Serial communications software

UARTS (USARTS) typically have an 8-Bit parallel interface to microprocessors. The processor writes to the UART's transmit register and reads from its receive register. A status register can be used to flag conditions such as:

- Transmitter busy (or available)
- A byte has arrived and is ready for reading.
- An error has occurred.

Control registers can be used to:

- Set the data rate (baud rate)
- Set the transmit word format
- Configure parity
- Configure handshaking

Sending a message to a remote host might typically take the following form:

```
void sendMessage(char *Message) {
        while (*Message) {
                while (TransmitterBusy()); // wait for transmitter to become available
                sendByte(*Message); // write the byte
                Message++; // move on to next byte in message
        }
}
```

The lower level functions sendByte and TransmitterBusy are architecture specific and may simply involve the reading/writing of a memory mapped UART register.

Receiving a message may take the following form:

```
void readMessage(char *Message,int MsgLen) {
        int ByteCounter = 0;
        while (ByteCounter < MsgLen -1) {
                while(ReceiverReady()==0);  // wait a byte to come
                Message[ByteCounter] = readByte();
                ByteCounter++;
        }
        Message[ByteCounter] = 0; // terminate the message string with NULL
}
```

(These functions are only indicative of the functions that may be used in reality).

In the case of both sendMessage and readMessage, the CPU loops waiting for the transmitter and receiver to be ready. Apart from being a waste of CPU time (and perhaps battery life as a result), this approach has a more significant problem. It may lock up the CPU. Consider the readMessage function. It keeps looping until a message of a particular (expected) length is received. What if the last byte of this message is never sent or lost in transit? The processor would be locked in this routing. A time-out could be arranged of course but this would again involve a lengthy waiting period for the CPU.

As a result of the poor use of CPU time, this “polling” of UART status is not particularly suitable for practical use. A more efficient approach involves the use of interrupts.

UART's are often connected to the interrupt subsystem of microprocessors and microcontrollers. Interrupts can be generated when the transmitter is idle or a byte has been received (or indeed when some error has occurred). Using interrupts, the serial communications software can quietly build and send messages in the background while the CPU spends its time either asleep (saving power) or performing other operations.

Interrupts driven serial communications software must buffer incoming data and outgoing data. Such buffering is commonly performed using circular buffers. The code below shows how such buffering might be performed in a generic manner.

Note the way interrupts are disabled and enabled at certain points. The reason for this is that the processor may be interrupt while it is updating the Head, Tail or count variables in the transmit or receive buffers. This could potentially cause problems if an interrupts were to occur while the buffers are being updated as part of a WriteCom or ReadCom operation. The areas during which interrupts are disabled are commonly referred to as critical sections. It is good practice to minimize the extent of critical sections as the processor will not respond to interrupts while executing code within them.

```
#define MAXBUFFER   64
typedef struct tagComBuffer{
    unsigned char Buffer[MAXBUFFER];
    unsigned Head,Tail;
    unsigned Count;
} ComBuffer;

ComBuffer ComRXBuffer,ComTXBuffer;
int PutBuf(ComBuffer  *Buf,unsigned char Data);
unsigned char GetBuf(ComBuffer  *Buf);
unsigned GetBufCount(ComBuffer  *Buf);
unsigned ComOpen;
unsigned ComError;
unsigned ComBusy;

void enableInterrupts() {
// perform platorm specific enabling of interrupts
}
void disableInterrupts() {
// perform platorm specific disabling of interrupts
}
int TransmitterBusy() {
// perform platorm specific check to see if transmitter is busy
// return 0 if free.
}
void sendByte(unsigned char theByte) {
// perform platorm specific write to UART transmitter register
}
unsigned char readByte() {
// perform platorm specific read of UART receiver register
}
void initUART() {
// perform platorm specific initialization
}
void uart_rx_handler (void)
{
// interrupt handler for UART receiver
  if (PutBuf(&ComRXBuffer,readByte()) )
    ComError = 1; // if PutBuf returns a non-zero value then there is an error
}

void usart_tx_handler (void)
{
  if (GetBufCount(&ComTXBuffer))
            sendByte(GetBuf(&ComTXBuffer));
}

extern int ReadCom(int Max,unsigned char *Buffer)
{
  unsigned i;
  if (!ComOpen)
    return (-1);
  i=0;
```

```
  while ((i < Max-1) && (GetBufCount(&ComRXBuffer)))
    Buffer[i++] = GetBuf(&ComRXBuffer);
  if (i>0)
  {
    Buffer[i]=0;
    return(i);
  }
  else {
    return(0);
  }
};
extern int WriteCom(int Count,unsigned char *Buffer)
{
  unsigned i,BufLen;
  if (!ComOpen)
    return (-1);
  if (!Buffer[0]) // blank string?
    return(0);
  BufLen = GetBufCount(&ComTXBuffer);
  for(i=0;i<Count;i++)
    PutBuf(&ComTXBuffer,Buffer[i]);
  // might have to kick-start interrupt driven comms if the UART is idle
  if ( (!BufLen) && !(TransmitterBusy()) )
        sendByte(GetBuf(&ComTXBuffer));
  return 0;
};

int PutBuf(ComBuffer *Buf,unsigned char Data)
{
    if ( (Buf->Head==Buf->Tail) && (Buf->Count!=0))
                  return(1);  /* OverFlow */
    disableInterrupts();
    Buf->Buffer[Buf->Head++] = Data;
    Buf->Count++;
    if (Buf->Head==MAXBUFFER)
                  Buf->Head=0;
    enableInterrupts();
    return(0);
};
unsigned char GetBuf(ComBuffer *Buf)
{
    unsigned char Data;
    if ( Buf->Count==0 )
                  return (0);
            disableInterrupts();
    Data = Buf->Buffer[Buf->Tail++];
    if (Buf->Tail == MAXBUFFER)
                  Buf->Tail = 0;
    Buf->Count--;
            enableInterrupts();
    return (Data);
}
unsigned int GetBufCount(ComBuffer *Buf)
{
```

```
    return Buf->Count;
};

void main(void)
{
  char rxdata;
  initUART();
  WriteCom(12,"Hello World!");
  while(1) {
    if (GetBufCount(&ComRXBuffer) ){
      rxdata = GetBuf(&ComRXBuffer); // read what the users sends
      WriteCom(11,"You sent : ");
      WriteCom(1, &rxdata); // echo it back
      rxdata = 0x0d;
      WriteCom(1, &rxdata); // send carriage return
      rxdata = 0x0a;
      WriteCom(1, &rxdata); // send line feed

    }
  }
}
```