Using a Z8051 UART to Implement a 1-Wire® Master with Multiple Slaves

Abstract

This application note describes how to use the UART peripherals in Zilog's family of Z8051 MCUs as 1-Wire bus Masters. Also described are the required electrical interface, the UART settings, the relationship between the UART and 1-Wire signals, and the search algorithm used to identify multiple slaves. For this application, the following three 1-Wire slave devices are used to demonstrate the 1-Wire protocol while operating in a multislave configuration:

- DS18S20: a digital thermometer
- DS2417: a time chip with interrupt
- DS24B33: a 4KB EEPROM

> **Note:** The source code file associated with this application note, AN0346-SC01.zip, is available for download from the Zilog website. This source code is compiled using the Small Device C Compiler (SDCC) version 3.1.0 and tested using Zilog's Z51F3220 Development Kit. For this source code to work properly on other Z8051 MCUs, you may be required to make minor modifications.

Overview of the 1-Wire Protocol

A 1-Wire bus utilizes a single wire for power and signaling. This bus operates in an open-drain environment; therefore a pull-up resistor is required. The bus also operates in a 2.0V–5.5V range. The communication is asynchronous, half-duplex, and strictly follows a Master-Slave scheme. Only one Master – and either one or several slave devices – should be connected on the bus. Only one data bit can be transmitted on the bus for every time period of at least 60µs.

For more detail about these pull-up resistor values, refer to the documentation that describes the 1-Wire slave devices you are using. For more information about the Maxim/Dallas 1-Wire interface, visit Maxim's 1-Wire Devices page.

Discussion

When using 1-Wire communication, the Master must initiate bit transmission by pulling the bus Low which, in turn, synchronizes the timing logic of all units. This section discusses the five basic signals that are significant to a 1-Wire operation and how these sig-

nals are generated by the Z8051 UART module. This section also shows the commands and algorithm used to identify slaves in a multislave 1-Wire setup.

Reset and Presence Signal

A *Reset and Presence signal* is illustrated in Figure 1. When issuing a Reset signal, the Master pulls the bus Low for at least 480µs. If a slave is present, a response will be received by the Master. This response is called the Presence signal, and it occurs when the bus is pulled Low by the slave(s) within 60µs after the Master releases the bus. If the Presence signal is not received by the Master, the Master will assume that there is no device(s)/slave(s) present on the bus.
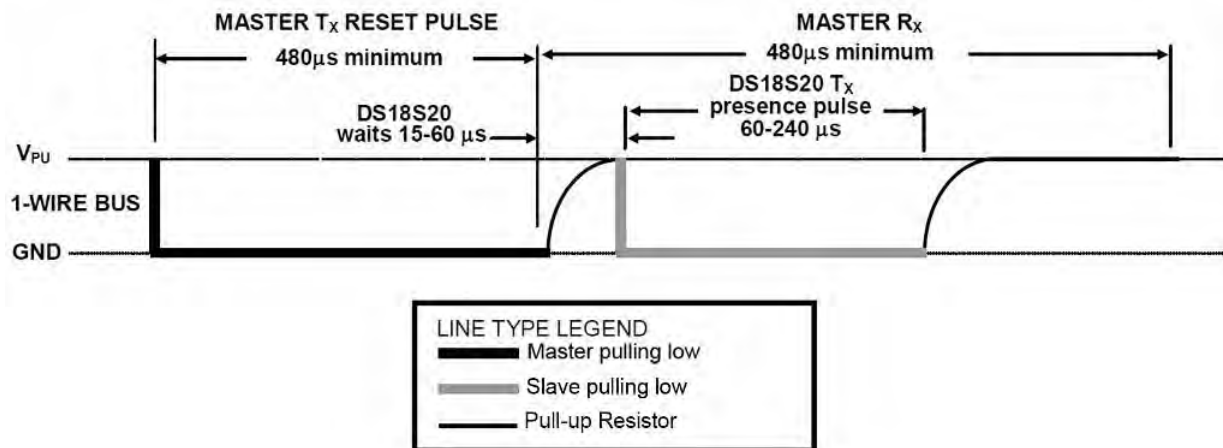

Figure 1. Reset and Presence Timing Diagram

Write Signals

The Write 0 and Write 1 signals are shown in Figure 2. To send a Write 0 signal in the bus, the Master must pull the bus Low for a period of 60µs to 120µs. To send a Write 1 signal in the bus, the Master must pull the bus Low for 1–15 µs, then release the bus for the remainder of the time period.
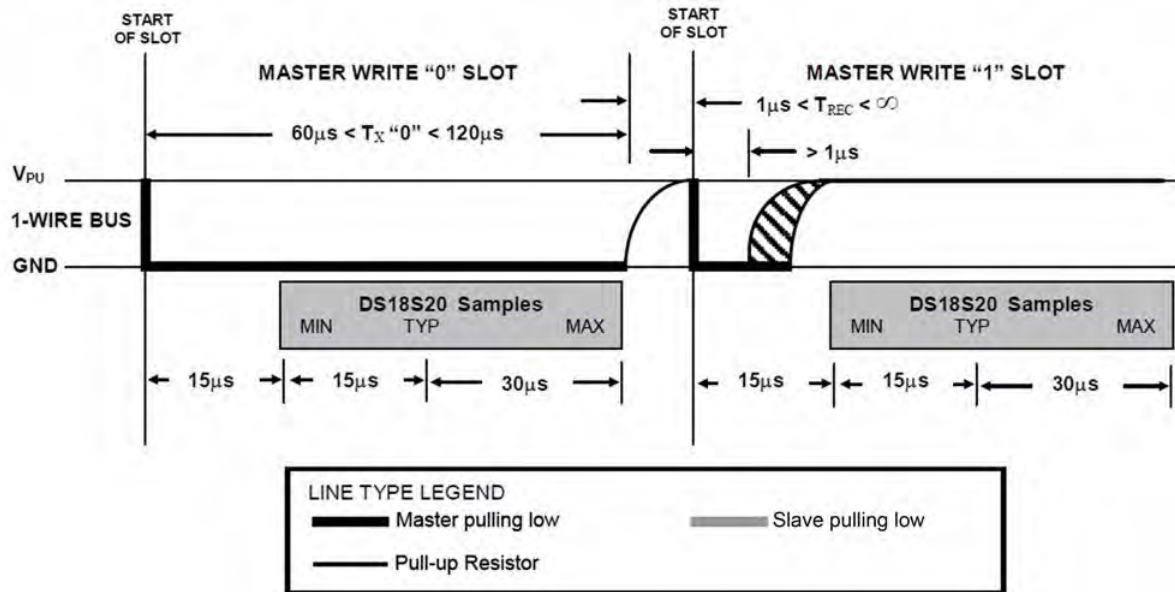
Figure 2. Write 0 and Write 1 Timing Diagram

Read Signal

A read signal is shown in Figure 3. To execute the read signal, the Master must pull the bus Low for at least 1µs. If the slave sends a 0, the slave will hold the bus Low; otherwise it will release the bus. The bus should be sampled 15µs after the bus is pulled Low. The Master reads a 0 if the data line is Low when sampled, and reads a 1 if the data line is High.

