Application Note

Using the USI Block as a UART on the Z8051 MCU

AN035801-0313

Abstract

This application note describes how to configure Zilog's Z8051 Universal Serial Interface (USI) peripheral to operate as a Universal Asynchronous Receiver Transmitter (UART). C code drivers for both polled and interrupt-driven method are provided.

Note: The source code file associated with this application note, AN0358-SC01.zip, is available for download from the Zilog website. Using the files contained in this .zip file requires installation of Zilog's Z8051 Software and Documentation, which is available free for download from the **Downloadable Software** category of the Zilog Store. This source code can be compiled using both Keil µVision 4 and Small Device C Compiler (SDCC) version 3.1.0. This source code has been tested using Zilog's Z51F3220 Development Kit. For this source code to work properly with other Z8051 MCUs, minor modifications to the source code may be required.

USI and UART: An Overview

The Universal Serial Interface (USI) peripheral on Zilog's Z8051 MCUs provides the hardware resources necessary for synchronous, asynchronous and two-wire serial communication. Compared to other software-based serial communication solutions, the USI uses less code space and performs at higher transfer rates. In this document, the USI is configured to operate as an Universal Asynchronous Receiver Transmitter (UART).

The UART is a highly flexible serial communication device that supports serial frames of 5, 6, 7, 8 or 9 data bits. Each serial frame is preceded by a start bit and is followed by one or two stop bits. Inclusion of a parity bit is optional; it is placed after the data frame and before the stop bit(s).

Discussion

When configured correctly, the USI peripheral on a Z8051 MCU can operate efficiently when performing UART serial communications. This section discusses the UART's data format, configuration, and other UART-related details.

Data Format

A serial frame is defined to be a series of character of data bits with synchronization bits (start and stop bits) and, optionally, a parity bit for error detection. The Z8051 UART supports all combinations of the following as valid frame formats:

- 1 start bit
- 5, 6, 7, 8 or 9 data bits
- Even, odd, or no parity bits
- 1 or 2 stop bits

The frame begins with the a start bit and is followed in sequence by the least significant data bit (lsb) and the next data bits – up to nine succeeding bits – finally ending with the most significant bit (msb). If a parity function is enabled, the parity bit is inserted between the last data bit and the stop bit. This parity bit is calculated by performing an exclusive OR of all of the data bits. If odd parity is used, the result of this exclusive OR is inverted. A High-to-Low transition on the data pin is considered to be the start bit. When a complete frame is transmitted, it can be directly followed by a new frame; otherwise, the communication line can be set to the IDLE state; this IDLE state is the High state of the data pin. Figure 1 shows the UART frame format.

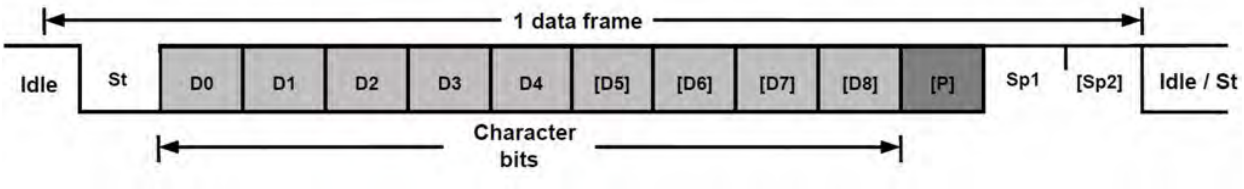

Figure 1. UART Frame Format

Transmitter (TX)

The transmitter consists of a single write buffer, a serial shift register, a parity generator, and control logic for handling serial frame formats. The UART transmitter is enabled by setting the TXE bit in the USIxCR2 Register. When this transmitter is enabled, the TXD pin should be set to the TXD function for the serial output pin of the UART.

Receiver (RX)

The receiver is the most complex part of the UART module, due to its clock and data recovery unit; this recovery unit is used for data reception. In addition to the recovery unit, the receiver includes a parity checker, a shift register, a two-level receive FIFO (USIxDR) and the control logic. The receiver can detect a frame error, a data overrun, and parity errors. The UART receiver is enabled by setting the RXE bit in the USIxCR2 Register. When this receiver is enabled, the RXD pin should be set to the RXD function for the serial input pin of the UART.

Baud Rate

The clock generation logic generates the base clock for the transmitter and receiver. For a UART, this clock generation logic is used for synchronizing the internally-generated baud rate clock to the incoming asynchronous serial frame on the RXD pin. The baud rate is set

in the USI Baud Rate Generation Register (USIxBD). Table 1 shows the equations for calculating this baud rate register setting.

