Keywords: DS5230, DS5250, DS80C310, DS80C320, DS87C520, DS87C530, DS80C390, DS80C400, DS89C430, DS89C450, high-speed microcontroller, ultra high-speed microcontroller, UART, RS-232, serial port, dual data pointer, FIFO, example co

APPLICATION NOTE 603

Implementing a Serial Port FIFO Using Dual Data Pointers

By: Kevin Self
Mar 12, 2002

Abstract: The Dallas Semiconductor high-speed microcontroller family allows the system designer to optimize serial communications via the internal UART. This application note demonstrates the use of a circular buffer utilizing the dual data pointer found in these enhanced 8051 microcontrollers. Example assembly code is provided to show the implementation of a simple 256-byte serial port circular buffer.

Introduction

The architecture of the original 8051 microprocessor included a standard universal synchronous/asynchronous receiver/transmitter (USART, more commonly known as UART). This peripheral allowed the device to communicate via an RS-232 interface at a variety of baud rates. One of the perceived shortcomings of the 8051 UART implementation is that its receive and transmit buffers are only one deep, i.e., software must retrieve a byte from the receive buffer before it is overridden by the next received character. It is possible to implement a fast software FIFO in many Dallas Semiconductor and Maxim microcontrollers, increasing the utility of the serial ports.

This application note demonstrates a simple 256-byte circular buffer, but the principle can be extended to buffers up to 64kB in length. Data is received by the serial port and stored in the buffer until acted upon by a user-supplied routine. When the buffer becomes full, the receiver, via software flow control, will signal the host to halt transmissions. When the user-supplied routine has cleared space in the buffer it will signal the host to resume transmissions. The example in this application note is generic enough to be adapted to a wide variety of user applications. Assembly source code to accompany the example is provided on the website in the file AN603_SW.A51.

Hardware Enhancements That Enable Fifo Construction

The High-Speed and Ultra-High-Speed Microcontroller families have many features that greatly simplify the implementation of a software FIFO. All of these features seek to minimize the software overhead associated with storing and retrieving the data. First among the features is the extremely high-speed nature of the Dallas Semiconductor and Maxim microcontrollers. Their enhanced 4 clock and 1 clockper-machine cycle cores minimize the time spent servicing the buffer, and by extension allow the main application to run faster.

Another significant feature is the dual data pointers. The original 8051 only had one data pointer, which made it difficult to simultaneously implement input and output pointers. Each time the system needed to switch between pointers it would have to save off the current value and load the value of the other pointer. This introduces a significant delay when attempting to quickly move data in and out of the buffer. With two data pointers one can be assigned to the input (insert) pointer and the other to the output (remove) pointer of the buffer, eliminating the delay associated with juggling the pointers. In addition, some microcontrollers incorporate enhanced data pointers that can automatically increment the data pointer following the execution of certain data pointer-related instructions. This saves additional machine cycles per buffer access.

Speed can be further increased by the use, when available, of the internal 1kB MOVX SRAM whose data can be accessed in a single machine cycle. Locating the circular buffer in this memory allows the implementation of a very fast buffer.

Software Description

This example describes the basic architecture of a circular FIFO buffer for a serial port. Data is received through serial port 0 and stored in a circular buffer in MOVX memory, where it remains until extracted and manipulated by a user-defined algorithm. This example uses the Dallas and Maxim microcontroller dual data pointers. DPTR (also known as DPTR0) is used as the input pointer, and always designates where new information received through the serial port should be stored. DPTR1 is the output pointer, showing where the application software will retrieve the next unprocessed byte. The accompanying software should be considered a skeleton of such an example.

A few assumptions were made that simplified the software and reduced its size:

1. The buffer length is 256 bytes, and begins on 0000h and ends on 00FFh. Software can quickly check for a nonzero data pointer high byte to indicate a rollover.
2. The input and output pointers only increment, although the software could be easily modified to support bidirectional pointers.
3. Feedback on the buffer status following reception of a character is restricted to a 1-byte remaining warning and 0-byte remaining shutdown approach. This go/no-go approach allows for very fast error checking, which minimizes the time spent in the buffer input and output routines. Buffer status following reading a character from the buffer is the responsibility of the user-supplied algorithm.
4. Error notification with the host is performed via software flow control. This is a common transmission protocol that is simple to implement and takes little processor overhead.

When the program starts, both data pointers are initialized to the beginning of the buffer. The serial port is also initialized, following which the software flow control 'ready' character (XON: 11h) is sent to the host to allow transmission of data. The software then waits until either a new character is received, or the application needs to read a byte from the buffer. This example only shows a shell of the routine that reads the byte from the buffer; the actual code would be defined by the application.

