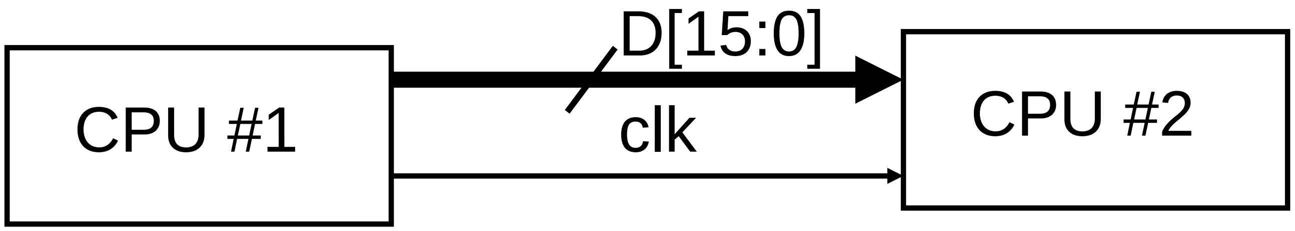
Parallel IO (Input/Output)

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



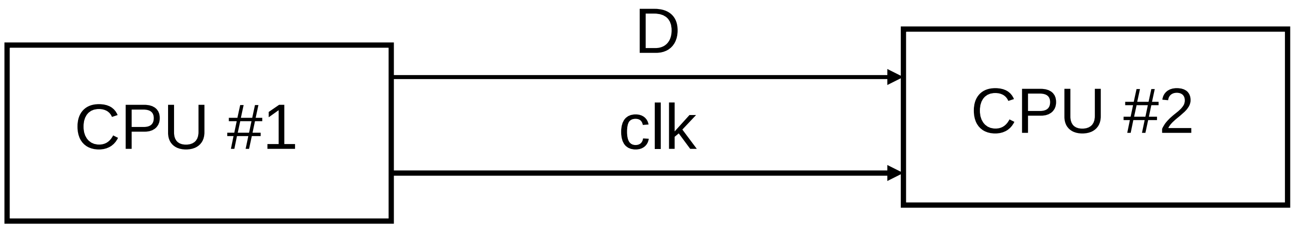
A 16-bit data channel is shown above. If data is transferred each rising clock edge, and clock rate is 300 MHz, then the **data transfer rate (bandwidth)** in bytes/sec is:

2 Bytes/clock period = 2Bytes /(1/300e06)s

= 2B * 300e06/s = 600e06B/s = 600×10^6 B/s

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 x 10^6 B/s?

600 x 10^6 B/s = 600 x 10^6 B/s * 8 bits/1Byte = 4800 x 10^6 b/s

1 bit/clock period = 4800 x 10^6 b/s

1 bit * Clock Frequency = 4800 x 10^6 b/s

Clock Frequency = 4800 x 10^6 Hz = 4.8 GHz

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.

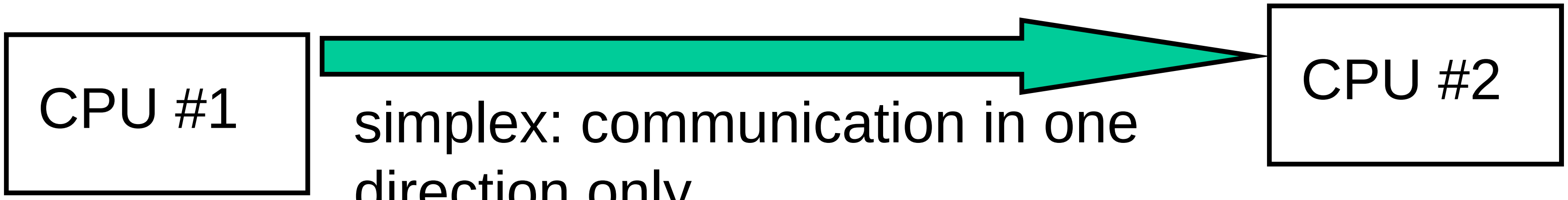
Serial IO Pros/Cons

Pros: Cheap, very few wires needed. Good for long distance ("inches to feet") interconnect.

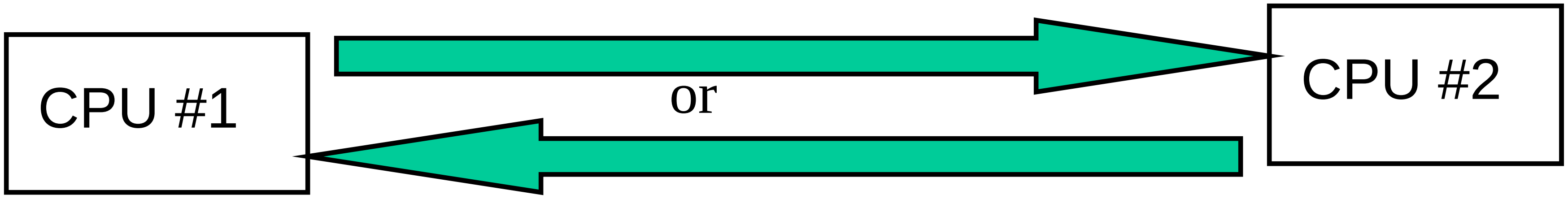
Cons: Speed; the fastest serial link will typically have lower bandwidth than the fastest parallel link. However, due to faster integrated circuit technology, new fast serial IO standards (USB2/USB3, Firewire, SATA) have replaced older parallel IO standards.

simplex vs half-duplex vs full-duplex

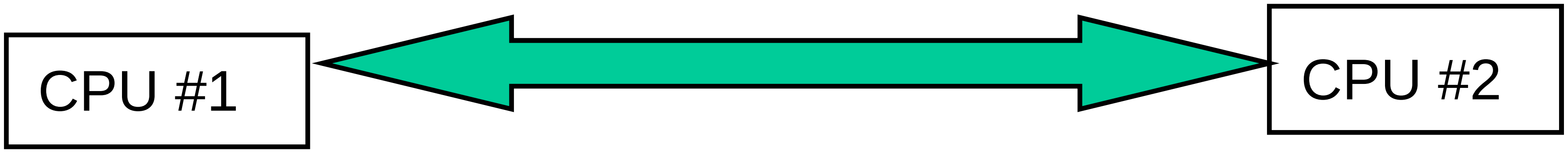
For communication channels



simplex: communication in one direction only



Half-duplex: communication in either direction, but only one way at a time

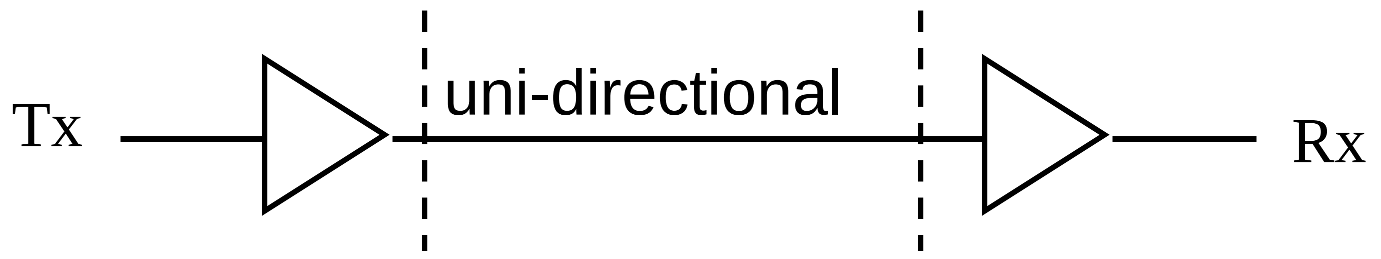


Full-duplex: communication in both directions at same time.

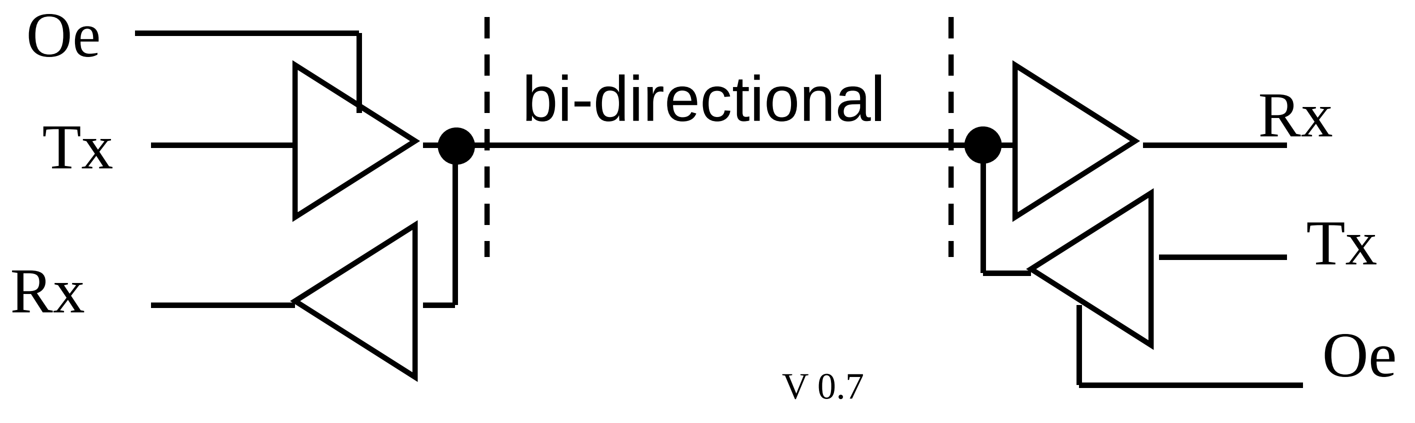
Wires: Simplex, Half-duplex

For wires:

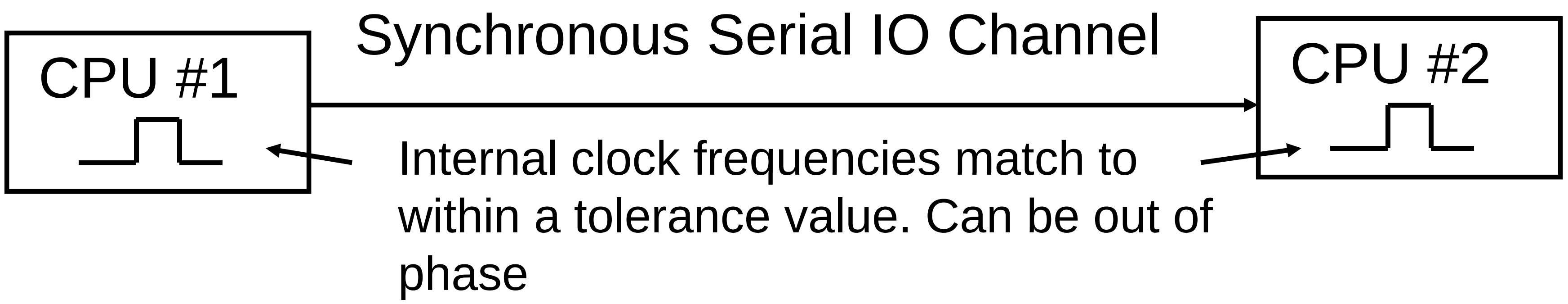
simplex wire: communication occurs only in one direction.