Table 1. UART Baud Rate Calculation

Operating Mode	Equation for Calculating Baud Rate
Normal Mode	$\text{Baud Rate} = \frac{f_X}{16 * (USI * BD + 1)}$
Double Speed Mode	$\text{Baud Rate} = \frac{f_X}{8 * (USI * BD + 1)}$

Flags

Three flags that are essential for UART operation are the UART Transmit Complete flag (TXC), the Data Register Empty flag (DRE) and the Receive Complete (RXC) flag. All of these flags can be found in the USI Status Register 1.

The DRE flag indicates whether the transmit buffer is ready to receive new data. This flag bit is set when the transmit buffer is empty, and cleared when the transmit buffer contains data to be transmitted but has not yet been moved into the shift register. This flag can also be cleared by writing a 0 to this bit position.

The TXC flag is set when the entire frame in the transmit shift register has been shifted out and there is no new data currently present in the transmit buffer. This flag is automatically cleared when the interrupt service routine of a TXC interrupt is executed.

The RXC flag is set when there are unread data in the receive buffer; it is cleared when all of the data in the receive buffer are read. The RXC flag is cleared by reading the data; the user is not required to clear this flag manually.

All three of these flags can be used to generate an interrupt.

Registers

Six registers are important for allowing the USI Module operate as a UART peripheral; these registers are listed in Table 2.

Table 2. USI Registers Related to the UART

USI Registers	Description
USIxBD	**USI Baud Rate Generation Register** The value of this register is used to generate the internal baud rate in UART Mode.
USIxDR	**USI Data Register** UART data are written to this register.
USIxCR1	**USI Control Register 1** Controls or changes the behavior of USI

Table 2. USI Registers Related to the UART (Continued)

USI Registers	Description
USIxCR2	**USI Control Register 2** Controls or changes the behavior of USI.
USIxCR3	**USI Control Register 3** Controls or changes the behavior of USI.
USIxST1	**USI Status Register 1** Operation status of the UART can be determined by reading this register.

Hardware Implementation

The Z51F32220 Development Board used in this application is connected to a PC using a USB-to-USB mini connector. An on-chip debugger is connected to the Development Board when downloading the application software to the MCU. Figure 2 shows an example of this hardware setup.