When the serial port receives a character, the serial port 0 interrupt service routine (ISR) is called. First, the routine retrieves the byte from the serial port and stores in the buffer at the location indicated by the input pointer (DPTR). Then the subroutine Increment_DPTx is called, which increments the selected data pointer and then performs two operations. First, it ensures that the incremented value has not exceeded the upper limit of the buffer. If so, it resets the pointer value to the beginning of the buffer. Second, it checks to see if the input pointer is in danger of overwriting the output pointer, a situation which could be encountered if the host was not emptying the buffer fast enough.

Overflow Detection And Handling

The overflow detection and handling routine is shown in Figure 1. It is called after either the input or output pointers are incremented. First, the overflow routine determines the distance between the input and output pointers. If the distance is one byte, then the software declares an overflow warning. This sets the overflow flag and also transmits an XOFF to the other unit to stop the transmission until the buffer empties again. The serial port receiver is left enabled to allow for the possibility that the sending unit was in the middle of a transmission when the XOFF was received. Declaring the warning state one byte shy of the buffer-full state allows for this last character to be received.

If the distance is zero bytes then the software declares an overflow shutdown. This sets the overflow flag, transmits an XOFF, and also disables the serial port receiver to prevent corruption of the data in the buffer. The only way to exit the overflow warning or shutdown states is for the buffer output routine to remove data until the distance increases to two.

If the distance is not one byte and not zero bytes then no overflow was detected. The routine then checks the state of the overflow flag. If the flag is not set then the routine exits. If the overflow flag was set, then the current state indicates that the buffer has just transitioned from an overflow warning state to no warning. If this is the case then the buffer is ready to begin receiving data again, so the routine clears the overflow flag, re-enables the serial port receiver, and transmits an XON character.

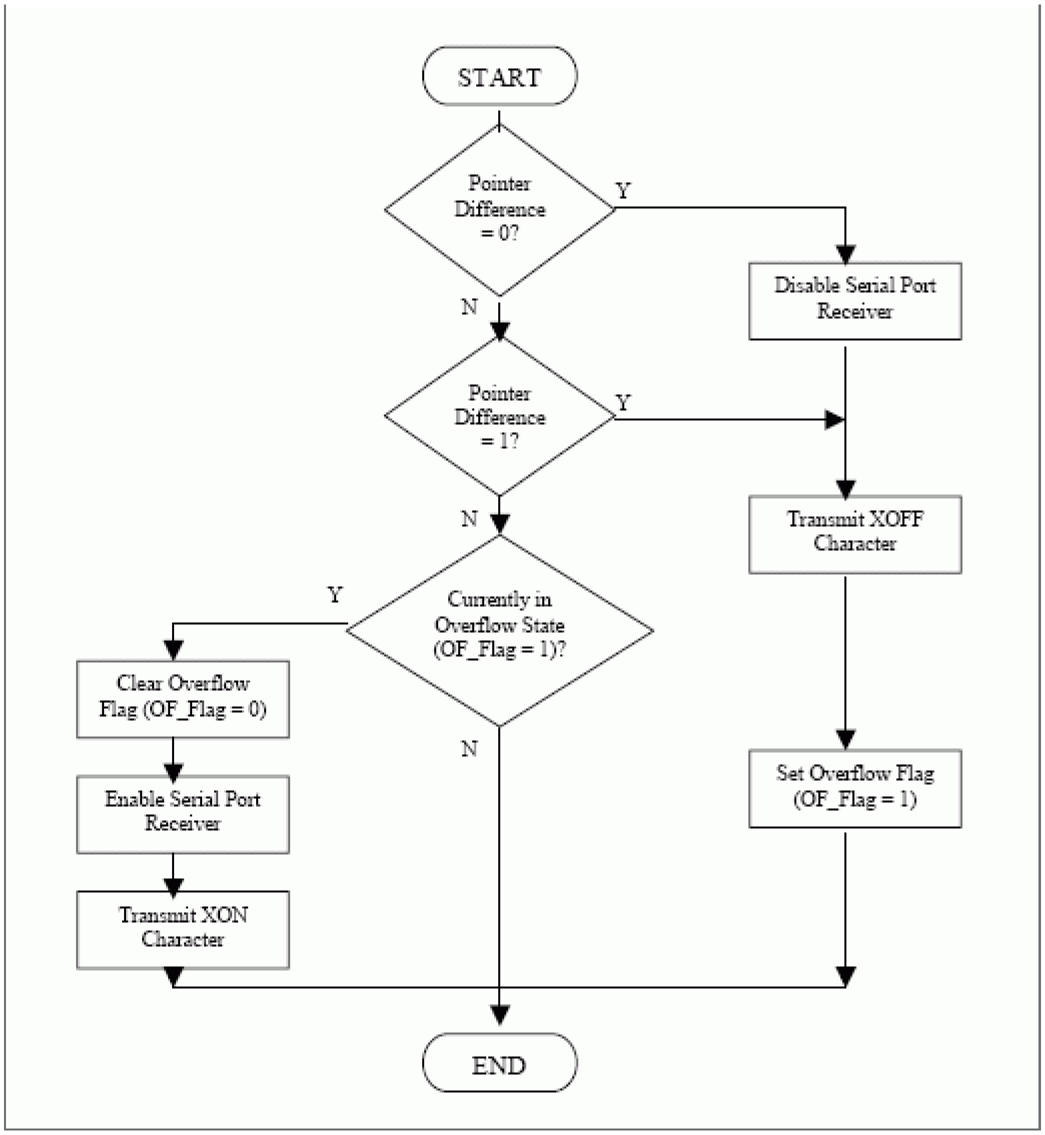
START
Pointer Difference = 0?
Y
N
Disable Serial Port Receiver
Pointer Difference = 1?
Y
N
Transmit XOFF Character
Currently in Overflow State (OF_Flag = 1)?
Y
N
Clear Overflow Flag (OF_Flag = 0)
Enable Serial Port Receiver
Transmit XON Character
Set Overflow Flag (OF_Flag = 1)
END

