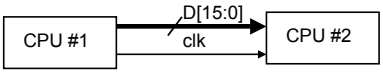
Parallel IO

Parallel IO – data sent over a group of parallel wires. Typically, a clock is used for synchronization.

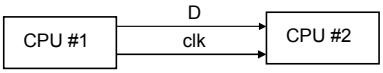


A 16-bit data channel is shown above. If data is transferred each rising clock edge, and clock rate is 300 MHz, then the **data transfer rate (bandwidth)** in bytes/sec is:

2 Bytes/clock period = 2 /(1/300e06)s
= 2 * 300e06/s = 600e06/s
= 600 MB/s (MB = MBytes)

Serial IO

Serial IO – data sent one bit at a time, over a single wire. A clock may or may not be used for synchronization



Question: Assuming one bit is sent each rising clock edge, how fast does the clock have to be achieve 600 MB/s?

600 MByte/s = 600 MBytes/s * 8 bits/1Byte = 4800Mb/s

Clock period = 1/4800e06
Clock Frequence = 1/clock period = 4800e06 = 4.8e09 = 4.8GHz

Parallel vs. Serial IO

Parallel IO Pros/Cons

Pros: Speed, can increase bandwidth by either making data channel wider or increasing clock frequency

Cons: Expensive (wires cost money!). Short distance only – long parallel wire causes crosstalk, data corruption.

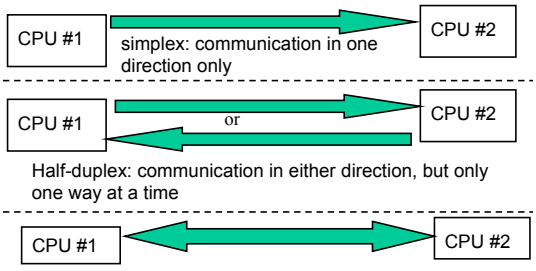
Serial IO Pros/Cons

Pros: Cheap, very few wires needed. Good for long distance interconnect.

Cons: Speed; the fastest serial link will typically have lower bandwidth than the fastest parallel link. However, for long distances (meters), new fast serial IO standards (USB2, Firewire) have replaced older parallel IO standards.

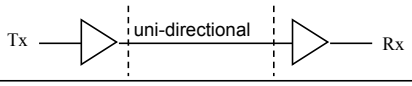
simplex vs half-duplex vs full-duplex

For communication channels

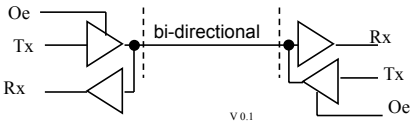


Wires: Simplex, Half-duplex

For wires:
simplex wire: communication occurs only in one direction.

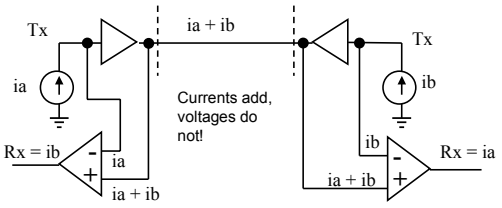


half-duplex wire: communication can occur in either direction, but with voltage signaling only one direction at a time.

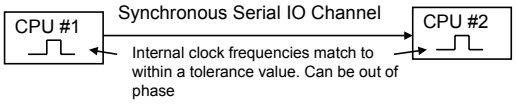


Wires: Full Duplex

Current mode signaling allows full duplex communication over a single wire. Used for communication in some advanced chipsets.



Synchronous Serial IO



Synchronous serial IO either

(a) sends the clock as a separate wire

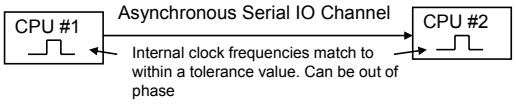
OR

(b) receiver (CPU #2) extracts clock from data stream or uses a Phase-Locked-Loop (PLL) and changes in the data stream to synchronize internal clock (phase alignment) to data stream.

For PLL synchronization, the data line must be guaranteed to have a minimum number of state changes ($0 \rightarrow 1$ or $1 \rightarrow 0$) within a particular time interval (*transition density*).

Synchronous serial IO can achieve high speeds; all new high speed serial standards are synchronous.

Asynchronous Serial IO



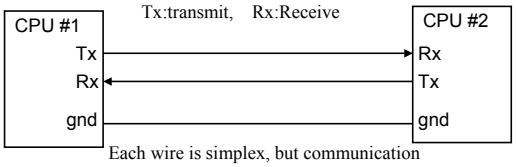
Asynchronous Serial I/O does not transmit the clock on a separate wire nor does it guarantee a particular transition densisty (ie., the data line could remain in the same state, either '1' or '0' for the duration of the transmission after the initial state change indicating start of transmission).

