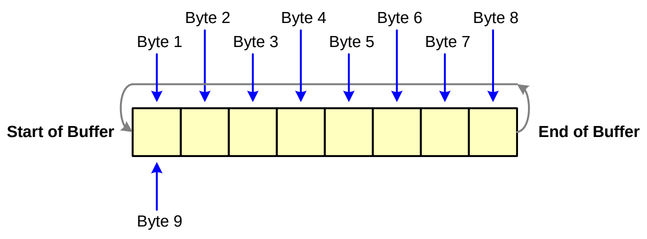
Amicus18 Interrupt Driven Buffered Software UART

Buffering of serial data is normally in the realm of a dedicated hardware UART peripheral, and indeed, there is sample code for the Amicus18 that does just this job. However, the Amicus18 microcontroller has a single UART peripheral, and this is normally used for communicating with the PC, and is the primary means of programming the microcontroller through its bootloader. It's only really available when the Amicus18 board is being used as an autonomous entity. i.e. not connected via the USB cable.

But even if we can't use the dedicated hardware UART, we can still enjoy the benefits of a buffered serial receiver that operates in the background while we get on with the main program in the foreground, thanks to the use of an interrupt.

Why would we require buffered serial data?

Asynchronous serial data relies on timing and not on a clock pulse, therefore it can be received without notification. Without buffering the incoming data, the program would need to know exactly when data was to be received, and wait in a tight loop until the data arrived. This is desirable in some, if not most programs, however, there is always the chance that while the microcontroller is performing tasks based upon the serial data received, the next lot of data is missed. A buffered receiver alleviates this by constantly looking for serial data transmissions in the background and storing what it receives in a section of RAM, ready for examining by the main program. The length of the section of RAM is finite, and is looped back upon itself when its length is reached, thus forming a ring buffer. This stops all the RAM from being used up by data. The general idea is shown below:



The above scenario is relatively simple when a hardware UART peripheral is being used, as the peripheral itself performs the task of receiving the serial data. However, this is further complicated in so much as we're using a software based UART which means we must capture the serial data ourselves bit by bit, and form the correct sequence into a byte before storing it in the buffer. Moreover, each bit must be captured within a certain time frame known as its baud rate or bit rate. And we can't stop the program from running while capturing and storing serial data. This is definately a job for an interrupt.

The interrupt in question is our good friend the timer overflow interrupt. This is an event that occurs when one of the timer peripherals overflows from either 255 to 0 for an 8-bit type, or 65535 to 0 for a 16-bit type. This is easily setup using one of the built in timer macros. Remember, once a timer peripheral is enabled, it will continue ticking away in the background every nth clock cycle of the microcontroller. The amount of clock cycles per timer tick is adjusted using a prescaler, normally in multiples of 2. i.e. 2, 4, 8, 16 etc. See the compiler's manual for a breakdown of the timer macros.

In this article, we're not going to go into any detail of timers, or indeed interrupts, i'll leave that for later articles. However, there is a wealth of information on the internet concerning both topics. Instead, we're going to concentrate on how to setup and use the buffered software serial interface.

The serial interface has been created as a self contained file that may be included into the main program, and replaces the compiler's Rsout and Rsin commands. You can download the Buffered Rsin.inc file from here:-

Amicus18 Interrupt Driven Buffered Software UART

Place the downloaded folder into the compiler's samples folder. The default location of this is "C:\Documents and Settings\User's name\Amicus\Samples" for a Windows XP OS, and "C:\Users\User's name\Amicus\Samples" for a Windows Vista OS.

Don't forget to change the "User's name" text to whatever user name you have given to the PC.