Figure 3. Read Timing Diagram

Generating Signals Using the UART

The basic signals discussed earlier in this discussion, when generated using the UART module on Zilog's Z8051 MCUs, require both the Transmitter (TXD) and Receiver (RXD) to be connected to the 1-Wire bus. Additionally, an external open-collector or open-drain buffer is required to allow the slave devices to pull the bus Low when the UART output is High. Figure 4 shows a sample buffer made of discrete components.
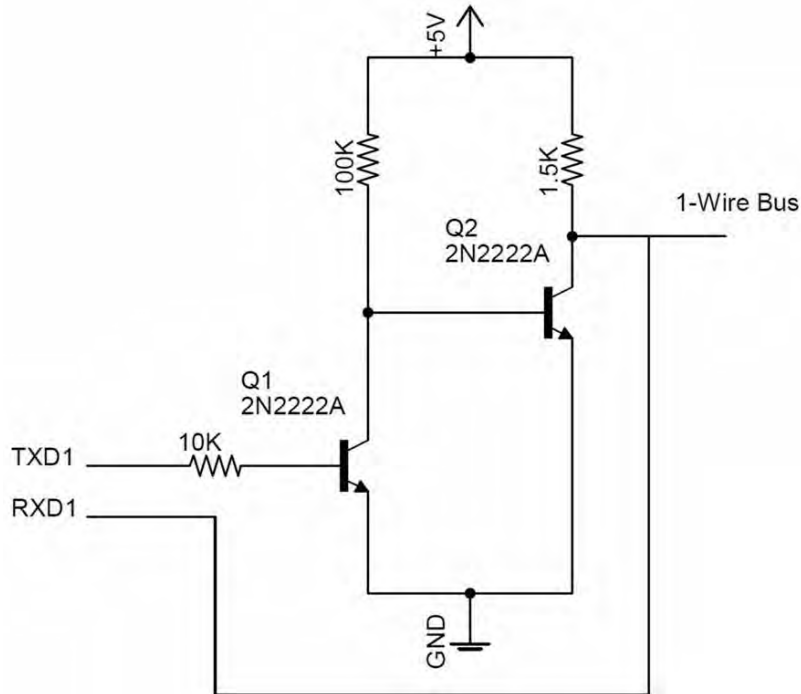
Figure 4. Sample Open-Collector Buffer Circuit

UART Configuration

For the UART module to communicate with a 1-Wire slave device, it should be configured to read and write a data format of 8 data bits, no parity and 1 stop bit; two baud rates, 9600 bps and 115200 bps, are used. Zilog recommends using an 11.0592 MHz crystal oscillator as the system clock to effect a zero-percent error on data transmission at the 115200 baud rate.

Reset and Presence Signal

To generate a Reset and Presence signal in the UART, the baud rate must be set to 9600 bps and an F0h byte must be transmitted. If the received byte is equal to the transmitted byte, there is no slave connected or present on the bus. However, if the value of the received byte ranges from 10h to 90h, one or more slaves are present. See Figure 5.
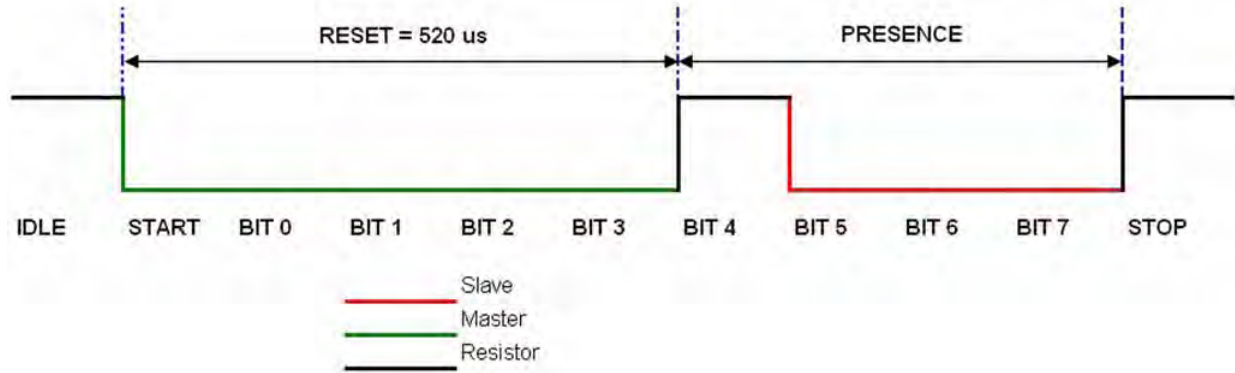
Figure 5. Reset and Presence Signal in the UART

Write 0 and Write 1 Signals

Write 0 and Write 1 signals are shown on Figure 6. Using a baud rate of 115200 bps, the UART transmitter must send `00h` for a Write 0 and `FFh` for a Write 1.
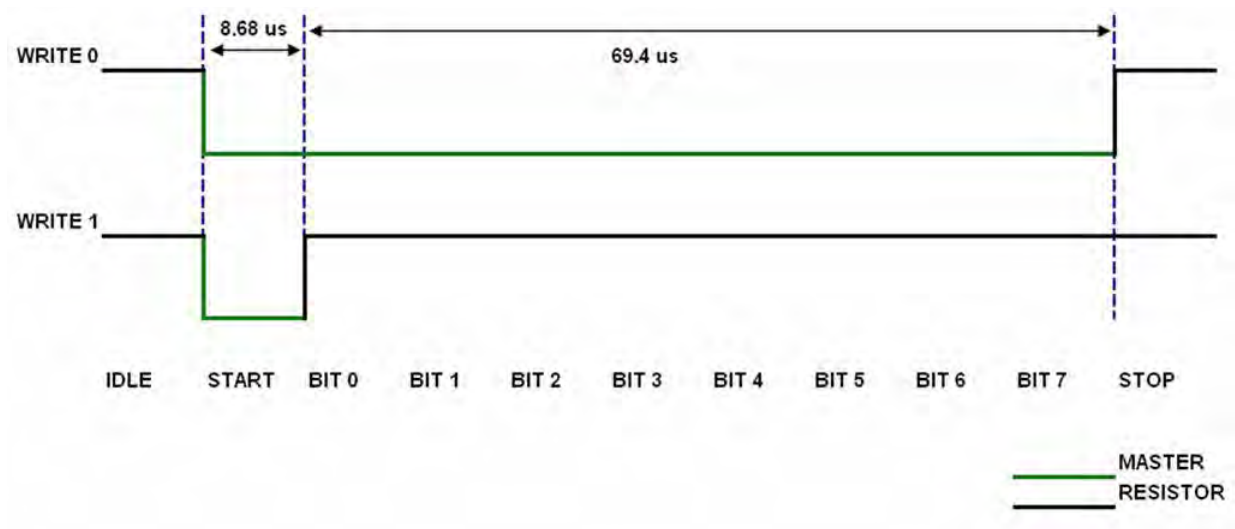

