



USART

Using the USART in Asynchronous Mode

Getting Started - USART

© 2001

In this presentation we will examine the use of the USART in the Asynchronous Mode of operation.



USART - Main Functions

- Universal Synchronous Asynchronous Receiver Transmitter
 - Can be synchronous or asynchronous
 - Can receive and transmit
 - Full duplex asynchronous operation
- Most common use
 - RS-232 communications to a PC serial port

Note: Needs driver for level shifting

Getting Started - USART

© 2001

USART stands for Universal Synchronous Asynchronous Receiver Transmitter. It is sometimes called the Serial Communications Interface or SCI.

Synchronous operation uses a clock and data line while there is no separate clock accompanying the data for Asynchronous transmission.

Since there is no clock signal in asynchronous operation, one pin can be used for transmission and another pin can be used for reception. Both transmission and reception can occur at the same time — this is known as full duplex operation. Transmission and reception can be independently enabled. However, when the serial port is enabled, the USART will control both pins and one cannot be used for general purpose I/O when the other is being used for transmission or reception.

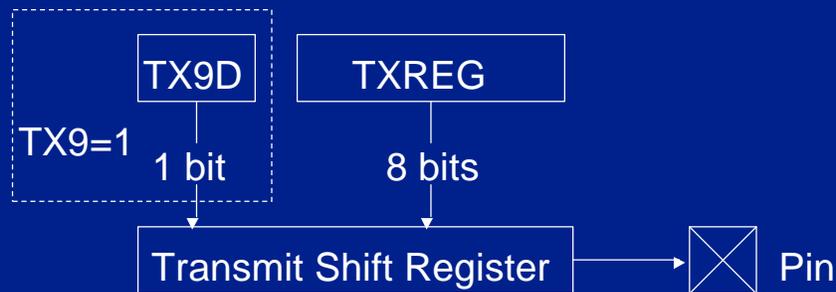
The USART is most commonly used in the asynchronous mode. In this presentation we will deal exclusively with asynchronous operation.

The most common use of the USART in asynchronous mode is to communicate to a PC serial port using the RS-232 protocol. Please note that a driver is required to interface to RS-232 voltage levels and the PICmicro® MCU should not be directly connected to RS-232 signals.

The USART can both transmit and receive, and we will now briefly look at how this is implemented in the USART.

USART - Block Diagram

- Simplified transmission block diagram

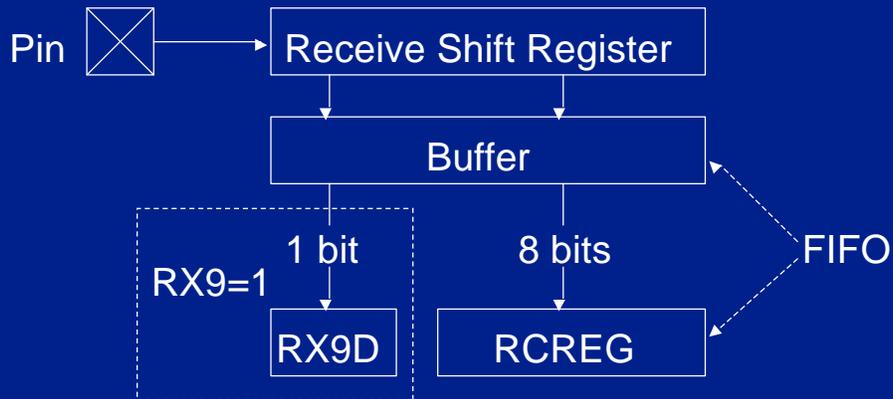


The USART can be configured to transmit eight or nine data bits by the TX9 bit in the TXSTA register. If nine bits are to be transmitted, the ninth data bit must be placed in the TX9D bit of the TXSTA register before writing the other eight bits to the TXREG register. Once data has been written to TXREG, the eight or nine bits are moved into the transmit shift register. From there they are clocked out onto the TX pin preceded by a start bit and followed by a stop bit.

The use of a separate transmit shift register allows new data to be written to the TXREG register while the previous data is still being transmitted. This allows the maximum throughput to be achieved.

USART - Block Diagram

- Simplified reception block diagram

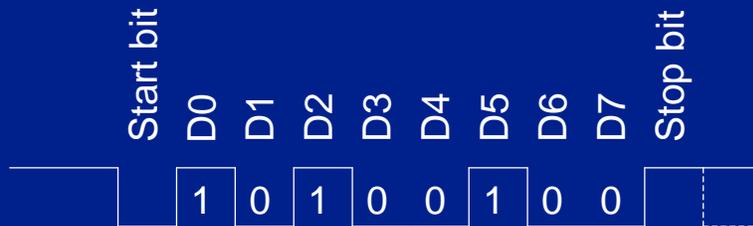


The USART can be configured to receive eight or nine bits by the RX9 bit in the RCSTA register. After the detection of a start bit, eight or nine bits of serial data are shifted from the RX pin into the receive shift register one bit at a time. After the last bit has been shifted in, the stop bit is checked and the data is moved into the buffer which passes the data through to the RCREG register if it is empty. The buffer and RCREG register therefore form a two element FIFO. If nine bit reception is enabled, the ninth bit is passed into the RX9D bit in the RCSTA register in the same way as the other eight bits of data are passed into the RCREG register.