Figure 1. Overflow Handler Flowchart.

Example Assembly Code

The following assembly language code example was compiled and simulated using the Keil Software µVision2 Integrated Development Environment. The header file reg320.inc is a file containing the names of registers and bits specific to the target microcontroller (in this case the DS80C320), and a similar file will usually be supplied with your assembler for use with the specific microcontroller in use.

```
;************************************
;
; AN603_SW.A51
; This software shows the implementation of a simple 256-byte serial port
; circular buffer to accompany Application Note 603. An overview is provided
; in the application note text.
;************************************
;
; If using a device that supports auto DPTR increment feature:
; 1) Modify DPS register as needed to activate auto DPTR increment feature.
; Place the instruction in the 'Init buffers' section.
; 2) Comment out all INC DPTR instructions.
;************************************
;
```

```
; If using a device that incorporates internal MOVX SRAM:
; 1) Modify PMR register as needed to activate internal MOVX SRAM.
; 2) Place the instruction in the 'Init buffers' section or
; compiler startup section.
;************************************
$include(reg320.inc) ;Include file with 320 SFR register and bit extentions
BufferStart EQU 0000h ;This example uses a 256 byte buffer
BufferEnd EQU 00FFh ;
XON EQU 011h ;Flow control start character
XOFF EQU 013h ;Flow control stop character
OF_Flag EQU FL ;Buffer overflow condition flag. May be changed to any
; bit-addressable location.
Buf_count EQU R7 ;Number of elements in buffer. May be changed to any
; register.
;*** Interrupt Vectors
org 0h ;Reset Vector
ljmp start
;***************
org 23h ;Serial Port 0 Vector
ljmp SerialPort0ISR
;***************
org 100h
;***************
;Start of main loop. This initializes the pointers, serial port, and
; would also contain the user code that pulls data out of buffer
; and manipulates it.
;***************
start:
;Init buffers
MOV DPTR, #Bufferstart ;Initialize the Input pointer
INC DPS ;Initialize the Output Pointer
MOV DPTR, #Bufferstart ;
MOV Buf_Count, #0h ;Initialize buffer data counter
;Init Serial Port 0 for mode 1, 9600 baud
MOV TMOD, #020h ;Timer 1 mode 2
MOV TH1, #0FDh ;9600 baud at 11.059 MHz
MOV SCON0, #050h ;SP0 Mode 1 with receive enabled
SETB TR1 ;Start T1 to enable serial port
ORL IE, #090h ;Enable Serial Port 0 and global interrupts
MOV SBUF0, #XON ;Signal to host that we're ready
;Demonstration routine to remove data from the buffer. This is a rough
; example and the final code will be highly customized by the user.
MOV A, Buf_Count ;If empty, there's no data to retrieve.
JZ Buf_Empty
Buf_Not_Empty:
ORL DPS, #01h ;DPTR1(Output pointer), pull char out of buffer,
MOVX A, @DPTR ; and increment the output pointer.
DEC Buf_Count ;Decrement counter to show one less char in buffer
CALL Increment_DPTx
Buf_Empty:
;End demonstration routine
sjmp $
;*************
;Serial Port 0 Interrupt Service Routine - This routine receives data via
; the serial port and places it in the buffer at the location indicated by
; DPTR0. It then calls the routine to increment and correct (if necessary)
; the data pointer. If called by the transmit interrupt, it simply clears
; the TI flag and returns.
;*************
SerialPort0ISR:
JB RI, RECEIVE_INT
TRANSMIT_INT: ;Transmitter called the interrupt.
CLR TI ;Transmit interrupt only called following XON/XOFF
RETI ; transmission. Just clear the flag and exit.