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



Synchronous Serial IO


Synchronous Serial IO Channel
CPU #1
CPU #2
Internal clock frequencies match to within a tolerance value. Can be out of phase

Synchronous serial IO either

(a) sends the clock as a separate wire

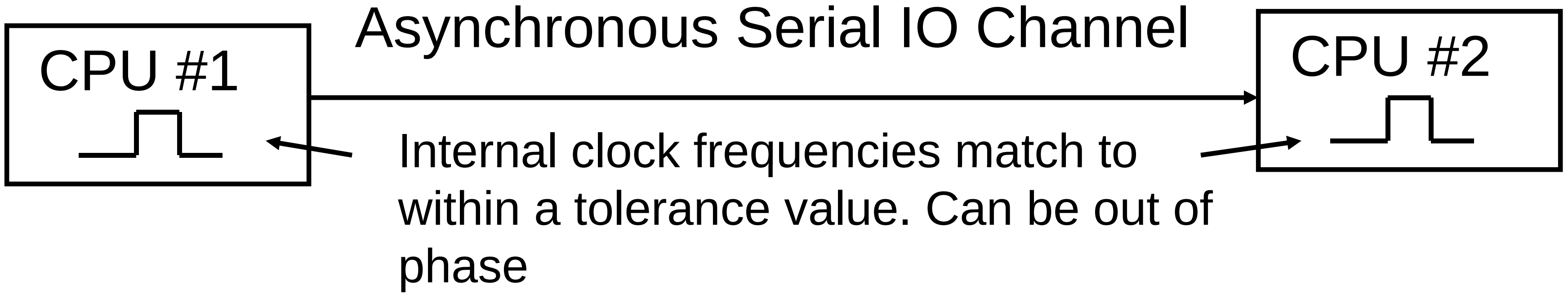
OR

(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 IO Channel
CPU #1
CPU #2
Internal clock frequencies match to within a tolerance value. Can be out of phase

Asynchronous Serial I/O does not transmit the clock on a separate wire nor does it guarantee a particular transition density (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).

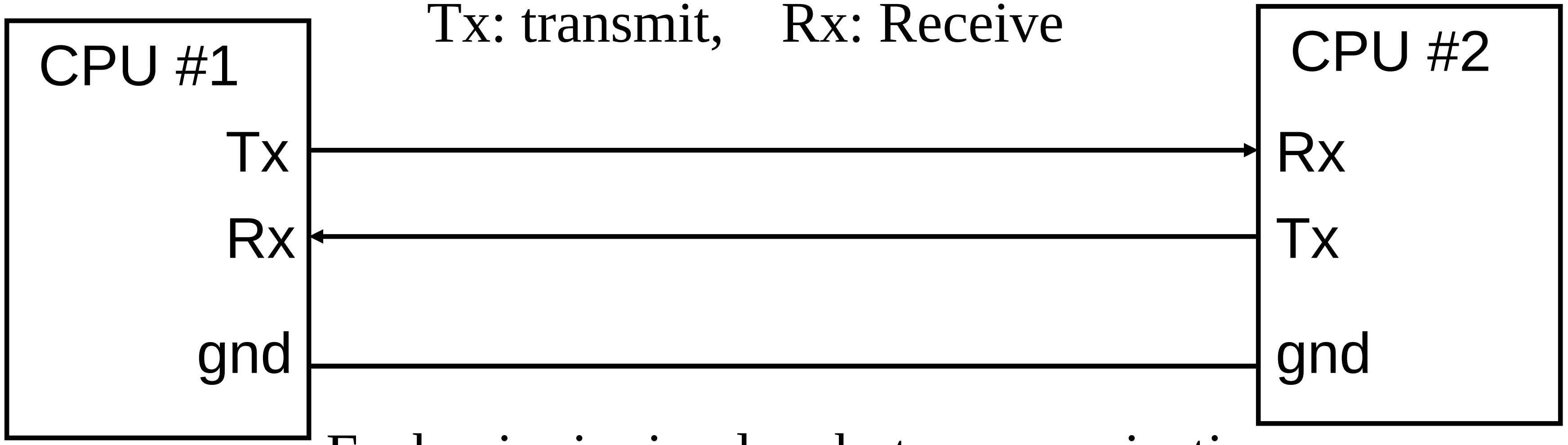
Asynchronous Serial I/O is used in older standards, is easy to implement, but is slower than synchronous serial standards.

A Three-Wire Async Serial Interface

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

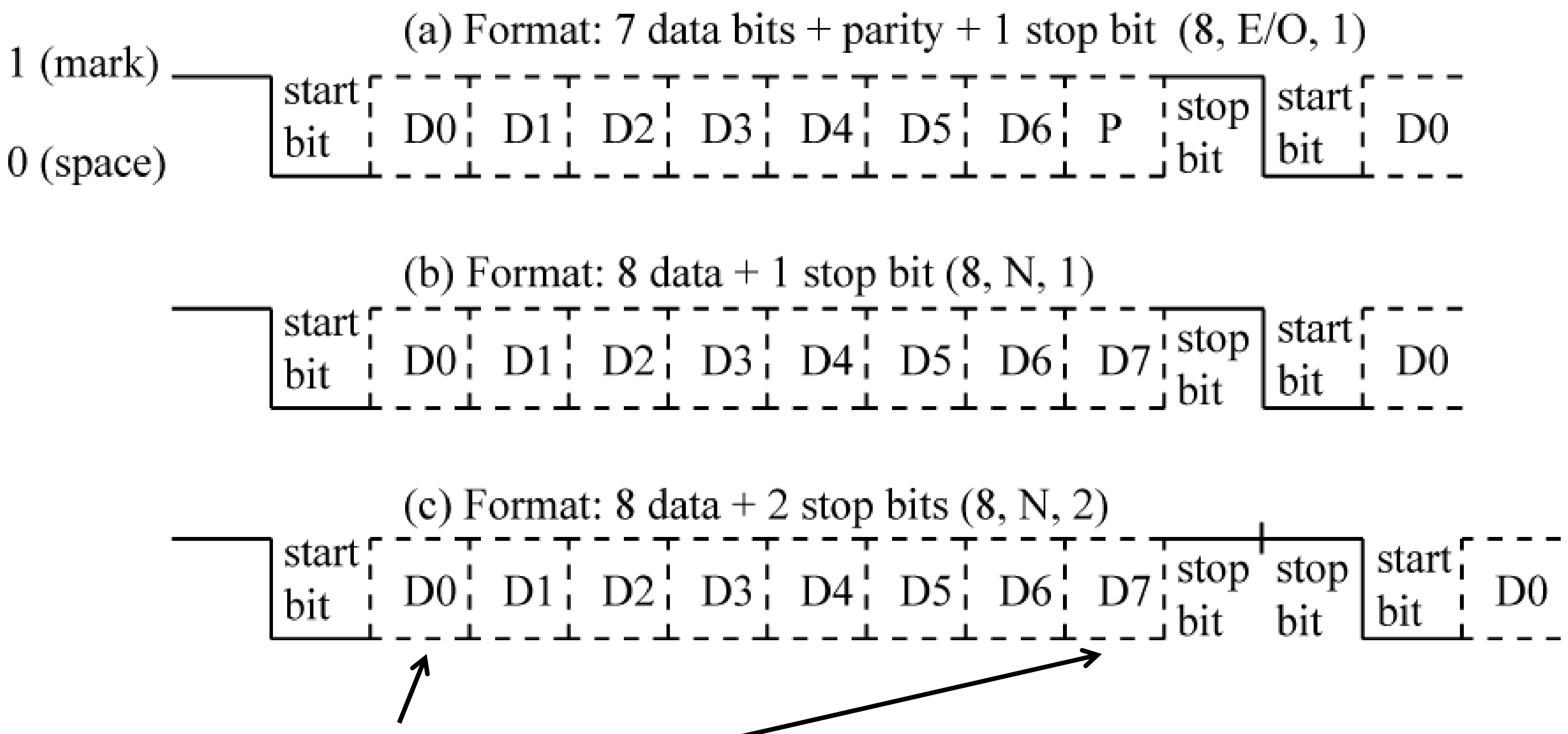
A version of this interface standard is known as RS-232 (there are more wires defined in the standard and different voltage levels; we will only use 3 wires)

Tx: transmit,   Rx: Receive



Each wire is simplex, but communication channel is full duplex

Asynchronous Serial Data Formats



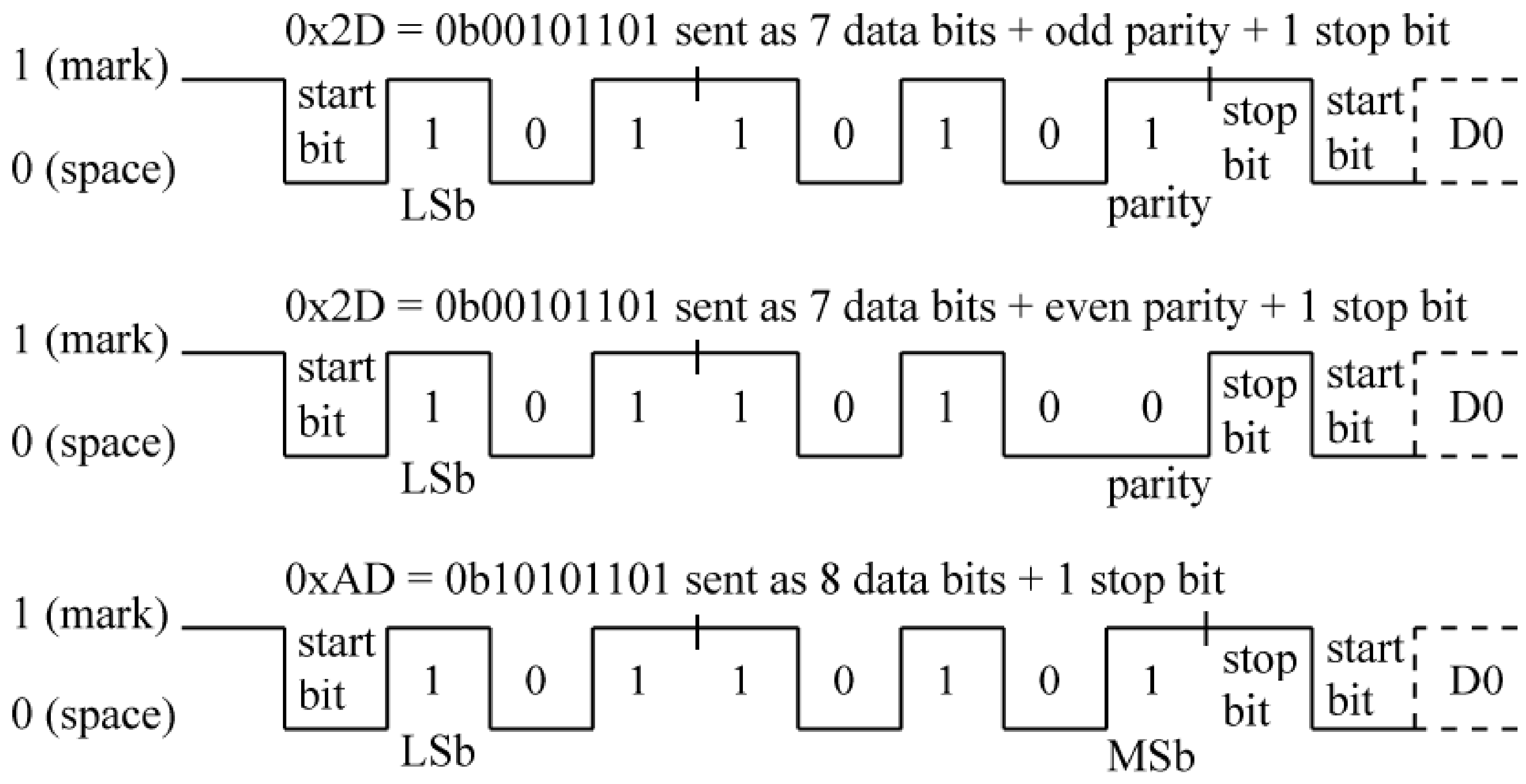
Data sent LSb to MSb.