Asynchronous Serial I/O is used in older standards, is easy to implement, but is slower than synchronous serial standards.

A Three-Wire Async Serial Interface

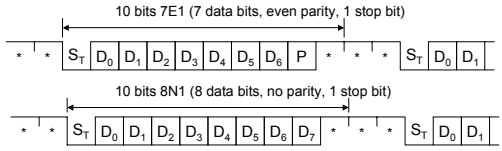
We will use a three-wire asynchronous serial interface to connect the PIC to an external PC.

This interface standard is known as RS-232 (there are more wires defined in the standard, we will only use 3 wires)



Each wire is simplex, but communication channel is full duplex

Asynchronous Serial Data Frame



Mark – A Constant Logic-1 Denoted by * *Space* – A Constant Logic-0

Standard is for Serial Line to CONSTANTLY be Driven to a MARK While Inactive

S_T – start bit
D_0 – LSB
$D_6(D_7)$ – MSB
P – parity bit
* - stop bit – a "mark"

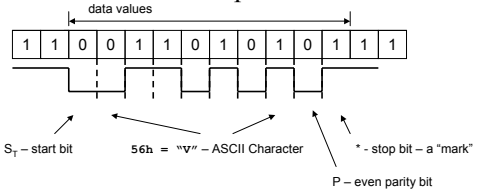
1 ASCII char. = 7 bits
(e.g. `DEL = 7fh` – higher is PC specific)
Typical is 10 bits for asynchronous transfer
Serial data with even/odd parity

Serial Data Receiver Starts Processing When:

1) high to low is sensed (start bit detection)
2) following (7 or 8) bits represent a character
3) parity bit for error detection
4) stop bit is detected (a "mark")

slide by Prof. Mitch Thornton

Example



When Receiver "sees" a Start Bit (high to low transition):

1) Local Timer Starts
2) Each bit sampled at midpoint in time (± % clock tolerance)
3) Maximum tolerance is ± ½ of 1 bit time interval over 10 intervals

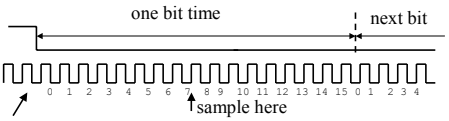
= (½)/10 = **5%**

slide by Prof. Mitch Thornton

Parity

- A **parity** bit is an extra bit added to a data frame to detect a single bit error
  - A single bit error is when one bit of the frame was received incorrectly (read as '0' when should have been '1', or vice-versa).
  - Not guaranteed to detect multi-bit errors
- Odd parity – parity bit value makes the total number of '1' bits in the frame odd
  - For 7-bit data value 0x56 (1010110), odd parity bit = '1'
- Even parity – parity bit value makes the total number of '1' bits in the frame even
  - For 7-bit data value 0x56 (1010110), even parity bit = '0'

Receiver Sampling



Receiver clock; period usually either 64x or 16x bit time (above is 16x).

At start bit, internal 4-bit counter set to 0. Sample at mid-point of bit time (counter value 7 or 8, some receivers sample at 7,8 and 9 and only accept bit if all values are the same – do this for glitch rejection).

Receiver/Transmitter clocks not perfectly matched. Our tolerance is ½ bit time (50%) spread over entire frame. Assuming a 10 bit frame, maximum mismatch between Rx/Tx clocks is 50%/10 = 5%,

Baud Rate vs Bits Per Second

- Baud rate is the rate at which signaling events are sent
- Bits per second (bps) is the number of bits transferred per second (any type of bits, data or overhead bits)
- If only a '1' or '0' is sent for each signaling event, then baud rate = bps
- However, could use a signaling protocol that transfers multiple bits per signaling event
  - i.e., use 4 different voltage levels, send two bits of data per signaling event (00 = -15v, 01= -5v, 10=+5v, 11 = 5v).
  - In this case, bit rate will be double the baud rate
- The effective data rate is the rate at which data is transferred, minus the overhead bits (ie. start and stop bits).

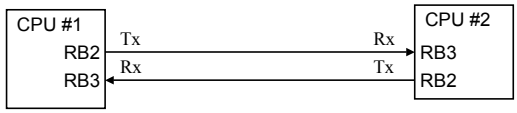
Common Baud Rates

Baud Rate	Divisor for 14.7456 MHz
115200	128
57600	256
38400	512
19200	1024
9600	2048
4800	4096
1200	16384

The PIC oscillator is divided down in order to provide the Tx/Rx clocks.

The divisor values on the right show that the commonly-used baud rates are even multiples of 14.7456MHz. This means these baud rates can be accurately reproduced by the PIC using this external oscillator frequency.

Software-driven Serial I/O



Can implement a serial link via software subroutines.

Must be able to implement software delay loops that can accurately delay for 1-bit time.

Does not require extra hardware on part of μP, but the processor operation is consumed by the send/receive operation.

