretty trivial:

trisa = 0; // set all bits of port a output

lata = data; // send data to the led's

Parallel I/O

If we have an input device connected to bit 4 of port b (such as a switch that will make the I/O pin 0 or 5 volts)

- Make bit 4 of port b into an input
- Read in the data

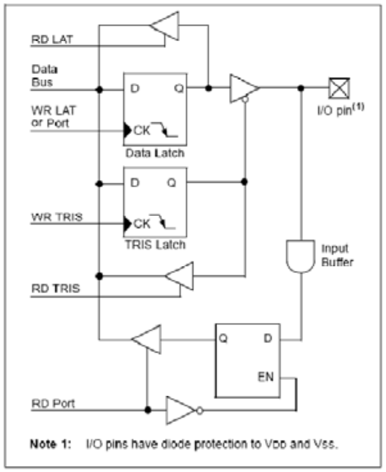
Again, this is pretty trivial:

```
trisb |= 00010000b;  // set bit 4 input
data = portb;  // read in the port
```

Note:

- reading portb reads the input lines; reading latb reads the output latch.

Parallel I/O



More Power!!!!

There will be many situations where the limited ability of an output port to source/sink current is insufficient to drive the output device being controlled, or the required voltage is higher than 5 volts.

Examples include:

- Higher current LEDs
- Turning motors on and off
- Activating relays, solenoids, etc.

The solutions to these problems is to use an external device such as a transistor to provide sufficient current/voltage for the device, and use the limited power of the microcontroller output to turn the external device on and off.

Serial I/O

One of the most common interfaces found on computers is the serial interface.

Bytes of data are sent in a serial fashion, that is, one bit at a time.

The standard for serial interface is called RS-232. It is used to send data over distances on the order of 25 feet.

It is gradually being replaced in modern computers by the faster and more flexible USB (Universal Serial Bus) interface.

RS-232 is still very common, very simple to implement, and well supported by software.

Serial I/O

Why worry about serial I/O in a microcontroller?

- **There are many applications where either the major function of an embedded system, or an auxiliary function, is the logging of data. Microcontrollers typically don't have enough memory to store significant amounts of data, but a serial link can be used to allow a laptop or other computer to log data sent to it by the microcontroller.**
- **If you are developing software for an embedded control application, and do not have an integral display (such as the LCD display on the systems we are using in the lab), a serial connection allows data to be written to a computer for debugging purposes. It is worth the little extra hardware to have this ability.**

Serial I/O

Why worry about serial I/O in a microcontroller?

- **RS-232 is the simplest form of serial I/O, but there are many other serial I/O standards that evolved from the desire to interface things to microcontrollers without using up lots of I/O port pins.**
- **Other common serial interfaces are SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), and various other forms of one wire and three wire interfaces.**

Serial I/O

There are several things that must be done to connect a microcontroller to a computer.

- **You need an application running on the computer that can accept serial data. A commonly available application is HyperTerminal, which is a simple terminal emulation program that is standard software on all PC's.**
- **Data sent between the microcontroller and the computer is sent using particular voltages (that are different from the standard voltages found on a microcontroller.) This requires level conversion.**
- **You need software in the microcontroller that will read and write serial data.**

Serial Data Format

If a data byte with the bits labeled b_7 – b_0 is sent over a serial link, the format looks like this:

Start Bit	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	Stop Bit

The line is normally high, and the start bit begins a transmission by going low. Each bit of the byte being sent follows as a 1 or a 0. Finally, the stop bit is sent as a 1.

Serial Data Format

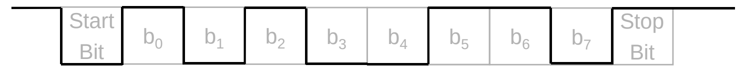
If a data byte with the bits labeled b_7 – b_0 is sent over a serial link, the format looks like this:

Start Bit	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	Stop Bit

The line is normally high, and the start bit begins a transmission by going low. Each bit of the byte being sent follows as a 1 or a 0. Finally, the stop bit is sent as a 1.

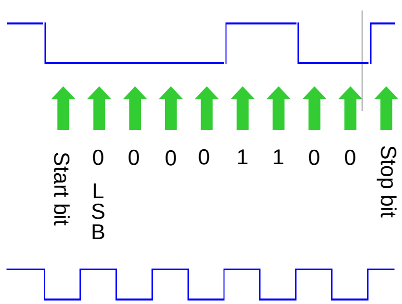
Serial Data Format

Example: Suppose we are sending the ASCII for the character "e", which is 65_H. The bits sent would look like:



The start bit signals the beginning of the transmission, and the stop bit ends the transmission.

Logic Analyzer Trace



Serial data

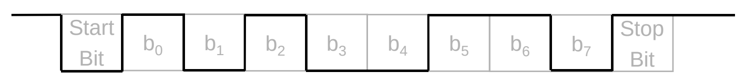
Data is: 00110000
30H
ASCII for the number 0

Baud rate clock (not present external to the microcontroller)

Serial Data Format

The example shown is for sending data with 8 bits and no parity (referred to as 8-none). Another possibility (since standard ASCII characters are only 7 bits long) is to send 7 bits of data, and one parity.

The parity can be set so that there are an even (called even parity) or odd (called odd parity) number of 1's in the transmission. This allows for simple error checking.



Serial Data Format

This form of serial communications is called asynchronous, because there is no common clock between the sender and the receiver.

The sender and receiver must be set to the same “baud” rate. In its simplest form, the baud rate is the number of bits sent per second.

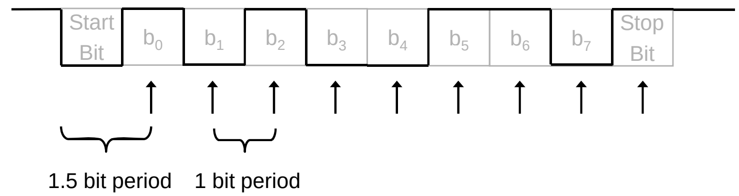
9600 baud is 9600 bits per second.

Serial Data Format

At 9600 baud, each bit period is about 104 μsec. The sender sends bits for that length of time.

The receiver watches for the start bit. When that transition occurs, it checks for bits based on its local clock, checking the bit in the center of each period.

If the clocks are close enough, the checks won’t drift out of the correct bit period by the end of the reception.



Connectors and Line Format

The standard connector for a serial interface today is the DB-9, a 9 pin connector. (Older serial interfaces use DB-25 connectors.)

There are separate wires for transmit and receive, a signal ground wire, and several flow control signals. In most applications today, the flow control signals can be ignored.

The two ends of the connection are called Data Terminal Equipment (DTE) (e.g. a computer) and Data Communications Equipment (DCE) (e.g. a modem).

Connectors and Line Format

If DCE is connected to DTE, a straight through cable is used.