On the PIC24 µC, will use 8 data bits.

Parity

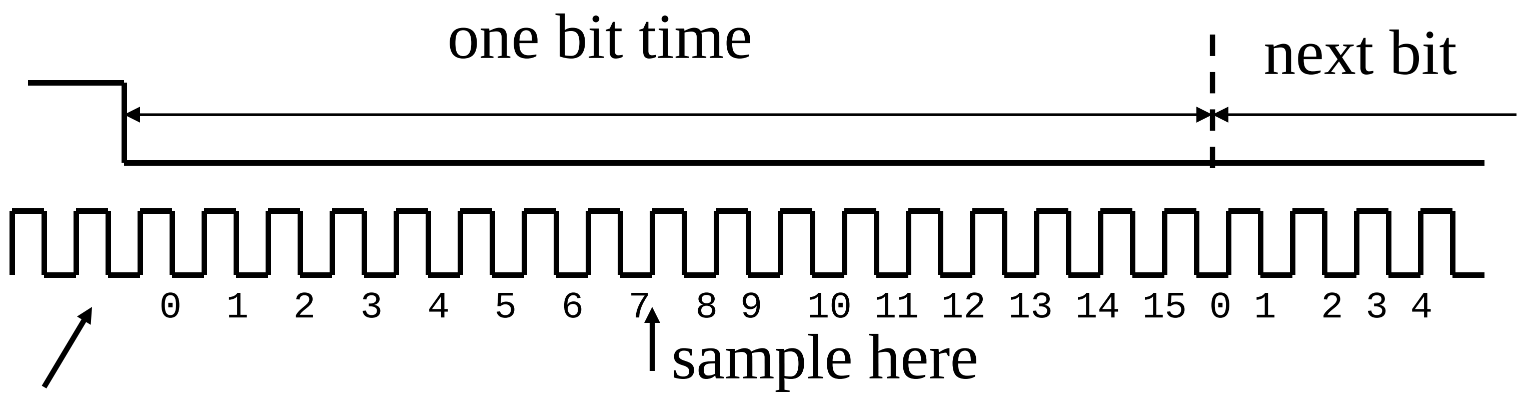
- A **parity** bit is an extra bit added to a data frame to detect a single bit error
  - A single bit error is when one bit of the frame was received incorrectly (read as ‘0’ when should have been ‘1’, or vice-versa).
  - Not guaranteed to detect multi-bit errors
- Odd parity – parity bit value makes 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’

Example



0x2D = 0b00101101 sent as 7 data bits + odd parity + 1 stop bit
1 (mark)
0 (space)
start bit
1
0
1
1
0
1
0
1
stop bit
start bit
D0
LSb
parity
0x2D = 0b00101101 sent as 7 data bits + even parity + 1 stop bit
1 (mark)
0 (space)
start bit
1
0
1
1
0
1
0
0
stop bit
start bit
D0
LSb
parity
0xAD = 0b10101101 sent as 8 data bits + 1 stop bit
1 (mark)
0 (space)
start bit
1
0
1
1
0
1
0
1
stop bit
start bit
D0
LSb
MSb

Receiver Sampling



Receiver clock; period is either 4x or 16x the bit time (above is 16x).

At start bit, an internal 4-bit counter set to 0. Data is sampled at the mid-point of a 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 are 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 = +15v).
  - 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).

Sample Calculations

1. What is a bit time (in μs) for a baud rate of 57600 ?

Bit time = 1/baud rate = 1/57600 = 1.736 x 10^{-5} seconds
Convert to μs,
1.736 x 10^{-5} s x 10^{6} μs /s = 1.736 x 10^{1} μs = 17.36 μs

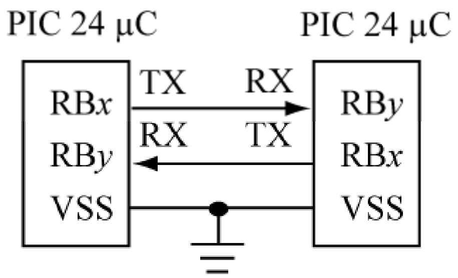
---

2. How long does it take to send 20 bytes at a baud rate of 19200 assuming a data format of 1 start + 8 data + 1 stop? Give the answer in milliseconds (ms).

Total bits = 20 x (1 + 8 + 1) = 20 x 10 = 200 bits.
Total time = bits x 1 bittime = 200 x 1/19200 = 0.01042 seconds
Convert to milliseconds:
0.01042 s x 1000 ms/s = 10.42 ms

Software-driven Serial I/O

```
/// Soft UART Config
#define CONFIG_STX() CONFIG_RB2_AS_DIG_OUTPUT()
#define STX _LATB2    //STX state
#define CONFIG_SRX() CONFIG_RB3_AS_DIG_INPUT()
#define SRX _RB3      //SRX state
#define DEFAULT_SOFT_BAUDRATE 19200
uint16 u16_softBaudRate = DEFAULT_SOFT_BAUDRATE;
```



```
void doBitDelay (uint16 u16_baudRate) {
  if (u16_baudRate == 9600) { DELAY_US(106); }
  else if (u16_baudRate == 19200) DELAY_US(52);
}
```

Add more tests to support additional baud rates

1/19,200 seconds

```
void doBitHalfDelay (uint16 u16_baudRate) {
  if (u16_baudRate == 9600) { DELAY_US(53); }
  else if (u16_baudRate == 19200) DELAY_US(26);
}
```

(a) Send start+8 data+stop

```
void outCharSoft(uint8 u8_c)
{
 uint8 u8_i;
 STX = 0;
 doBitDelay(u16_softBaudRate);
 for (u8_i=0;u8_i<8;u8_i++) {
  if (u8_c & 0x01)
    STX = 1;
  else STX = 0;
  doBitDelay(u16_softBaudRate);
  u8_c = u8_c >> 1;
 }
 STX = 1;
 doBitDelay(u16_softBaudRate);
}
```

Send start bit

Send 8 data bits, LSb to MSb

One data bit

Right shift for next bit

Send stop bit

(b) Receive start+8 data+stop

```
uint8 inCharSoft(void) {
 uint8 u8_i, u8_c;
  u8_c = 0x00;
 while(SRX) doHeartbeat();
 doBitHalfDelay(u16_softBaudRate);
 for (u8_i=0;u8_i<8;u8_i++) {
  doBitDelay(u16_softBaudRate);
  if (SRX) u8_c = u8_c | 0x80;
  if (u8_i != 7)
     u8_c = u8_c >> 1;
 }
 doBitDelay(u16_softBaudRate);
 return(u8_c);
}
```

Wait for start bit (exit when SRX = = 0)

Wait for middle of bit time

Read a bit

Right shift for next bit

Receive stop bit, no error check (assume it is a logic 1)

Works, but not an efficient use of CPU resources.

A serial protocol implemented by software and parallel I/O pins is called ‘bit-banging’.

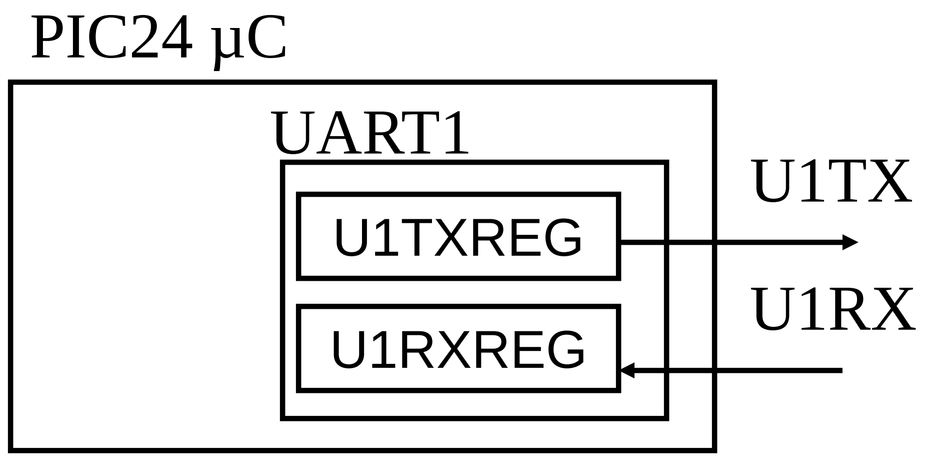
PIC24 µC UART*x*

UART → **U**niversal **A**synchronous **R**eceiver **T**ransmitter

Hardware module that implements asynchronous serial IO. Referred to as UART*x*, as there may be multiple UART modules on one PIC24 µC

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

Will always use 8-bit, no parity for PIC24 asynchronous serial IO.



