Application Note

A Nonblocking UART for Z8 Encore! MCUs

AN034901-0813

Abstract

Many applications cannot use an interrupt-driven UART due to critical timing requirements within their interrupt routines, yet they may still require the functions of a UART. For example, if a system is monitoring the charging of a battery with a timer interrupt every 100ms, a UART interrupt can cause these timer interrupts to become lost, consequently impacting the monitor timing of the battery-charging operation. Previous solutions would typically require the main function to contain all noncritical code, such as user input or output through the UART. However, if checking for user input/output is *blocking* in function, then no other noncritical code will be executed within the main function until user input is received.

This document contains sample code files for initializing the UART and to manage a noninterrupt UART for nonblocking transmit and receive functionality. Both circular and linear buffer implementations are also introduced to facilitate the buffering of UART data streams.

Note: The source code file associated with this application note, AN0349-SC01.zip, is available free for download from the Zilog website. This source code has been tested with version 5.0.0 of ZDSII for Z8 Encore! XP MCUs. Subsequent releases of ZDSII may require you to modify the code supplied with this application note.

Features

This application offers the following features:

- Noninterrupt-driven UART configuration
- Nonblocking receive and transmit functions
- UART linear buffer handling with notification flag
- Circular buffer handing with state flag
- Z8F1680 MCU

Discussion

A Universal Asynchronous Receiver/Transmitter (UART) is a full-duplex communication channel capable of performing asynchronous data transfers. The UART uses a single 8-bit data mode with selectable parity. Features of the UART include:

- 8-bit asynchronous data transfer
- Selectable even/odd parity generation and checking
- Option of one or two stop bits

- Separate transmit and receive interrupts
- Separate transmit and receive enables
- Framing, parity, overrun and break detection
- 16-bit baud rate generators (BRG)
- Selectable MULTIPROCESSOR (9-Bit) Mode with three configurable interrupt schemes
- Baud Rate Generator Timer Mode
- Driver enable output for external bus transceivers UART

The UART consists of three primary functional blocks: transmitter, receiver, and baud rate generator. The UART's transmitter and receiver each function independently but use the same baud rate and data format. Figure 1 shows the UART architecture.

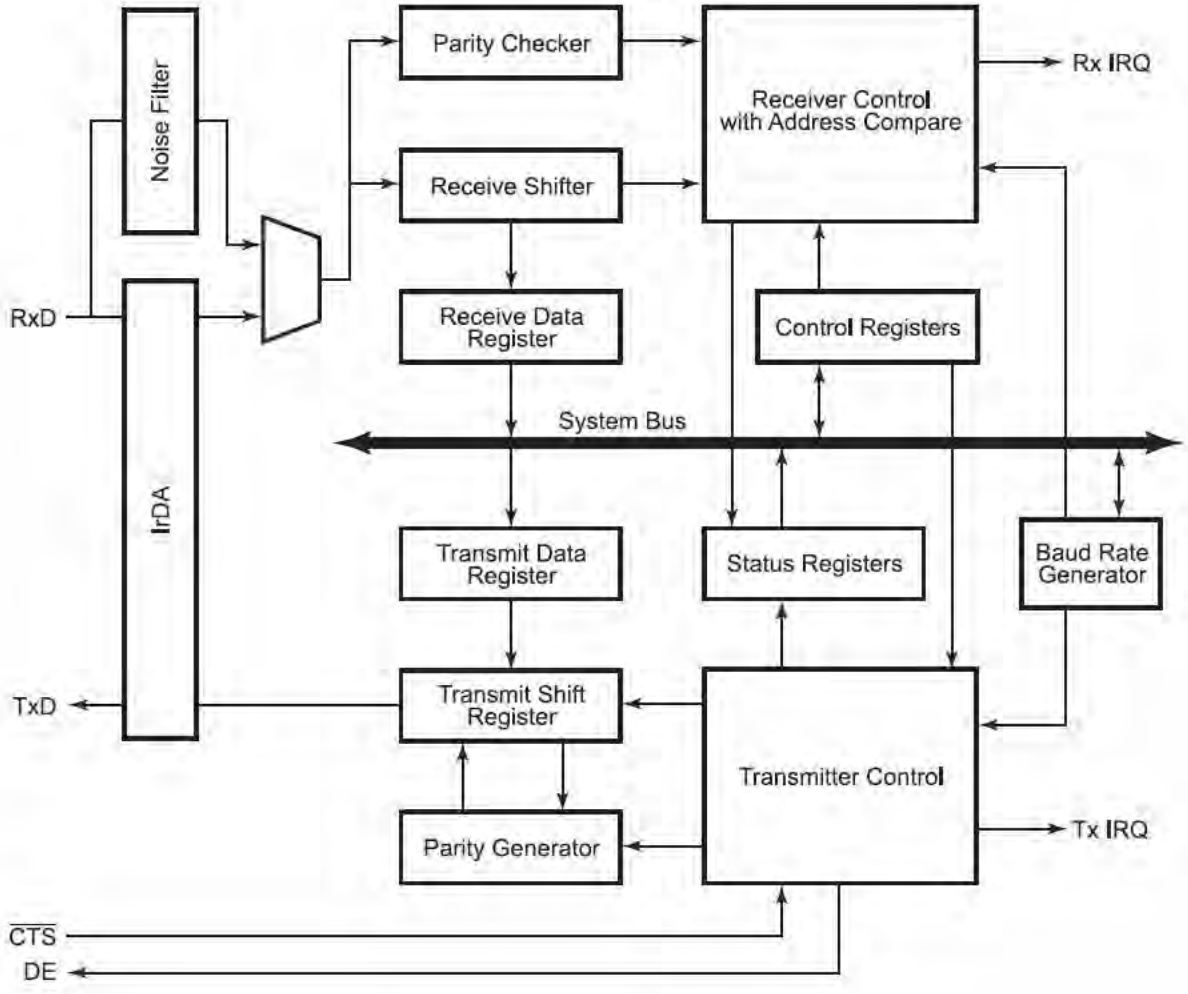


Figure 1. The UART Architecture of the Z8F1680 Series MCU

The Z8F1680 MCU's UART function is a full-duplex communication channel capable of handling asynchronous data transfers. A reliable UART communication is affected by two factors: system clock speed, and desired baud rate. Ensure that the UART baud rate errors never exceed 5%.

Note: To learn more about calculating UART baud rates for the Z8F1680 MCU, refer to the LIN-UART chapter of the Z8F1680 Product Specification (PS0250).

Noninterrupt and Nonblocking Functions

Noninterrupt functions are functions that do not use Interrupt Service Routines (ISR) and do not affect other routines' processing times. Nonblocking functions are functions that will execute if a flag is set but will otherwise exit.

A noninterrupting, nonblocking routine must be called repeatedly inside a function or loop. To implement a noninterrupting, nonblocking UART, both the receive and transmit functions are called at every iteration of the loop, or called at a specific time interval, to process the data in both the RX and TX buffers.

For this application, a time interval between 38µs (minimum) to 250µs (maximum) was observed during testing to cause this noninterrupting and nonblocking UART method to function properly with a baud rate of 115200bps and 255-character buffer size settings. A different time interval will exist under a different baud rate and buffer size.

In a circular buffer implementation, a time interval must be implemented to ensure that incoming data will not overwrite existing data when the buffer is full. Conversely, in a linear buffer implementation, a time interval must be implemented to ensure that incoming data will not be discarded when the buffer is full.

When receiving data, the application monitors if the receive flag is set, gets the data from the receive register, and saves it in the incoming buffer. If the flag is not set, the application will exit the receive function and continue other routines. When transmitting data, the application checks to determine if the transmit flag is clear, after which data is placed onto the output register. If a transmission is currently ongoing, or if the transmit flag is set, the application exits the transmit function and executes other routines.

Circular and Linear Buffer Implementations

A buffer is generally used as temporary data storage, usually for streaming data. A circular buffer (or ring buffer) is a temporary data storage method with a memory allocation scheme in which the buffer can be of a fixed size, and each memory location can be reused when the index pointer has returned to its starting location. This circular buffer method is widely used and exists in several versions, depending on application requirements. While data is being written to the buffer, the write pointer increments, and the data counter also increments. Similarly, while data is being read from the buffer, the read pointer increments and the data counter decrements.