The use of a separate receive shift register and a FIFO buffer allows time for the software running on the PICmicro MCU to read out the received data before an overrun error occurs. It is possible to have received two bytes and be busy receiving a third byte before the data in the RCREG register is read.

USART - Signals

- Asynchronous 8 bit waveform example
 - Data is H'25' = B'00100101'

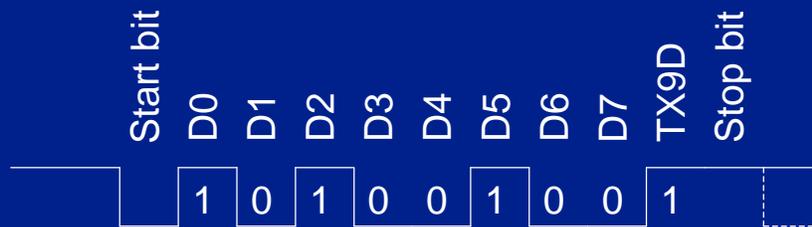


The USART outputs and inputs logic level signals on the TX and RX pins of the PICmicro MCU. The signal is high when no transmission or reception is in progress and goes low when the transmission starts. This low going transition is used by the receiver to synchronize to the incoming data. The signal stays low for the duration of the start bit and is followed by the data bits, least significant bit first. In the case of an eight-bit transfer, there are eight data bits and the last data bit is followed by the stop bit which is high. The transmission therefore ends with the pin high. After the stop bit has completed, the start bit of the next transmission can occur as shown by the dotted lines.

There are several things to note about this waveform, which represents the signal on the TX or RX pins of the microcontroller. The start bit is a zero and the stop bit is a one. The data is sent least significant bit first so the bit pattern looks backwards in comparison to the way it appears when written as a binary number. The data is not inverted even though RS-232 uses negative voltages to represent a logic one. Generally, when using the USART for RS-232 communications, the signals must be inverted and level shifted through a transceiver chip of some sort.

USART - Signals

- Asynchronous 9 bit waveform example
 - Data is H'25' = B'00100101'
 - TX9D = 1



In the case of a nine bit asynchronous transmission, the ninth bit occurs after the eighth data bit and is followed by the stop bit. Having the ninth bit in this position makes it easy to implement RS-232 data formats that require parity or two stop bits. Even parity can be implemented in software by changing the ninth data bit to make the total number of ones in the data an even number. Similarly, odd parity can be implemented by making the total number of ones in the data an odd number. If two stop bits are required for an eight bit transmission, this can be achieved by setting the ninth data bit to one. The ninth bit then acts as the first stop bit and the normal stop bit becomes the second stop bit.

Another advantage of having a ninth data bit is that it can be used as an address indicator. This is commonly implemented on the RS-485 protocol. Each device on a serial bus is assigned a specific address and monitors the data for transmissions with the ninth bit set. When the ninth bit is set, the software on the PICmicro compares the received data to its own address. If the addresses match, the software can enable reception of the data that follows the address, but if the addresses do not match, the data that follows is ignored. On some PICmicro devices the USART has hardware built in to check the address bit and can ignore transmissions that do not have this bit set. These USARTs are referred to as addressable USARTs.



USART - Signals

- Voltage levels
 - USART uses logic level signals
 - RS-232 uses positive and negative voltages
 - Use interface chip such as MAX232

The signals on the USART pins of the microcontroller use logic levels. This means that for a five volt supply, the signals will be close to five volts when they are high and close to ground when they are low. When communicating with other logic devices, these signals can be used directly. In many applications, particularly with asynchronous communications, transmission standards such as RS-232 and RS-485 require different voltage levels to be used. For example, RS-232 uses a voltage below minus five volts to represent a logic one and a voltage above five volts to represent a logic zero. For RS-232, an interface chip such as Microchip's TC232 device is recommended to convert the signals to the required levels.

- Registers used to control USART
 - SPBRG - Baud rate generator
 - TXSTA - Transmit status and control
 - RCSTA - Receive status and control
 - TXREG - Transmit data register
 - RCREG - Receive data register
 - PIR1 - Peripheral interrupt flag register
 - PIE1 - Peripheral interrupt enable register

There are several registers used to control the USART and these will be discussed in more detail later in the presentation.