UART Registers

- U1RXREG – holds a received character; read this to get character
- U1TXREG – write to this register to send a character
- U1STA, U1MODE – contains configuration bits and status bits for the module.
- U1BRG – sets the baud rate

UARTx Transmitter

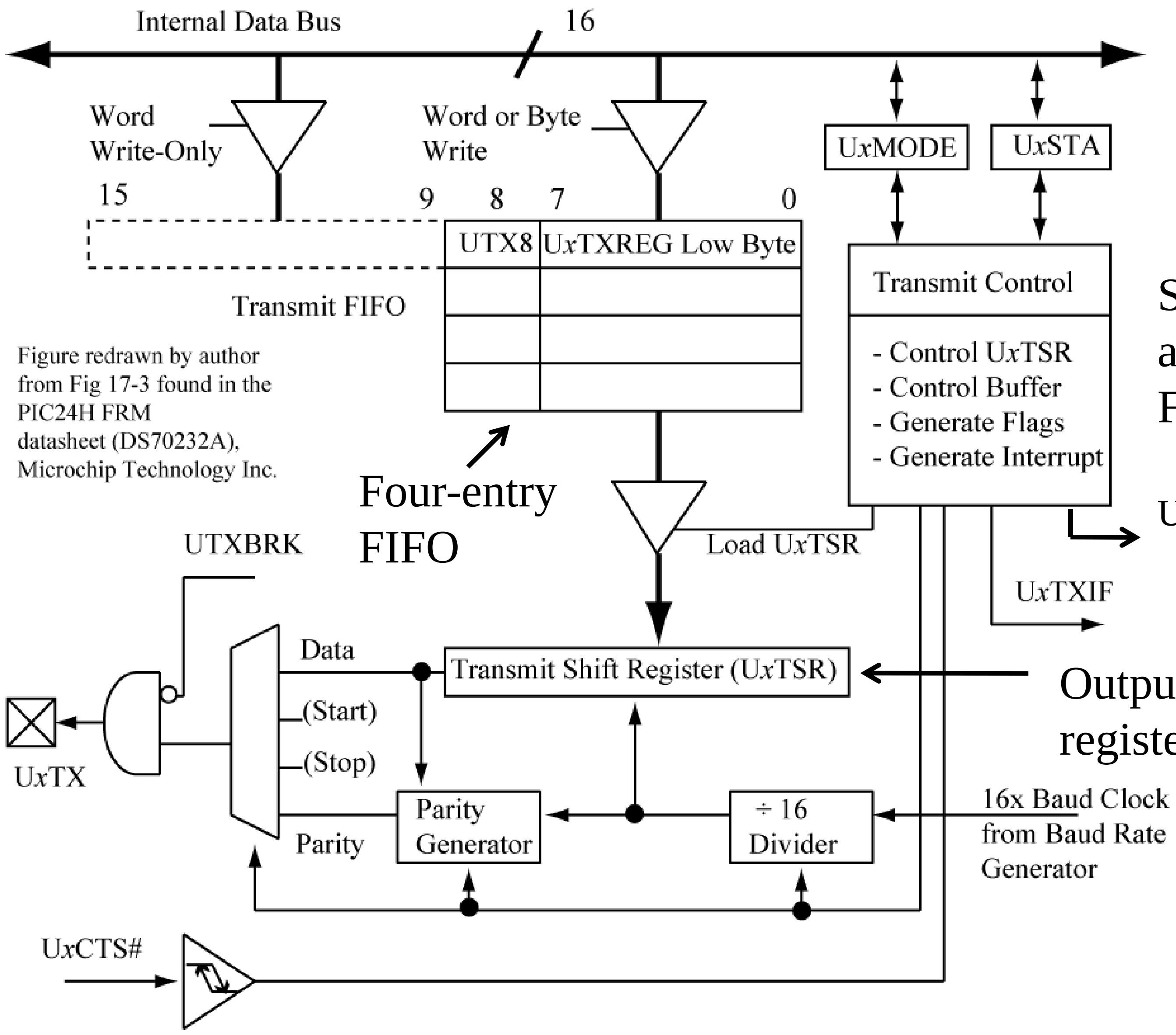
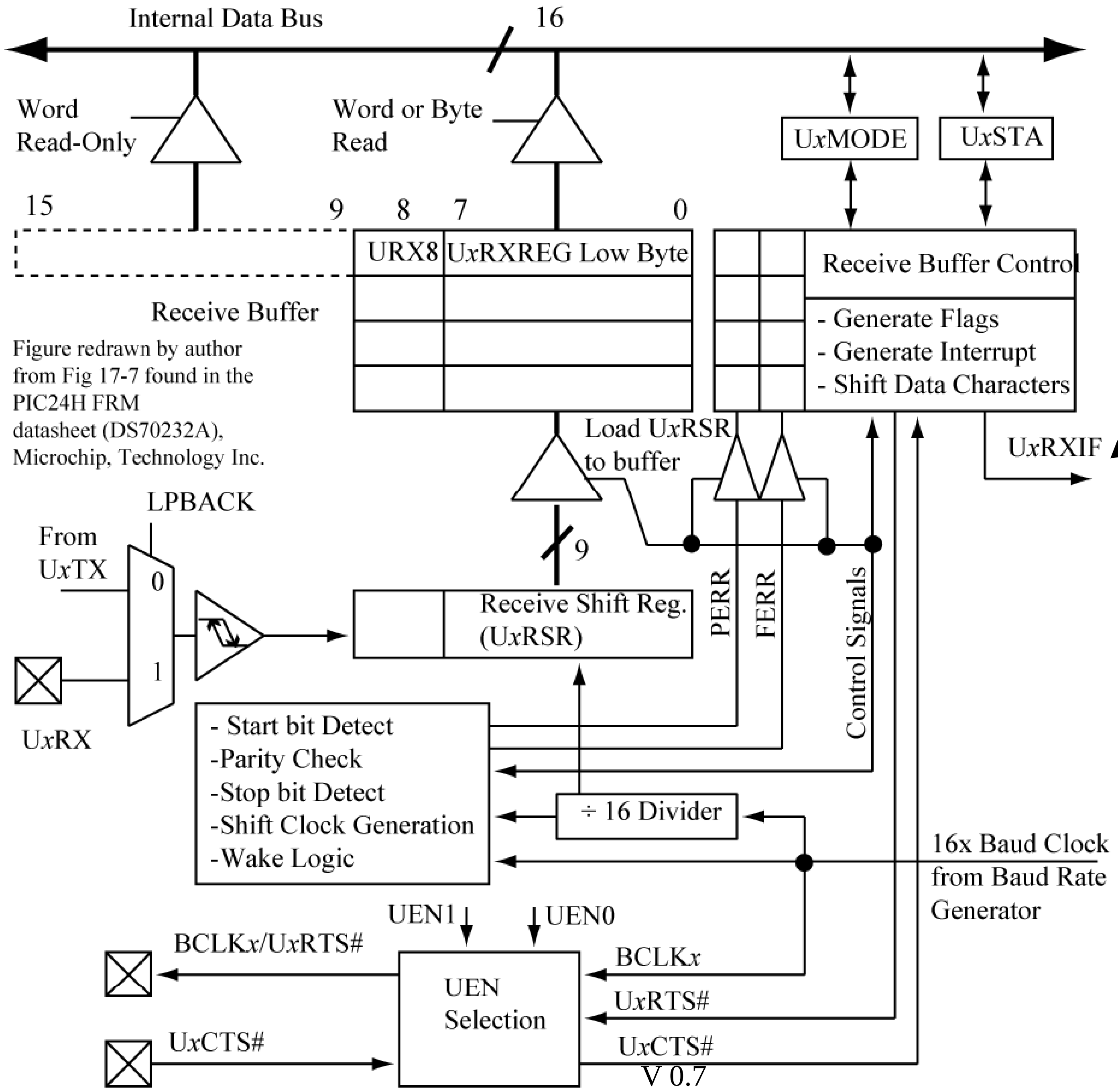


Figure redrawn by author from Fig 17-3 found in the PIC24H FRM datasheet (DS70232A), Microchip Technology Inc.

UARTx Receiver



Data available in FIFO

Receiver Error Conditions

Framing Error (FERR): The stop bit for a value was read as a ‘0’ instead of a ‘1’.

Parity Error (PERR): Wrong parity received.

Overrun Error (OERR): Input FIFO has overrun, happens on start bit of sixth character (four characters in FIFO, one character in receive shift register).

The Framing, Parity errors have to be checked before the character is read as they match up with the byte that is currently available in the FIFO.

Baud Rate Clock Generation

UxBRG register is period register for baud rate generation

Baud Rate = FCY /(S x (U*x*BRG+1))

Where S = 16 (low speed mode) or S = 4 (high speed mode).

Typically, we solve for U*x*BRG value given S and desired Baud Rate:

UxBRG = [FCY /(S x BR)] – 1

In our code, this is done via a *C* funcion, and we choose S = 16. Below is a simplified version of the code:

```
static inline void CONFIG_BAUDRATE_UART1(uint32 baudRate) {
  uint32 brg = (FCY/baudRate/16) - 1;
  U1MODEbits.BRGH = 0;
  U1BRG = brg;
}
```

$\longleftarrow$ Cannot exceed 65535 (0xFFFF) since U1BRG is a 16-bit register!

Example Baud Rates

(a) Using internal oscillator with PLL to achieve FOSC = 80 MHz (FCY = 40 MHz)
% Error does not account for internal oscillator frequency error.

Baud Rate	UxBRG (High Speed, BRGH = 1)	Actual	% Error	UxBRG (Low Speed, BRGH = 0)	Actual	% Error
230400	42	232558.1	0.9%	10	227272.7	-1.4%
115200	86	114942.5	-0.2%	21	113636.4	-1.4%
57600	173	57471.3	-0.2%	42	58139.53	0.9%
38400	259	38461.5	0.2%	64	38461.54	0.2%
19200	520	19193.9	0.0%	129	19230.77	0.2%
9600	1041	9596.9	0.0%	259	9615.385	0.2%
4800	2082	4800.8	0.0%	520	4798.464	0.0%