Another form of buffer is the linear buffer, which is similar to circular buffer with the exception that the index pointer does not return to the starting location. When the index

pointer reaches its buffer size, the buffer discards the incoming data and waits for the buffer to be read. If the buffer is read, the read pointer starts to increment until it reached the write pointer. When the read pointer is equal to the write pointer, the buffer is empty and ready to accept data.

Software Implementation

This section discusses a number of factors important to the implementation of the UART, including its initialization, receive, and transmit and main routines.

UART Initialization

The following routine demonstrates how to configure a nonblocking, noninterrupting UART using the 8-N-1 format. The *BAUDRATE* value in this routine is set to 115.2Kbps using a 11.05920MHz system clock.

```
void UART_Init(void)
{
 PADD |= 0x30;// Setup ports for alternate function
 PAAF |= 0x30;
 PAAFS1 &= ~0x30;
 U0BRH = (UINT8)((BAUDRATE & 0xFF00) >> 8); // Set up baud rate at
 U0BRL = (UINT8)(BAUDRATE & 0x00FF);        // 115Kbps
 U0CTL0 = 0xC0;           // Receive enable, no parity, 1 stop bit
}
```

UART Receive Routine

The following routine demonstrates how to handle data received from the UART Receive Data Register. It does not use any ISR, and is nonblocking in function. The data received from this register is transferred to the input buffer. This function is called at every pass through the main routine. As a result, the user can get and interpret the data from the `RBUF_InBuff` input buffer.

```
void UART0_Rx(void)
{
 if((U0STAT0 & 0x80) == 0x80)

  UINT8 uctemp = U0RXD;

  RBUF_AddByteToInBuffer(uctemp);  // add Rx data to input buffer

}
```

UART Transmit Routine

The following routine demonstrates how to use the output buffer for handling data to be transmitted via the UART Transmit Data Register. This function does not use interrupts,

and is also nonblocking. This function is called at every pass through the main routine. Data must be present in the buffer before starting the transmission.

```
void UART_Tx0(void)
{
 // If there is data to transmit
 if((U0STAT0 & 0x06) && ( ucRBUF_GetLengthOutBuffer() > 0 ))
 {
  U0TXD = ucRBUF_GetByteFromOutBuffer(); // get Tx data from
                                         // output buffer
 }
}
```

There are two build options available for this source code: a circular buffer option and a linear buffer option. The selection is implemented in the `RBUF.h` file, as shown below.

```
#define CIRCULAR 1 // Uncomment if circular buffer, otherwise
                   // linear
```

The option to use a circular or linear buffer is a matter of user preference. To implement a circular buffer, simply uncomment the `#define` statement shown above, and comment out the line for linear buffer implementation.

Adding a Byte to the Buffer

Figure 2 shows the initial state of an empty buffer, and indicates the location of the read and write pointers.

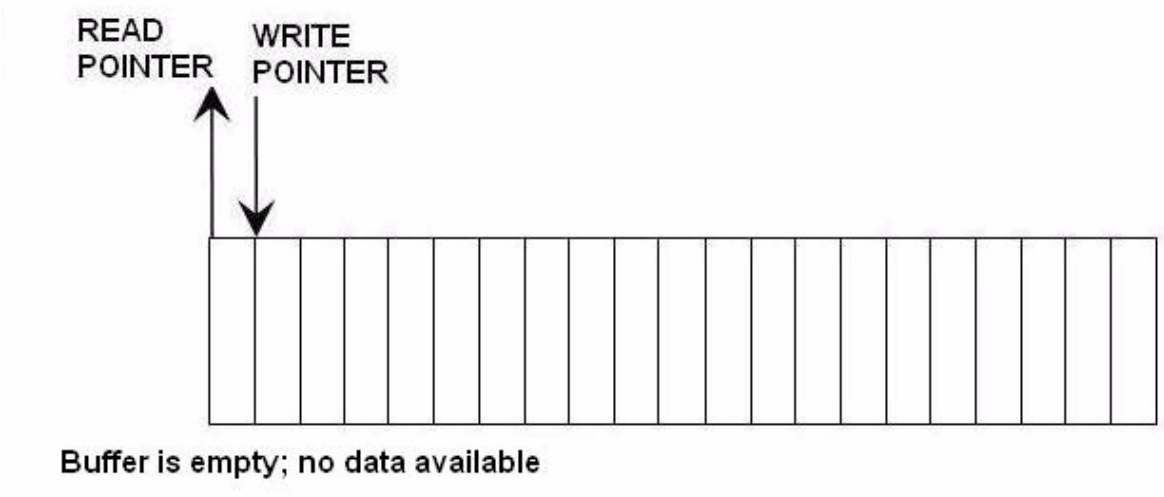


Figure 2. Initial State of an Empty Buffer

As bytes are added to a buffer, its buffer write pointer will increment until it reaches the end of the buffer, indicating that the buffer is full; see Figure 3.

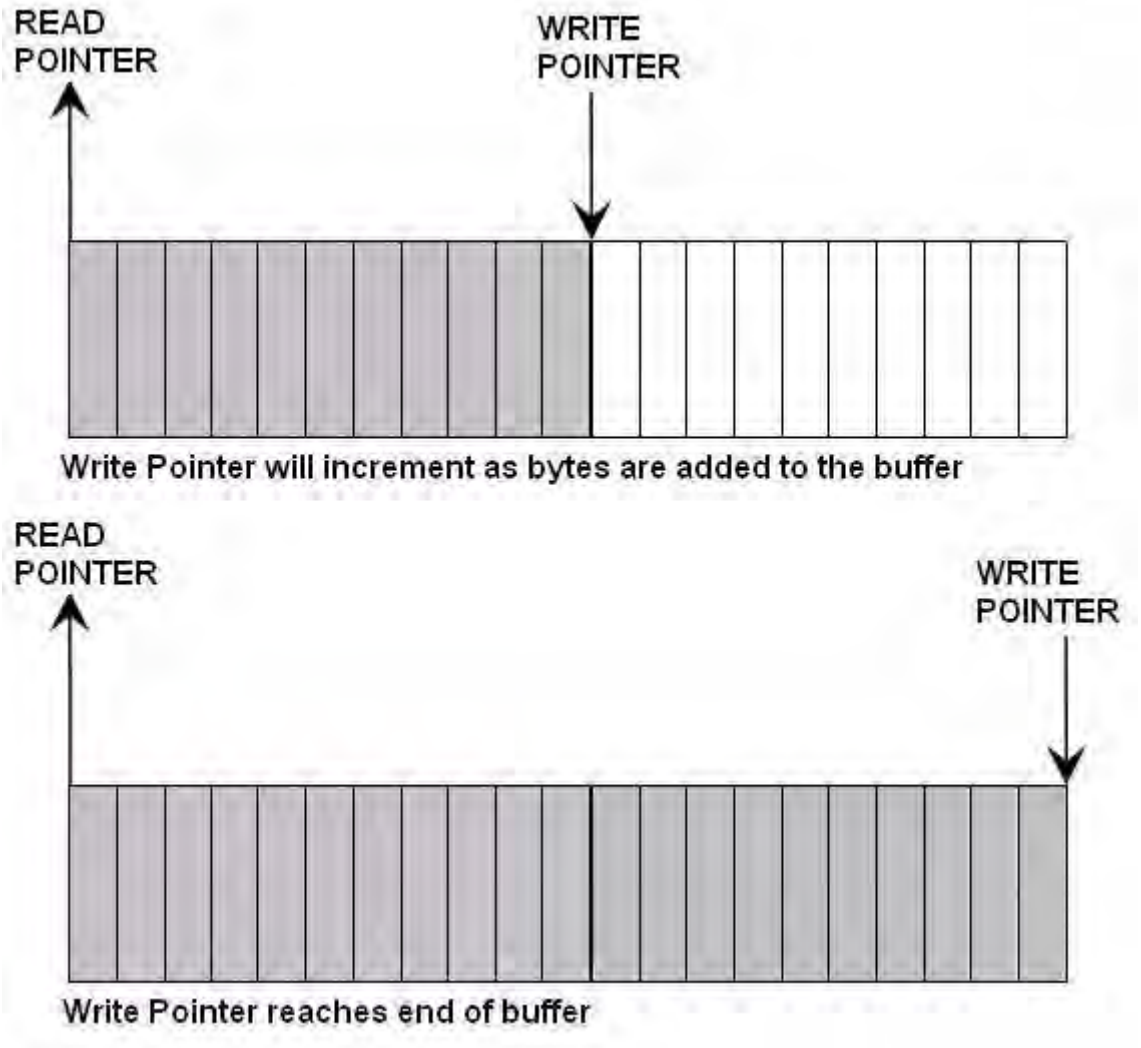


Figure 3. Buffer Write Implementation

Circular and linear buffer implementations differ from one another when a write pointer reaches the end of a buffer. In a circular buffer, buffer write pointers will automatically loop to the beginning of the buffer when full; see Figure 4.