The SPBRG register allows the baud rate to be set.

The TXSTA and RCSTA registers are used to control transmission and reception but there are some overlapping functions and both registers are always used.

The TXREG and RCREG registers are used write data to be transmitted and to read the received data.

The PIR1 and PIE1 registers contain the interrupt flag bits and enable bits to allow the USART to generate interrupts. Interrupts are often used when the PICmicro is busy executing code and data needs to be transmitted or received in the background. The interrupt flags are not only used for interrupts but can also be read during normal operation to determine whether data has been received or can be transmitted.



USART - Baud Rate

- Baud rate generator registers
 - SPBRG register
 - BRGH bit in TXSTA register
 - SYNC bit in TXSTA register

The rate at which data is transmitted or received must always be set using the baud rate generator unless the USART is being used in synchronous slave mode.

The baud rate is set by writing to the SPBRG register. The SYNC bit selects between synchronous and asynchronous modes, and these modes have different baud rates for a particular value in the SPBRG register. For asynchronous mode, the SYNC bit must be cleared and the BRGH bit is used to select between high and low speed options for greater flexibility in setting the baud rate.



USART - Baud rate

- Formulas for baud rate
 - Baud rate = $F_{osc}/(16(SPBRG+1))$, BRGH=1
 - Baud rate = $F_{osc}/(64(SPBRG+1))$, BRGH=0

- Formulas for SPBRG
 - $SPBRG = (F_{osc}/(16 \times \text{Baud rate})) - 1$, BRGH=1
 - $SPBRG = (F_{osc}/(64 \times \text{Baud rate})) - 1$, BRGH=0

The top two formulas show how the baud rate is set by the value in the SPBRG register and the BRGH bit.

More important for the user, however, is to be able to calculate the value to place in the SPBRG register to achieve a desired baud rate. The bottom two formulas can be used to do this. The SPBRG register can have a value of zero to 255 and must always be an integer value. When these formulas yield a value for SPBRG that is not an integer, there will be a difference between the desired baud rate and the rate that can actually be achieved. By calculating the actual baud rate using the nearest integer value of SPBRG, the error can be determined. Whether this error is acceptable usually depends on the application.



USART - Baud rate

- Example calculation
 - 4MHz oscillator
 - 9600 baud asynchronous
 - For BRGH = 1
$$\text{SPBRG} = 4000000 / (16 \times 9600) - 1 = 25.04$$
 - For BRGH = 0
$$\text{SPBRG} = 4000000 / (64 \times 9600) - 1 = 5.51$$
- Best choice is BRGH = 1, SPBRG = 25

Getting Started - USART

© 2001

As an example of a baud rate calculation, consider the case of a microcontroller operating at 4MHz that is required to communicate at 9600 baud with a serial port on a PC. The USART would then be used in asynchronous mode.

When BRGH is set to zero, the ideal value of SPBRG is calculated as 5.51. Since this differs from the closest integer value of six by approximately nine percent, this will cause a corresponding error in the baud rate.

When BRGH is set to one, the ideal value of SPBRG is calculated as 25.04. This is very close to the integer value of 25 which must be used. Setting SPBRG to 25 will give a baud rate of 9615 which is within two tenths of a percent of the desired baud rate.

Note that to get an accurate and stable baud rate, an accurate and stable oscillator is required. A crystal or ceramic resonator usually works well but an RC oscillator is seldom accurate enough for reliable asynchronous communications. It is not advisable to use an RC oscillator when doing RS-232 communications to a PC, for example.



USART - Baud Rate

- Setting up the baud rate

```
BANKSEL    SPBRG
movlw     D'25'
movwf    SPBRG
BANKSEL    TXSTA
bsf      TXSTA, BRGH
```

- BANKSEL is an assembler directive
 - Generates code to change banks

This is an example of how to write code to set up the baud rate. The correct bank must be selected to access the SPBRG register. The required data, decimal 25 in this example, is moved into the working register which is then moved into the SPBRG register. Finally, the BRGH bit in the TXSTA register is set. In many cases the BRGH bit will be set when writing to the TXSTA register to set up other features of the USART which will be discussed later.

This example uses the BANKSEL directive to select the correct bank. It is not a PICmicro instruction but will cause the assembler or linker to generate the appropriate instructions to change the bank. Please see the MPASM User's Guide for more information.



USART - TXSTA Register

- TXSTA - Transmit status and control
 - CSRC - Clock source select
 - TX9 - 9 bit transmission enable
 - TXEN - Transmit enable
 - SYNC - Synchronous mode select
 - BRGH - High baud rate select
 - TRMT - Transmit shift register status
 - TX9D - 9th bit of data to transmit

Getting Started - USART

© 2001

The TXSTA register is mainly used to control transmissions but does have other functions. For example, the SYNC bit selects between synchronous and asynchronous modes for both transmission and reception.