Figure 6. Write 0 and Write 1 Signals in the UART

Read Signal

Using a baud rate of 115200 bps, a read signal is generated using the UART module by transmitting `FFh` on the bus. The start bit produced by the UART tells the slave(s) that the Master is performing a read operation. If the slave should transmit a 0 bit, the slave must pull the line Low until the Master has finished sampling the LSB. If the slave did not pull the line Low, the Master will read a 1. Figure 7 illustrates the read signal in the UART.
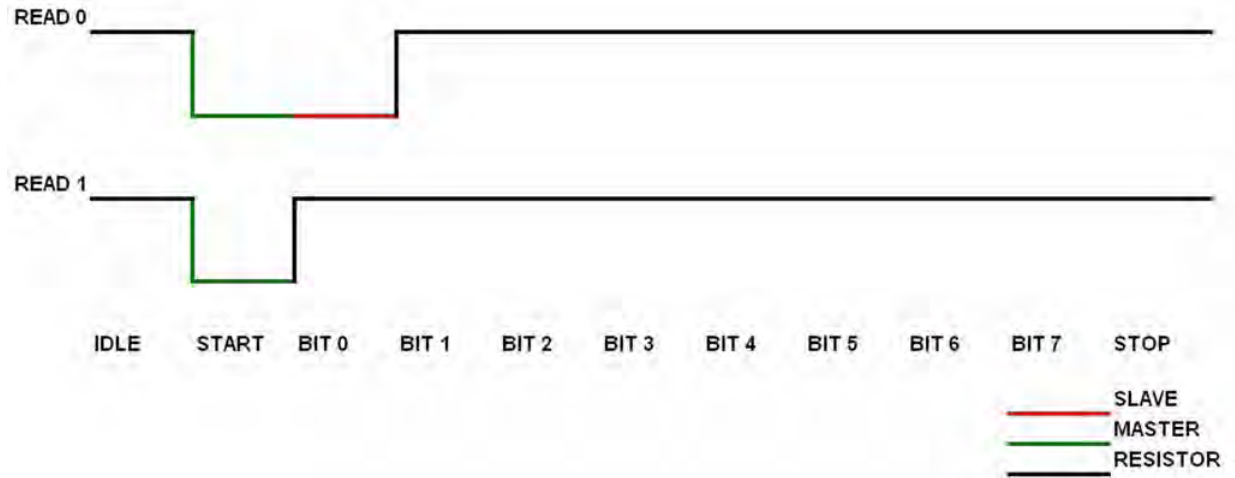
Figure 7. Read Signal in the UART

ROM Commands

All 1-Wire devices feature a unique 64-bit stored identifier in ROM. This identifier is used to facilitate addressing of each devices connected on the bus, and is divided into three parts. The first 8 bits represent the family code, the next 48 bits are for the serial number, and the final 8 bits are a CRC computed from the preceding 56 bits.[1] The ROM commands listed in Table 1 are used to operate within this 64-bit ROM space.

Table 1. ROM Commands

Command	Code	Purpose
SEARCH ROM	F0h	Identify all contents of the slave's 64-bit ROM.
MATCH ROM	55h	Communicate to a specific device.
READ ROM	33h	Identification.
SKIP ROM	CCh	Skip addressing.
OVERDRIVE SKIP ROM	3Ch	Overdrive version of SKIP ROM.
OVERDRIVE MATCH ROM	69h	Overdrive version of MATCH ROM.

Search Algorithm

A search algorithm is very important to a 1-Wire system that involves multiple slaves. If the ROM numbers of the slave devices on a 1-Wire network are not known, they can be discovered by using a search algorithm. By using this particular algorithm, the MCU can easily address which specific slave device it will communicate with. Table 2 shows the events that occur in the search algorithm.

---

1. CRC computation is outside the scope of this document.

Table 2. Master and Slave Activity in the Search Algorithm

Master	Slave(s)
1-Wire reset	Responds with a presence pulse.
Write search command	Readies for search.
Read bus for 'AND' first bit	Each slave sends bit 1 of its ROM number.
Read bus for 'AND complement bit	Each slave sends the complement of the bit 1 of its ROM number.
Master writes bit 1 direction on the bus	Each slave receives the bit written by the Master, if bit read is not the same as bit 1 of its ROM number then the Slave goes into a wait state.

The events listed in Table 2 will repeat until the 64th bit of the slave's ROM number is found. Upon examination of Table 3, it becomes evident that if all of the participating devices show the same value in a particular bit position, then there is only one direction that the path can branch to.

Table 3. Search Path Direction

Search Bit Position vs. Last Discrepancy	Path Taken
=	Take the '1' path
<	Take the same path as last time (from the last ROM number found)
>	Take the '0' path

A condition in which zeroes exist in a bit position can be termed a *discrepancy*; this discrepancy can be the key to finding devices in subsequent searches. On the first pass, the search algorithm specifies that when there is a discrepancy (bit/complement = 0/0), the 0 path is taken.

> **Note:** Another algorithm could be devised to use the 1 path first. The bit position for the last (previous) discrepancy is recorded for use in the next search. Table 3 describes the paths that are taken on subsequent searches when a discrepancy occurs.

When considering which search algorithm to employ when using three slave devices, let us first assume devices with a 2-bit ROM number only, as indicated in Figure 8.

Figure 8. 2-Bit Sample Slave ROM Number

The FIRST Operation

The search begins when the Master issues a Reset; all slave devices will respond for presence detection. The Master then sends the search command to the slaves; the search for the first device will follow. This search is also known as the *FIRST* operation.

Figures 9 through 11 present simple search examples using three slave devices. Figure 9 shows the *FIRST* operation, which searches on the 1-Wire bus for the first device. Upon initialization, Last Discrepancy is set to 0. The Master will initiate a read operation, and the slaves will respond by sending Bit 1 of their two-bit ROM number. The Master reads 0 (an AND of all of the slaves' Bit 1 data). The Master next initiates a read operation, and the slaves send the Master's Bit 1 complement. The Master reads a 0 (an AND of the bits sent by the slaves). The Master then evaluates the two bits (that were sent by the slaves) for path direction. Because the Master has received zeroes in each of these two bits, a discrepancy occurs. Looking at Table 3 on page 8, the Master should send a 0 because the bit position is 1 and is therefore greater than the discrepancy, which is 0.
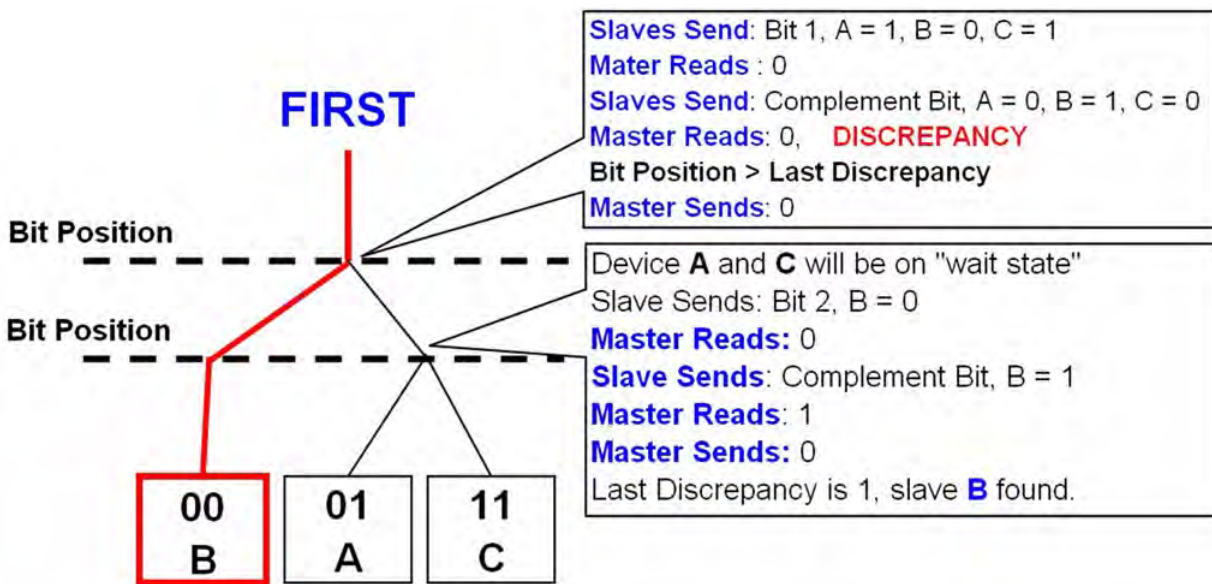

Figure 9. FIRST Operation to Find First Slave

The slaves will go into a wait state when their Bits 1 are not 0. The remaining active slave, which we'll call Slave B, should show 0 in Bit 1. The Master then initiates a read operation for Bit 2, and Slave B clears Bit 2 to 0. The Master reads Bit 2 from Slave B and reinitiates a read for the complement bit. The Master receives a 1; the bits received by the Master are 0 and 1. The Master then sends a 0, once again being the only direction that the path can branch to. The *FIRST* operation is now complete.