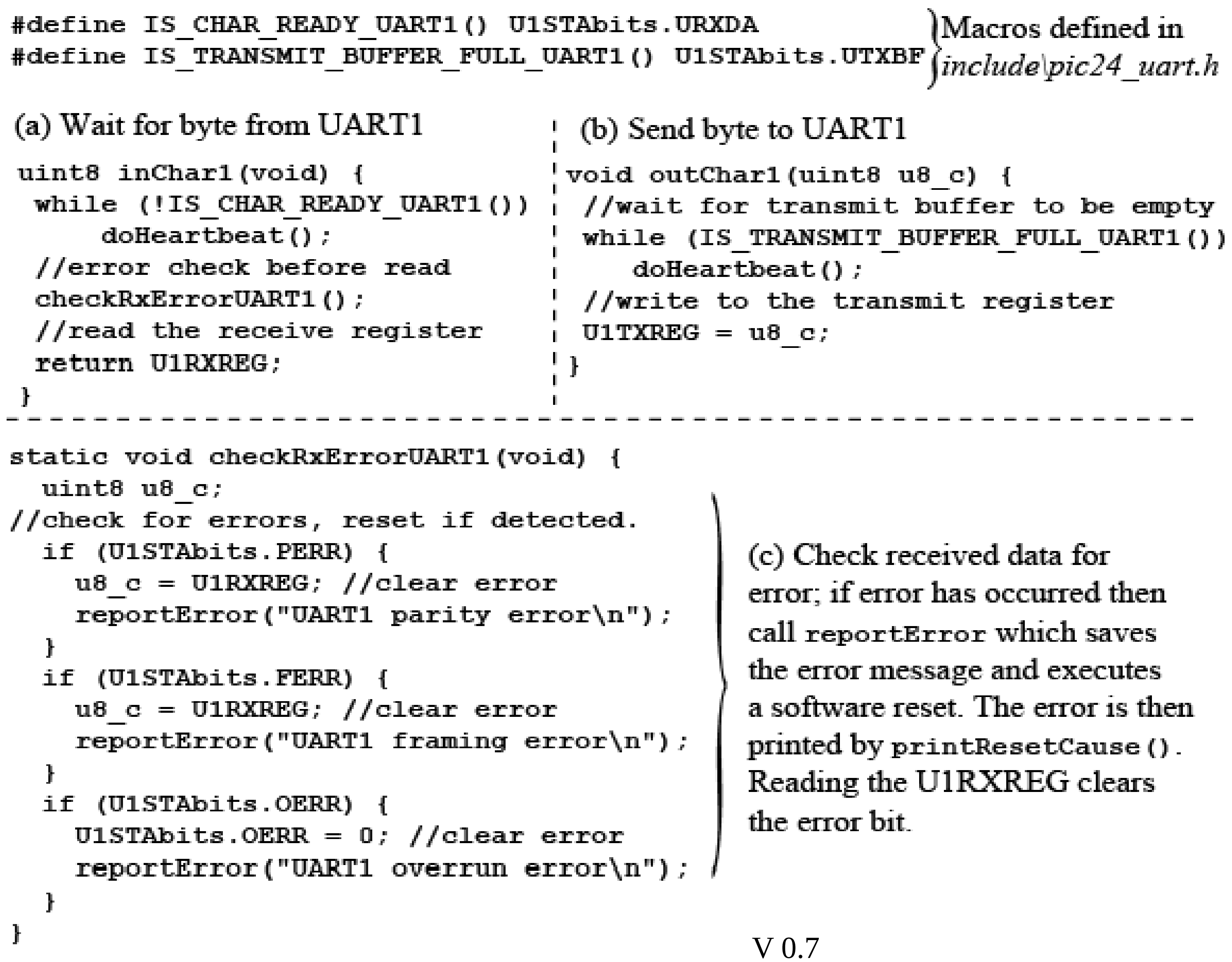
(b) Using external 7.3728 MHz crystal with internal PLL to achieve FOSC = 10 x crystal freq
(FOSC = 73.728 MHz, FCY = 36.864 MHz)

Baud Rate	UxBRG (High Speed, BRGH = 1)	Actual	% Error	UxBRG (Low Speed, BRGH = 0)	Actual	% Error
230400	39	230400.0	0.0%	9	230400	0.0%
115200	79	115200.0	0.0%	19	115200	0.0%
57600	159	57600.0	0.0%	39	57600	0.0%
38400	239	38400.0	0.0%	59	38400	0.0%
19200	479	19200.0	0.0%	119	19200	0.0%
9600	959	9600.0	0.0%	239	9600	0.0%
4800	1919	4800.0	0.0%	479	4800	0.0%

Because sender/receiver must both agree, need 'standard' baud rates.

The 2-3% inherent error in internal fast RC oscillator means that at low FCY clock rates (< 4 MHz, perhaps to save power) , it is difficult to match higher baud rates.

`inChar1/outChar1` UART1 Functions



```
#define IS_CHAR_READY_UART1() U1STAbits.URXDA
#define IS_TRANSMIT_BUFFER_FULL_UART1() U1STAbits.UTXBF
```

Macros defined in *include\pic24_uart.h*

(a) Wait for byte from UART1

```
uint8 inChar1(void) {
 while (!IS_CHAR_READY_UART1())
     doHeartbeat();
 //error check before read
 checkRxErrorUART1();
 //read the receive register
 return U1RXREG;
}
```

(b) Send byte to UART1

```
void outChar1(uint8 u8_c) {
 //wait for transmit buffer to be empty
 while (IS_TRANSMIT_BUFFER_FULL_UART1())
    doHeartbeat();
 //write to the transmit register
 U1TXREG = u8_c;
}
```

```
static void checkRxErrorUART1(void) {
  uint8 u8_c;
//check for errors, reset if detected.
  if (U1STAbits.PERR) {
    u8_c = U1RXREG; //clear error
    reportError("UART1 parity error\n");
  }
  if (U1STAbits.FERR) {
    u8_c = U1RXREG; //clear error
    reportError("UART1 framing error\n");
  }
  if (U1STAbits.OERR) {
    U1STAbits.OERR = 0; //clear error
    reportError("UART1 overrun error\n");
  }
}
```

(c) Check received data for error; if error has occurred then call `reportError` which saves the error message and executes a software reset. The error is then printed by `printResetCause()`. Reading the U1RXREG clears the error bit.

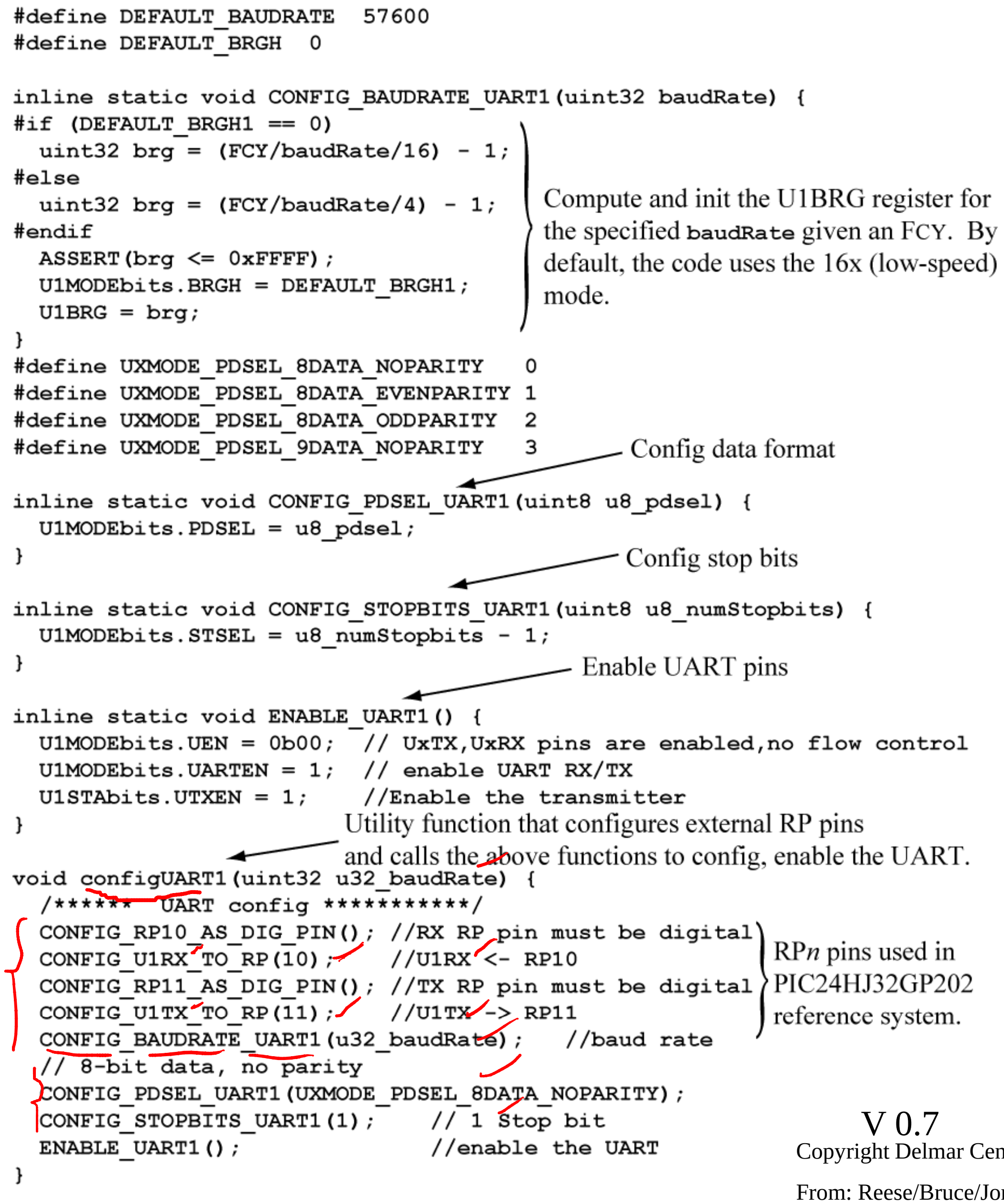
The functions contained in *pic24_uar*t.c and *pic24_uart.h*

Functions provided for four UART modules

`inchar1()`, **`inchar2()`**, **`inchar3()`**,

..etc.

UART1 Configuration



```
#define DEFAULT_BAUDRATE  57600
#define DEFAULT_BRGH  0

inline static void CONFIG_BAUDRATE_UART1(uint32 baudRate) {
#if (DEFAULT_BRGH1 == 0)
  uint32 brg = (FCY/baudRate/16) - 1;
#else
  uint32 brg = (FCY/baudRate/4) - 1;
#endif
  ASSERT(brg <= 0xFFFF);
  U1MODEbits.BRGH = DEFAULT_BRGH1;
  U1BRG = brg;
}
#define UXMODE_PDSEL_8DATA_NOPARITY   0
#define UXMODE_PDSEL_8DATA_EVENPARITY 1
#define UXMODE_PDSEL_8DATA_ODDPARITY  2
#define UXMODE_PDSEL_9DATA_NOPARITY   3

inline static void CONFIG_PDSEL_UART1(uint8 u8_pdsel) {
  U1MODEbits.PDSEL = u8_pdsel;
}

inline static void CONFIG_STOPBITS_UART1(uint8 u8_numStopbits) {
  U1MODEbits.STSEL = u8_numStopbits - 1;
}

inline static void ENABLE_UART1() {
  U1MODEbits.UEN = 0b00;  // UxTX,UxRX pins are enabled,no flow control
  U1MODEbits.UARTEN = 1;  // enable UART RX/TX
  U1STAbits.UTXEN = 1;    //Enable the transmitter
}

void configUART1(uint32 u32_baudRate) {
  /******  UART config ***********/
  CONFIG_RP10_AS_DIG_PIN(); //RX RP pin must be digital
  CONFIG_U1RX_TO_RP(10);    //U1RX <- RP10
  CONFIG_RP11_AS_DIG_PIN(); //TX RP pin must be digital
  CONFIG_U1TX_TO_RP(11);    //U1TX -> RP11
  CONFIG_BAUDRATE_UART1(u32_baudRate);   //baud rate
  // 8-bit data, no parity
  CONFIG_PDSEL_UART1(UXMODE_PDSEL_8DATA_NOPARITY);
  CONFIG_STOPBITS_UART1(1);    // 1 Stop bit
  ENABLE_UART1();              //enable the UART
}
```

Compute and init the U1BRG register for the specified `baudRate` given an FCY. By default, the code uses the 16x (low-speed) mode.