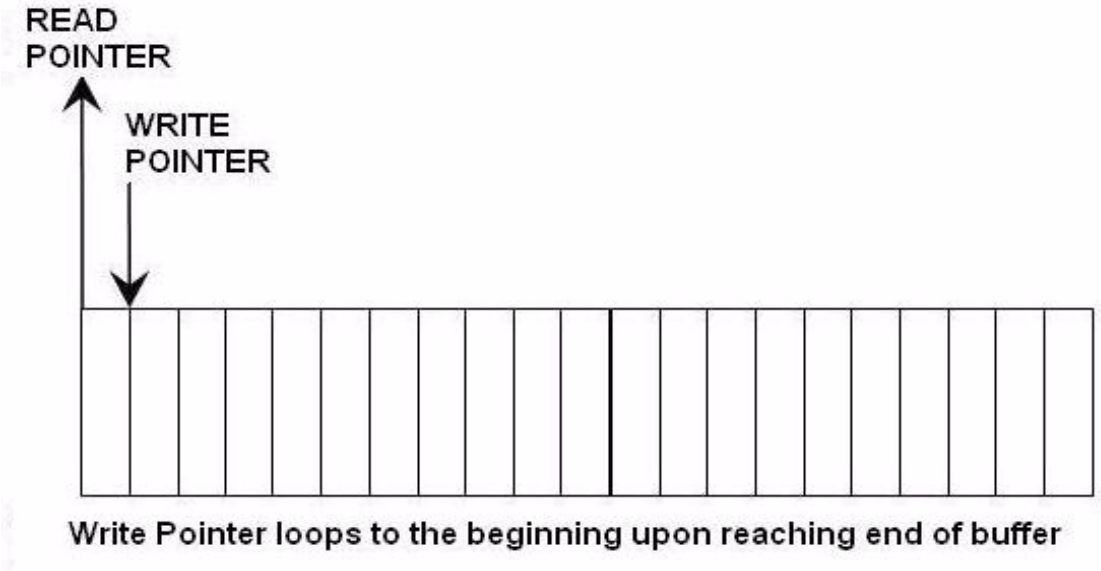


Figure 4. Buffer Write Implementation

In a linear buffer, if the buffer write pointer reaches the end of the buffer, the buffer becomes full and will discard any incoming data.

The following routines add a byte to the output buffer, and checks for an available buffer before writing.

```
#ifdef CIRCULAR                        // for circular buffer
void RBUF_AddByteToOutBuffer(UINT8 ucdata)
{

 if(( ((ucRBUF_OutWrPtr + 1) % RBUF_OUT_BUFFERSIZE) !=
ucRBUF_OutRdPtr ) && (ucRBUF_Flag & OUTBUFF_FULL) !=
OUTBUFF_FULL))
 {
  ucRBUF_OutBuff[ucRBUF_OutWrPtr] = ucdata;
  ucRBUF_OutWrPtr = (ucRBUF_OutWrPtr + 1) % RBUF_OUT_BUFFERSIZE;
  ucRBUF_OutLength++;
  ucRBUF_Flag |= OUTBUFF_HASDATA;   //outbuffer has data
 }

 if(ucRBUF_OutLength == RBUF_OUT_BUFFERSIZE)
 {
  ucRBUF_Flag |= OUTBUFF_FULL;      //outbuffer full
 }
}
#else                                  // for linear buffer
void RBUF_AddByteToOutBuffer(UINT8 ucdata)
{
```

```
 if( (ucRBUF_OutLength != RBUF_OUT_BUFFERSIZE) &&
 ((ucRBUF_Flag & OUTBUFF_FULL) != OUTBUFF_FULL))
 {
  ucRBUF_OutBuff[ucRBUF_OutWrPtr] = ucdata;
  ucRBUF_OutWrPtr = ucRBUF_OutLength;
  ucRBUF_OutLength++;
  ucRBUF_Flag |= OUTBUFF_HASDATA;//outbuffer has data
 }

if(ucRBUF_OutLength == RBUF_OUT_BUFFERSIZE)
 {
 ucRBUF_Flag |= OUTBUFF_FULL;    //outbuffer full
 }
}
#endif
```

The following routine adds a byte to the input buffer.

```
#ifdef CIRCULAR // for circular buffer
void RBUF_AddByteToInBuffer(UINT8 ucdata)
{
 if((((ucRBUF_InWrPtr + 1) % RBUF_IN_BUFFERSIZE) != ucRBUF_InRdPtr
 ) &&
 ((ucRBUF_Flag & INBUFF_FULL) != INBUFF_FULL))
 {
  ucRBUF_InBuff[ucRBUF_InWrPtr] = ucdata;
  ucRBUF_InWrPtr = (ucRBUF_InWrPtr + 1) % RBUF_IN_BUFFERSIZE;
  ucRBUF_InLength++;
  ucRBUF_Flag |= INBUFF_HASDATA;//inbuffer has data
 }

 if(ucRBUF_InLength == RBUF_IN_BUFFERSIZE)
 {
  ucRBUF_Flag |= INBUFF_FULL;    //inbuffer is full
 }
}
#else // for linear buffer
void RBUF_AddByteToInBuffer(UINT8 ucdata)
{
 if( (ucRBUF_InLength != RBUF_IN_BUFFERSIZE) &&
 ((ucRBUF_Flag & INBUFF_FULL) != INBUFF_FULL))
 {
  ucRBUF_InBuff[ucRBUF_InWrPtr] = ucdata;
  ucRBUF_InWrPtr = ucRBUF_InLength;
  ucRBUF_InLength++;
  ucRBUF_Flag |= INBUFF_HASDATA;//inbuffer has data
 }

 if(ucRBUF_InLength == RBUF_IN_BUFFERSIZE)
```

```
 {
  ucRBUF_Flag |= INBUFF_FULL;    //inbuffer is full
 }
}
```

Getting a Byte from the Buffer

Figure 5 shows the initial state of an unread buffer and indicates the location of the read and write pointers.

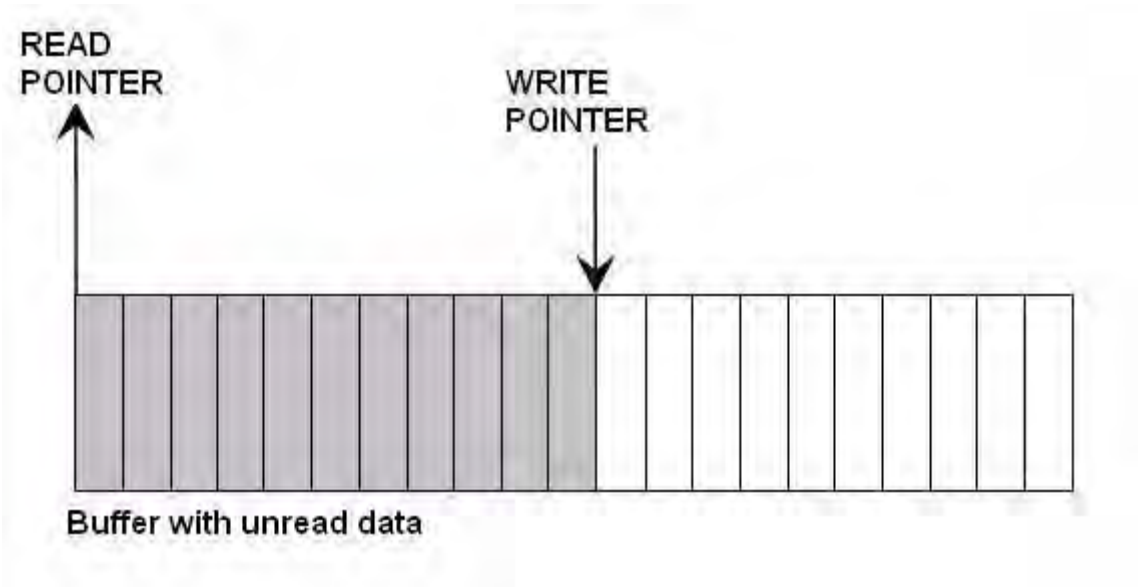


Figure 5. Initial State of the Buffer to be Read

A buffer read pointer will increment after each read until it reaches the buffer write pointer, indicating that the buffer is empty, as shown in Figure 6.