Modem Cable - Straight Cable DB9 to DB9

DTE Device (Computer) DB9		DTE to DCE Connections	DCE Device (Modem) DB9	
Pin# DB9 RS-232 Signal Names		Signal Direction	Pin# DB9 RS-232 Signal Names	
#1 Carrier Detector (DCD)	CD	←	#1 Carrier Detector (DCD)	CD
#2 Receive Data (Rx)	RD	←	#2 Receive Data (Rx)	RD
#3 Transmit Data (Tx)	TD	→	#3 Transmit Data (Tx)	TD
#4 Data Terminal Ready	DTR	→	#4 Data Terminal Ready	DTR
#5 Signal Ground/Common (SG)	GND		#5 Signal Ground/Common (SG)	GND
#6 Data Set Ready	DSR	←	#6 Data Set Ready	DSR
#7 Request to Send	RTS	→	#7 Request to Send	RTS
#8 Clear to Send	CTS	←	#8 Clear to Send	CTS
#9 Ring Indicator	RI	←	#9 Ring Indicator	RI
Soldered to DB9 Metal - Shield	FGND		Soldered to DB9 Metal - Shield	FGND

Connectors and Line Format

- It is much less clear today what is DTE and what is DCE, and it is not uncommon to connect like devices over a serial link (computer to computer for example.)
- If this is the situation, the cable has to cross signals so that one ends transmit is connected to the other ends receive, and vice versa.
- Practical advice: If you serial link doesn't work, and the setup of each end is the same (baud rate, number of bits, parity, flow control, etc.) try swapping the wires on pins 2 and 3 on your cable.

Connectors and Line Format

- The actual signals sent are not 5 volt signals. RS-232 sends a negative voltage (typically -12 volts) to signify a "1" and a positive voltage (typically +12 volts) to signify a "0".
- This used to be a pain, because it meant that in addition to the 5 volt supply for you microcontroller, you needed a +12 and a -12 volt supply for the serial connection.
- Modern technology has come to the rescue with a device called the MAX232, which will take in a signal that is 0 or 5 volts and put out a signal that is +12 or -12 volts (using only a 5 volt supply!)

The USART

- USART stands for Universal Synchronous Asynchronous Receiver transmitter. It is a hardware device built into computers and microcontrollers that accepts a byte from the computer and shifts the byte our serially, and accepts a serial set of bit and gives it to the computer in parallel.
- There is a USART inside the 18F4620, and it is a common feature of microcontrollers. (Note that if your application calls for a serial connection, you should choose a microcontroller with a built in USART.)

The USART

- The USART greatly simplifies the task of serial communications. It is set up for the desired baud rate and number of bits, and then the microcontroller need only give it the byte to send, and the USART does the rest.
- On the receive side, the USART receives the serial bit stream and gives the corresponding byte to the microcontroller.
- In the 18F4620 USART, the transmitter and receiver are functionally separate, but share the same baud rate generator

Setting the Baud Rate

The baud rate is the speed with which the bits are transmitted. Both ends of the serial connection need to be set to the same baud rate.

There are a number of standard baud rates.

The baud rate in the 18F4620 is controlled by a number of registers:

TABLE 18-2: REGISTERS ASSOCIATED WITH BAUD RATE GENERATOR

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	51
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	51
BAUDCON	ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	51
SPBRGH	EUSART Baud Rate Generator Register High Byte								51
SPBRG	EUSART Baud Rate Generator Register Low Byte								51

Legend: — = unimplemented, read as '0'. Shaded cells are not used by the BRG.

Setting the Baud Rate

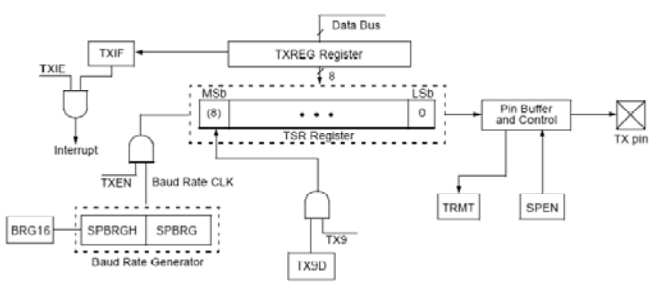
These table found in the documentation tell you how to set SPBRG for the desired baud rate.

There is often more than one choice for a given rate, so you should use the one with the smallest error.

BAUD RATE (K)	SYNC = 0, BRGH = 1, BRG16 = 1 or SYNC = 1, BRG16 = 1											
	Fosc = 40.000 MHz			Fosc = 20.000 MHz			Fosc = 10.000 MHz			Fosc = 8.000 MHz		
	Actual Rate (K)	% Error	SPBRG value (decimal)	Actual Rate (K)	% Error	SPBRG value (decimal)	Actual Rate (K)	% Error	SPBRG value (decimal)	Actual Rate (K)	% Error	SPBRG value (decimal)
0.3	0.300	0.00	33332	0.300	0.00	16665	0.300	0.00	8332	300	-0.01	6665
1.2	1.200	0.00	8332	1.200	0.02	4165	1.200	0.02	2082	1200	-0.04	1665
2.4	2.400	0.02	4165	2.400	0.02	2082	2.402	0.06	1040	2400	-0.04	832
9.6	9.606	0.06	1040	9.596	-0.03	520	9.615	0.16	259	9615	-0.16	207
19.2	19.193	-0.03	520	19.231	0.16	259	19.231	0.16	129	19230	-0.16	103
57.6	57.803	0.35	172	57.471	-0.22	86	58.140	0.94	42	57142	0.79	34
115.2	114.943	-0.22	86	116.279	0.94	42	113.636	-1.36	21	117647	-2.12	16

The 18F4620 USART Transmitter

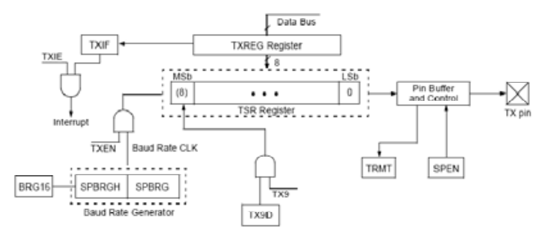
The transmitter of the USART looks like this. The main function is to shift out a byte serially, and this is done by the TSR register.



The 18F4620 USART Transmitter

There are a number of interrupt flags, interrupt enables, and other bits that are associated with the transmitter:

- TXEN – Transmitter Enable
- SPEN Serial Port Enable
- TXIF and TXIE – Transmit interrupt flag and enable
- TRMT – Transmitter empty (MT) flag
- RC6 – External pin on the 18F4620

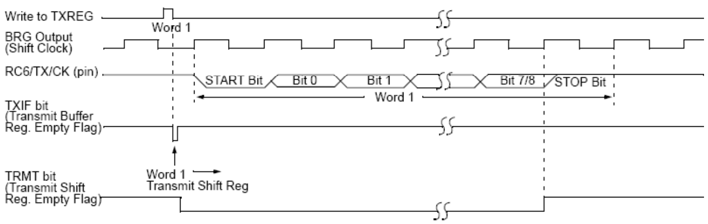


The 18F4620 USART Transmitter

Serial transmission looks like this:

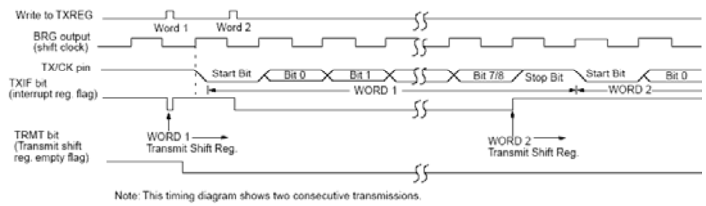
Note:

- **TXIF indicates that the TXREG is empty and can accept another byte**
- **TRMT indicates that the transmit shift register is empty.**
- **These are slightly different things.**



The 18F4620 USART Transmitter

The difference between TRMT and TXIF can be seen by looking at back to back serial transmissions. The TXIF tells us that we can write another byte to be sent to TXREG. (Note that we can either of these flags, and don't have to use interrupts.)