RECEIVE_INT: ;Receiver called the interrupt.
PUSH ACC
ANL DPS, #0FEh ;Clear SEL to select DPTR0
MOV A, SBUF0 ;Get new character and store in buffer
MOVX @DPTR, A
INC Buf Count ;Increment counter to show one more char in buffer
```

```
CALL Increment_DPTx
CLR RI
POP ACC
RETI ;End of Interrupt
;*************
;Increment Data Pointer. Corrects for incrementing past end of buffer and also
; checks pointer positions to make sure an overflow hasn't occurred. If
overflow
; has occurred, it triggers an 'overflow warning' or 'overflow shutdown' as
; appropriate. Note that SEL must be configured prior to calling this routine
; so that the correct data pointer gets modified.
;*************
Increment_DPTx:
PUSH ACC
INC DPTR ;First, go ahead and inc the pointer
ROLLOVER_CHECK: ;Now, check for a rollover past BufferEnd
MOV A, DPS ;Need separate routines for DPTR and DPTR1
JB ACC.0, Check_DPTR1
Check_DPTR0:
MOV A, DPH ;If DPH<>0, then rollover ocurred
JZ Calc_Dist_DPTR1_to_DPTR0 ;If this routine returns with carry set
JMP ROLLOVER ;DPTR0 now exceeds BufferEnd. Correct by
Check_DPTR1:
MOV A, DPH1 ;If DPH1<>0, then rollover ocurred
JZ Calc_Dist_DPTR1_to_DPTR0 ;If this routine returns with carry set
ROLLOVER: ;Active DPTR now exceeds BufferEnd. Correct
MOV DPTR, #BufferStart ; by resetting pointer to bufferstart.
; Fall through to Calc_Dist_DPTR1_to_DPTR0
;*************
;Calculate distance between pointers to check for a buffer overflow.
; Based on the difference, there are 4 cases:
; 1) If difference=1 or -1, then declare warning, set flag, send XOFF.
; 2) If difference=0, then we received a character after XOFF was sent.
; Declare shutdown, set OF_Flag =1, send XOFF, and disable receiver.
; 3) If not case 1 or 2 and OF_Flag =1, previous overflow has cleared, so
send
; XON, reenable receiver, and clear OF_Flag =0.
; 4) If not case 1, 2, or 3, then do nothing and exit.
;*************
Calc_Dist_DPTR1_to_DPTR0:
CLR C ;Check the distance between pointers.
MOV A, DPL1 ; If DPL1-DPL =1 or -1, declare overflow warning
SUBB A, DPL ; If DPL1-DPL =0, declare overflow shutdown as
JZ OVERFLOW_HANDLER ; we received an extra byte while in the
CJNE A, #01h, NO_OVERFLOW ; warning condition.
OVERFLOW_HANDLER: ;Character received while distance is 0 or 1
JNB OF_Flag, OVERFLOW_WARNING ; bytes. If 1 byte distance (warning), then
; send XOFF to host but leave receiver enabled in
OVERFLOW_SHUTDOWN: ; case host is currently transmitting. If 0 bytes
CLR REN ; distance (shutdown) then do same but disable
OVERFLOW_WARNING: ; receiver to prevent buffer corruption.
MOV SBUF0, #XOFF ; 0 or 1 byte distance determination is done by
SETB OF_Flag ; monitoring the overflow flag (OF_Flag).
JMP END_INC_DPTR
NO_OVERFLOW: ;No overflow detected. If overflow flag previously
JNB OF_Flag, END_INC_DPTR ; set, clear it, reenable serial port, and issue
MOV SBUF0, #XON ; XON character to host to restart communication.
SETB REN ; Otherwise there is nothing that needs to be done.
CLR OF_Flag
END_INC_DPTR:
POP ACC
RET
end
```

Related Parts		
DS5230	IP Security Microcontroller	
DS5250	High-Speed Secure Microcontroller	
DS80C310	High-Speed Microcontroller	Free Samples
DS80C320	High-Speed/Low-Power Microcontrollers	Free Samples
DS80C390	Dual CAN High-Speed Microprocessor	Free Samples
DS80C400	Network Microcontroller	Free Samples
DS87C520	EPROM/ROM High-Speed Microcontrollers	Free Samples
DS87C530	EPROM Microcontrollers with Real-Time Clock	Free Samples
DS89C430	Ultra-High-Speed Flash Microcontrollers	Free Samples
DS89C450	Ultra-High-Speed Flash Microcontrollers	Free Samples