Once the include file is placed within the main program, it will setup the interrupt and manipulate it's variables ready for use:

```
' Interrupt driven software serial UART Demo
'
' The size of the Rsin Buffer (max 255) (must be issued before the "Buffered RSIn" include)
$define Rsin_BufferSize 30
'
' Configure the Alphanumeric LCD for use with the LCD Shield
'
    Declare LCD_DTPin = PORTB.4                ' LCD's data port starts at bit-4 of PortB
    Declare LCD_ENPin = PORTB.3                ' LCD's EN pin connection
    Declare LCD_RSPin = PORTB.1                ' LCD's RS pin connection
    Declare LCD_Interface = 4                  ' LCD operating in 4-bit mode
    Declare LCD_Lines = 2                      ' LCD has 2 lines
    Declare LCD_Type = Alphanumeric            ' LCD type is alphanumeric (Hitachi)
    Declare LCD_CommandUs = 2000   ' Delay between command values sent to the LCD (in us)
    Declare LCD_DataUs = 50        ' Delay between data values sent to the LCD (in us)
'
' Create some variables for the demonstration
'
    Dim ByteVar As Byte                        ' Byte to be received and displayed
    Dim bLCD_Xpos As Byte                      ' LCD's X position
    Dim bLCD_Ypos As Byte                      ' LCD's Y position

'----------------------------------------------------------------------------------
    Include "Buffered Rsin.inc"      ' Load the software UART routines into the program
                                     ' Preferably after the user variables have been created
'----------------------------------------------------------------------------------
    GoTo Main                                  ' Jump over the subroutines
'----------------------------------------------------------------------------------
' Display a character on the LCD with boundary checks
' Input     : ByteVar holds the character to display
' Output    : None
' Notes     : Moves to alternate lines when the amount of character per line is exceeded
'
DisplayChar:
    If bLCD_Xpos > 16 Then                     ' Have we reached the end of the line?
        bLCD_Xpos = 1                          ' Yes. So reset the X position
        If bLCD_Ypos = 1 Then                  ' Are we on line 1 of the LCD?
            Inc bLCD_Ypos                      ' Yes. So move to line 2
        Else                                   ' Otherwise...
            Dec bLCD_Ypos                      ' Move to line 1
        EndIf
        Cursor bLCD_Ypos, bLCD_Xpos            ' Move the cursor
    EndIf
    Print ByteVar                              ' Display the character
    Inc bLCD_Xpos                              ' Increment the X position
    Return
'----------------------------------------------------------------------------------
' Main demo loop
Main:
    DelayMS 100                                ' Wait for the LCD to stabilise
    Cls                                        ' Clear the LCD
    RSOut "Type some characters in the terminal window\r",_
          "and they will appear on the LCD.\r",_
          "Note that no characters are missed\r",_
          "even though there is a stupidly large delay within the receiving loop.\r\r"
    SetTimeout(5000)                     ' Set the Rsin timeout to 5 seconds

    bLCD_Xpos = 1                        ' Reset the LCD cursor's X position
    bLCD_Ypos = 1                        ' Reset the LCD cursor's Y position
    While 1 = 1                          ' Create an infinite loop
        While 1 = 1                      ' Create another infinite loop
            'ByteVar = RSIn             ' Receive a byte from the UART (without timeout)
            ByteVar = RSIn,{TimedOut}  ' Receive a byte from the UART (with timeout)
            DelayMS 200         ' << Add a stupidly large delay between characters received
            GoSub DisplayChar            ' Display the byte on the LCD
        Wend                             ' Close the loop
TimedOut:  ' Come here if a timeout occured
        Cls                              ' Clear the LCD
        Print At 2,1,"Timed Out"         ' Display the timeout message on line 2 of the LCD
        DelayMS 400                      ' Clear the LCD
        Cls
        bLCD_Xpos = 1                    ' Reset the LCD cursor's X position
        bLCD_Ypos = 1                    ' Reset the LCD cursor's Y position
    Wend                                 ' Do it forever
```

The above program, although it looks complex, is a simple demonstration of the serial mechanism. It waits for characters typed into the Amicus18 serial terminal and displays them on an alphanumeric LCD such as the Amicus LCD shield, or the layout shown within the compiler's manual (*see the Print section*). If no characters are received within 5 second, a timeout message is displayed and the process of looking for characters is continued.

Notice that there is a very large, and normally impractical, delay of 200 ms between the reception of serial data and its display. This would normally be sufficient to miss virtually every single piece of data received from the serial interface, however, all the data is buffered, and what the Rsin command is really doing is dipping into the buffer and grabbing a previously received byte. This mechanism works well unless the amount of characters received fills the buffer before they are read. However, increasing the buffer size can alleviate this sort of problem to a certain extent.

Finer points of the interrupt driven software serial routines

The buffer can be increased or decreased in size by issuing $define Rsin_BufferSize. The maximum size is 255 bytes. The $define must be placed in the program before the "Buffered Rsin.inc" file is included.

A new psuedo command has been added to the serial mechanism that allows the timeout value of Rsin to be adjusted dynamically instead of only at compile time through a Declare. This is by the use of:

```
SetTimeout(Amount of Time)
```

The amount of time parameter is in milliseconds, with a maximum limit of 65535, and can be a constant or a variable. Once SetTimeout has been issued, the Rsin command will maintain the timeout until a new SetTimeout command is issued.

Another pseudo command will clear the serial buffer and reset the variables assigned to it. This is:

```
ClearSerialBuffer()
```

Unlike the standard Rsout and Rsin commands, the baud rate for each must remain the same, and only non-inverted serial is supported. However, this is very straightforward to modify. The standard Rsout and Rsin declares are also used by the new routines.

It must be remembered that there is an interrupt occuring in the background, which means that the foreground program will operate a little slower than usual. This will not be noticable in most programs, unless time critical command are used. i.e. DelayUs, DelayMs, Serin, Serout etc...

A higher Baud rate will increase the frequency of the interrupt, thus slowing down the foreground even further. The maximum Baud rate is 38400.