18F4620 Transmitter Registers

The registers involved with serial transmission are:

- **SPBRGH and SPBRGL – Get the correct value for the desired baud rate based on the system clock speed and BRGH.**
- **TXREG – Location to place a byte to be transmitted out the serial port.**
- **TXSTA – Transmitter status register.**
- **RCSAT – Receiver status register. SPEN (serial port enable bit is found here.)**
- **PIE1 and PIR1 – Peripheral Interrupt Enable register and Peripheral Interrupt Register (home to TXIE and TXIF respectively).**
- **INTCON – Global interrupt enable and peripheral interrupt enable.**

18F4620 Transmitter Registers

TABLE 18-5: REGISTERS ASSOCIATED WITH ASYNCHRONOUS TRANSMISSION

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
PIR1	PSPIF[(1)]	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	52
PIE1	PSPIE[(1)]	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	52
IPR1	PSPIP[(1)]	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	52
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	51
TXREG	EUSART Transmit Register								51
TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	51
BAUDCON	ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	51
SPBRGH	EUSART Baud Rate Generator Register High Byte								51
SPBRG	EUSART Baud Rate Generator Register Low Byte								51

Transmitter Status Register

The main register for serial transmission is TXSTA.

REGISTER 18-1: TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7							bit 0

bit 7 **CSRC:** Clock Source Select bit
Asynchronous mode:
Don't care.
Synchronous mode:
1 = Master mode (clock generated internally from BRG)
0 = Slave mode (clock from external source)

bit 6 **TX9:** 9-bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN:** Transmit Enable bit
1 = Transmit enabled
0 = Transmit disabled

Note: SREN/CREN overrides TXEN in Sync mode.

bit 3 **SENDB:** Send Break Character bit
Asynchronous mode:
1 = Send Sync Break on next transmission (cleared by hardware upon completion)
0 = Sync Break transmission completed
Synchronous mode:
Don't care.

bit 2 **BRGH:** High Baud Rate Select bit
Asynchronous mode:
1 = High speed
0 = Low speed
Synchronous mode:
Unused in this mode.

bit 1 **TRMT:** Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** 9th bit of Transmit Data
Can be address/data bit or a parity bit.

BAUDCON register

The BUADCON register is associated with both transmit and receive.

REGISTER 18-3: BAUDCON: BAUD RATE CONTROL REGISTER

R/W-0	R-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN
bit 7							bit 0

bit 7 **ABDOVF**: Auto-Baud Acquisition Rollover Status bit
1 = A BRG rollover has occurred during Auto-Baud I
(must be cleared in software)
0 = No BRG rollover has occurred

bit 6 **RCIDL**: Receive Operation Idle Status bit
1 = Receive operation is Idle
0 = Receive operation is active

bit 5 **Unimplemented:** Read as '0'

bit 4 **SCKP**: Synchronous Clock Polarity Select bit
Asynchronous mode:
Unused in this mode.
Synchronous mode:
1 = Idle state for clock (CK) is a high level
0 = Idle state for clock (CK) is a low level

bit 3 **BRG16**: 16-bit Baud Rate Register Enable bit
1 = 16-bit Baud Rate Generator – SPBRGH and SPBRG
0 = 8-bit Baud Rate Generator – SPBRG only (Compatible mode), SI

bit 2 **Unimplemented:** Read as '0'

bit 1 **WUE:** Wake-up Enable bit
Asynchronous mode:
1 = EUSART will continue to sample the RX pin – interrupt gener
cleared in hardware on following rising edge
0 = RX pin not monitored or rising edge detected
Synchronous mode:
Unused in this mode.

bit 0 **ABDEN**: Auto-Baud Detect Enable bit
Asynchronous mode:
1 = Enable baud rate measurement on the next character. Requires
(55h); cleared in hardware upon completion
0 = Baud rate measurement disabled or completed
Synchronous mode:
Unused in this mode.

Transmitter

To setup the serial transmitter:

- **Set SPBRGH and SPBRGL based on the system clock, and your choice of BRGH and BRG16.**
- **Set TXSTA for 8 bit asynchronous transmission with the correct value of BRGH.**
- **Set the correct values into the BAUDCON register.**
- **Enable the serial port (bit SPEN found in RXSTA)**

To send serial data:

- **Be sure that the TXREG is empty (either by polling TXIF or TRMT)**
- **Write the byte to be sent to TXREG.**

Transmitter and Interrupts

The determination of whether to use interrupts for serial transmission depends on the application.

Interrupts are most useful when you are sending a string of characters that you have already created. If this is the case, you can design your software to use the TXIF interrupt to load the next character in the string and send it.

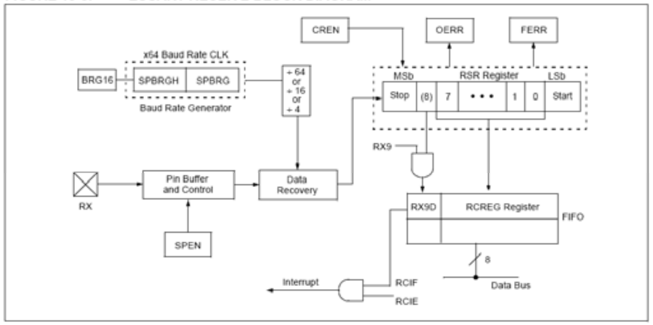
For many other applications, it is easier just to poll either TXIF or TRMT.

If using interrupts, note that in addition to setting TXIE to enable the TXIF interrupt, you need to enable global interrupts (GIE) and also peripheral interrupts (PIE).

The 18F4620 USART Receiver

The receiver of the USART looks like this. The main function is to shift in a byte serially, and this is done by the RSR register.

FIGURE 18-6: EUSART RECEIVE BLOCK DIAGRAM

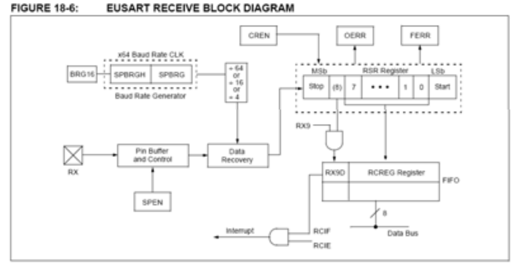


The 18F4620 USART Receiver

The basic setup is the same as the transmitter.