This approach is not-so-fondly referred to as *bit-banging*.

putch(c)

```
/* assume RB2 is Tx line and
is already a '1' */
void putch(c)
unsigned char c;
{
  char i;
  bitclear(PORTB,2);
  delay_1bit();
  for(i=0;i<8;i++) {
    if (bittst(c,0))
      bitset(PORTB,2);
    else bitclr(PORTB,2);
    delay_1bit();
    c = c >> 1;
  }
  bitset(PORTB,2)
  delay_1bit();
}
```

putch(c) -- send one character over software serial link

send start bit

send 8-data bits, this does parallel-to-serial conversion

Check LSB value, send 0 or 1

right shift to send LSB to MSB

send stop bit, leave line in '1' condition

getch()

```
/* assume RB3 is Rx line */
unsigned char getch()
{
  unsigned char c;
  c = 0x00;
  while(bittst(PORTB,3));
  delay_onehalf_bit();
  for(i=0;i<8;i++) {
    delay_1bit()
    if (bittst(PORTB,3)) c = c | 0x80;
    if (i != 7) c = c >> 1;
  }
  return(c);
}
```

getch() -- receive one character over software serial link

Wait for start bit

Wait until middle of bit time

Input bit was '1', set MSB.

Right shift as bits are sent LSB to MSB

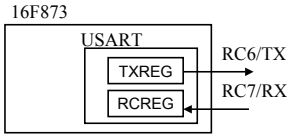
PIC16F873 USART

USART → **U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter

Hardware module in PIC that implements both synchronous and asynchronous serial IO. We will use asynchronous mode.

Frees the processor from having to implement software delay loops; receive/transmit done by USART while processor can do other tasks.

Will always use 8-bit, no parity for PIC serial IO.



USART Registers

- TXREG – holds a received character; read this to get character
- RCREG – write to this register to send a character
- RCSTA – contains status bits for received character
- SPBRG and TXSTA control baud rate
  - TXSTA status bits also select between async/sync IO, enable TX transmission
- PIR1 register contains status bits
  - TXIF (transmit interrupt flag), '1' if TXREG is empty
  - RCIF (receive interrupt flag), '1' if RCREG is full

RCIF, TXIF Bits

TABLE 10-5: REGISTERS ASSOCIATED WITH ASYNCHRONOUS TRANSMIS

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0Bh, 8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	R0IF
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF

Will be a '1' when RCREG has a character.

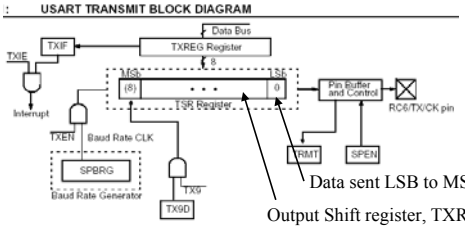
Wait until RCIF=1, then read RCREG to get received character.

Will be a '0' if TXREG is full (last character written to TXREG has not been sent yet).

Wait until TXREG=1, then write character to TXREG

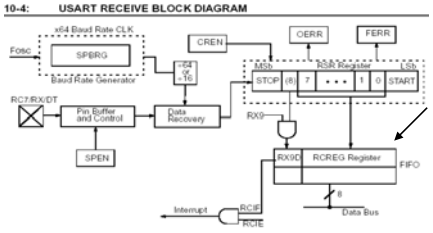
Transmit Hardware

TXREG is **double-buffered**. If TSR is empty, after write to TXREG, then TXREG transferred to TSR and TREG empty again (TXIF = 1). Do not have to wait for last character to be sent before writing new character.



Receive Hardware

RCREG is also a buffered register via 2 deep FIFO. This gives the processor more freedom in how fast it responds to received characters.



Two-deep FIFO. Can hold 2 characters, while 3rd character is being shifted into RSR register.

getch()/putch() (USART)

```
/* return 8 bit char
from Receive port */

unsigned char getch ()
{
  unsigned char c;
  /* wait until character
is received */
  /* while (!RCIF) */
  while (!bittst(PIR1,5));
  c = RCREG;
  return(c);
}
```

```
/* send 8 bit char to
Transmit port */

void putch (c)
unsigned char c;
{
  /* wait until transmit
reg empty */
  /* while (!TXIF) */
  while (!bittst(PIR1,4));
  TXREG = c;
}
```

These subroutines much simpler than software-based serial I/O. The *putch*/*getch* single character functions is used by the library function *printf()*.

Baud Rate Control

The baud rate is controlled by the 8-bit value in the SBPRG register and the BRGH bit (bit 2 in TXSTA register).

Baud_Rate = Fosc/ [K*(SBPRG+1)]

or

SBPRG = (Fosc/[K*Baud_Rate]) – 1

K = 16 if BRGH = 1 (high speed mode), then K = 16

K = 64 if If BRGH = 0 (low speed mode)

Baud Rate Examples

Desired baud rate of 9600, Fosc = 14.7456 MHz

What is SBPRG value for high speed mode?