The CSRC bit has no effect in asynchronous mode.

The TX9 bit enables nine bit transmission. If this bit is set, the TX9D bit in the TXSTA register will be transmitted in addition to the eight bits of data that was written to the TXREG register.

The TXEN bit enables transmissions. Once this bit has been set, writes to the TXREG register will cause a transmission to be initiated if the serial port has been enabled.

The SYNC bit should be cleared to select asynchronous operation.

The BRGH bit selects between the high and low speed baud rate options in asynchronous mode. This allows a greater range of baud rates to be selected.

The TRMT bit indicates that there is data in the transmit shift register. This means that there is a transmission in progress. Data that is written to the TXREG register is loaded into the transmit shift register when this register is empty. The transmit shift register is internal and is not a readable register.



USART - RCSTA Register

- RCSTA - Receive status and control
 - SPEN - Serial port enable
 - RX9 - 9 bit receive enable
 - SREN - Single receive enable
 - CREN - Continuous receive enable
 - ADDEN - Address detect enable
 - FERR - Framing error
 - OERR - Overrun error
 - RX9D - 9th bit of data received

Getting Started - USART

© 2001

The RCSTA register is mainly used to control reception but does have other functions. For example, the SPEN bit is used to enable the entire serial port, both for transmissions and receptions.

Setting SPEN also configures both port pins associated with the USART to their USART functions.

The RX9 bit enables nine bit reception. This causes the ninth data bit received to be loaded into the RX9D bit in the RCSTA register.

The SREN bit has no effect in asynchronous mode.

The CREN bit enables reception of data continuously while it is set and disables reception when cleared.

The ADDEN bit enables address detection in nine bit asynchronous mode and is only available on parts with an addressable USART. When this bit is set, only data that has the ninth data bit set will be received.

The FERR bit indicates a framing error which means that the stop bit was not detected. The framing error is associated with a particular byte in RCREG and once the RCREG has been read, the FERR bit will be meaningless until the next data is received into RCREG. Framing errors are often caused by incorrect baud rates.

The OERR bit indicates an overrun error which means that a complete byte was received when the FIFO was still full with the two previous bytes. The new data will be lost and no further data will be received until the CREN bit has been cleared and set again by software. The two bytes in the FIFO can still be read from RCREG however.



USART - Setting up

- Setting up 8 bit asynchronous mode
 - High baud rate, no interrupts

```
BANKSEL TXSTA
movlw   B'00100100'
movwf   TXSTA
BANKSEL RCSTA
movlw   B'10010000'
movwf   RCSTA
```

We will now look at some examples of code used to set up and use the USART in asynchronous mode. To set up eight bit asynchronous mode, this code sets the TXEN bit to enable transmission, the CREN bit to enable reception and the SPEN bit to enable the serial port. The BRGH bit is set to select the high baud rate. The TX9 and RX9 bits are cleared since eight bit operation is desired and the SYNC bit is cleared for asynchronous operation.

The PORT pins used by the USART must be set as inputs. This is the default state after a reset but it is good practice to explicitly set the pins to inputs anyway.

This code and the two examples that follow do not set up the interrupts but this will be covered later.



USART - Setting up

- Setting up 9 bit asynchronous mode
 - High baud rate, no address detect or interrupts

```
BANKSEL TXSTA
movlw   B'01100100'
movwf   TXSTA
BANKSEL RCSTA
movlw   B'11010000'
movwf   RCSTA
```

To set up nine bit asynchronous mode, this code sets the TXEN bit to enable transmission, the CREN bit to enable reception and the SPEN bit to enable the serial port. The BRGH bit is set to select the high baud rate. The TX9 and RX9 bits are set since nine bit operation is desired and the SYNC bit is cleared for asynchronous operation.



USART - Setting up

- Setting up 9 bit asynchronous mode
 - High baud rate, address detect, no interrupts

```
BANKSEL TXSTA
movlw   B'01100100'
movwf   TXSTA
BANKSEL RCSTA
movlw   B'11011000'
movwf   RCSTA
```

To set up nine bit asynchronous mode with address detect, this code sets the TXEN bit to enable transmission, the CREN bit to enable reception and the SPEN bit to enable the serial port. The BRGH bit is set to select the high baud rate. The TX9, RX9 and ADDEN bits are set since nine bit operation and address detection is desired and the SYNC bit is cleared for asynchronous operation.



USART - Transmitting

- Waiting until a byte can be transmitted

```
                BANKSEL  PIR1
WaitTX:         btfss    PIR1, TXIF
                goto     WaitTX
```

We will now look at some code examples for transmitting data with the USART.