- Baud rate is based on the same SPBRG value as the transmitter.
- Data shifts in serially, and can be read from RCREG.
- Data comes in on external pin RC7.
- RCIE and RCIF are the Receiver Interrupt Enable and Receiver Interrupt flag respectively.

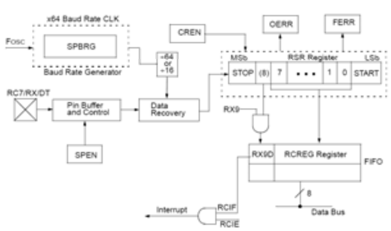
FIGURE 18-6: EUSART RECEIVE BLOCK DIAGRAM



The 18F4620 USART Receiver

The receiver is a little more complicated to deal with, for several reasons:

- When a byte is appears in the register is not under the control of the receiver, but depends on whatever is sending the data.
- The receiver must be able to detect bad things that might happen during transmission.



The 18F4620 USART Receiver

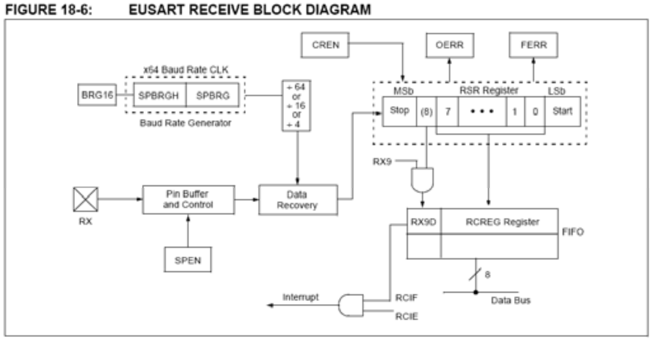
Receiver Complications:

- Since an external source is determining when bytes are sent, the microcontroller must be checking for data and reading it from the receiver, otherwise an error called and "overrun" will occur. This means that more bytes were received than can be held in the receiver for your program to read, and thus you missed some data.
- Almost every USART stops receiving data when this happens, and sets a flag (called the overrun error flag or OERR).
- If this occurs, you must reset the receiver to clear the error.
- To help this occur less frequently, the RCREG in the 18F4620 is a 2 byte FIFO (First In First Out) register that can hold to successive receptions.
- Another type of error that can be detected is a framing error, where the receiver doesn't find the stop bit where it is expected. (This often means a baud rate mismatch between transmitter and receiver.

The 18F4620 USART Receiver

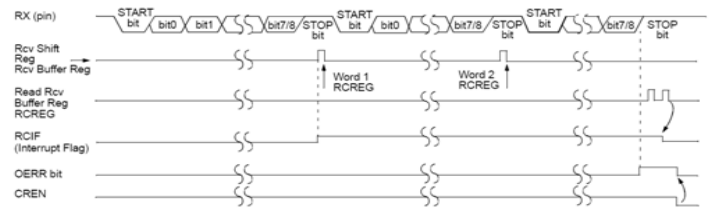
Notice the framing error and overrun error flags, and also that RCREG is actually a 2 byte FIFO.

FIGURE 18-6: EUSART RECEIVE BLOCK DIAGRAM



The 18F4620 USART Receiver

The timing diagram:



Note: This timing diagram shows three words appearing on the RX input. The RCREG (receive buffer) is read after the third word, causing the OERR (overrun) bit to be set.

18F4620 Receiver Registers

The registers involved with serial reception are:

- **SPBRGH and SPBRGL – Gets the correct value for the desired baud rate based on the system clock speed and BRGH.**
- **RCREG – 2 byte FIFO that hold the received data for reading by the microcontroller.**
- **RCSTA – Receiver status register.**
- **TXSTA – Home of BRGH**
- **PIE1 and PIR1 – Peripheral Interrupt Enable register and Peripheral Interrupt Register (home to TXIE and TXIF respectively).**
- **INTCON – Global interrupt enable (GIE) and peripheral interrupt enable (PIE).**

18F4620 Receiver Registers

TABLE 18-6: REGISTERS ASSOCIATED WITH ASYNCHRONOUS RECEPTION

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	52
PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	52
IPR1	PSPIP(1)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	52
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	51
RCREG	EUSART Receive Register								51
TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	51
BAUDCON	ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	51
SPBRGH	EUSART Baud Rate Generator Register High Byte								51
SPBRG	EUSART Baud Rate Generator Register Low Byte								51

Receiver Status Register

The main register for serial reception is TXSTA.

RCSTA: RECEIVE STATUS AND CONTROL REGISTER (ADDRESS 18h)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

bit 7 **SPEN:** Serial Port Enable bit
1 = Serial port enabled (configures RC7/RX/DT and RC6/TX/CK pins as serial port pins)
0 = Serial port disabled

bit 6 **RX9**: 9-bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN**: Single Receive Enable bit
Asynchronous mode:
Don't care
Synchronous mode - master:
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.
Synchronous mode - slave:
Don't care

bit 4 **CREN**: Continuous Receive Enable bit
Asynchronous mode:
1 = Enables continuous receive
0 = Disables continuous receive

Receiver Status Register

The main register for serial reception is TXSTA.

RCSTA: RECEIVE STATUS AND CONTROL REGISTER (ADDRESS 18h)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

bit 3 **ADDEN:** Address Detect Enable bit
Asynchronous mode 9-bit (RX9 = 1):
1 = Enables address detection, enables interrupt and load of the receive buffer when RSR<8> is set
0 = Disables address detection, all bytes are received, and ninth bit can be used as parity bit

bit 2 **FERR**: Framing Error bit
1 = Framing error (can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

bit 1 **OERR**: Overrun Error bit
1 = Overrun error (can be cleared by clearing bit CREN)
0 = No overrun error

bit 0 **RX9D:** 9th bit of Received Data (can be parity bit, but must be calculated by user firmware)

Receiver

To setup the serial receiver:

- **Set SPBRG based on the system clock, and your choice of BRGH.**
- **Be sure BRGH is set appropriately (in TXSTA).**
- **Set RXSTA for 8 bit asynchronous transmission and enable the serial port (SPEN)**

To receive serial data:

- **Wait for data to appear (you can use interrupts or poll the RCIF)**
- **Read data from the RCREG.**
- **Be sure your software can deal with errors (overrun and framing, particularly the former.)**

Receiver and Interrupts

The determination of whether to use interrupts for serial reception depends on the application, but it is often more advantageous in reception, since the microcontroller does not know when data is going to occur.

For some applications, it is easier just to poll either RCIF to see when data is available.

It is wise to avoid turning on data reception until you are ready to handle it, otherwise overrun errors may occur.

It is also wise to check for that error as part of your routine, particularly if you are polling RCIF.

If using interrupts, note that in addition to setting RCIE to enable the RCIF interrupt, you need to enable global interrupts (GIE) and also peripheral interrupts (PIE).

Serial I/O Software

Using the USART functions on a microcontroller, we can write software routines the do the basic functions of writing and reading characters to and from the serial port. We will assume that we are talking to a terminal application.