As a result, the identified bits are 00, which is the ROM number of Slave B. The first slave device found is B. Last Discrepancy is set to 1, because the discrepancy occurs only once in the first search.

The NEXT Operation

After finding the first device, the *NEXT* operation will occur, in which the search algorithm is executed a second time to search for the next device. The *NEXT* operation is usually performed after a *FIRST* operation or another *NEXT* operation; the state essentially remains unchanged from the previous search and before performing another search. See Figures 10 and 11.
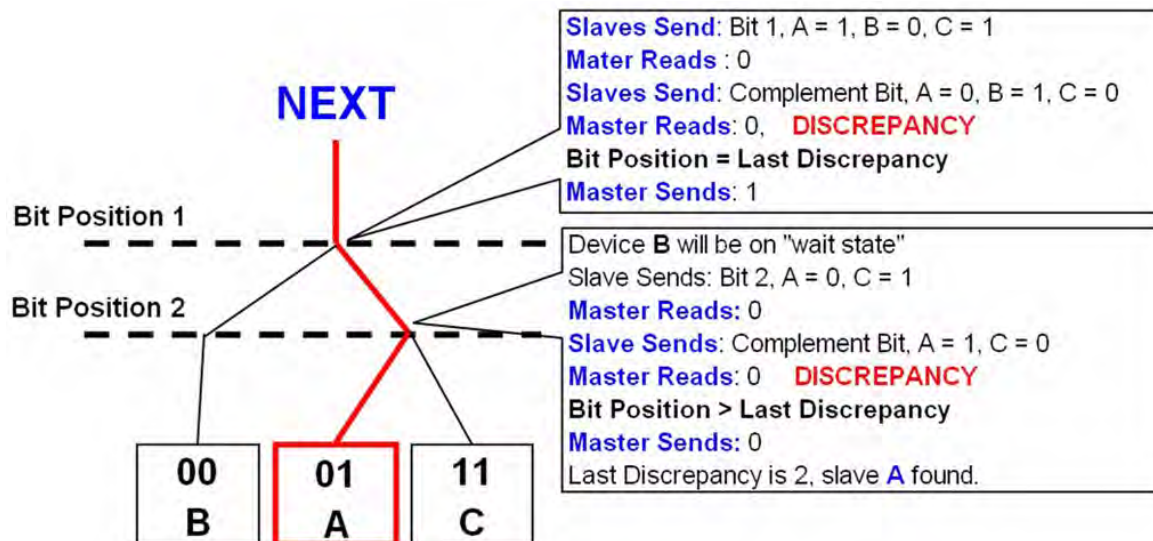

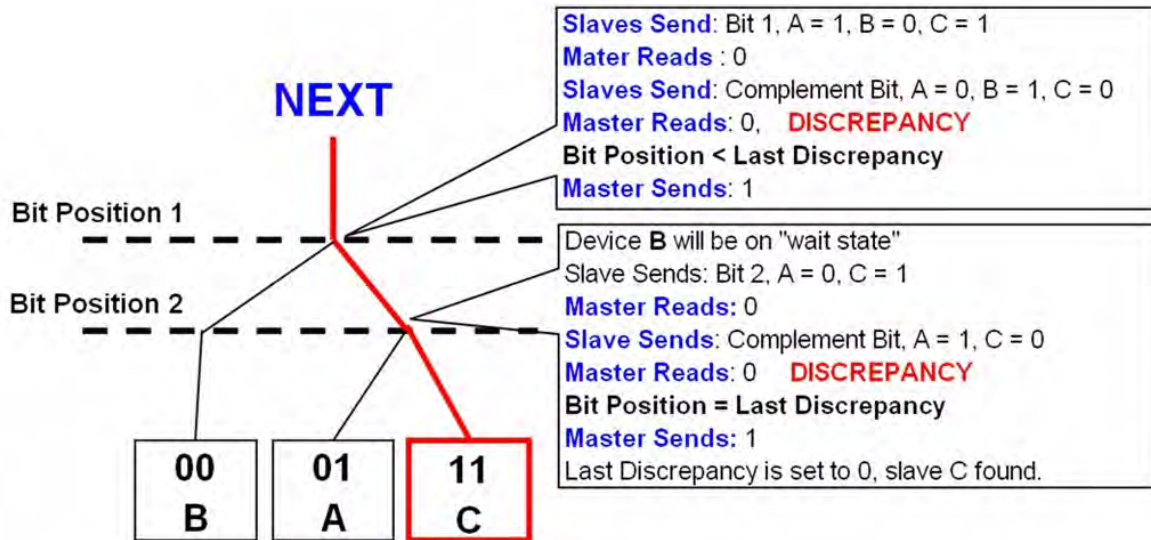**Figure 10. NEXT Operation To Find Second Slave**

Figure 11. NEXT Operation To Find Last Slave

Hardware Implementation

This section discusses the physical interface between the Z8051 MCU and the 1-Wire slave devices, and how the Z8051 MCU is configured and programmed to write and read data effectively over a 1-Wire bus with three slave devices.

For this application, a Z51F3220 MCU is used as a Master while DS18S20, DS24B33 and DS2417 are used as slaves.