Config data format

Config stop bits

Enable UART pins

Utility function that configures external RP pins and calls the above functions to config, enable the UART.

RP*n* pins used in PIC24HJ32GP202 reference system.

U1TX, U1RX pins must be mapped to RPx remappable pins.

Also, these pins must be configured for digital operation.

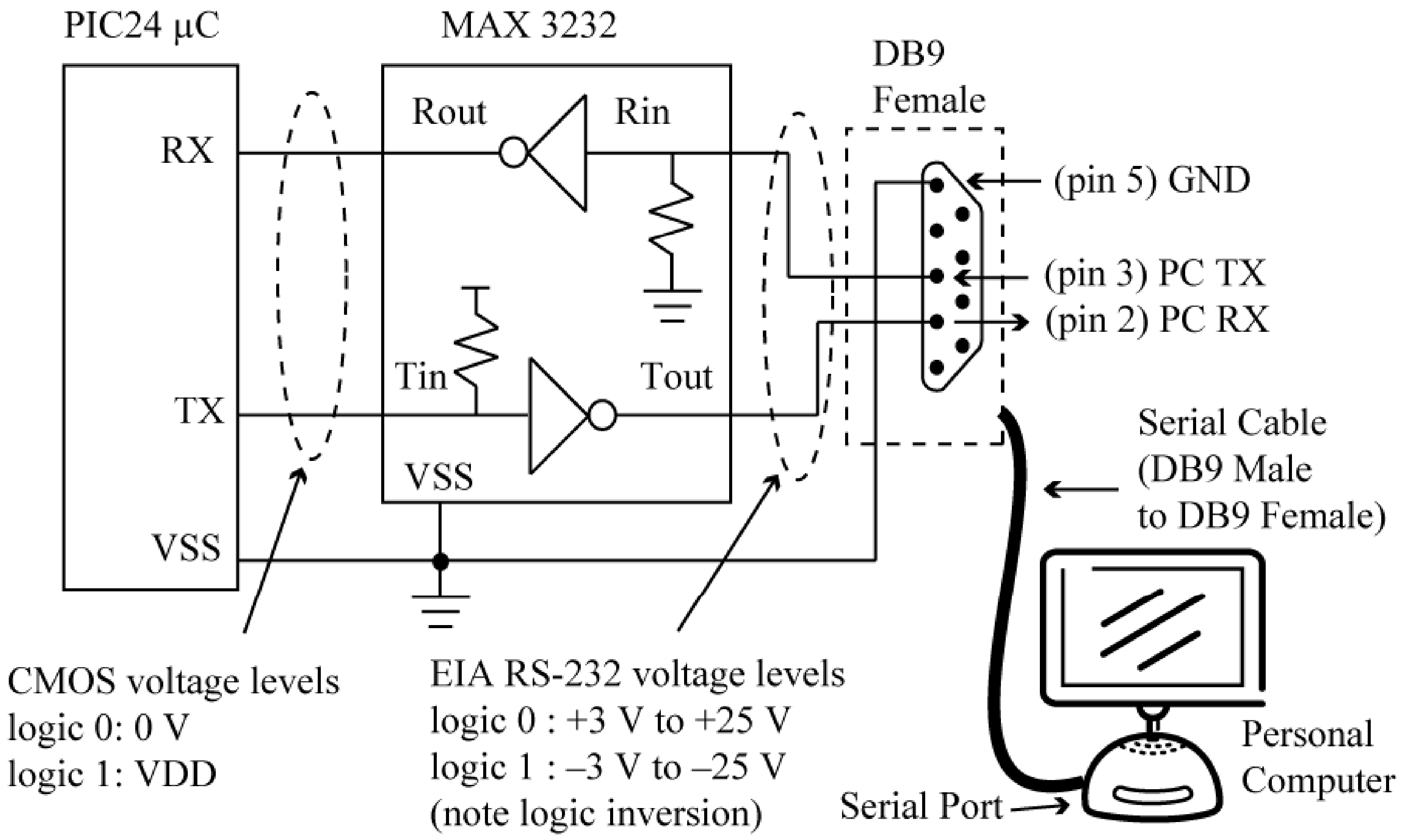
`inChar/outChar` Serial Functions

```
uint8 inChar(void) {
 switch (__C30_UART) {
#if (NUM_UARTS >= 1)
      case 1 : return inChar1();
#endif
#if (NUM_UARTS >= 2)
      case 2 : return inChar2();
#endif
#if (NUM_UARTS >= 3)
      case 3 : return inChar3();
#endif
#if (NUM_UARTS >= 4)
      case 4 : return inChar4();
#endif
 default :
    REPORT_ERROR("Invalid UART");
   }
}
```

```
 void outChar(uint8 u8_c) {
    switch (__C30_UART)
    {
#if (NUM_UARTS >= 1)
      case 1 : outChar1(u8_c); break;
#endif
#if (NUM_UARTS >= 2)
      case 2 : outChar2(u8_c); break;
#endif
#if (NUM_UARTS >= 3)
      case 3 : outChar3(u8_c); break;
#endif
#if (NUM_UARTS >= 4)
      case 4 : outChar4(u8_c); break;
#endif
      default :
       REPORT_ERROR("Invalid UART");
    }
  }
```

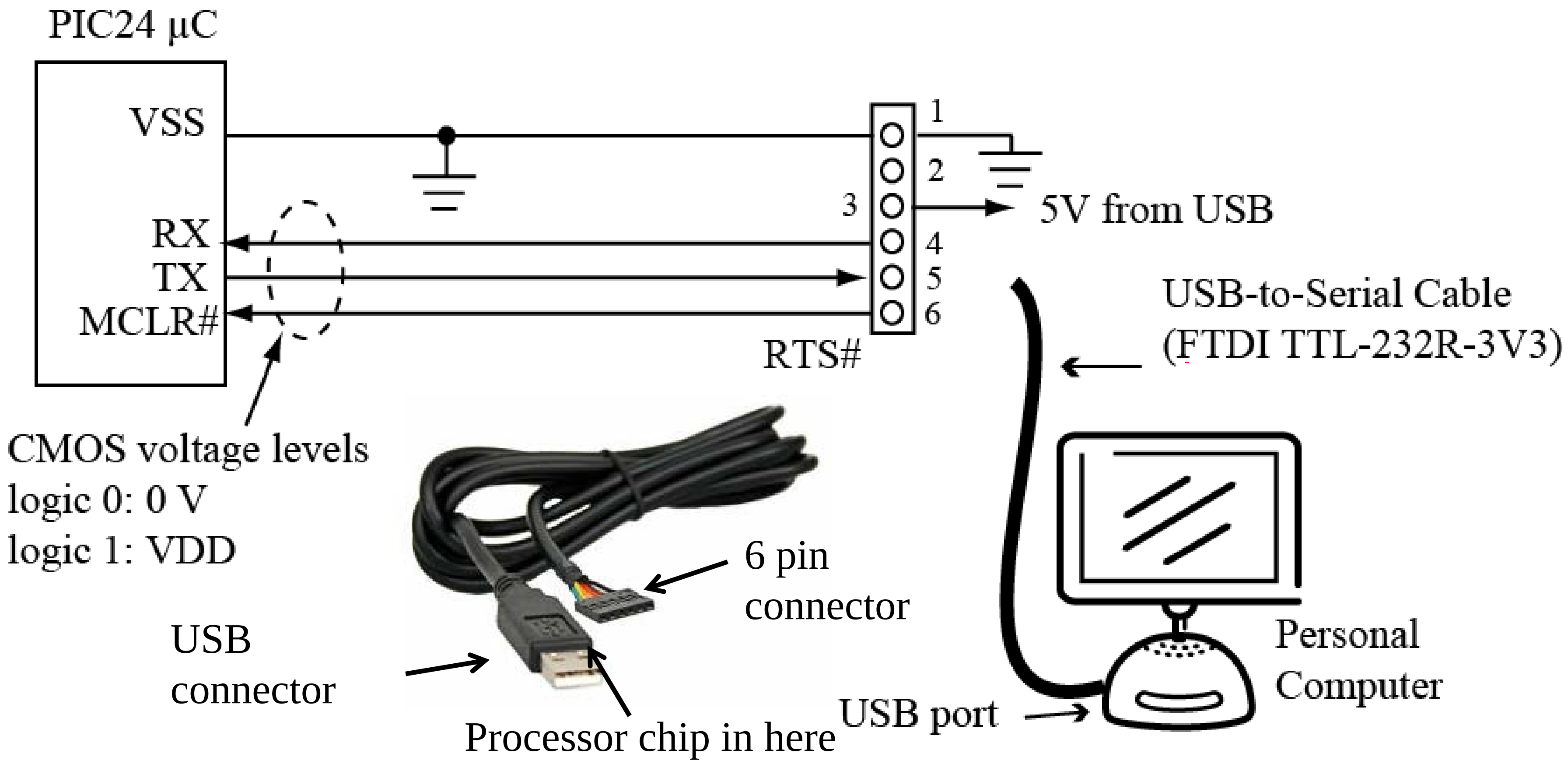
These functions in *pic24_serial.c*, *pic2_serial.h* and are used by **`outString()`**, **`inString()`**, etc. Can dynamically change which UART*x* the functions target by using the **`__C30_UART`** variable.

RS-232: PIC24 µC to PC Serial IO Connection



The diagram above shows a PIC24 to PC serial connection via an RS-232 serial cable. This requires conversion from RS-232 voltage levels to CMOS voltage levels.