We need three low level routines:

- Initialize the USART
- putc() -- send a character out on the usart
- getc() -- gets a character from the usart

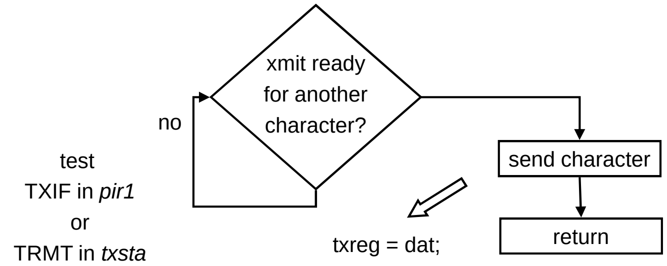
We have already discussed the registers that need to be set up in the init routine.

putc

This function sends a character to the terminal.

It is logical to have the prototype be:

- `void putc(char dat)`

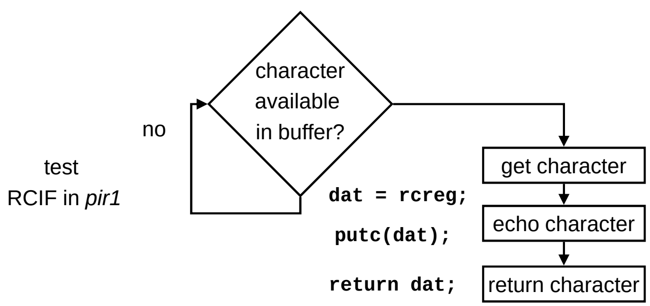


getc

This function gets a character to the terminal.

It is logical to have the prototype be:

- `char getc(void)`



Bit Variables

The compiler allows you to define bits. For example, TXIF is a bit in PIR1 that indicates that the transmit register can be written to.

We can define this as a bit as:

```
volatile bit txif@pir1.4
```

We can then use this like any variable

```
if (txif) ...
```

Bits

Warning: The system.h file defines bit names associated with all these registers as follows:

```
/////// PIR1 Bits ///////////////////////////////////////
#define TMR1IF              0x00000000
#define TMR2IF              0x00000001
#define CCP1IF              0x00000002
#define SSPIF               0x00000003
#define TXIF                0x00000004
#define RCIF                0x00000005
#define ADIF                0x00000006
#define PSPIF               0x00000007
```

Note that these are simply offsets that indicate the position in the register of the bit.

Bits

The C code `if (TXIF)` is equivalent to `if (4)` which is always true.

These definitions are for using the complier supplied functions such as set_bit

```
//Helper macros
#define clear_bit( reg, bitNumb)  ((reg) &= ~(1 << (bitNumb)))
#define set_bit( reg, bitNumb )   ((reg) |= (1 << (bitNumb)))
#define test_bit( reg, bitNumb )  ((reg) & (1 << (bitNumb)))
```


Advice

The following code snippets do the same thing:

```
volatile bit tbmt@pir1.4
...
if(tbmt) ...
```

and

```
if (pir1 & (1 << 4)) ...
```

If you every want me to look at your code and help you find a bug, use the former.

Outline

- **Lab 5 preview**
- **Functions**
- ➢ **Interrupts**
- **Timer/counters**
- **A/D Conversion**
- **Serial I/O**
- **Pulse Width Modulation**
- **Parallel I/O**

Interrupts

A special kind of function is the interrupt function.

An interrupt is a signal that an event (in our context, and event that happened in the hardware) has occurred.

The interrupt function is the software that reacts to that hardware occurrence.

We are going to first talk a little about what can generate an interrupt in our microcontroller, and then more specifically about handling the interrupts that occur.

Why Use Interrupts

Interrupts are most useful for events that happen asynchronously.

Suppose our project has a sensor that detects when the cup has been removed from the automatic drink dispenser. We could have our software in a look constantly checking this switch.

Often, however, there are other things that we need to be doing, so a better approach might be to use interrupts.

18F4620 Interrupts

There are lots of things that can generate an interrupt on the 4620. These include events such as a timer turning over (counting from FFFF to 0000), a character arriving in the USART, etc.

There are also certain port bits that have interrupt functions associated with them so that an external event can cause the interrupt.

The interrupts in this device are divided into two groups, regular interrupts and peripheral interrupts.

There is also now priority associated with the interrupts.

18F4620 Interrupts

How do I know if the interrupt I want to use is peripheral or not? Read the damn manual! (oops, sorry!)

Case: USART receive:

- We saw that the RCIF (Receive interrupt flag) occurred when a character shows up in the usart.
- Looking at the manual, we can see that this is a peripheral interrupt because of the name of the register in which it lives and the requirement of PEIE (Peripheral interrupt enable) be set as can be seen in the manual.

INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
IPR1	PSPIP(1)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D

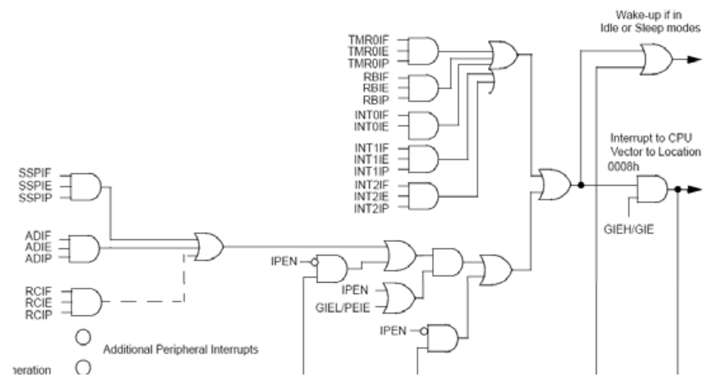
18F4620 Interrupts

Case: INT1:

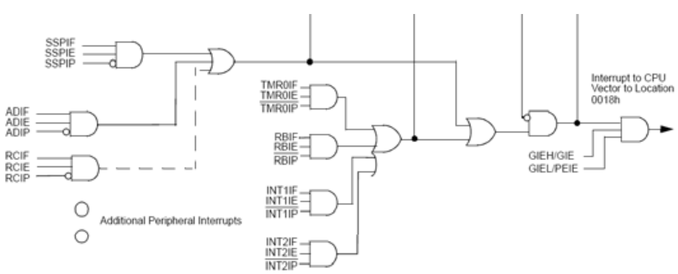
- **Bit 1 of port B can be set up as an edge triggered interrupt. (Edge triggered means that the interrupt occurs on the transition of the signal from one logic lever to the other. Which transition is configurable.)**
- **INT1 is controlled by bits in the registers shown below.**

PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
LATB	PORTB Data Latch Register (Read and Write to Data Latch)							
TRISB	PORTB Data Direction Control Register							
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
INTCON2	RBPU	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
INTCON3	INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF
ADCON1	—	—	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0

High Priority Interrupts in the 4620



Low Priority Interrupts in the 4620



18F4620 Interrupts

Why so many bits associated with an interrupt? Here's the code:

- **xxxIF is the Interrupt Flag. A bit that gets set when the interrupting event occurs. Note that interrupts don't have to be enabled for the bit to get set.**
- **xxxIE is the Interrupt Enable. For an event to interrupt the processor, it must be enabled.**
- **xxxP is set to indicate whether the interrupt is high Priority or low priority.**

To have an event interrupt the processor, you must have the interrupt enabled, global interrupts enabled, and if it is a peripheral interrupt, peripheral interupts enabled.