Before a byte can be transmitted, a check should be made to ensure that the previous byte written to the TXREG register does not get overwritten. Data is only moved from the TXREG when the transmit shift register is empty so if a transmission is in progress, data will stay in the TXREG until the previous data has been transmitted. The TXIF flag gets set when the data in the TXREG gets moved into the transmit shift register so this bit must be tested before new data is written to TXREG.

The code tests the TXIF bit in the PIR1 register and keeps looping back until the bit is detected high. In some situations, the processor cannot spare the time to wait for the current transmission to end and in this case the code might return to a main program instead of waiting in a loop. An alternative is to use interrupts to detect when data can be written.



USART - Transmitting

- Loading the 9th data bit to be transmitted
 - 9th data bit in LSb of DataBits

```
BANKSEL  DataBits
rrf      DataBits,W
BANKSEL  TXSTA
bcf      TXSTA, TX9D
btfsc   STATUS, C
bsf     TXSTA, TX9D
```

Getting Started - USART

© 2001

After the TXIF bit has been tested to check that new data can be transmitted, the data can be written. If a nine bit transmission is required, the ninth bit must be written before the other eight bits are written to TXREG since writing to TXREG will immediately initiate a transmission if the transmit shift register is empty.

In this example the ninth data bit is assumed to be in the least significant bit of a general purpose register labeled DataBits. A rotate instruction is used to get this bit into the carry flag in the STATUS register which is then tested to set or clear the TX9D bit which holds the ninth data bit.

This code can be simplified if the DataBits register is in the same bank as the TXSTA register. In this example they are assumed to be in different banks and the STATUS register is used for intermediate storage of the bit since it can be accessed in all banks.

The method used to load the ninth data bit also depends on the particular application. For example if this bit is to be used as another stop bit, it can be set once when initializing the USART, and will never need to be changed.



USART - Transmitting

- Loading the byte and transmitting data
 - 8 data bits in DataByte

```
BANKSEL  DataByte
movf     DataByte,W
BANKSEL  TXREG
movwf   TXREG
```

To initiate the transmission, data must be written to the TXREG. In this example, the data to be transmitted is assumed to be in a general purpose register labeled DataByte. The code moves the data from DataByte into the working register and then into the TXREG register. This initiates the transmission.



USART - Receiving

- Waiting for a byte to be received

```
                BANKSEL  PIR1
WaitRX:        btfss    PIR1,RCIF
                goto     WaitRX
```

We will now look at some code examples for receiving data with the USART.

Before reading the RCREG, a check should be made to determine whether new data has been received. When there is new data in the RCREG register, the RCIF bit in the PIR1 register will be set.

The code tests the RCIF bit in the PIR1 register and keeps looping back until the bit is detected high. In many situations, the processor cannot spare the time to wait for the the next data to be received and there is also the danger of being stuck in an endless loop if data cannot be received for some reason. It is usually better to return to the main program instead of waiting in a loop. An alternative is to use interrupts to detect when data has been received.



USART - Receiving

- Reading the 9th data bit
 - 9th data bit is placed in LSb of DataBits

```
BANKSEL RCSTA
rrf      RCSTA,W
BANKSEL DataBits
bcf      DataBits,0
btfsc   STATUS,C
bsf      DataBits,0
```

Getting Started - USART

© 2001

After the RCIF bit has been tested to check that new data has been received, the data can be read. If a nine bit reception is being performed, the ninth bit in RX9D must be read before the other eight bits are read from RCREG since reading from RCREG will immediately allow the next data in the FIFO to be loaded into RCREG and the RX9D bit.

In this example it is assumed that the ninth data bit should be loaded into the least significant bit of a general purpose register labeled DataBits. Since the ninth data bit, RX9D, is the least significant bit of RCSTA, a rotate instruction is used to place this bit into the carry flag of the STATUS register. The least significant bit of the DataBits register is then set or cleared depending on the carry flag.

This code can be simplified if the DataBits register is in the same bank as the RCSTA register. The STATUS register is used for intermediate storage since it can be accessed in all banks.

The method used to read the ninth data bit also depends on the particular application. For example if this bit is to be used as another stop bit, it can be tested directly and cause a branch to an error handling routine if it is zero.



USART - Receiving

- Reading a received byte
 - 8 bits of data are placed in DataByte

```
BANKSEL  RCREG
movf     RCREG,W
BANKSEL  DataByte
movwf   DataByte
```

After detecting that new data has been received and reading the ninth data bit if required, the eight bits of data can be read from the RCREG register.

In this example, it is assumed that the data received should be put into a general purpose register labeled DataByte. The code moves the data from the RCREG register into the working register and then into the DataByte register.