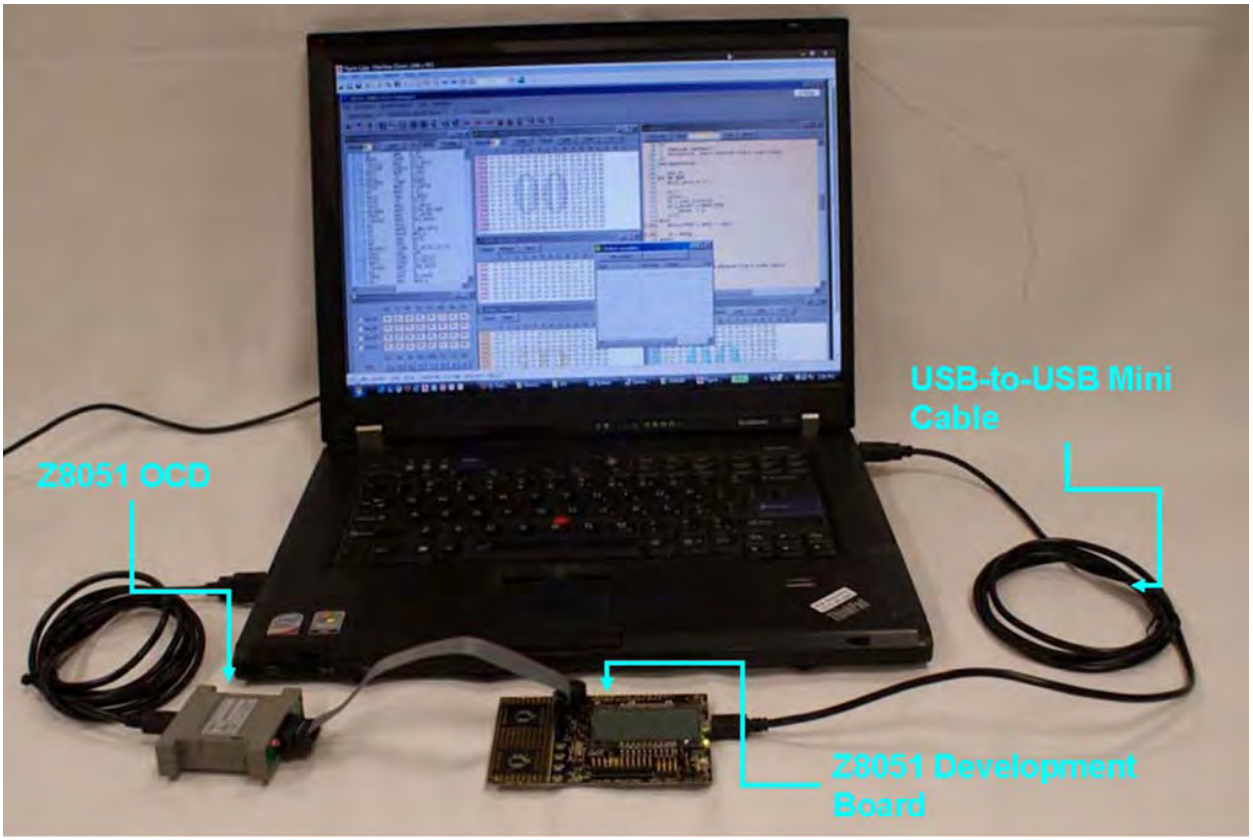

Figure 2. Hardware Setup

The USB-to-USB mini cable provides two functions: it powers the Development Board and serves as the communication line between the PC and the MCU. On the Development Board, the cable is connected to the FTDI UART-to-USB chip, which is connected to the UART pins.

Software Implementation

The software for this application is designed to show an example for both polled and interrupt-driven UART communication. It is also designed to work for both Keil and SDCC compilers. Listed below are brief descriptions of the four pertinent UART source code files, each of which is more fully described below.

uart.c. UART driver source.

uart.h. UART driver header file.

uart_example.c. Contains the main program for the UART-polled and interrupt-driven methods.

Clock Generator

The routine that sets up the clock generator can be found in the `osc.c` file. It is important to set the clock generator, because it produces the basic clock pulses that allow the system clock to drive the CPU and peripheral hardware such as the USI. In this application, the clock generator was configured to use the 16MHz calibrated internal RC Oscillator.

UART Routines

The UART routines can be found in the `uart.c` file. These routines initialize the UART, set the UART baud rate, and send and receive data with both the polling and interrupt-driven methods. Table 3 lists the functions included in the `uart.c` file.

Table 3. UART Routines

Function Name	Description
void Uart_Initialize (UINT32 ulFrequency, UINT32 ulBaudrate)	This routine initializes the USI to operate in UART mode. It also sets the baud rate. The default UART setting is 8-bits data, no parity, and 1 stop bit. This routine also initializes the buffers if the UART interrupt macros are defined.
INT8 Uart_GetChar (VOID)	This routine gets the data from the UART data register. Return value is a character. This function is used when the UART is configured to work in polled method.
VOID putchar (INT8 cData)	This routine puts a character to the UART data register. This function is called when printf function (SDCC) is used. No return value.
INT8 putchar (INT8 cData)	This routine puts a character to the UART data register. This function is called when printf function (KEIL) is used. Return value is a character.

Table 3. UART Routines (Continued)

Function Name	Description
VOID Uart_PutString (INT8 *acString, UINT ucLength)	This routine puts a string to the TX buffer and triggers the UART Transmit interrupt. This routine will only work if TX is set to interrupt mode.
VOID Uart_PutStrToTXBuffer (INT8 *acStr, UINT ucLen)	This routine puts a string to the TX buffer. This routine will only work if TX is set to interrupt mode.
VOID Uart_PutByteToTXBuffer (INT8 cData)	This routine puts a byte to the TX buffer. This routine will only work if TX is set to interrupt mode.
VOID Uart_StartTransmit (VOID)	This routine triggers the TX interrupt. This routine will only work if TX is set to interrupt mode.
UINT8 Uart_GetLengthTxBuffer (VOID)	This routine returns the length of the TX buffer. This routine will only work if TX is set to interrupt mode.
INT8 Uart_GetByteFromTXBuffer (VOID)	This routine gets a byte from the TX buffer. The return value is a character. This routine will only work if TX is set to interrupt mode.
VOID Uart_GetStrFromTXBuffer (INT8 *cStr, UINT ucLen)	This routine gets a string from the TX buffer. This routine will only work if the TX is set to interrupt mode.
UINT8 Uart_GetLengthRXBuffer (VOID)	This routine returns the length of the RX Buffer. This routine will only work if RX is set to interrupt mode.
VOID PutByteToRXBuffer (INT8 cData)	This routine puts a byte to the RX buffer. This routine will only work if RX is set to interrupt mode.
VOID Uart_PutStrToRXBuffer (INT8 *cStr, UINT ucLen)	This routine puts a string to the RX buffer. This routine will only work if RX is set to interrupt mode.
INT8 Uart_GetByteFromRXBuffer (VOID)	This routine gets a byte from the RX buffer. This routine will only work if RX is set to interrupt mode.
VOID Uart_GetStrFromRXBuffer (INT8 *cStr, UINT ucLen)	This routine gets a string from the RX buffer. This routine will only work if RX is set to interrupt mode.