USB to Serial: PIC24 µC to PC Connection



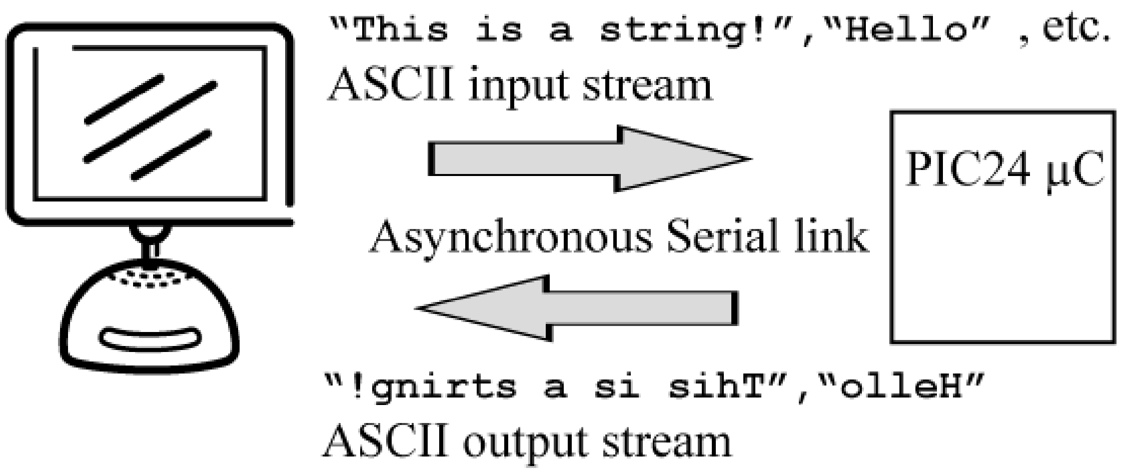
RS232 connectors on PCs have been replaced by USB, so now use a USB-to-Serial connector. This has a processor built into the connector that converts from USB to asynchronous serial.

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

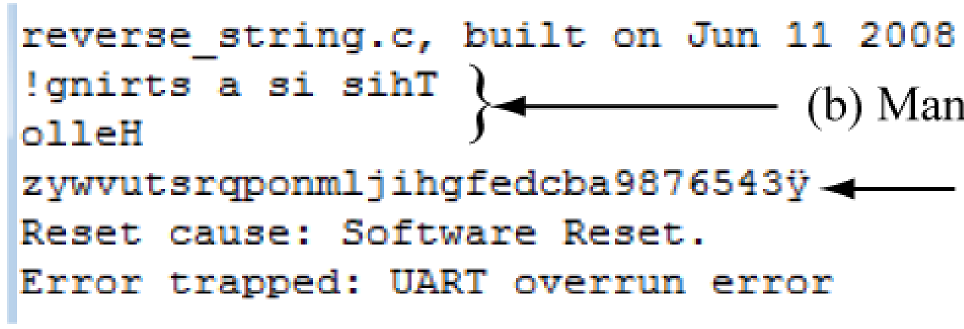
Interrupt Driven UART*x* Receive

(a) While the result is printing, more data is being sent, and the data is lost unless buffered.



```
int main (void) {
 char  sz_1[BUFSIZE+1];
 char  sz_2[BUFSIZE+1];

 configBasic(HELLO_MSG);
 while(1) {
  //input a string
  inString(sz_1,BUFSIZE);
  //reverse it, then print.
  reverseString(sz_1, sz_2);
  outString(sz_2);
  outString("\n");
 }
}
```



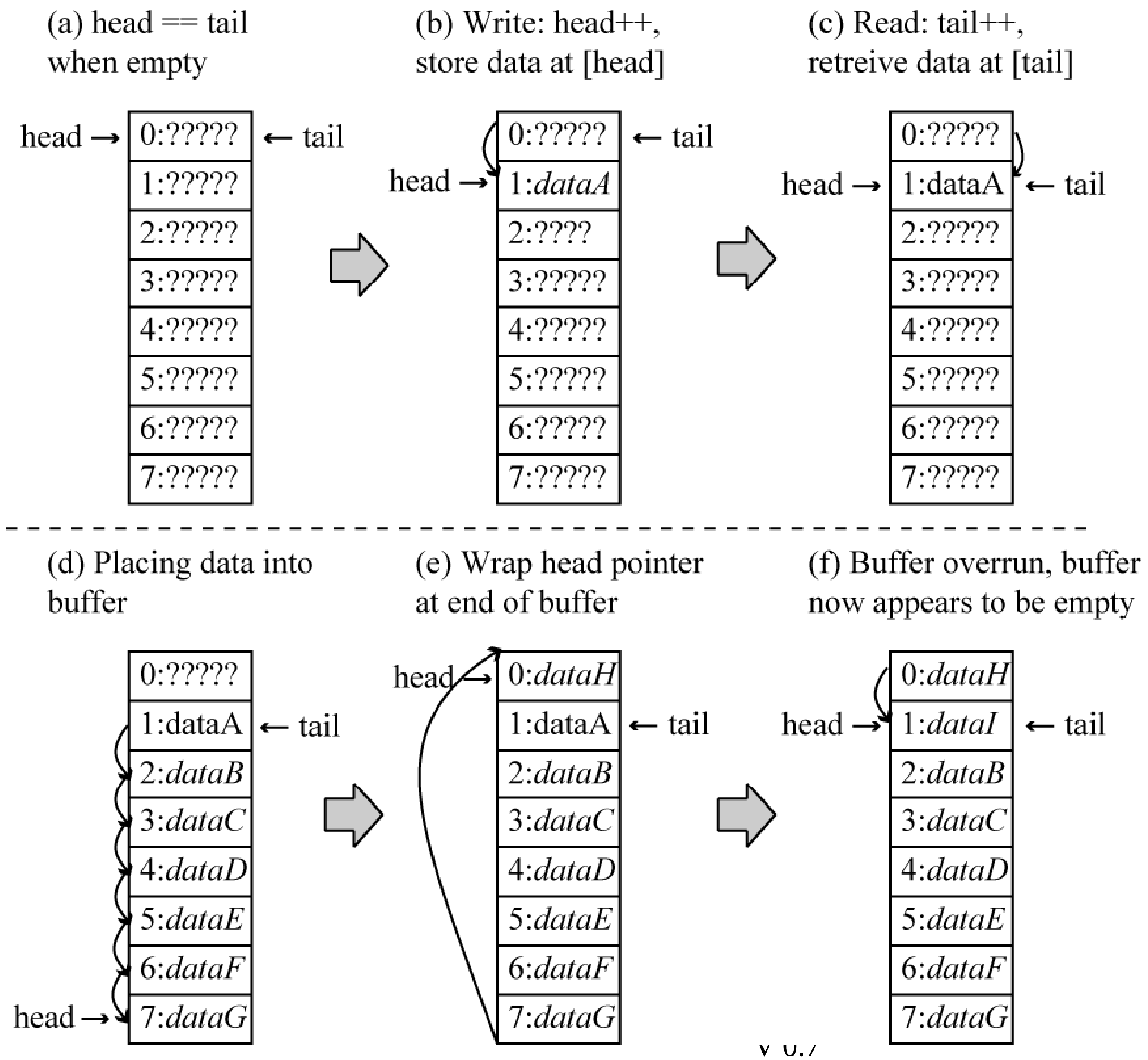
```
reverse_string.c, built on Jun 11 2008
!gnirts a si sihT
olleH
zywvutsrqponmljihgfedcba9876543ÿ
Reset cause: Software Reset.
Error trapped: UART overrun error
```

(b) Manually typed strings, no overrun.

(c) String is copied from paste buffer, so data arrives at a fast rate and causes UART receiver overrun.

V 0.7

Software FIFO Operation



Use a software FIFO to buffer input data read by UART1 receive ISR.

Software FIFO Implementation (UART1 Receive)

```
#ifdef UART1_RX_INTERRUPT
//Interrupt driven RX
#ifndef UART1_RX_FIFO_SIZE
#define UART1_RX_FIFO_SIZE 32  //choose a size
#endif

#ifndef UART1_RX_INTERRUPT_PRIORITY
#define UART1_RX_INTERRUPT_PRIORITY 1
#endif

volatile uint8 au8_rxFifo1[UART1_RX_FIFO_SIZE];
volatile uint16 u16_rxFifo1Head = 0;
volatile uint16 u16_rxFifo1Tail = 0;

uint8 inChar1(void) {
  while (u16_rxFifo1Head == u16_rxFifo1Tail) {   //wait for character
    doHeartbeat();
  } ;
  u16_rxFifo1Tail++;
  if (u16_rxFifo1Tail == UART1_RX_FIFO_SIZE) u16_rxFifo1Tail=0; //wrap
  return au8_rxFifo1[u16_rxFifo1Tail];  //return the character
}
```

(a) Defined by user in MPLAB project to select polled or interrupt driven RX.

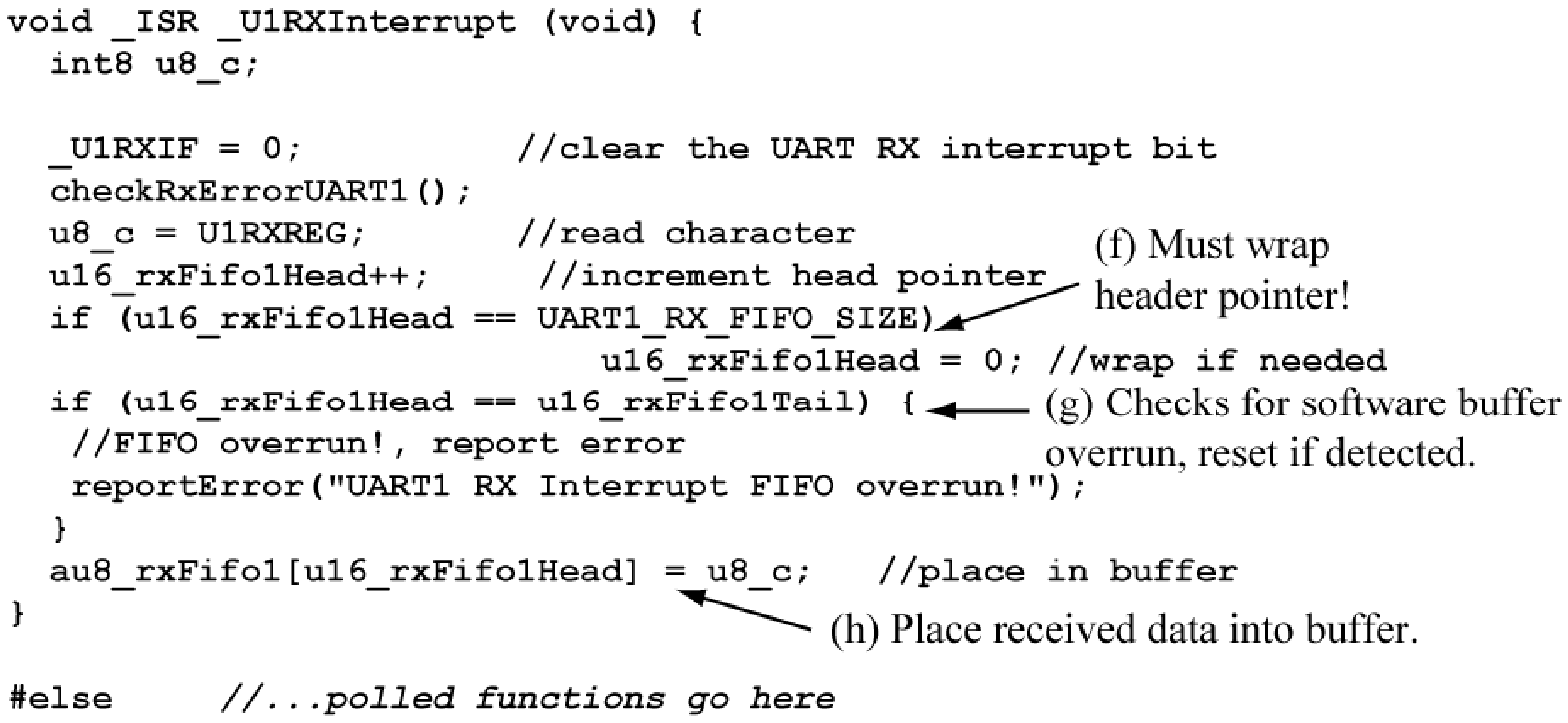
(b) Default FIFO buffer size and interrupt priority, can be overridden by user.