As shown in Figure 12, TXD1 is connected to the open-collector buffer circuit that consists of resistors and NPN transistors. This circuit enables the slaves to pull the line Low when the UART is in an idle state.
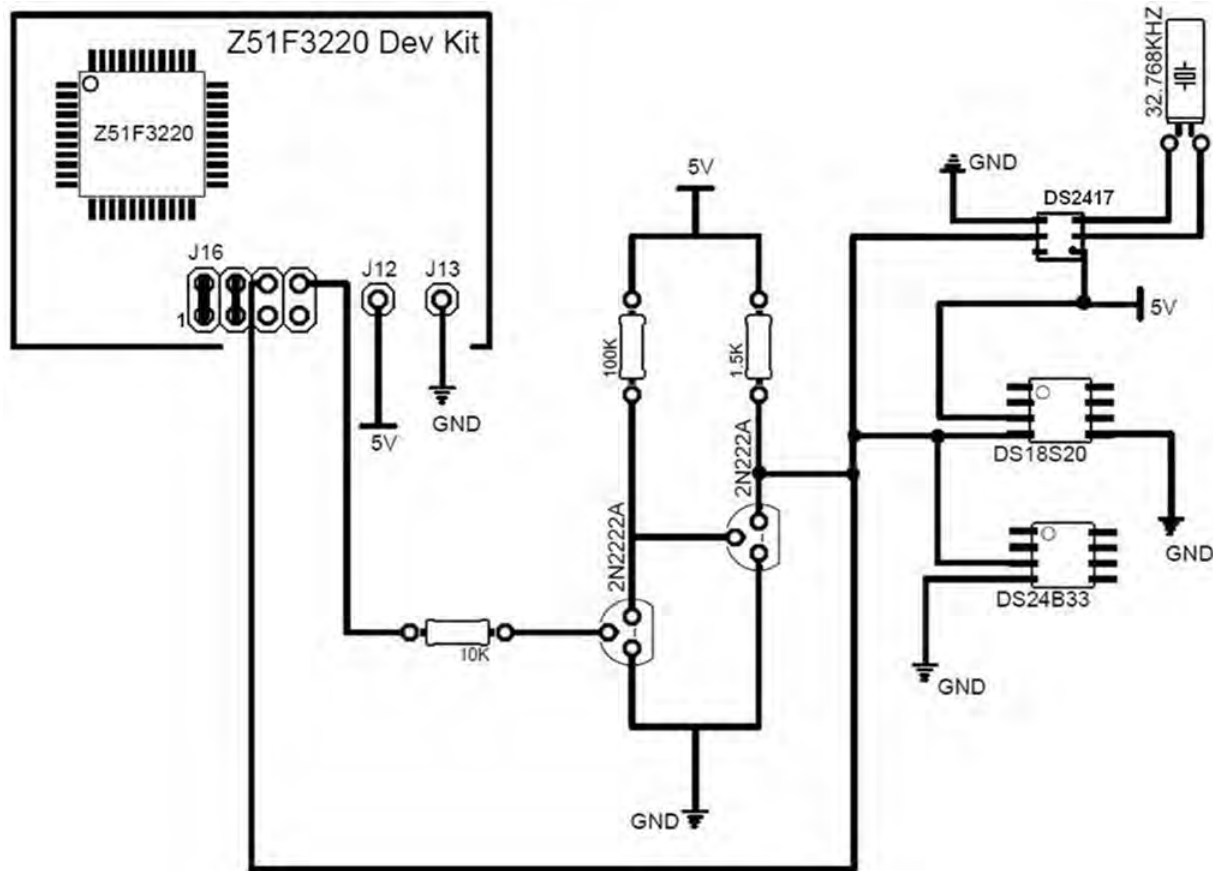
Figure 12. MCU and Slave Connection

In an ideal situation, a 1-Wire device will obtain its power and its as data in a single bus. However, some devices require additional current that may cause an unacceptable voltage drop across the weak 1-Wire pull-up resistor or require more current than can be supplied by the bus. For this reason, the DS18S20 and DS2417 slaves are powered by an external voltage supply, while the DS24B33 slave is the only one powered by the 1-Wire bus. The current in the 1-Wire bus is sufficient to power-up the DS24B33 slave during read and write operations.

Software Implementation

The required peripherals and oscillator must first be initialized; the brief routine that follows shows the order of initialization.

```
void Init(void)
{
    //Initialize Ports
```

```
    PORT_Init();

// Initialize System Clock
InitOscillator();

// Initialize UART
InitUart();

}// Init
```

The ports/pins that are used for the UART and System Clock must be configured. P20 and P10, P40 and P41 are configured to use their alternate functions as UART1 and UART0, respectively. P50 and P51 are configured as X_{OUT} and X_{IN}.

```
void PORT_Init (void)
{
  // Initialize port to use Sub functions.
  P1FSRL = RXD1EN;
  P2FSRL = TXD1EN;
  P4FSR = (RXD0EN | TXD0EN);

  // XTAL Function
  P5FSR = (XIN | XOUT);
}// PORT_Init
```

Next, the MCU is configured to use the Main External Oscillator, running at 11.0592 MHz. The proper initialization for the oscillator is to set the OSCCR Register before setting the SCCR. Not performing this initialization sequence causes the oscillator to work only when the OCD is hooked up, and not when it is unplugged.

```
void OSC_Init (        unsigned char OSCCRval,
                       unsigned char SCCRval)
{
  OSCCR = OSCCRval;
  SCCR = SCCRval;
}// OSC_Init
```

The final part of the initialization is to configure how the UART will operate. The routine that follows shows how the UART is configured.

```
void InitUart (void)
{
  UART_Init(UART0,
        FREQUENCY,DEFBAUD,
        (UART_MODE | NOPARITY | USIZE8),
        (RXCIE | TXE | RXE |USIEN),
        ~(MASTER|LOOPS|DISXCK|USISSEN|FXCH0|USISB|SITX8|
        USIRX8));
```

```
       UART_Init(UART1,
       FREQUENCY,
       DEFBAUD,
       (UART_MODE|NOPARITY|USIZE8),
       (TXE | RXE | USIEN),
       ~(MASTER|LOOPS|DISXCK|USISSEN|FXCH0|USISB|USITX8| USIRX8));
    }// InitUart
```

UART0 is configured with an initial baud rate of 9600, 8 data bits, no parity and 1 stop bit. UART1 is configured with a baud rate of 9600, 8 data bits, no parity, and with RXD1 set to operate in interrupt mode.

1-Wire Communication-Related Functions

In this application, all functions related to 1-Wire communication are written in the Z51lWireComm.c file. Table 4 shows the functions used in a 1-Wire communication.

Table 4. 1-Wire Communication-Related Functions

Function Name	Description
OW_SendDataBit (unsigned char TxBit)	Sends FFh in the bus if the TxBit is 1, 00h if it is 0. No return value.
OW_GetDataBit (void)	Gets the bit sent by the slave(s). Returns 1 if the received byte in RXD1 is 0xFF; otherwise, returns 0.
OW_ClearUSI1ST1 (void)	Clears the USI1ST1 Register of UART1. This action is particularly significant because RXD1 is tied to TXD1.
OW_SendDataByte(unsigned char* DataByte, unsigned char NumberOfBytes)	Sends bytes of data in the bus. This data is contained in the array, and the number of bytes to send is defined by the second passed parameter. Uses the OW_SendDataBit function to send these bytes one bit at a time. No return value.
OW_GetDataByte(unsigned char* DataByte, unsigned char NumberOfBytes)	Calls the OW_GetDataBit to get the bits sent by the slaves: these bits are arranged to form a byte which will then be stored in an array. The value of the second parameter determines how many bytes to get.
OW_SendFunctionCommand (unsigned char Command)	Sends the function command to the slave devices.
OW_SendROMCommand (unsigned char Command)	Sends the ROM command to the slave devices.
OW_SendROMCode(unsigned char* ROMCodes, unsigned char DeviceNumber)	Sends the 64-bit ROM number of the slave identified via the device number parameter. The 64-bit ROM number is contained in the array.
OW_GetFamilyCode (unsigned char* ROMCodes, unsigned char DeviceNumber)	Extracts the 8-bit family code from the ROM codes contained in the array. The corresponding family code required to extract is identified via the device number parameter. Returns the family code extracted.

Table 4. 1-Wire Communication-Related Functions (Continued)

Function Name	Description
OW_Reset (void)	Sends the Reset signal to the slave devices and waits for the Presence signal from the slave devices. Returns 1 if successful; 0 otherwise.
OW_Search(unsigned char* ROMCodes)	Performs the Search algorithm, then saves the 64-bit ROM numbers identified into an array and returns the number of devices found.