UART Switches

UART switches are macros that are defined or commented out depending on which UART is used (0 or 1) and how each will be used (interrupt-driven or polling). These switches are written in the `uart.h` file, together with the function prototypes of the routines.

The following routine offers an example of a UART interrupt switch macro:

```
// Both RX and TX works as interrupt-driven
#define __INTERRUPT_DRIVEN__

// UART 0 is used
#define __UART0__

// UART RX only works as interrupt-driven
#define __UART_RX_INT__
```

```
// UART TX only works as interrupt- driven
#define __UART_TX_INT__
```

The __UART0__ macro should not be commented out if UART 0 is used. By the same token, the __INTERRUPT_DRIVEN__ macro should not be commented out if it is preferred that both the TX and RX of the UART must function in interrupt-driven mode. However, if either TX or RX must function in interrupt-driven mode, either the __UART_RX_INT__ macro or the __UART_TX_INT__ macro should be commented out. If the polled method is used, the __INTERRUPT_DRIVEN__ macro should be commented out.

Polled UART Implementation

An example of a polling UART operation can be found in the uart_example.c file. The following routine shows how UART0 is used with the polled method.

```
while (1)
{
// Display Message to the Terminal
cStrLength = sprintf(acBuffer,"\r[Loop: %5i ]Press 'O' to turn LED
"On or 'F' to turn it Off: ", iCount++);

printf(acBuffer);           // Directly print to Terminal
cRead = Uart_GetChar ();    // immediately read
cRead = toupper(cRead);     // Convert read character to uppercase

if(cRead == 'O')            // If read character is 'O'
{
   LED_ON;                  // LEDs ON
}
else if(cRead == 'F')       // If read character is 'F'
{
   LED_OFF;                 // LEDs OFF
}
Else                        // Otherwise
{}                          // Do nothing
 }
```

The above routine actively samples the status of the cRead variable, which is equal to the return value of the Uart_GetChar() function that gets data from the UART Data Register and executes the corresponding action required for every value of cRead. If cRead is O, the indicator LEDs illuminate. If cRead is F, then the LEDs will turn off; otherwise, no action will be taken.

Interrupt-Driven UART Implementation

An example of an interrupt-driven UART data transfer is written in the uart.c file, as indicated in the following code fragment. To facilitate data input/output via the UART, a circular buffer is used for storing data. Separate buffers are used to handle receive and transmit

data. The buffer size variables, RX_BUFFER_SIZE and TX_BUFFER_SIZE, can be changed to suit user requirements.

```
UINT8 xdata aucUart_InBuffer [RX_BUFFER_SIZE];   // Input Buffer
UINT8 xdata aucUart_OutBuffer [TX_BUFFER_SIZE];  // Output buffer
```

The following code demonstrates how to handle data received from the UART Data Register. The data received from this register is transferred to the buffer. It is up to the user to determine how to get and interpret the data from the `aucUart_InBuffer` input buffer.

```
VOID RX0_Isr(VOID) interrupt RX0_VECT
{
 INT8 cData = USIDR;         // Read RX Data Register
 if( (USIST1 & (DOR | FE | PE) == (DOR | FE | PE)) )
                             // Error Occurred
 {
  return;                    // Just read data to clear USIxST1
 }
 else
 {
  Uart_PutByteToRXBuffer(cData);  // Put data to RX Buffer
 }
}                            // RX_isr
```

Similarly, the code that follows demonstrates how to use the `aucUart_OutBuffer` for handling data to be transmitted via the UART Transmit Data Register. Data must be placed into this buffer before starting a transmission.

```
Uart_PutStrToTXBuffer(acString, ucLength); // Place the string in
TX buffer
Uart_StartTransmit();        // Trigger UART TX interrupt

////////////////////////////////////////////////////////////////

VOID TX0_Isr (VOID) interrupt TX0_VECT
{
 if (! (USIST1 & DRE))       // If USIxST1 Data Register
 {                           // Empty Flag is Zero
                             // Do nothing
                             // Not empty yet; just skip
 }
 else                        // Otherwise
 {
  if (Uart_GetLengthTxBuffer () > 0)
  {
   USIDR = Uart_GetByteFromTXBuffer ();
  }                          // To USI Data Register (TX)
   else                      // Otherwise
   {}                        // Do nothing
```

```
  }
}                                   // TX_isr
```

Circular Buffer Implementation

A buffer is generally used for temporary data storage, and usually for streaming data. Similarly, a circular (or ring) buffer stores temporary data 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 the starting location. This buffering scheme is widely used and has several existing versions, each of which varies depending on application requirements. This section describes a simple buffering mechanism.