USART - Flow Control

- Flow control
 - Controls rate of received data
 - Controls rate of transmitted data
 - Implemented in software
- Types of flow control
 - XON/XOFF
 - Hardware

Getting Started - USART

© 2001

We will now consider flow control.

The USART will receive data as fast as the baud rate allows. In some circumstances, the software that must read the data from the RCREG register may not be able to do so as fast as the data is being received. In this case, there is a need for the PICmicro to tell the transmitting device to suspend transmission of data temporarily.

Similarly, the PICmicro may need to be told to suspend transmission temporarily. This is done by means of flow control. There are two common methods of flow control, XON/XOFF and hardware.

XON/XOFF flow control can be implemented completely in software with no external hardware, but full duplex communications is required. When incoming data needs to be suspended, an XOFF byte is transmitted back to the other device that is transmitting the data being received. To start the other device transmitting again, an XON byte is transmitted. XON and XOFF are standard ASCII control characters. This means that when sending raw data instead of ASCII text, care must be taken to ensure that XON and XOFF characters are not accidentally sent with the data.

Hardware flow control uses extra signals to control the flow of data and they are defined as part of the RS-232 communications standard, for example. To implement hardware flow control on a PICmicro, extra I/O pins must be used. Generally, an output pin is controlled by the receiving device to indicate that the transmitting device should suspend or resume transmissions. The transmitting device tests an input pin before a transmission to determine whether data can be sent.



USART - Interrupts

- Advantages of using interrupts
 - No polling or waiting for data
 - Background tasks easier
 - Fast response
 - Reduced chance of overflow errors
 - More common for receiving data

Getting Started - USART

© 2001

Interrupts are useful to minimize the time that the software spends polling to check for received data or testing whether a new transmission can be started. This can make implementing other tasks easier since they do not have to stop to test the USART.

The software can respond faster to incoming data since it does not wait the polling interval before detecting that there is new data.

Because of the faster response, data spends less time in RCREG waiting to be read and overflow errors are less likely to occur.

Typically, interrupts are used to receive data without being used to transmit data. In most software it is impossible to know when data will be received and interrupts provide a convenient means to avoid the need to continuously check for new received data.



USART - Interrupts

- Interrupt flags and enable bits
 - INTCON - Interrupt control register
GIE, PEIE

The USART interrupts are controlled by three registers.

The INTCON register contains the GIE and PEIE bits. These are the global interrupt enable and peripheral interrupt enable bits and both must be set in order for the receive or transmit interrupts to occur.



USART - Interrupts

- Interrupt flags and enable bits
 - INTCON - Interrupt control register
GIE, PEIE
 - PIE1 - Peripheral interrupt enable register
TXIE, RCIE

The PIE1 register contains the TXIE and RCIE bits and these are the transmit and receive interrupt enable bits. They allow the transmit and receive interrupts to be independently enabled or disabled.

The PIR1 register contains the TXIF and RCIF bits and these are the transmit and receive interrupt flag bits. When one of these bits get set while the appropriate interrupt enable bits are set, an interrupt will occur.

- Interrupt flags and enable bits
 - INTCON - Interrupt control register
GIE, PEIE
 - PIE1 - Peripheral interrupt enable register
TXIE, RCIE
 - PIR1 - Peripheral interrupt flag register
TXIF, RCIF

The PIR1 register contains the TXIF and RCIF bits and these are the transmit and receive interrupt flag bits. When one of these bits get set while the appropriate interrupt enable bits are set, an interrupt will occur.

The RCIF bit gets set when new data is available in RCREG and gets cleared when all data is emptied from the FIFO. Reading RCREG in the interrupt routine therefore clears the flag automatically if there is no other data in the FIFO.

The TXIF bit gets cleared when data is written to TXREG and gets set when this data moves into the transmit shift register to get transmitted. The interrupt therefore occurs when new data can be transmitted. When the last byte of data has been written to TXREG, the TXIE bit should be cleared to stop the interrupts from occurring. The interrupt can be enabled again when new data needs to be transmitted and this will immediately cause an interrupt.

- Enabling the interrupts

```
BANKSEL  PIE1
bsf      PIE1, TXIE
bsf      PIE1, RCIE
bsf      INTCON, PEIE
bsf      INTCON, GIE
```

This code example shows how the transmit and receive interrupts are enabled by setting the appropriate bits in the INTCON and PIE1 registers. Note that the INTCON register can be accessed in all banks.

There is no need to clear the interrupt flags since these are controlled by the USART hardware.

This code may be combined with other initialization code that enables other interrupts. In that case it may be more efficient to write to the registers instead of using several bit set instructions.