Interrupts

REGISTER 9-1: INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

bit 7 **GIE/GIEH:** Global Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked interrupts
0 = Disables all interrupts
When IPEN = 1:
1 = Enables all high priority interrupts
0 = Disables all interrupts

bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts
When IPEN = 1:
1 = Enables all low priority peripheral interrupts
0 = Disables all low priority peripheral interrupts

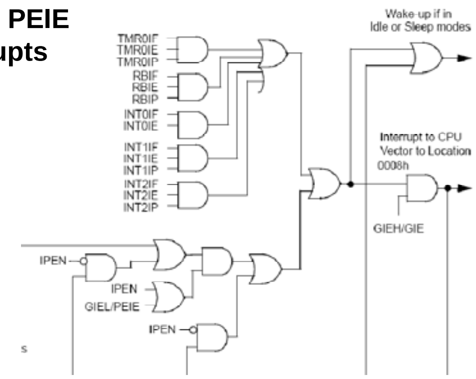
REGISTER 4-1: RCON: RESET CONTROL REGISTER

R/W-0	R/W-1[1]	U-0	R/W-1	R-1	R-1	R/W-0[2]	R/W-0
IPEN	SBOREN	—	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
bit 7							bit 0

bit 7 **IPEN:** Interrupt Priority Enable bit
1 = Enable priority levels on interrupts
0 = Disable priority levels on interrupts (PIC16CXXX Compatibility mode)

Interrupts

IPEN, GIEH, and PEIE control interrupts in bulk.



Handling Interrupts in Software

There are several things you need to do to use an interrupt:

- **Setup the particular hardware function to generate an interrupt.**
- **Enable that specific hardware interrupt to occur.**
- **Enable interrupts in general to occur (GEI).**
- **Perhaps enable peripheral interrupts (PEIE)**
- **Set IPEN as desired.**
- **Write software to do what you need to do when the interrupt occurs**

Handling Interrupts in Software

There is a predefined function called interrupt which is declared:

```
void interrupt (void);
```

Upon interrupt, the software execution switches from whatever it was doing, and executes this function. This works like any other function call, except it occurs asynchronously based on some hardware event, not because it was called by a line in your program.

The interrupt routine should precede the main routine in your code.

Handling Interrupts in Software

Notice that there are no arguments and nothing is returned.

```
void interrupt (void);
```

The interrupt routine will be able to access all of the variables defined in system.h since these are global variables.

If there are other variables that you want your interrupt routine and your main routine to share, you will have to make them global also.

Generic Interrupt Routine

Consider using both RBIF and INT1. The general form of your interrupt routine would include:

```
void interrupt(void)
{
  if (rcif) // see if receive char interrupt
  {
    rcif = 0;  // clear interrupt bit
              // do the interrupt stuff for rcif
  }
 if (int1) // see if interrupt int1 caused
  {
    int1 = 0;  // clear interrupt bit
              // do the interrupt stuff for int1
  }
}
```


Generic Interrupt Routine

Note that I am using bit variables to make my life easier. I would define

```
volatile bit rcif@pir1.5
volatile bit int1if@intcon3.1
```

: PIR1: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

INTCON3: INTERRUPT CONTROL REGISTER 3

R/W-1	R/W-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF
bit 7							bit 0

Generic Interrupt Routine

I may be strange but I think it is easier and produces more readable code if you do:

```
 if (rcif) // see if receive char interrupt
{
   rcif = 0;  // clear interrupt bit
```

rather than

```
 if (pir1 & 0x20) // see if receive char int
{
   pir1 &= 11011111b;  // clear interrupt bit
```

Interrupt Routine Comments

We need to have our interrupt routine determine which interrupt caused it to get there. We are only using one here, but it is good practice.

The interrupt routine needs to clear the interrupt flag that caused the interrupt. If not, upon exit from the interrupt routine, the flag would still be set and the interrupt would occur immediately!!!

Outline

- Lab 5 preview
- Functions
- Interrupts
- ➢ Timer/counters
- A/D Conversion
- Serial I/O
- Parallel I/O

Timer / Counters

A timer / counter is a register inside the microcontroller that increments.

If it increments based on the system clock it is called a timer.

If it increments based on some external signal, it is called a counter.

As a timer, the register will allow you to determine how long an event was.

As a counter, the register will allow you to determine how many events occurred.

Counter Application

Suppose you are precisely positioning something via a motor turning a screw drive. An shaft encoder can give you a pulse for every n^{th} of a turn the shaft makes.

By counting these pulses, you can determine the position.

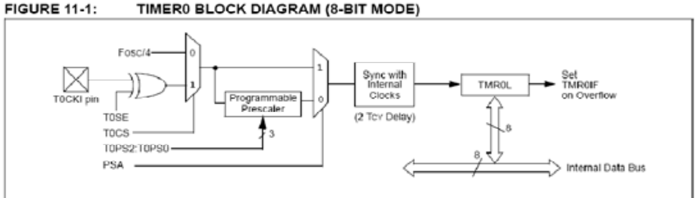
Note that one of the functions related to timer/counters is a compare function, which can be combined with a counter to tell you when a particular value of the count is reached.

Timer 0 in the 4620

Timer 0 looks like this. Clock source is either an external signal (T0CLK) or the system clock (FOSC/4). (T0CS decides which.)

There is a pre-scaler, which can further divide the clock rate.

FIGURE 11-1: TIMER0 BLOCK DIAGRAM (8-BIT MODE)

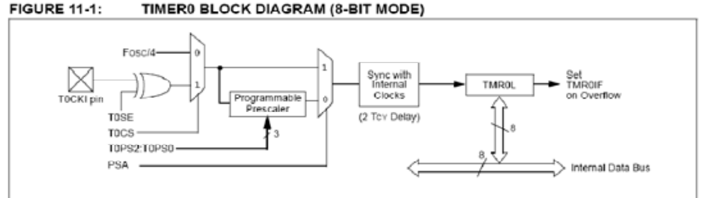


Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

Timer 0 in the 4620

Finally, the signal goes to the TMR0 register, and when it overflows (switches from FF to 00) an interrupt occurs.

FIGURE 11-1: TIMER0 BLOCK DIAGRAM (8-BIT MODE)

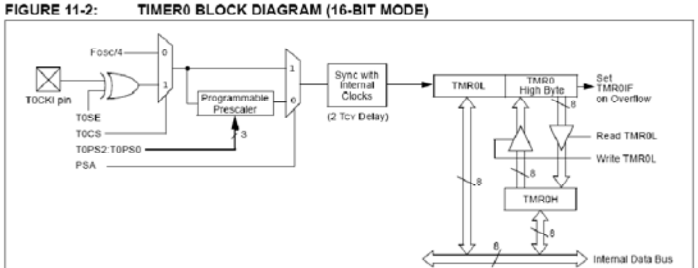


Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

Timer 0 in the 4620

There is also a 16 bit mode for this timer. 16 bits allows for a longer count

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

Timer 0 setup

Registers/Bits associated with timer 0.