To initialize circular buffers, a memory segment or an array of predefined length, is initialized. This memory segment is where buffered data will be stored.

```
#define RX_BUFFER_SIZE    (100)
UINT8 xdata aucUart_InBuffer [RX_BUFFER_SIZE];
```

To facilitate how the circular buffer is managed, two index pointers and a data counter are initialized, as indicated in the following code fragment.

```
UINT8 ucUart_InBufferRdPtr;   // RX pointer to next read loc
UINT8 ucUart_InBufferWrPtr;   // RX pointer to next write loc
UINT8 ucUart_InBufferLength;  // RX length
```

Upon initialization, the buffer is empty and the pointers are at the beginning of the buffer, as indicated in Figure 3.

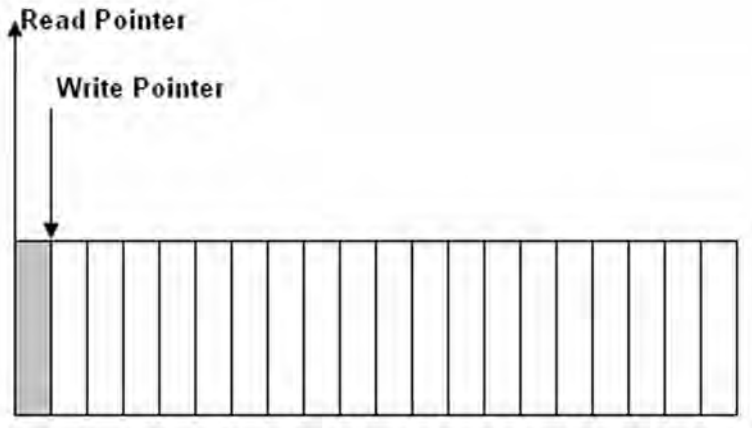

Figure 3. Initializing the Buffer

While data is being written to the buffer, the write pointer and the data counter both increment. Similarly, while data is being read from the buffer, the read pointer increments and the data counter decrements, as indicated in Figure 4 and the code that follows.

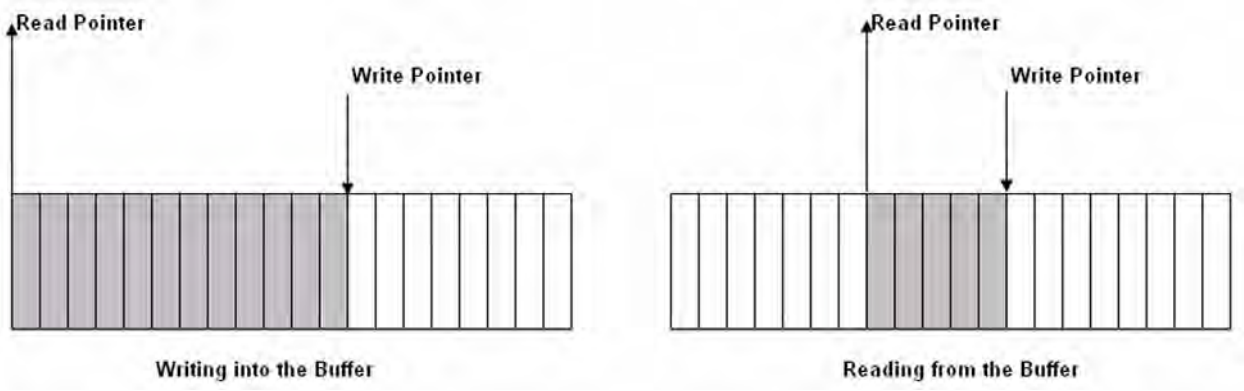

Figure 4. Read/Write Operations of the Buffer

```
VOID Uart_PutByteToRXBuffer (INT8 cData)
{
                              // Check if RX buffer is full
 if((ucUart_InBufferWrPtr + 1)%RX_BUFFER_SIZE) !=
ucUart_InBufferRdPtr)
 {
                              // Add byte to RX Buffer
  aucUart_InBuffer[ucUart_InBufferWrPtr] = cData;
                              // Increment RX write pointer
  ucUart_InBufferWrPtr = (ucUart_InBufferWrPtr + 1) %
RX_BUFFER_SIZE;
                              // Increment RX Buffer length count
  ucUart_InBufferLength++;
 }
 else                         // if RX buffer full
 {}                           // Do nothing
}                             // Uart_PutByteToRXBuffer


INT8 Uart_GetByteFromRXBuffer (VOID)
{
                              // Get a byte from RX Buffer
                              INT8 cData =
  aucUart_InBuffer[ucUart_InBufferRdPtr];
                              // Increment RX write pointer
  ucUart_InBufferRdPtr = (ucUart_InBufferRdPtr + 1) %
RX_BUFFER_SIZE;
                              // Decrement RX buffer length count
  ucUart_InBufferLength--;
                              // return obtained byte
  return (cData);
}                             // Uart_GetByteFromRXBuffer
```

When the read or write pointer reaches the end of the buffer, it will return to the start, causing a wrap-around effect. As a result, the data that has been previously fetched using the read operation will be overwritten. See Figure 5.