(c) Wait until data is available in buffer, which is placed there by the ISR.

(d) Must wrap tail pointer!

(e) Return the data from the buffer.

Software FIFO Implementation (UART1 Receive) (cont.)



```
void _ISR _U1RXInterrupt (void) {
  int8 u8_c;

  _U1RXIF = 0;            //clear the UART RX interrupt bit
  checkRxErrorUART1();
  u8_c = U1RXREG;         //read character
  u16_rxFifo1Head++;       //increment head pointer
  if (u16_rxFifo1Head == UART1_RX_FIFO_SIZE)
                                u16_rxFifo1Head = 0; //wrap if needed
  if (u16_rxFifo1Head == u16_rxFifo1Tail) {
   //FIFO overrun!, report error
   reportError("UART1 RX Interrupt FIFO overrun!");
  }
  au8_rxFifo1[u16_rxFifo1Head] = u8_c;   //place in buffer
}

#else     //...polled functions go here
```

(f) Must wrap header pointer!
(g) Checks for software buffer overrun, reset if detected.
(h) Place received data into buffer.

The **_U1RXInterrupt** ISR places received data into the software FIFO, the **inChar1()** function takes data out of the software FIFO. The software FIFO can overrun, signal an error when that happens!

Software FIFO Implementation (UART1 Transmit)

```
#ifdef UART1_TX_INTERRUPT
//Interrupt driven TX
#ifndef UART1_TX_FIFO_SIZE
#define UART1_TX_FIFO_SIZE 32  //choose a size
#endif

#ifndef UART1_TX_INTERRUPT_PRIORITY
#define UART1_TX_INTERRUPT_PRIORITY 1
#endif

volatile uint8 au8_txFifo1[UART1_TX_FIFO_SIZE];
volatile uint16 u16_txFifo1Head = 0;
volatile uint16 u16_txFifo1Tail = 0;

void outChar1(uint8 u8_c) {
  uint16 u16_tmp;
  u16_tmp = u16_txFifo1Head;
  u16_tmp++;
  if (u16_tmp == UART1_TX_FIFO_SIZE) u16_tmp = 0; //wrap if needed
  while (u16_tmp == u16_txFifo1Tail){
   doHeartbeat();
  }

  au8_txFifo1[u16_tmp] = u8_c; //write to buffer
  u16_txFifo1Head = u16_tmp;    //update head
  _U1TXIE = 1;                 //enable interrupt
}
```

(a) Defined by user in MPLAB project to select polled or interrupt driven TX

(b) Default FIFO buffer size and interrupt priority, can be overridden by user.

(c) Copy head pointer to temporary as we do not want to modify the head pointer value examined by the background ISR.

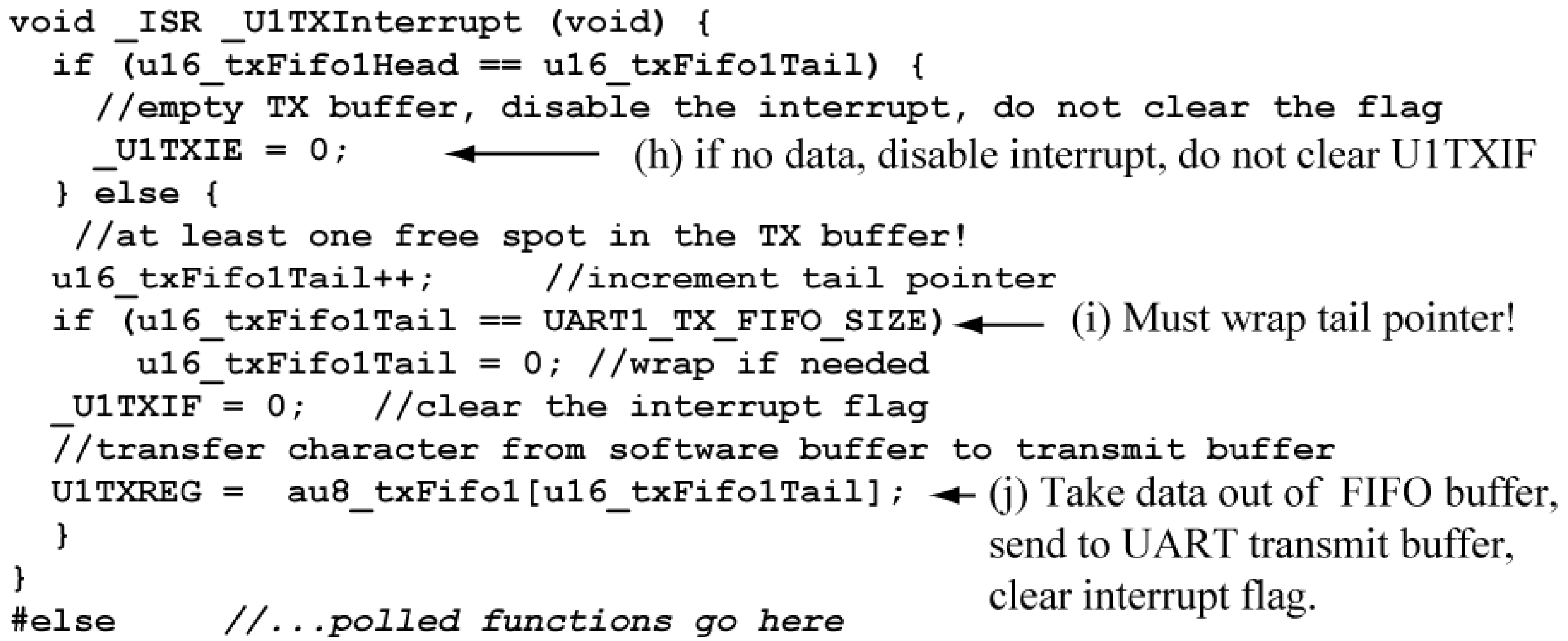
(d) Must wrap head pointer!

(e) Wait until space available in transmit buffer

(f) Place data in buffer

(g) Enable TX interrupt so that ISR can take data out.

Software FIFO Implementation (UART1 Transmit)(cont.)



```
void _ISR _U1TXInterrupt (void) {
  if (u16_txFifo1Head == u16_txFifo1Tail) {
    //empty TX buffer, disable the interrupt, do not clear the flag
    _U1TXIE = 0;
  } else {
   //at least one free spot in the TX buffer!
  u16_txFifo1Tail++;     //increment tail pointer
  if (u16_txFifo1Tail == UART1_TX_FIFO_SIZE)
      u16_txFifo1Tail = 0; //wrap if needed
  _U1TXIF = 0;   //clear the interrupt flag
  //transfer character from software buffer to transmit buffer
  U1TXREG =  au8_txFifo1[u16_txFifo1Tail];
  }
}
#else     //...polled functions go here
```

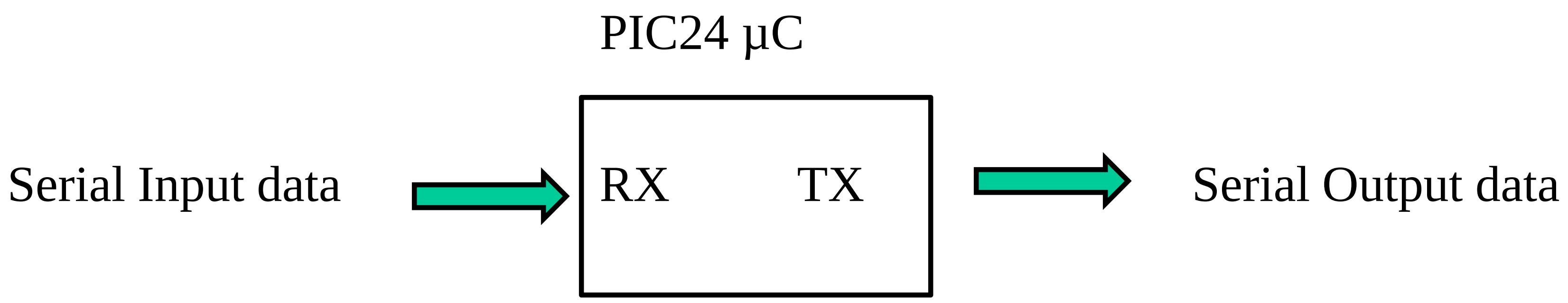
(h) if no data, disable interrupt, do not clear U1TXIF

(i) Must wrap tail pointer!

(j) Take data out of FIFO buffer, send to UART transmit buffer, clear interrupt flag.

The _**U1TXInterrupt** ISR takes data out of the software FIFO, the **outChar1()** function places data into the software FIFO. The _**U1TXInterrupt** ISR is enabled by **outChar1()** function when data placed into the software FIFO, and the ISR is triggerred by _U1TXIE if free spots in hardware transmit FIFO. The _**U1TXInterrupt** ISR disables itself if nothing left in the transmit FIFO.

When does an input Software FIFO help?



RX, TX operate at same baud rate. If one output character is sent for each input character (like in 'reverse string' operation), then the INPUT BANDWIDTH is the same as the OUTPUT BANDWIDTH.

The processing time is NON-ZERO, so we have:

INPUT bandwidth +   processing time > OUTPUT bandwidth

This means the software FIFO can help in **bursty** conditions (bursts of data arrive). Average input data rate, since data is not arriving in continuous stream, will be less than processing time + output data rate.

If data is arriving in a CONTINOUS STREAM, no amount of buffering will prevent FIFO overflow if

input bandwidth + processing time > Output bandwidth!!!!

Software Input FIFOs are Generic!

Software Input FIFOs are generic; they can be used to buffer any type of input events, such as:

Keypad presses

Pulse width measurement data

Values read from an analog-to-digital converter

Software FIFOs are a useful mechanism for data buffering.

What do you have to know?

- Difference between async/sync serial IO
- Format of async serial IO frames
- Details of PIC24 UART 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
- Software FIFO operation
- PIC24 interrupt driven UART1 Receive and Transmit operation