TABLE 11-1: REGISTERS ASSOCIATED WITH TIMER0

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
TMR0L	Timer0 Register Low Byte								50
TMR0H	Timer0 Register High Byte								50
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	50

INTCON2: INTERRUPT CONTROL REGISTER 2

R/W-1	R/W-1	R/W-1	R/W-1	U-0	R/W-1	U-0	R/W-1
$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
bit 7							bit 0

Setting up a Timer

REGISTER 11-1: T0CON: TIMER0 CONTROL REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

bit 7 **TMR0ON:** Timer0 On/Off Control bit
1 = Enables Timer0
0 = Stops Timer0

bit 6 **T08BIT**: Timer0 8-bit/16-bit Control bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter

bit 5 **T0CS**: Timer0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKO)

bit 4 **T0SE**: Timer0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA**: Timer0 Prescaler Assignment bit
1 = Timer0 prescaler is NOT assigned. Timer0 clock input bypasses prescaler.
0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.

bit 2-0 **T0PS2:T0PS0**: Timer0 Prescaler Select bits
111 = 1:256 Prescale value
110 = 1:128 Prescale value
101 = 1:64 Prescale value
100 = 1:32 Prescale value
011 = 1:16 Prescale value
010 = 1:8 Prescale value
001 = 1:4 Prescale value
000 = 1:2 Prescale value

Timers

There are three other timers available in the 18F4620. They are either 8 or 16 bit timers. Each has a similar setup to timer 0.

There is also a watch dog timer that has a period that can be set from 4 msec to 131 seconds.

The WDT can be used to recover from the main code loop getting hung.

It can also be used to wake up the processor from sleep mode (a low power consumption mode).

The default is for the WDT to be enabled.

Tasks 5 and 6

Tasks 5 and 6 involve timers and interrupts.

Task 5 starts with a single interrupt timer combination, with task 6 adding a second.

Issues:

- **You will have to set up timer 0 to provide one interrupt per second. This involves getting the timer set up correctly, with the correct pre-scale and loading the correct value into the count register.**
- **Set all the bits correctly so that the processor can be interrupted.**
- **Limitations of the compiler.**

Tasks 5 and 6

The compiler does not allow you to call a function from two different execution threads. Thus, if you are using the LCD in the main program, you can't use it as part of your interrupt service routing.

You don't have to worry about using priority in the interrupts. The default is to have a single priority and that will work fine in tasks 5 and 6.

Tasks 5 and 6

Use semaphores (aka flags) to communicate between interrupt service routine and main program.

- Interrupt routine sets a flag (global) when the interrupt occurs.
- Main program is watching for the flag to be set:
  - Takes the appropriate action
  - Clears the flag.

Be sure and clear the interrupt flag or you will re-interrupt as soon as you return to the main program.

Tasks 5 and 6

To see if you are getting an interrupt you can:

- Increment and display a value on portd (the LED's)
- Use the logic analyzer.

```
/* setup bit a1 as an output and define
   a macro to toggle the bit */
volatile bit a1@PORTA.1;      // name bit porta bit 1
#define toggle_a1 a1=1; nop(); a1=0;  // define toggle
trisa.1 = 0;          // make port a bit 1 output
a1=0;                         // start at zero
```

- With this code included in your program, you can make bit 1 of port a toggle with the statement

```
toggle_a1;
```


SPI Interface

Many microcontrollers have other serial interfaces built into their hardware.

Common ones include I2C (Inter-Integrated Circuit) and SPI (Serial Peripheral Interface).

In the Microchip microcontrollers, these are part of the same hardware called a Synchronous Serial Port.

These interfaces typically have a single master and some number of slave devices.

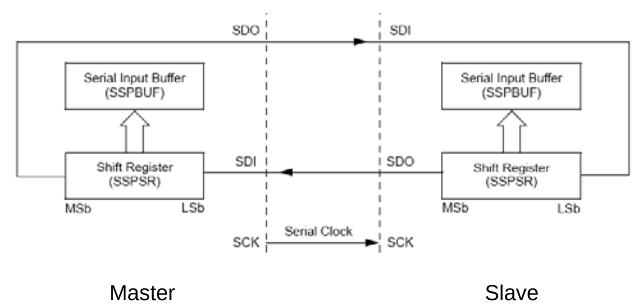
It is called synchronous because the two communicating devices share a common clock provided by the master device.

Since the Chipcon part uses an SPI interface, we will talk about that in a little detail.

SPI Interface

The basic idea of an SPI interface is to share information over a serial connection. It looks like the picture below.

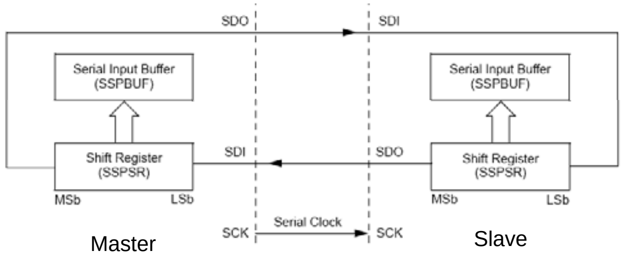
Note that the master is the source of the clock.



SPI Interface

Data is given to the buffers in parallel and is shifted across the link in serial.

This is how communications is done with the CC2420, and your are able to set various registers, and well as transfer the information that is carried in messages between ZigBee devices.

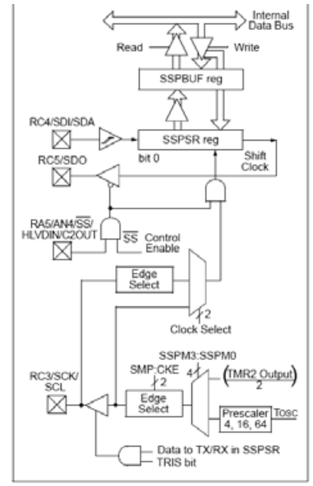


SPI Interface

A more detailed view of the hardware is shown.

Note:

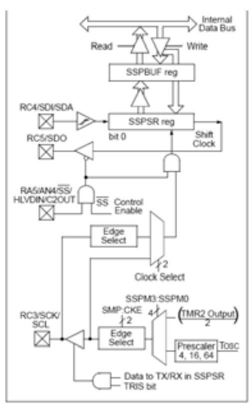
- Specific pins are used for the various signals
- The clock can be generated based on the system clock or an internal timer.
- Numerous registers are involved in the setup of the device.