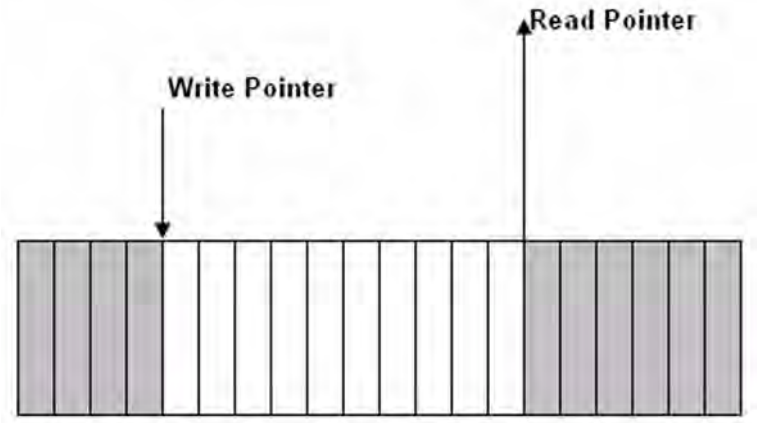

Figure 5. Buffer Wrap Around Effect

Equipment Used

This section provides a complete list of the hardware and software requirements for this application.

Hardware

The hardware tools required for this application are:

- Z51F3220 Development Kit
- Desktop or laptop PC with a minimum of three USB ports

Software

The software tools used to develop this application are:

- Keil µVision 4
- SDCC v3.1.0
- AN0358-SC01.zip file containing the project and source code files
- HyperTerminal or equivalent terminal emulation program

Testing the Application

This section lists the steps for demonstrating this application and testing the software.

Hardware Setup

To build, configure and test the hardware for this application, observe the following procedure.

1. Connect the On-Chip-Debugger (OCD) to the PC's USB port, and connect the 10-pin connector of the OCD to the Z51F3220 Development Board.
2. Connect one end of the USB-to-USB mini cable to the Z51F3220 Development Board, and connect the other end to the PC's USB port. The indicator LED near the USB mini port of the Development Board should illuminate to indicate that the Board is powered up.
3. On the Development Board, check to determine if the header J16 jumpers, pins 1–2 and 3–4, are in place. These jumpers will connect the UART0 TXD and RXD pins to the FTDI UART-to-USB chip.

Software Setup

To install, configure and test the software for this application, observe the following procedure.

1. Download and install the Z8051 Software and Documentation files if you haven't already done so. These files are available free for download from the **Downloadable Software** category of the Zilog Store.
2. After the Z8051 software is installed, download the AN0358-SC01.zip file from the Zilog website and unzip it into the following path, which was created during the installation process in Step 1:

   ```
   <Z8051 software installation folder>\samples
   ```

3. Open the `uart.h` file, which is located in the `<inc>` folder. If the UART will be interrupt-driven, ensure that the `__INTERRUPT_DRIVEN__` macro switch is defined. However, if the UART will function in polling mode, comment out the `__INTERRUPT_DRIVEN__` macro, then save the file.
4. The software created for this application is designed in a manner such that the source code must be compiled using either the Keil µVision 4 tool or the Small Device C Compiler (SDCC) tool. Select one of these two tools as your compiler, and compile the application software.
5. If the preferred compiler is SDCC, run the `build_sdcc.bat` file which is contained in the `sdcc` folder. Ensure that the `INSTALL_FOLDER` variable in the `build_sdcc.bat` file contains the correct installation path for `Z8051_2.1` in your computer. If not, change it to the correct path, then save the file.
6. A hex file will be created at the conclusion of the build. Load this hex file to the Z8051 MCU using Zilog's Z8051 OCD 1.147 software tool.

Note: To learn more about compiling and loading a hex file to a Z8051 MCU, please refer to the Z8051 Tools Product User Guide (PUG0033).

Demonstration

To demonstrate how the application works, observe the following procedure.