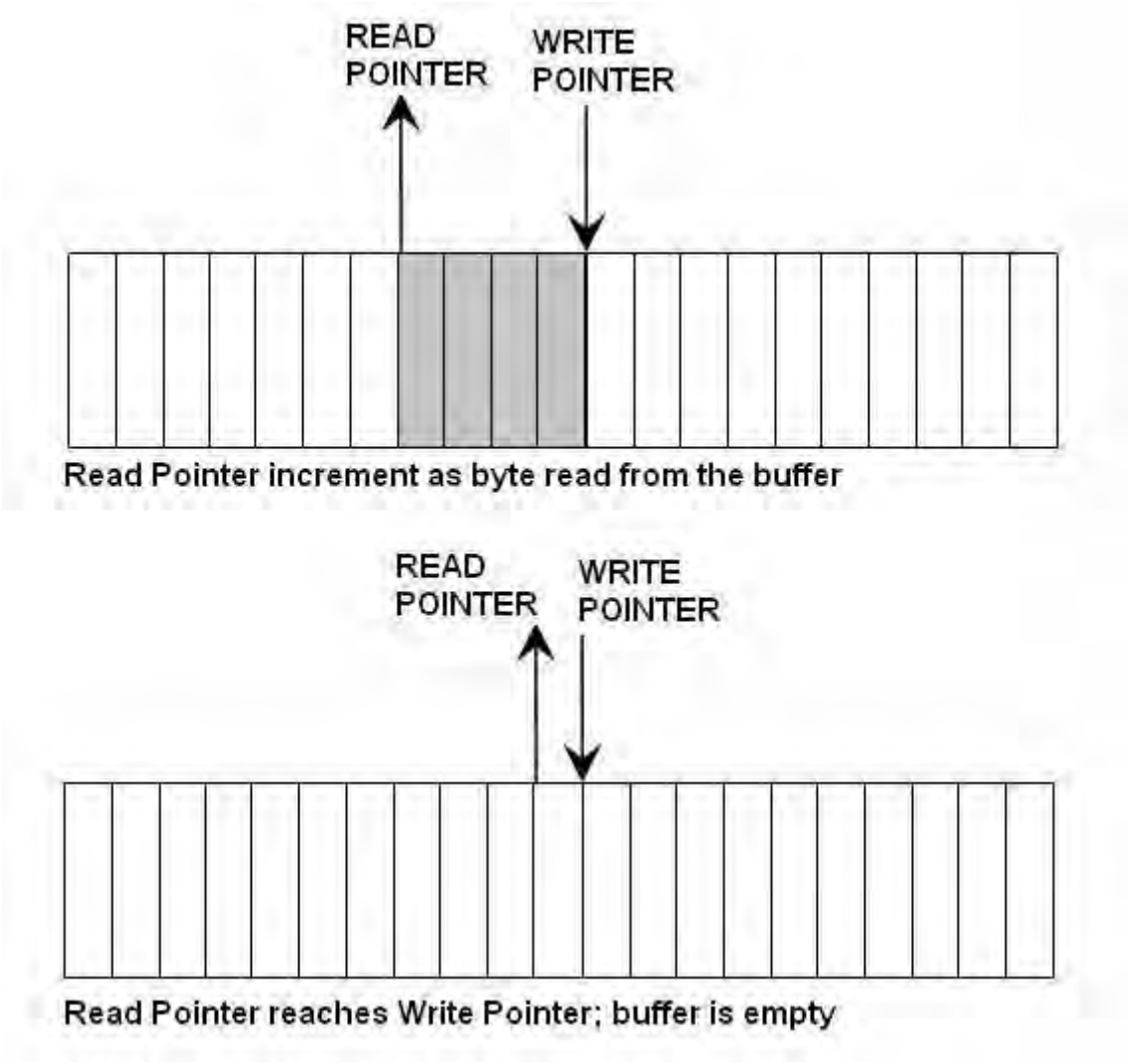


Figure 6. Buffer Read Implementation

Circular and linear buffer implementations differ from one another when the read pointer reaches the write pointer. In a circular buffer, a buffer read pointer will not loop to the beginning of the buffer when empty.

In a linear buffer, if the buffer read pointer reaches the buffer write pointer, the buffer is empty, and the read and write pointers are both reset to 0.

The following routine gets a byte from the output buffer and checks for available data in the buffer before reading.

```
#ifdef CIRCULAR                    //for circular buffer UINT8
ucRBUF_GetByteFromOutBuffer(void)
{
```

```
 if((ucRBUF_GetLengthOutBuffer()) &&
 (((ucRBUF_Flag & OUTBUFF_HASDATA) == OUTBUFF_HASDATA)||
 ((ucRBUF_Flag & OUTBUFF_FULL) == OUTBUFF_FULL)))
 {
  UINT8 ucdata = ucRBUF_OutBuff[ucRBUF_OutRdPtr];
  ucRBUF_OutRdPtr = (ucRBUF_OutRdPtr + 1) % RBUF_OUT_BUFFERSIZE;
  ucRBUF_OutLength--;
  return ucdata;
 }

if(ucRBUF_GetLengthOutBuffer() == 0)
 {
  //outbuffer is empty, no data available
  ucRBUF_Flag &= ~(OUTBUFF_FULL|OUTBUFF_HASDATA);;
 }

 return 0;
}
#else // for linear buffer
UINT8 ucRBUF_GetByteFromOutBuffer(void)
{
 if((ucRBUF_GetLengthOutBuffer()) &&
 (((ucRBUF_Flag & OUTBUFF_HASDATA) == OUTBUFF_HASDATA)||
 ((ucRBUF_Flag & OUTBUFF_FULL) == OUTBUFF_FULL)))
 {
  UINT8 ucdata = ucRBUF_OutBuff[ucRBUF_OutRdPtr];
  ucRBUF_OutLength--;

  if(ucRBUF_OutLength == 0)
  {
   ucRBUF_OutRdPtr = 0;
  }
  else
  {
  ucRBUF_OutRdPtr++;
  }

  if(ucRBUF_OutRdPtr==ucRBUF_OutWrPtr)
  {
   //outbuffer is empty, no data available
   ucRBUF_Flag &= ~(OUTBUFF_FULL|OUTBUFF_HASDATA);;
   ucRBUF_OutLength = 0;
   ucRBUF_OutRdPtr = 0;
  }

 return ucdata;
 }

return 0;
}
#endif
```

The following routine retrieves a byte from the input buffer.

```
#ifdef CIRCULAR // for circular buffer
UINT8 ucRBUF_GetByteFromInBuffer(void)
{
 if((ucRBUF_GetLengthInBuffer()) &&
 (((ucRBUF_Flag & INBUFF_HASDATA) == INBUFF_HASDATA)||
 ((ucRBUF_Flag & INBUFF_FULL) == INBUFF_FULL)))
 {
  UINT8 ucdata = ucRBUF_InBuff[ucRBUF_InRdPtr];
  ucRBUF_InRdPtr = (ucRBUF_InRdPtr + 1) % RBUF_IN_BUFFERSIZE;
  ucRBUF_InLength--;
  return ucdata;
 }

 if(ucRBUF_GetLengthInBuffer() == 0)
 {
  //inbuffer is empty, no data available
  ucRBUF_Flag &= ~(INBUFF_FULL|INBUFF_HASDATA);
 }

 return NULL;
}
#else                                  // for linear buffer
UINT8 ucRBUF_GetByteFromInBuffer(void)
{
 if((ucRBUF_GetLengthInBuffer()) &&
 (((ucRBUF_Flag & INBUFF_HASDATA) == INBUFF_HASDATA)||
 ((ucRBUF_Flag & INBUFF_FULL) == INBUFF_FULL)))
 {
  UINT8 ucdata = ucRBUF_InBuff[ucRBUF_InRdPtr];
  ucRBUF_InLength--;

  if(ucRBUF_InLength == 0)
  {
   ucRBUF_InRdPtr = 0;
  }
  else
  {
   ucRBUF_InRdPtr++;
  }

  if(ucRBUF_InRdPtr == ucRBUF_InWrPtr)
  {
   //inbuffer is empty, no data available
   ucRBUF_Flag &= ~(INBUFF_FULL|INBUFF_HASDATA);
   ucRBUF_InLength = 0;
   ucRBUF_InRdPtr = 0;
  }
```

```
  return ucdata;
 }

 return 0;
}
#endif
```