DS18S20 Operation-Related Functions

One of the slaves used in this application is the DS18S20 device, which is a digital thermometer that uses a single bus for data and power. The next routine, ds18s20_GetTemp, shows how to get data from this DS18S20 slave.

```
void ds18s20_GetTemp  (unsigned char *ds18s20ROMCode,
                       unsigned char *ds18s20Data,
                       unsigned char SlaveNumber)
{
  unsigned char ds18s20_Loop = 0;
  //Issue Reset and Presence Sequence
  while (! OW_Reset ());
  //Send ROM Command Match ROM
  OW_SendROMCommand (OWMATCHROM);
  //Send 64-bit ROM Code
  OW_SendROMCode (ds18s20ROMCode, SlaveNumber);
  //Send Function Command Convert Temperature
  OW_SendFunctionCommand (DS18S20CONVERTT);
  // Strobe for DS18S20 Response
  while ((OW_GetDataBit ()) == 0)
  //Delay
  for (ds18s20_Loop = 0; ds18s20_Loop <10; ds18s20_Loop)
  //Issue Reset and Presence Sequence
  while (! OW_Reset ());
  //Send ROM Command Match ROM
  OW_SendROMCommand (OWMATCHROM);
  //Send 64-bit ROM Code
  OW_SendROMCode (ds18s20ROMCode, SlaveNumber);
  //Send Function Command
  OW_SendFunctionCommand(DS18S20READSCRATCHPAD);
  // Read DS18S20 Scratch Pad for Temperature
  OW_GetDataByte(ds18s20Data, DS18S20TEMPBYTES);
  // Reset
  while (! OW_Reset ());
}// ds18s20_ConvertTemp
```

The function gets two bytes of data from the DS18S20 slave; the first byte contains temperature data. This data should be divided by two to get the temperature value (hex) in Celsius. The second byte is the sign byte. If this byte is equal to 00h, the temperature is positive. If this byte is instead FFh, the temperature is negative. This two-byte information is then stored in an array passed to the function from the main program. Figure 13 shows the two-byte data output that DS18S20 device will provide during a read operation.

Figure 13. Temperature Register Format

DS2417 Operation-Related Functions

The DS2417 device is a 1-Wire time chip with an interrupt featuring a one-second resolution, 32-bit binary counter. This slave outputs six bytes of data when it receives a read command; the first byte represents device control and the remaining five bytes represents clock data. Figure 14 shows the device control byte of the DS2417 device.
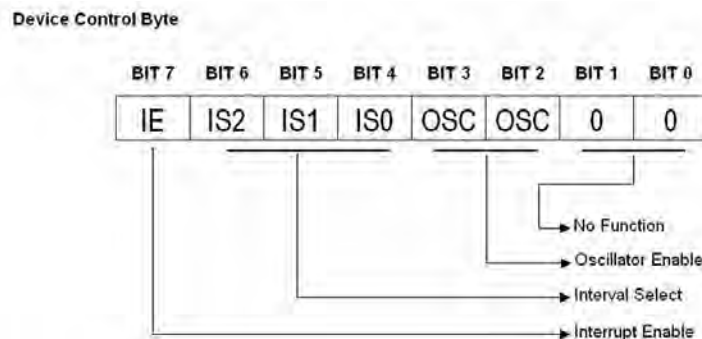
Figure 14. DS2417 Device Control Byte

The routine that follows shows how the control byte and clock are being set to their appropriate values. The control byte is set to enable the 32.768 KHz oscillator, which is connected to Pin 5 and Pin 6. The clock is next set to 0 by issuing a function command and sending four bytes of data which are all zeroes.

```
void ds2417_SetClock(unsigned char *ds2417ROMCode,
                     unsigned char SlaveNumber)
{
  unsigned char ds2417_Loop;
  unsigned char ds2417_Data[4];
```

```
// Reset
  while(!OW_Reset());
// Send ROM Command Match ROM
OW_SendROMCommand(OWMATCHROM);
// Send 64-bit ROM Code
OW_SendROMCode(ds2417ROMCode,SlaveNumber);
// Send Function Command Set Clock
  OW_SendFunctionCommand(DS2417SETCLOCK);
// Write Control Register First
  ds2417_Data[0] = DS2417CTRLBYTESET;
OW_SendDataByte (ds2417_Data, 1);
  // Write 0x00000000
for( ds2417_Loop = 1;
ds2417_Loop < DS2417TOTALBYTES;
ds2417_Loop++)
{
ds2417_Data[ds2417_Loop] = DS2417DEFAULTTIMESET;
}// for
  OW_SendDataByte (ds2417_Data,(DS2417TOTALBYTES - 1));
// Reset
  while(!OW_Reset());
 }// ds2417_SetClock
```

To get data from DS2417 a function command is also sent to the
  Slave. Then the Master initiates the read. The DS2417 will
  transmit the content of Control Register and information in
  the real-time clock counter. The following routine shows how
  the five-byte data is obtained

```
void ds2417_GetTime(unsigned char *ds2417ROMCode,
unsigned char *ds2417Data,
unsigned char SlaveNumber)
{
// Reset
while(!OW_Reset());

// Send ROM Command Match ROM
OW_SendROMCommand(OWMATCHROM);
// Send 64-bit ROM Code
OW_SendROMCode(ds2417ROMCode,SlaveNumber);
// Read Clock
OW_SendFunctionCommand(DS2417READCLOCK);
// Get Bytes
OW_GetDataByte(ds2417Data,DS2417TOTALBYTES);

// Reset
while(!OW_Reset());
}// ds2417_GetTime
```

These five bytes of data are then stored to the array passed to the function from the main program.

DS24B33 Operation-Related Functions

The DS24B33 slave is a 512-byte, 1-Wire EEPROM device. Data can be written, read, or erased in this device simply by using the 1-Wire protocol. Table 5 shows all of the functions used to communicate with this slave.

Table 5. DS24B33 Operation-Related Functions

Function Name	Description
ds24b33_WriteEEPROM(unsigned char* ds24b33ROMCode, unsigned char* ds18s20DataTemp, unsigned char* ds2417DataTime, unsigned char* ds24b33Address, unsigned char* ds24b33EndingAddress, unsigned char SlaveNumber)	Writes DS18S20 and DS2417 data to the EEPROM.
ds24b33_ReadEEPROM(unsigned char* ds24b33ROMCode, unsigned char SlaveNumber)	Reads the EEPROM.
ds24b33_EraseEEPROM(unsigned char* ds24b33ROMCode, unsigned char SlaveNumber)	Erase the EEPROM's contents; technically, write FFh to the entire contents of memory.
ds24b33_CopyScratchPad(unsigned char* ds24b33ROMCode, unsigned char SlaveNumber)	Sends a copy scratchpad function command to the slave, then sends the following three bytes of data as an authorization code: a two-byte target address and one byte for the ending address/data status.
ds24b33_WriteScratchPad(unsigned char* ds24b33ROMCode, unsigned char* ds24b33Address, unsigned char* ds24b33EndAddress, unsigned char SlaveNumber)	Sends a write scratchpad function command that writes the thirty-two-page scratchpad.
ds24b33_ReadScratchPad(unsigned char* ds24b33ROMCode, unsigned char SlaveNumber)	Reads the scratch pad.
ds24b33_TxData(unsigned char* ds24b33_DataCont1, unsigned char* ds24b33_DataCont2)	Arranges the required data to write to the EEPROM, combining the data from DS18S20 and DS2417 into one array.