1. Load the hex file of the preferred UART example to the MCU by using either Keil µVision4 or the Z8051 OCD 1.147 software tool.
2. After loading the software to the MCU, disconnect the 10-pin connector of the OCD and the USB-to-USB mini cable from the Development Board.
3. Power up the Development Board by reconnecting the USB-to-USB mini cable to the Development Board.
4. Configure the HyperTerminal (or equivalent) emulation program. In the application software, the MCU UART is set to communicate with 9600 baud, 8 bits data frame, no parity bits, and 1 stop bit. Essentially, configure the terminal emulation program to match the MCU's requirements.
5. Reset the MCU by pressing the reset switch on the Development Board. Note that for the reset pin to work, the **Enable/Reset Input** option in both the OCD and Keil configuration dialogs must be selected during the loading of the hex file to the MCU. See Figures 6 and 7.

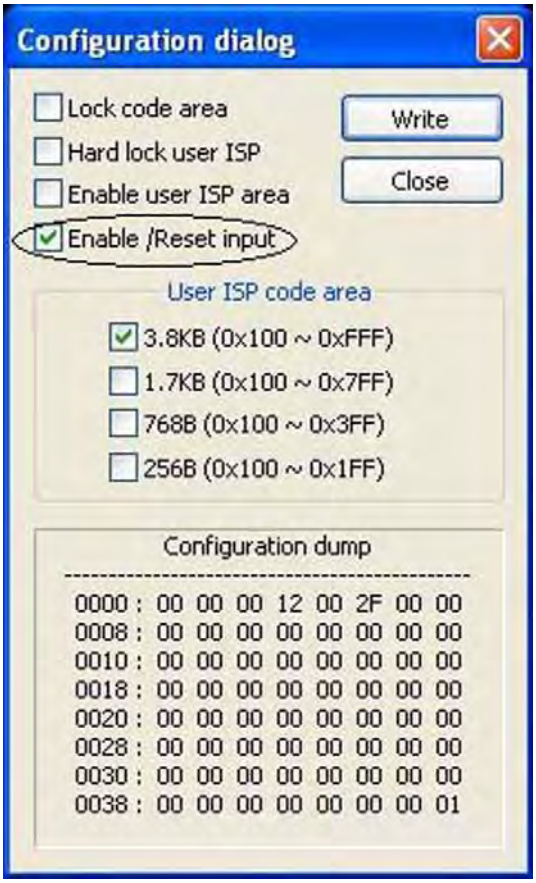

Figure 6. Configuration Dialog, Zilog Z8051 OCD 1.147

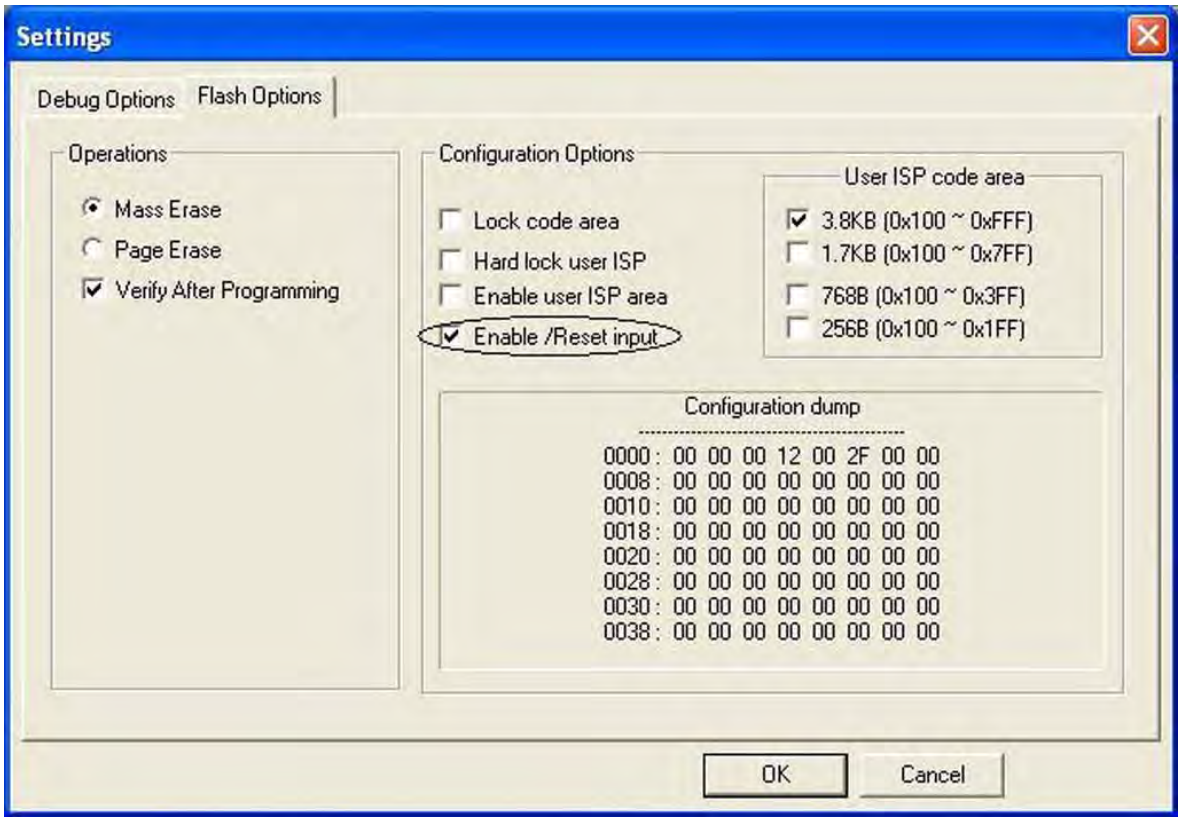

Figure 7. Configuration Options, Keil µVision4

6. The LEDs on the Development Board will illuminate, and HyperTerminal will display a message, as shown in Figure 8.

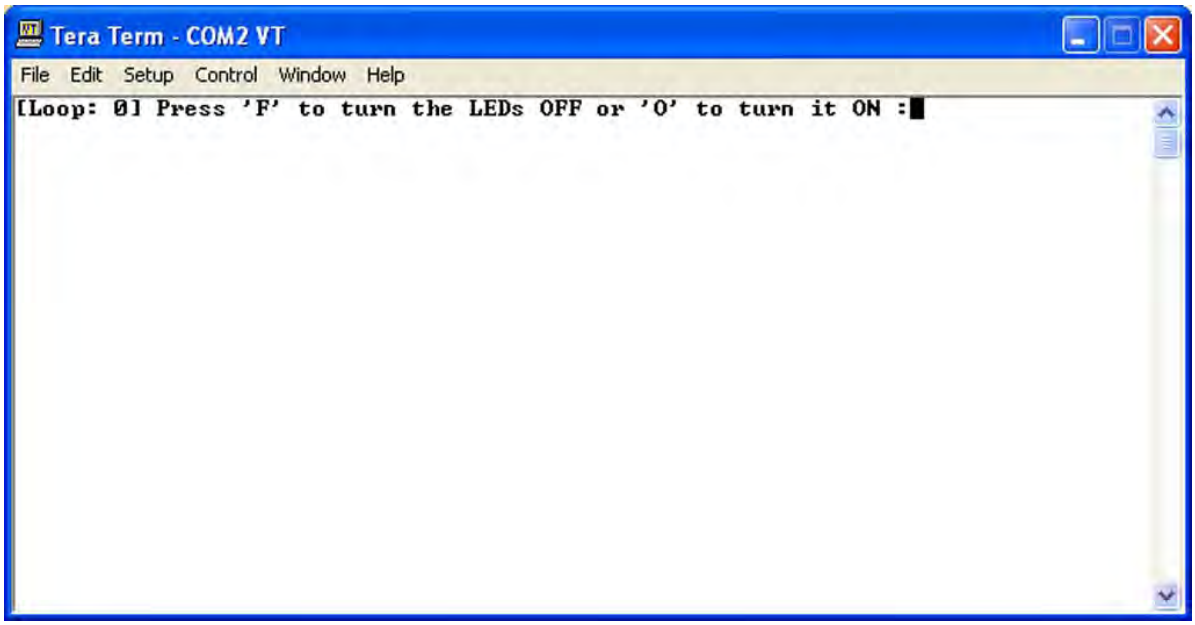

Figure 8. Terminal Display After Reset

7. Follow the instructions displayed in HyperTerminal. Press the O key on your keyboard to turn the LEDs ON, and press the F key to turn the LEDs OFF. Zilog recommends that you try both the polled and interrupt-driven UART examples. Observe the difference between the two UART operations by observing the behavior of the `Loop` value.

Results

The terminal emulation program shows that the Z8051 MCU's USI Module can be configured to work as a UART peripheral. As such, it can function in both Polled Mode and Interrupt-Driven Mode. In Polled Mode, the software will continuously check the status of the UART. As a result, the variable's Loop value will not increment until the UART receives a character. While operating in Interrupt-Driven Mode, the software will execute a loop until the UART receives a character; the interrupt service routine is then executed.

Summary

This document discusses how to use the USI Module of Zilog's Z8051 MCU as a UART. The software projects included with this application demonstrate that the USI can work as a UART, functioning in both the Polled and the Interrupt-Driven modes.

References

The following documents support this application, and are available free for download from the Zilog website.

- Z51F3220 Product Specification (PS0299)
- Z51F3220 Development Kit User Manual (UM0243)
- Z8051 Tools Product User Guide (PUG0033)

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.

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.