The Timer Function

In this application, a timer function is used to track the run time of the system, the average speed of input, and the period of time since the last input. Timer 0 is set to a continuous mode that ticks every millisecond. The following routine, which sets Timer 0 initialization, is contained in the `uart.h` file of the AN0349-SC01 source code.

```
//////////////////////////////////////////////////////////
// Peripheral Configuration Defines
//////////////////////////////////////////////////////////
#define T0RH_VAL    0x02  // Timer 0 reload value high for 1ms
                          // time out
#define T0RL_VAL    0xB3  // Timer 0 reload value low for 1ms time
                          // out

#define T0CTL0_VAL  0x00  // Timer 0 Control 0 Register value
                          // Timer Interrupt occurs on all defined
                          // Reload, Compare and Input Events
                          // Reset Input Capture Event
#define T0CTL1_VAL  0xA1  // Timer 0 Control 1 Register value
                          // Enabled Timer, Prescal=16, CONTINUOUS
                          // Mode
```

The following code is contained in the timer.c file of the AN0349-SC01 source code.

```
void TIMER_Init(void)
{
 T0H = 0x00;              // Timer 0 High Byte Register
 T0L = 0x01;              // Timer 0 Low Byte Register
 T0RH = T0RH_VAL;         // Reload High Byte Register
 T0RL = T0RL_VAL;         // Reload Low Byte Register
 T0CTL0 = T0CTL0_VAL;     // Load T0 Config0
 T0CTL1 = T0CTL1_VAL;     // Load T0 Config1
 IRQ0ENH |= 0x20;         // T0 interrupt and priority - Medium
 IRQ0ENL &= ~0x20;        // T0 interrupt and priority - Medium

}
```

For every tick of Timer 0, `ulTIMER_TimeCounter` and `ulTIMER_MinuteCounter` are incremented by 1. These two timer counters are represented in milliseconds, wherein the

values in the code that follows are equivalent to twenty-four hours and one minute, respectively.

```
void interrupt isrTIMER0(void)
{
 ulTIMER_TimeCounter++;         //increment hour counter

 //if counter for time monitoring reaches 24hrs
 if(ulTIMER_TimeCounter > HRS24)
 {
  ulTIMER_TimeCounter = RESET;  //counter reset and start again
 }

 ulTIMER_MinuteCounter++;       //increment minute counter

 if(ulTIMER_MinuteCounter == MINUTE) //if minute counter reaches
                                     //60000mS
 {
  ulTIMER_MinuteCounter = RESET; //reset counter
  ucTIMER_Flag |= ENABLED;      //0x01 minute flag to display time
 }
}
```

The Main Function

The main loop test starts with the user entering a string of characters in a terminal emulator (in this application, HyperTerminal is used). When the user presses the Enter key, the entered string will be displayed in the HyperTerminal window and include timing details such as the system time, the elapsed time, and the rate of input if display details are enabled; a command set is provided in this application to turn these timing details on or off.

However, if the user-entered string is a command, a notification will be displayed in the HyperTerminal window. Refer to Appendix B. Nonblocking UART Flowchart on page 29.

Table 1 list simple commands used to test the functionality of this application. To enable usage of this command set, press the Tab key, as indicated in Table 1.

Table 1. Nonblocking UART Application Commands

Command	Description
[TAB][MINUTEON][ENTER]	This command turns on the running display in one-minute intervals.
[TAB][MINUTEOF][ENTER]	This command turns off the running display.
[TAB][DETAILON][ENTER]	This command turns on the details display each time the user enters a string or character.
[TAB][DETAILOF][ENTER]	This command turns off the details display.

Note: Commands are case-sensitive.

Equipment Used

The tools used to build and test this application are:

- ZDSII – Z8 Encore! v5.0.0
- Z8F1680 MCU
- 5V DC power supply
- Serial cable
- The HyperTerminal terminal emulation program

Setup

This nonblocking UART application is implemented using a Z8 Encore! XP F1680 Series (28-Pin) Development Kit (Z8F16800128ZCOG). Contained in this Kit is the Z8 Encore! XP F1680 Series Development Board, to which a serial cable is connected, as shown in Figures 7 and 8. The serial cable connects the Board to a PC running HyperTerminal.

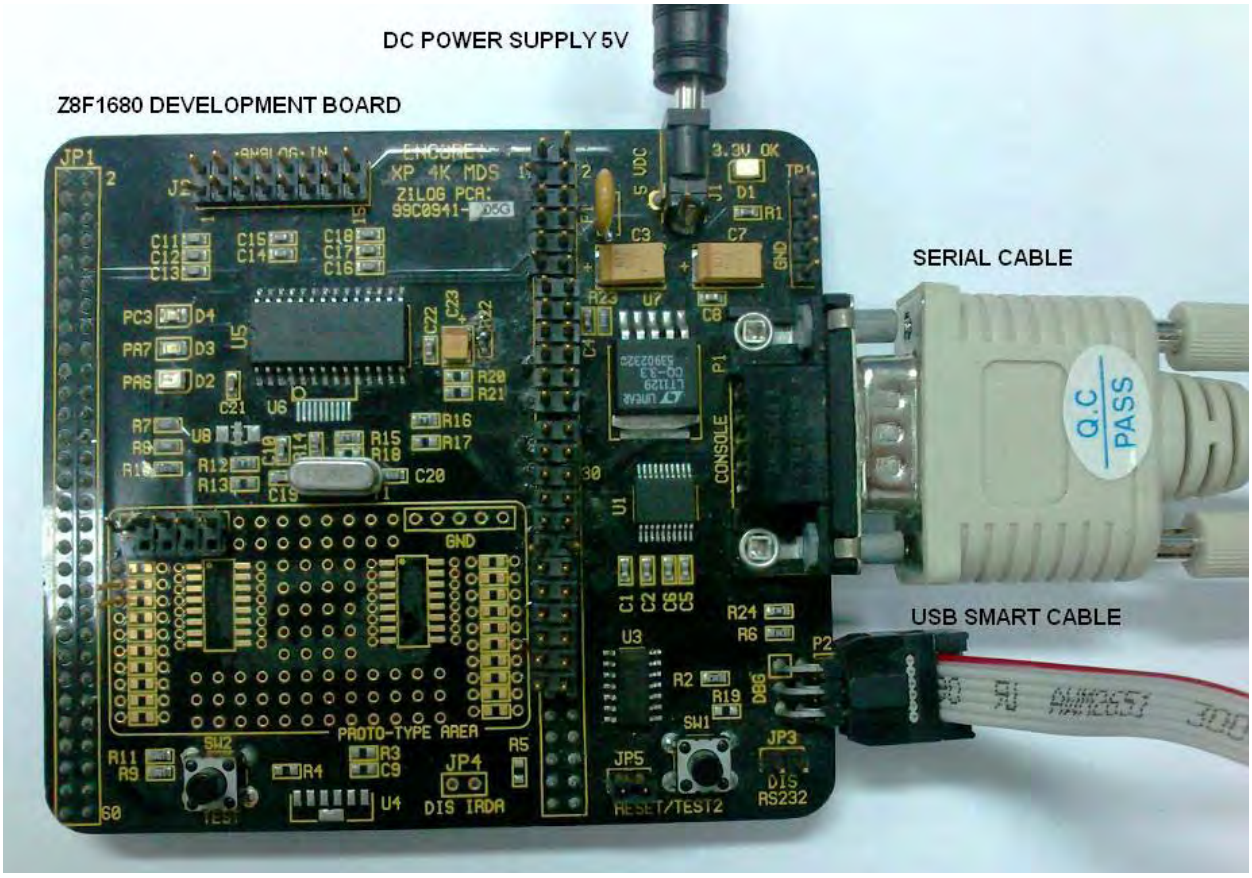


Figure 7. Hardware Setup

Figure 8 illustrates the hardware setup.