- Context saving
 - Interrupt can occur at any line of code
 - Could use same registers
 - STATUS and W critical
 - PCLATH if using more than one page
 - Automatic context saving on PIC18CXXX

When an interrupt occurs, the code at the interrupt vector immediately starts executing. This can happen at any time that the interrupts are enabled and the interrupt routine could use the same registers as the code that was interrupted. For example, only a very simple interrupt routine will not use the working register or affect the STATUS bits. It can be catastrophic for the interrupted code to have these registers changed unexpectedly. For this reason, it is important to save the data from any registers that may be changed by the interrupt routine and restore the contents of the registers before returning to the interrupted code. Another register that is often affected by interrupts is the PCLATH register if the software uses more than one page of program memory.

On the newer 18C devices there is automatic saving of critical registers for high priority interrupts.

- Context saving example

```
movwf    W_Save
movf     STATUS,W
clrf     STATUS
movwf    STATUS_Save
movf     PCLATH,W
movwf    PCLATH_Save
```

This code is an example of how the working register, STATUS register and PCLATH register can be saved at the start of an interrupt service routine. This should usually be the very first code at the interrupt vector but it is possible to first have some other code that does not affect these critical registers and does not require a specific bank.

The working register is saved first into a register called W_Save. Since the bank is not known at this point, W_Save must be in shared memory or all the possible locations of W_Save in all the banks must be reserved. In other words the software must not use any of the locations that W_Save could represent in each of the banks.

Once the working register has been saved, STATUS is moved into the working register. This can affect the Z bit in the STATUS register but note that the original contents of STATUS has been saved unchanged in the working register. The bank bits can now be cleared to select bank zero and the working register moved into the STATUS_Save register defined in bank zero. This now contains the original contents of STATUS.

Finally the PCLATH register is saved into the PCLATH_Save register defined in bank zero.

- Context restoring example

```
movf    PCLATH_Save,W
movwf   PCLATH
movf    STATUS_Save,W
movwf   STATUS
swapf   W_Save,F
swapf   W_Save,W
retfie
```

After executing the interrupt routine it is necessary to restore the critical registers to their original state and this code example shows how this can be done.

First PCLATH is restored from PCLATH_Save in bank zero. Then STATUS is restored from STATUS_Save in bank zero. At this point the STATUS register has been restored and it is critical not to change any of the STATUS bits. STATUS contains the correct bank for the data stored in W_Save but using a movf instruction to restore the working register would affect the Z bit in STATUS. To avoid this problem, the swapf instruction is used instead because it does not affect any STATUS flags. Two swapf instructions are required to swap the nibbles of W_Save and then swap them back to their original positions while putting the data into the working register.



USART - Interrupts

- Testing the flags and enable bits

```
BANKSEL  PIE1
btfss    PIE1, TXIE
goto     Cont
BANKSEL  PIR1
btfsc    PIR1, TXIF
goto     PutTXData
```

```
Cont:    ...
```

Getting Started - USART

© 2001

In the interrupt routine, after the context has been saved, the cause the interrupt must be determined.

In many cases it is not sufficient to test the interrupt flags, since these flags will get set regardless of whether the interrupt is enabled or not. For example if the TXIE bit is cleared to disable a transmit interrupt, and a timer interrupt occurs, the interrupt routine will get executed and the TXIF flag may be tested. This could cause the transmit interrupt code to get executed even though a transmit interrupt did not occur.

The safest solution is to test the interrupt enable flag as well as the interrupt flag itself, as shown in this example. The transmit interrupt routine PutTXData will only get executed if both the TXIE and the TXIF bits are set.

- Error flags
 - FERR - Framing Error
 - OERR - Overrun Error

- Software parity
 - Can use 9th bit

- Software checksum

There are several errors that can occur during serial communications and the USART can detect two types of errors automatically. These are indicated by two error flags in the RCSTA register for framing errors and overrun errors.

In addition, software can be used to detect other errors if parity or checksums are used. By using the ninth data bit as a parity bit, any single bit error in the data can be detected. A checksum on several bytes of data can provide an extra level of certainty about the validity of the data.

- Framing error
 - FERR bit is set
 - Stop bit detected as zero
 - Associated with the byte in the RCREG
 - FERR must be read before RCREG
 - Reading RCREG clears error

A framing error occurs when the stop bit is zero. The stop bit should always be one. The framing error is always associated with the byte in the RCREG and is passed through the FIFO in the same way as the data with which it is associated. Reading the RCREG allows the next data byte to be loaded into RCREG with its own framing error flag. For this reason it is essential to read the error flag before the data is read from RCREG, in the same way that the ninth data bit is read before the data in RCREG.

There is no need to clear the framing error flag since the FERR bit will be updated as soon as new data is received into RCREG.