Equipment Used

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

Hardware

The hardware tools used to develop this 1-Wire application are listed in Table 6.

Table 6. Application Hardware

Description	Quantity
1.5KΩ resistor	1
100KΩ resistor	1
10KΩ resistor	1
2N2222A, NPN Transistor	2
DS18S20, 1-Wire Digital Thermometer	1
DS24B33, 1-Wire 4Kb EEPROM	1
DS2417, 1-Wire Real Time Clock with Interrupt	1
32.768KHz, Crystal Oscillator	1
Connecting wires	4
Z51F3220 Development Kit	1
11.0592MHz Crystal Oscillator	1
USB-to-USB Mini Connector	1

Software

The software tools used to develop this 1-Wire application are:

- Zilog Z8051 On-Chip Debugger version 1.147

- SDCC v 3.1.0

- AN0346-SC01.zip, containing the project file and source codes.

- HyperTerminal or any equivalent communications and terminal emulation program.

Documentation

The following documents are each associated to the Z8051 MCU and/or 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)

Testing/Demonstrating the Application

This section discusses a methodology for demonstrating this application and testing the software.

Hardware Setup
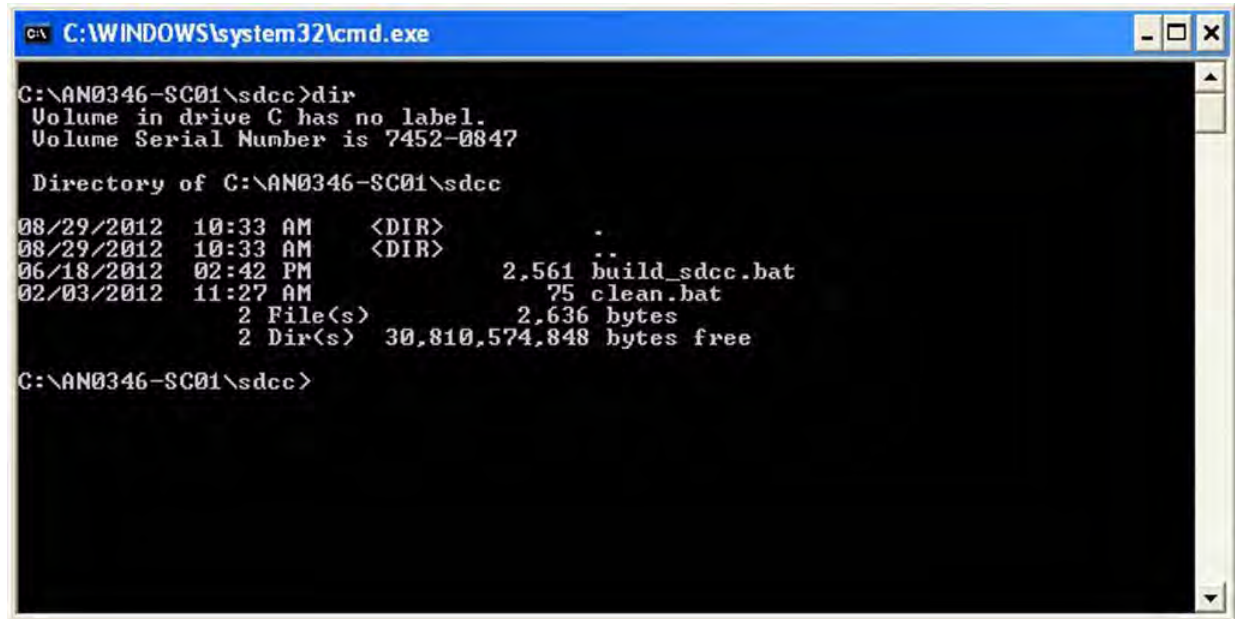
Observe the following procedure to set up the hardware.

1. Set up the open-collector buffer/driver on a protoboard; see Figure 4 on page 5 for reference.

2. Connect Pin 3 of the DS18S20 device and Pin 4 of the DS2417 device to V_{CC}.

3. Connect Pin 5 of the DS18S20 device and Pin 4 of the DS2417 to GND.

4. Connect Pin 4 of the DS18S20 device, Pin 3 of the DS24B33 device, and Pin 2 of the DS2417 device to the output of the buffer.

5. Connect the 32.768 KHz crystal oscillator to pins 5 and 6 of the DS2417 device.

6. Connect the V_{CC} of the buffer to J12, and connect the GND of the buffer to J13 on the Development Board.

7. Connect RXD1 (Pin 6 of J16) directly to the one-wire bus, and connect TXD1 (Pin 8 of J16) to the open-collector buffer. Jumpers should be placed in J16 pins 1–2 and 3–4. The following four wires must be connected between the Z51F3220 Development Board and the protoboard: V_{CC}, Ground, RXD1 and TXD1.

8. Replace the default 12 MHz crystal (Y1) on the Z51F3220 Board with an 11.0592 MHz crystal oscillator.

9. Connect the Zilog USB 2.0 OCD to J1 of the Z51F3220 Board.

10. To provide power to the Development Board, insert the USB connector into Port P1 of the Z51F3220 Board, and connect the other end of this cable to the PC.

Software Configuration

Observe the following procedure to correctly compile, load and run the software.

1. Download the AN0346-SC01.zip file from the Zilog website and unzip it to your PC's hard drive.

2. From the Windows **Start** menu, launch a command prompt. Browse to the AN0346-SC01 folder and open the Sdcc file contained inside it; see Figure 15.
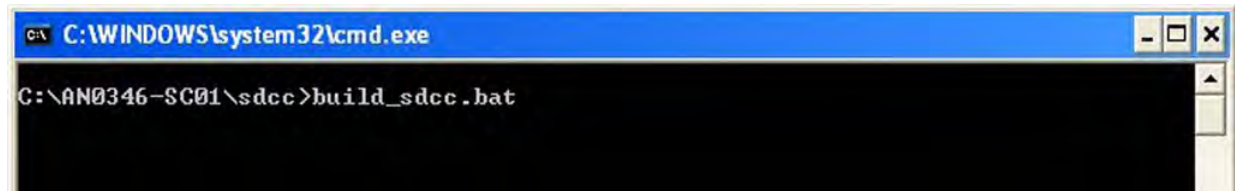
Figure 15. Opening the AN0346-SC01 Files within the Command Prompt

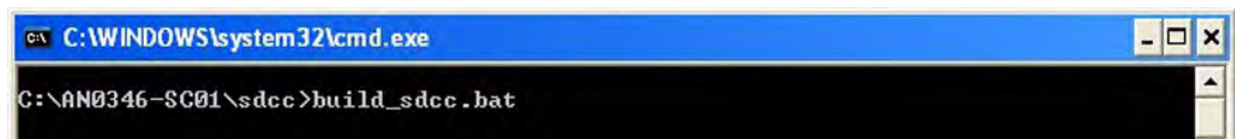3. Run the `build_sdcc` batch file to compile and link to the source code; see Figure 16.