Figure 8. Block Diagram

Testing Procedure

Observe the following procedure to configure and test the Nonblocking UART application.

1. Launch the HyperTerminal application on your PC. The Connection Description dialog box will appear. Enter an appropriate name in the **Name:** field of this dialog and click **OK**.
2. The COM Port Properties dialog box appears next. Configure the fields in this dialog to be the same as those shown in Figure 9, then click **OK**.

Figure 9. HyperTerminal Setup

3. Download ZDSII – Z8 Encore! v5.0.0 and install it on your PC's hard drive.
4. Download AN0349-SC01.zip from the Zilog website and unzip it to a convenient location on your PC.
5. Launch ZDSII. In the ZDSII menu bar, choose **Open Project...** from the **File** menu to display the Open dialog box. Browse to the AN0349-SC01 folder you created in Step 4, locate the `.zdsproj` file, and click **Open**.
6. When the project is open, navigate via the **Project** menu **to Settings**; the Project Settings dialog will appear. In the Code Generation pane, ensure that the **Limit Optimization** checkbox is selected, and that **Memory Model** is set to **Large**.
7. In the **Target** pane of the **Settings** dialog's **Debugger** panel, select **AN0349_Z8F1680**, and click **Setup** to open the Configure Target dialog. In this dialog's Clock pane, ensure that Source is set to **Internal** and that Frequency (MHz) is set to **11.05920**. Click **OK**.
8. A dialog box will appear, displaying the following message:

   ```
   The project settings have changed since the last build. Would
   you like to rebuild the affected files?
   ```

   Click **YES** to build the project.
9. In the ZDSII toolbar, click **GO**.

Note: Ensure that both the 5V power supply and the USB SmartCable are connected to the Z8 Encore! XP F1680 Series Development Board. The other end of USB SmartCable must be properly connected to the PC's USB port. To learn more about these connections, refer to the Z8 Encore! XP F1680 Series (28-Pin) Development Kit User Manual (UM0203) and to the USB SmartCable User Manual (UM0181).

10. The HyperTerminal dialog will prompt for a user entry, as shown in Figure 10.

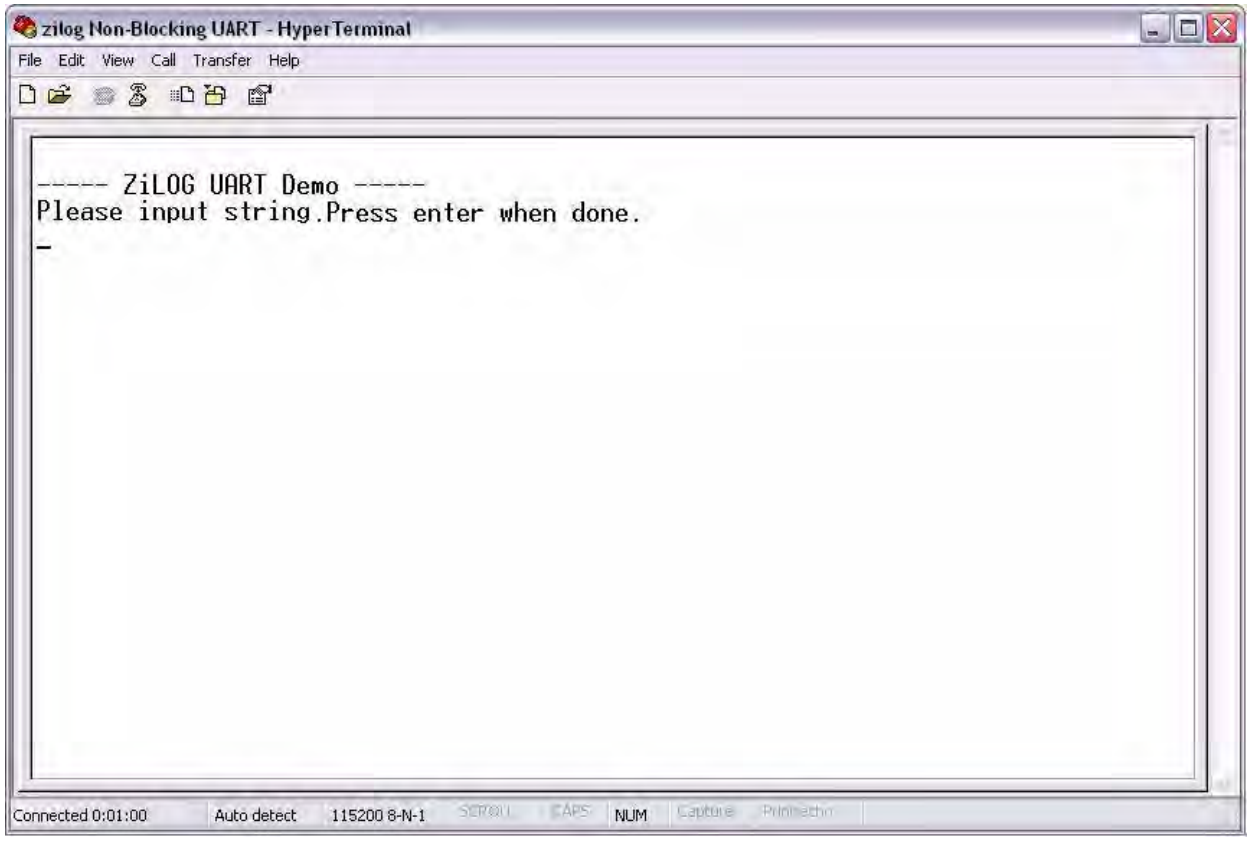


Figure 10. The User is Prompted to Enter a String

11. At the prompt, enter any character string, then press the Enter key. The string will display in the HyperTerminal window, as shown in Figure 11.

Figure 11. The Entry Screen

12. Press the Tab key to enable the command set, then enter the `DETAILON` command and press Enter. The HyperTerminal window will display `Details On!`, as shown in Figure 12.

Figure 12. The Details On Notification

13. Next, enter a new character string, then press the Enter key. Timing details are now included in the in HyperTerminal window, as shown in Figure 13.