- Overrun error
 - OERR bit is set
 - Two bytes already received and not read
 - Next byte received is discarded
 - CREN cleared and set again clears error

An overrun error occurs when the FIFO is full with two bytes that have already been received and a third byte has been clocked into the receive shift register. Since this third byte needs to be moved into the FIFO and there is no space available, it is discarded and an overrun error is indicated.

Overrun errors can be avoided by reading the incoming data from RCREG fast enough. Interrupts can often be used to ensure that data is read in time.

Once an overrun error occurs, no new data will be received until the receive logic has been reset by clearing the receive enable bit, CREN, and enabling it again. A common symptom of an overrun error is that the USART stops receiving unexpectedly, often after the first two bytes.

- Testing for errors

```
BANKSEL RCSTA
btfsc RCSTA,OERR
goto ErrOERR
btfsc RCSTA,FERR
goto ErrFERR
```

This code shows how the framing and overrun error bits can be tested. The code branches to the appropriate error handling routine when it detects that an error flag is set.

- Clearing framing errors

```
ErrFERR:  BANKSEL  RCREG
          movf    RCREG,W
          BANKSEL  ErrorCode
          movlw   FRAME_ERR
          movwf   ErrorCode
          call    ErrorHandler
```

This code shows how the framing error can be cleared, in effect, by reading the RCREG. Note that the error bit will remain set until new data has been received and loaded into the RCREG register.

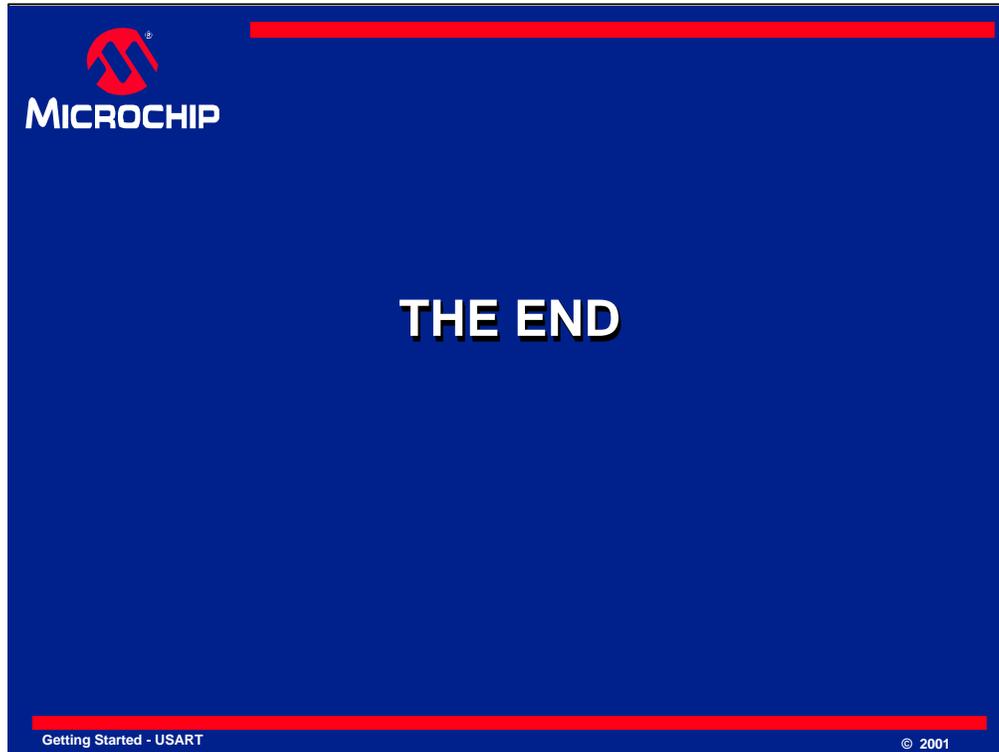
How the error is handled will depend entirely on the application. In this code, a general purpose register called ErrorCode is loaded with a predefined value represented by the label FRAME_ERR and a routine called ErrorHandler is called. This routine will take the appropriate action.

- Clearing overrun errors

```
ErrOERR:  BANKSEL  RCSTA
          bcf      RCSTA,CREN
          bsf      RCSTA,CREN
          BANKSEL  ErrorCode
          movlw   OVRUN_ERR
          movwf   ErrorCode
          call    ErrorHandler
```

This code shows how the overrun error can be cleared by clearing and setting the CREN bit. In some cases the two bytes in the FIFO will need to be read out first since they represent valid data.

How the error will be handled will again depend on the application. In this code, a general purpose register called ErrorCode is loaded with a predefined value represented by the label OVRUN_ERR and a routine called ErrorHandler is called. This routine will take the appropriate action.



That concludes the USART asynchronous mode getting started module. We hope that it has been helpful and will assist in future designs with our PICmicro devices. More information on the USART module can be found in the data sheets and reference manuals on our website. Thank you.