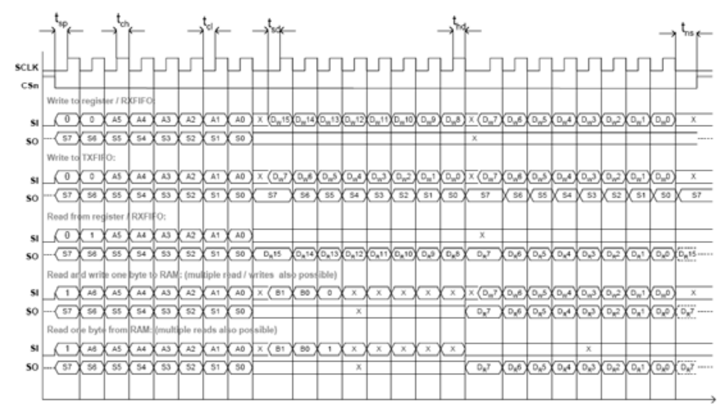
SPI Interface

Basic idea:

- (Master) When a byte is written into the SSPBUF and SSPSR, it gets shifted out on the SDO pin using a clock that is sent out on the SCK pin.
- (Slave) If the slave is selected, it will receive the serial stream and send back one of its own (a reply) that will show up in the SSPBUF register.
- (Master) Read the returning data.



SPI Interface to CC2420



Registers for SPI

TABLE 17-2: REGISTERS ASSOCIATED WITH SPI™ OPERATION

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	52
PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	52
IPR1	PSPIP(1)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	52
TRISA	TRISA7(2)	TRISA6(2)	PORTA Data Direction Control Register						52
TRISC	PORTC Data Direction Control Register								52
SSPBUF	SSP Receive Buffer/Transmit Register								50
SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	50
SSPSTAT	SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF	50

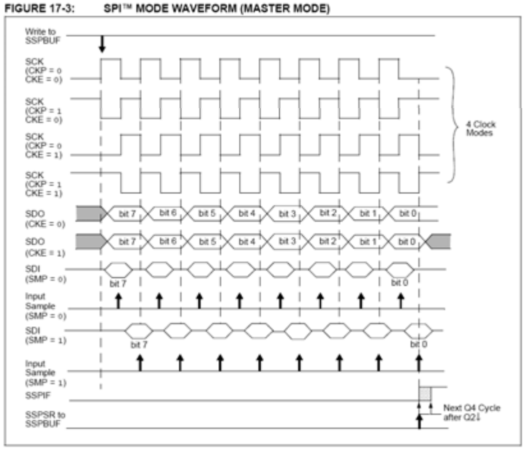
Legend: Shaded cells are not used by the MSSP in SPI mode.

Note 1: These bits are unimplemented on 28-pin devices and read as '0'.

2: PORTA<7:6> and their direction bits are individually configured as port pins based on various primary oscillator modes. When disabled, these bits read as '0'.

SPI Options

FIGURE 17-3: SPI™ MODE WAVEFORM (MASTER MODE)



SPI Software

In a similar fashion to the USART, you should think about having low level functions that initialize, write, and read the SPI.

Initialization involves setting up the clock speed, the edges used, etc.

Write and read are a little more complicated. Essentially, to read something, you need to write something.

We can see what's going on by looking at the timing diagram.

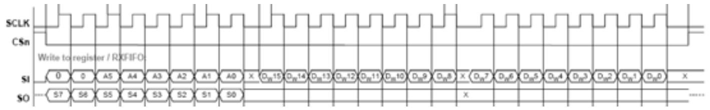
SPI Software

To write a register in the CC2420, you send a command that includes bits that specify which register (A5 – A0).

The appropriate value would be written to the SSPBUF, and it would automatically be shifted out.

As it is shifted out, a status byte is shifted in. When it is all there, a flag will be set telling you that a byte is present and needs to be read.

You read this byte, and then write in the high byte of the 16 bit word that is to be sent to the register.



Analog to Digital Conversion

There are many sensors that measure an analog real world value and produce a signal that is a voltage or current that is proportional to the value being measured.

Examples include:

- Strain Gauges
- Accelerometers
- Temperature Sensors

To use these external values as part of an embedded control application, we need a way to represents the value of the analog signal inside the microcontroller.

A/D in the 4620

The 18F4620 has a 10 bit successive approximation converter.

The analog source can be selected from one of 13 different pins.

There is also the capability to select different reference voltages which set the range of the analog input (maximum and minimum values.)

As usual, there are a number of different registers associated with using the A/D converter in the device.

A/D in the 4620

TABLE 19-2: REGISTERS ASSOCIATED WITH A/D OPERATION

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	52
PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	52
IPR1	PSPIP(1)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	52
PIR2	OSCFIF	CMIF	—	EEIF	BCLIF	HLVDIF	TMR3IF	CCP2IF	52
PIE2	OSCFIE	CMIE	—	EEIE	BCLIE	HLVDIE	TMR3IE	CCP2IE	52
IPR2	OSCFIP	CMIP	—	EEIP	BCLIP	HLVDIP	TMR3IP	CCP2IP	52
ADRESH	A/D Result Register High Byte								51
ADRESL	A/D Result Register Low Byte								51
ADCON0	—	—	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON	51
ADCON1	—	—	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0	51
ADCON2	ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0	51
PORTA	RA7(1)	RA6(1)	RA5	RA4	RA3	RA2	RA1	RA0	52
TRISA	TRISA7(2)	TRISA6(2)	PORTA Data Direction Control Register						52
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	52
TRISB	PORTB Data Direction Control Register								52
LATB	PORTB Data Latch Register (Read and Write to Data Latch)								52
PORTE(4)	—	—	—	—	RE3(3)	RE2	RE1	RE0	52
TRISE(4)	IBF	OBF	IBOV	PSPMODE	—	TRISE2	TRISE1	TRISE0	52
LATE(4)	—	—	—	—	—	PORTE Data Latch Register			52

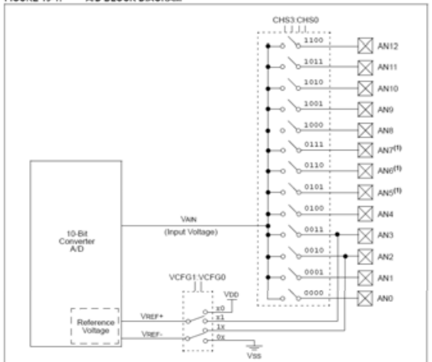
A/D in the 4620

Note:

- The analog signal must be allowed to settles before doing the conversion.
- Since it is a successive approximation converter, it is not the fastest converter in the world, and the conversion time must be chosen based on the system clock. (The device needs more cycles to convert (per bit) as the system clock speed goes up.)
- Some external signals can be used as references.
- Pins used as analog inputs must be setup as analog. (Note that analog is the default.)

A/D in the 4620

FIGURE 19-1: A/D BLOCK DIAGRAM



Note 1: Channels AN5 through AN7 are not available on 28-pin devices.
2: I/O pins have diode protection to VDD and VSS.

Takeaways

One bit set wrong can make something not work.

Use the tools you have to debug:

- **You have a fairly complete set of routines that print to the LCD display.**
- **You can easily make a similar set that prints to the terminal.**
- **If you are not sure you are setting a register correctly, print it out:**

```
LCD_bin(rcsta);
```

For hardware issues, the logic analyzer helps