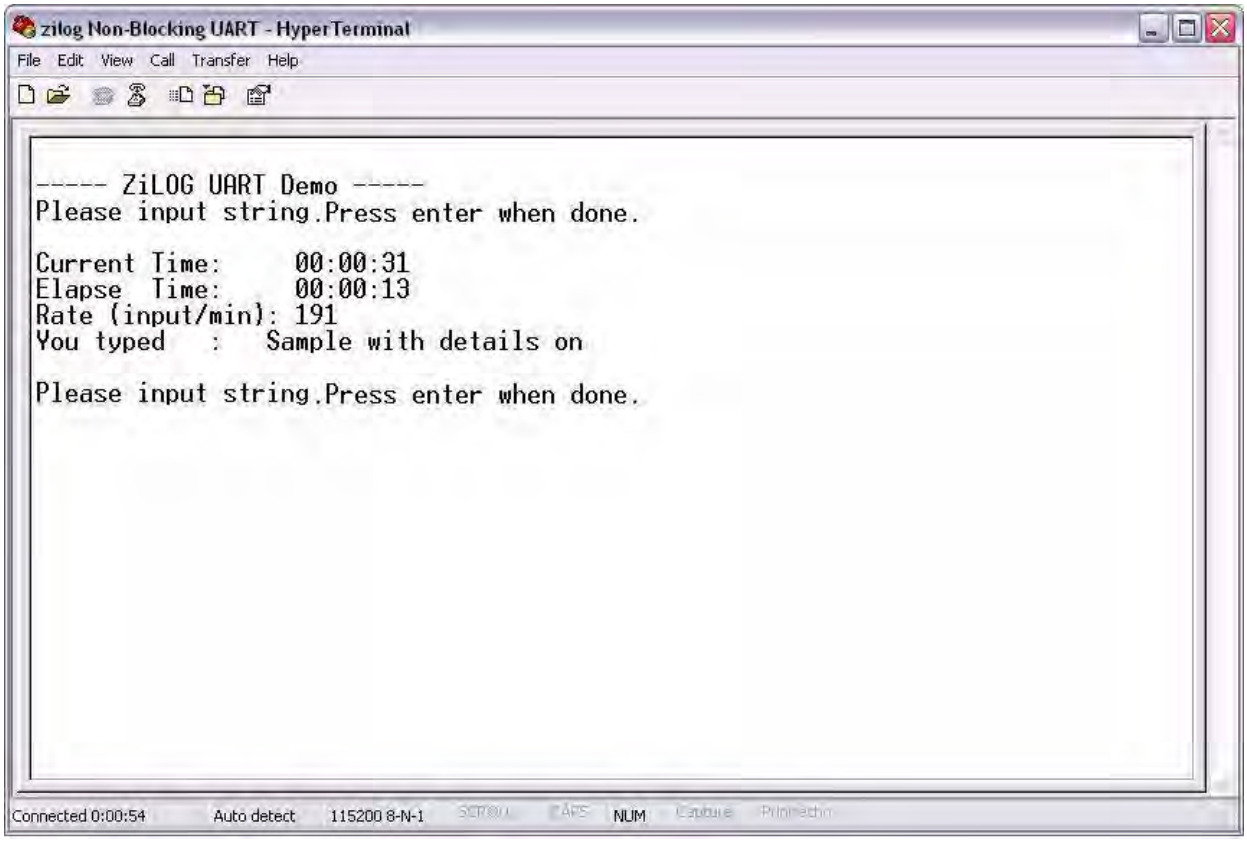


Figure 13. Timing Details Enabled Using the DETAILON Command

14. To turn off timing details, press the Tab key, then enter the `DETAILOF` command and press Enter. The HyperTerminal window will display `Details Off!`, as shown in Figure 14.

Figure 14. Showing Details Off notification

15. Press the Tab key, then enter the `MINUTEON` command and press Enter. The HyperTerminal window will display `Minute On!`, as shown in Figure 15.

Figure 15. Showing Minute On notification

16. Over the course of the next few minutes, the running time will appear in one-minute intervals in the HyperTerminal window, as shown in Figure 16.

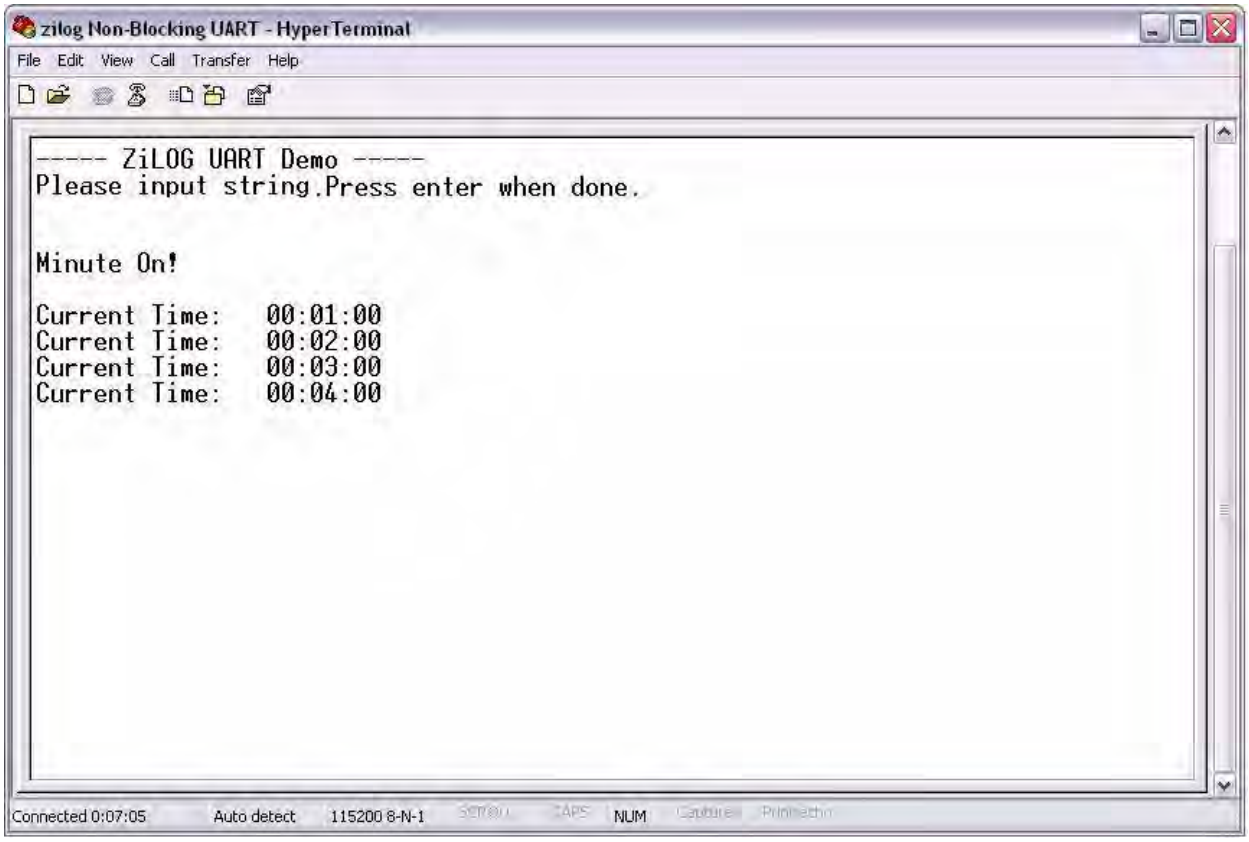


Figure 16. The Running Time at One-Minute Intervals

17. To turn off the one-minute running time display, press the Tab key and enter the following string: `MINUTEOF`. The HyperTerminal window will display `Minute Off!`, as shown in Figure 17.

Figure 17. The Minute Off! Message

18. Press the Tab key, then enter any alphabetic characters of any length and press Enter. An `Error Command!` notification will appear in the HyperTerminal window, as shown in Figure 18.