Figure 16. Running the Build Script

4. As indicated in Figure 17, you should see a confirmation that the program has success-fully compiled and linked. A hex file has now been created and it is located in the `Sdcc` directory.
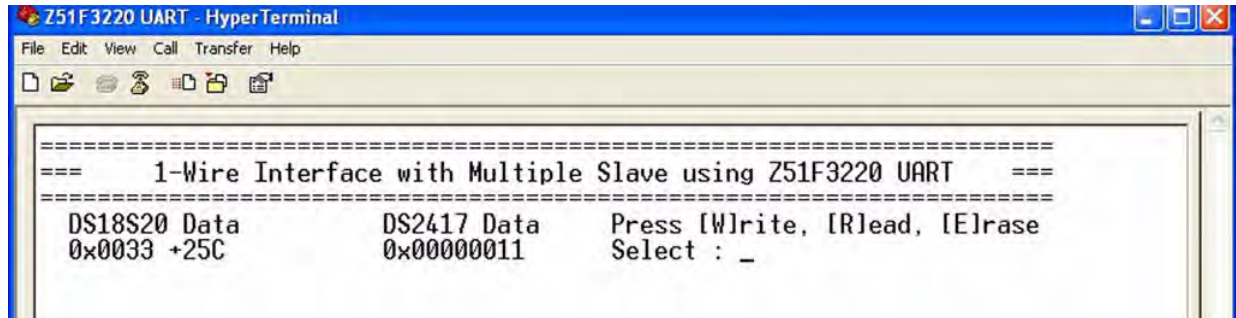

Figure 17. Build Notification

5. Next, launch the Zilog Z8051 OCD 1.147 software application to load the hex file to the MCU. For detailed instructions about loading a hex file, refer to the Z8051 Tools Product User Guide (PUG0033).

6. Disconnect the USB-to-USB mini cable and disconnect the OCD. Then, reconnect the USB-to-USB mini cable to the Z51F3220 Development Board to power it up.

7. After loading the hex file to the MCU, configure the terminal emulation program; HyperTerminal or an equivalent program can be used. Configure the terminal emulator to the following settings:

   – 9600 baud

   – 8 data bits

   – No parity bits

   – 1 stop bit

   – No flow control

8. Reset the MCU. Figure 18 shows the HyperTerminal display upon reset.
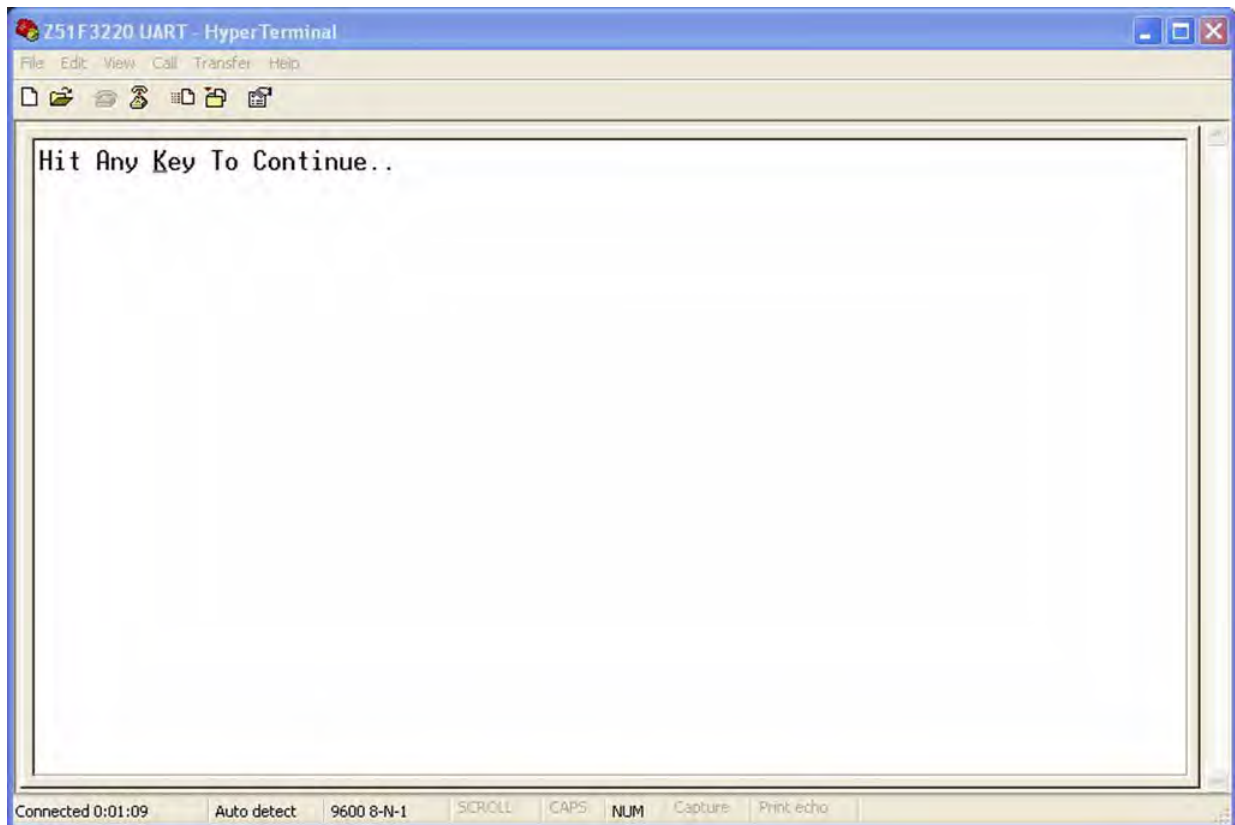
Figure 18. HyperTerminal Start-Up Display After Reset

9. Press any key on your keyboard. The image shown in Figure 19 will appear.

Figure 19. Main Program Output to HyperTerminal

10. Enter **W** to write the current DS18S20 and DS2417 data to EEPROM; see Figure 20.

Figure 20. EEPROM Write

11. Enter **R** to read the EEPROM content, as shown in Figure 21.

Figure 21. EEPROM Read

12. Enter **E** to erase the contents of the EEPROM; see Figure 22.

```
====================================================================
===        1-Wire Interface with Multiple Slave using Z51F3220 UART       ===
====================================================================
  DS18S20 Data          DS2417 Data        Press [W]rite, [R]ead, [E]rase
    0x0033 +25C           0x0000005E          Select : E
Erasing EEPROM Contents...  [OK]
```

Figure 22. EEPROM Erase

Results

The HyperTerminal display shows that the Z8051 MCU's UART module can be used as a 1-Wire master. If configured correctly, it satisfies the timing requirement of the 1-Wire protocol. For this application, the data from DS18S20 and DS2417 slave devices is successfully obtained by the 1-Wire master, and is both written to and read by the DS24B33 slave device.

Summary

This document discusses the implementation of a 1-Wire interface with multiple slaves using Zilog's Z8051 microcontroller. These slaves, the DS18S20, DS2417 and DS24B33 devices, are all Dallas Semiconductor products. The 1-Wire implementation is successful in that the transmission of data up to the bit level passes the timing requirements of the 1-Wire protocol.

References

The following documents are each associated to the Z8051 MCU and/or are available free for download from the Zilog website.

Zilog Documentation

*   Z51F3220 Product Specification (PS0299)

*   Z51F3220 Development Kit User Manual (UM0243)

*   Z8051 Tools Product User Guide (PUG0033)

*   Develop a Dallas 1-Wire Master Using the Z8F1680 Series of MCUs Application Note (AN0331)

Additional Documentation

*   Application Note 214, *Using UART to Implement a 1-Wire Bus Master*, Dallas Semiconductor, 2002

*   Application Note 187, *1-Wire Search Algorithm*, Dallas Semiconductor, 2002

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