SBPRG = (14.7456e06/[16*9600]) – 1
= 95

What is SBPRG value for low speed mode?

SBPRG = (14.7456e06/[64*9600]) – 1
= 23

Enabling Async Serial IO

1. Configure serial port pins (RC6/TX, RC7/RX) via SPEN bit (RCSTA:7 =1). To be on the safe side, also set TRISC7=0, TRISC6 =0, setting RC7, RC6 of the parallel port logic to inputs.
2. Select high or low speed baud rate via BRGH bit (bit 2) of TXSTA register
3. Select async mode SYNC bit (TXSTA:4 = 0)
4. Select 8-bit transmit via TX9 bit (TXSTA:6 = 0)
5. Select 8-bit receive via RX9 bit (RCSTA:6 = 0)
6. Enable transmit port via TXEN bit (TXSTA:5 = 1)
7. Enable receive port via CREN bit (RCSTA:4 = 1)

Example *C* code to Enable Serial I/O

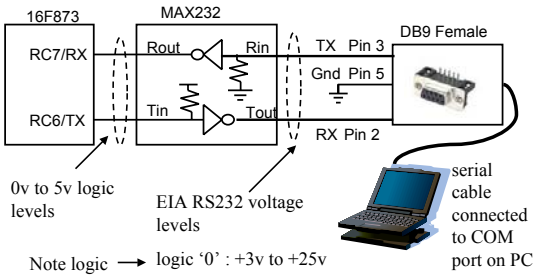
```
/* setup Async communication */

bitset(RCSTA, 7); /* serial port enable */
bitset(TRISC, 7); /* RC7 input */
bitset(TRISC, 6); /* RC6 input */
bitset(TXSTA, 2); /* high speed mode */
SPBRG = 95;       /* 9600 baud, high speed mode */
bitclr(TXSTA, 4); /* async mode */
bitclr(TXSTA, 6); /* 8-bit transmit */
bitclr(RCSTA, 6); /* 8-bit reception*/
bitset(TXSTA, 5); /* transmit enable*/
bitset(RCSTA, 4); /* enable receive */
```

At this point, ready to call getch()/putch() to perform serial IO.

Receive Error Conditions

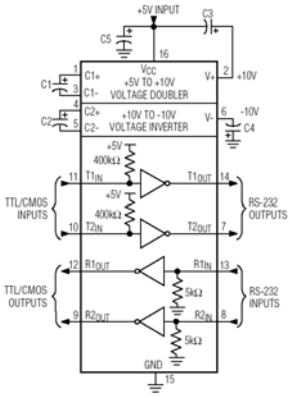
- FERR bit (RCSTA:2) is set when a framing error is detected
  - A framing error occurs when a STOP bit is detected as a '0' value.
  - This happens is actual baud rate slower than expected baud rate.
- OERR bit (RCSTA:1) is set when an overrun error is detected
  - Waited too long to read RCREG and FIFO fills up
  - Set when stop bit of 3rd byte is detected (2 bytes in FIFO, and 3rd byte is shifted in)
  - All receive activity is stopped; to reset, clear CREN (RCSTA:4), then set CREN.

PIC to PC Serial IO Connection



What is EIA-RS232?

- An interface standard originally used to connect PCs to modems
  - A modem is a device used to send digital data over phone lines
  - The standard defines voltage levels, cable length, connector pinouts, etc
- There are other signals in the standard beside TX, RX, Gnd
  - The other signals are used for modem control (Data Carrier Detect, Ring Indicator, etc) and flow control (flow control signals are used to determine if a device is ready to accept data or not)
  - We will not cover the other signals in the RS232 standard



MAXIM 232/232A RS232 driver/receiver

Converts RS232 voltage levels to digital levels and vice-versa

External capacitors used with internal charge pump circuit to produce +/- 10V from 5V supply

Hyperterminal

Will use Hyperterminal program on PC to communicate with PIC.

Under Programs→Accessories →Communications → Hyperterminal

When configuring Hyperterminal connection, must know port number (COM1/COM2/etc), baud rate, data bits (8), parity (none), stop bits (1), and flow control(none)

On PC lab machines, use COM1

Very important to set flow control to **none** since we are only using a 3-wire connection and not using the handshaking lines in the RS232 standard. If you forget this, then will not receive any characters.

What do you have to know?

- Difference between async/sync serial IO
- Format of async serial IO frames
- Details of PIC 16F873 USART operation for asynchronous IO
- Definitions of simplex, half-duplex, full-duplex
- What is meant by RS-232 and the need for voltage conversion between RS-232 and digital levels
- PIC16F873 to PC serial port interfacing