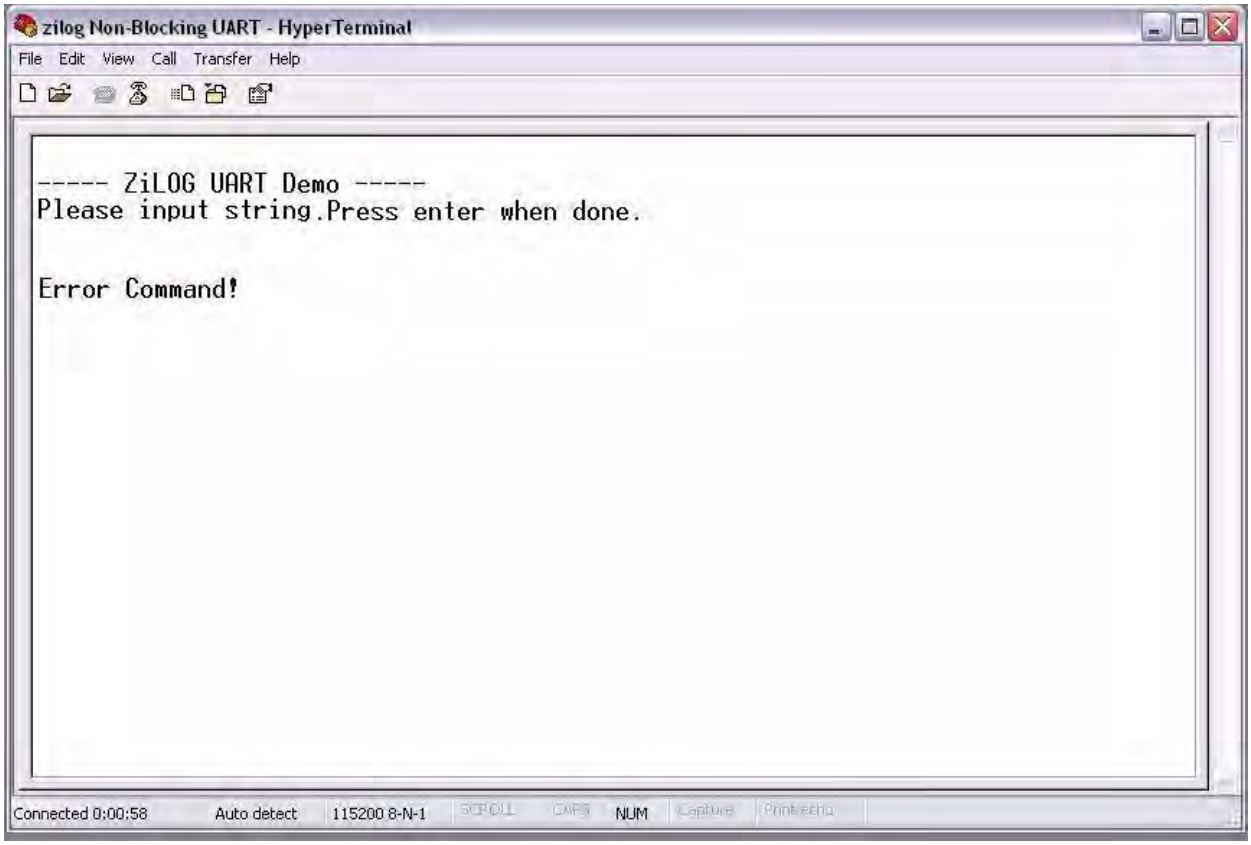


Figure 18. The Error Command Message

Note: The `Error Command!` notification will result if the command you enter at the HyperTerminal prompt is misspelled or is not a listed command. If such an error occurs, press the Tab key, enter the correct command, and press Enter.

Results

The application firmware has been tested using ZDSII – Z8 Encore! v5.0.0 and the Z8 Encore! XP F1680 Series (28-Pin) Development Kit. Nonblocking and noninterrupt UART functions were achieved based on testing results. UART transmission and reception were executed in a timely manner. A list of commands can be used to show the timing details. The source code is modular and can be reused and implemented according to user requirements.

Summary

This application demonstrates the use of noninterrupt and nonblocking functionality with a UART peripheral. The application can send any alphabetic/alphanumeric data to the UART without being interrupted and blocked by other functions. To implement this scheme, UART transmit and receive functions are called upon a strict time interval of 38µs (minimum) to 250µs (maximum) to ensure that the data is received and transmitted correctly. A list of commands is provided in this application to turn the display of these timing details on or off. Circular and linear buffer implementations are provided for the user's discretion.

References

Documents that support this application are listed below. Each of these documents can be obtained from the Zilog website by clicking the link associated with its document number.

- Z8 Encore! XP F1680 Series Product Specification (PS0250)
- Zilog Developer Studio II - Z8 Encore! User Manual (UM0130)
- eZ8 CPU User Manual (UM0128)
- An Interrupt-Driven UART for Z8 Encore! XP and Z8 Encore! MC MCUs Application Note (AN0330)

Appendix A. Nonblocking UART Functions

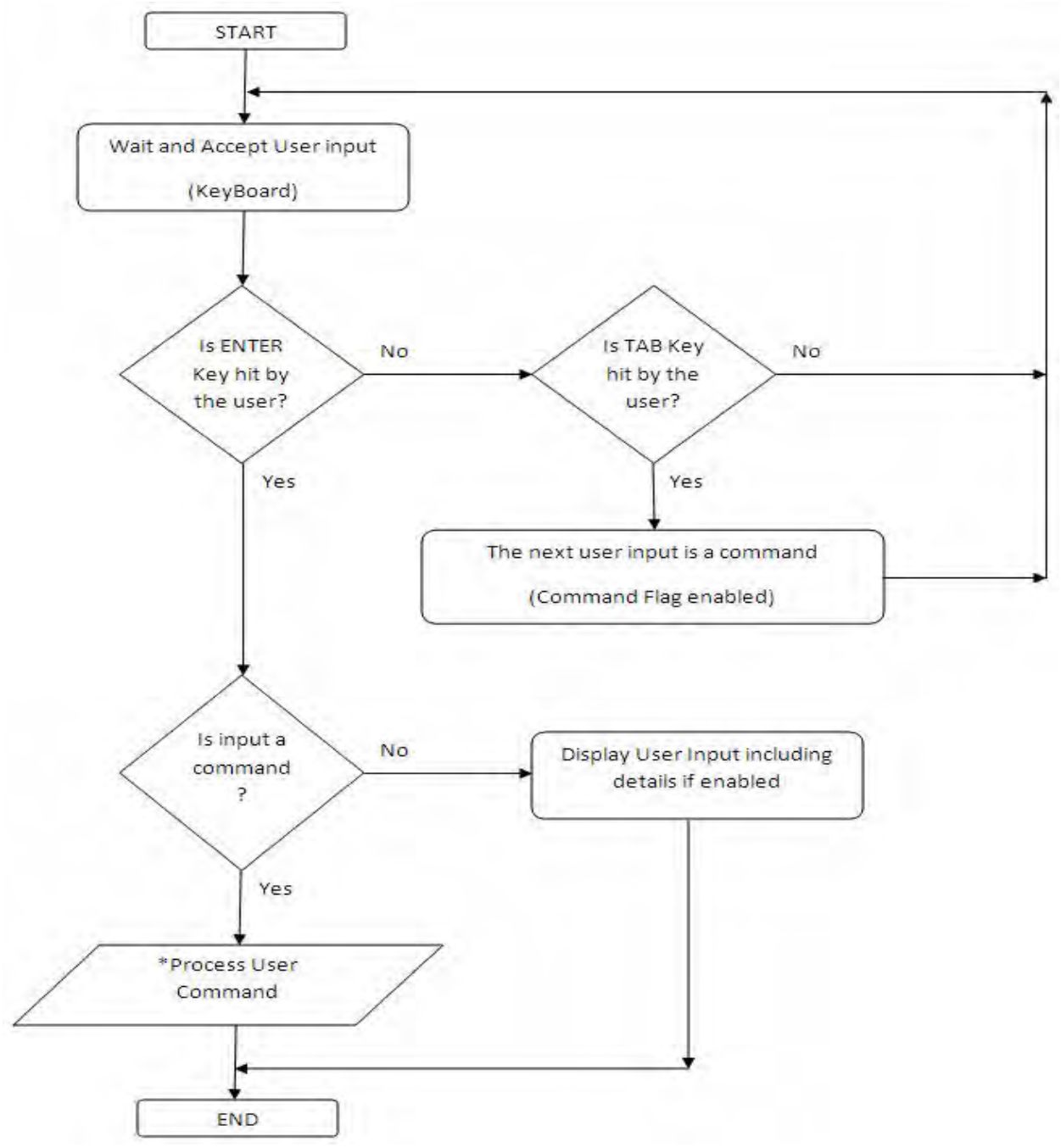
Table 2 lists the nonblocking functions developed for this application.

Table 2. Nonblocking UART Function and User (Receive and Transmit)

Function Name	Description
UINT8 RBUF_GetLengthOutBuffer (void)	Returns the current length of the OUTPUT buffer.
void RBUF_ AddByteToOutBuffer(UINT8 data)	Writes 1 byte of data into the OUTPUT buffer.
void RBUF_AddStrToOutBuffer (UINT8 *data, UINT8 len)	Writes a series or array of byte data into the OUTPUT buffer.
UINT8 RBUF_GetByteFromOut Buffer(void)	Reads 1 byte of data from the OUTPUT buffer.
void RBUF_GetStrFromOutBuffer (UINT8 *data, UINT8 len)	Reads a series or array of byte data from the OUTPUT buffer.
UINT8 RBUF_GetLengthInBuffer (void)	Returns the current length of the INPUT buffer.
void RBUF_ AddByteToInBuffer(UINT8 data)	Writes 1 byte of data into the INPUT buffer.
void RBUF_AddStrToInBuffer (UINT8 *data, UINT8 len)	Writes a series or array of byte data into the INPUT buffer.
UINT8 RBUF_GetByteFromInBuffer(void)	Reads 1 byte of data from the INPUT buffer.
void RBUF_GetStrFromInBuffer (UINT8 *data, UINT8 len)	Reads a series or array of byte data from the INPUT buffer.

Appendix B. Nonblocking UART Flowchart

Figure 19 presents the basic flow of the Nonblocking UART routine.



Note: *Refer to Table 1 on page 14 for a list of these commands.

Figure 19. Flow of the Nonblocking UART Routine

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.