Technical Note

Using Serial Multi-Drop with eZ80Acclaim!™ MCUs

TN001401-0603

General Overview

This Technical Note discusses setting up devices in ZiLOG's eZ80Acclaim!™ product line for 9-bit multi-drop mode. This setup consists of one master device and two or more slave devices sharing the same serial bus. Each slave or peripheral is assigned a unique address and command set. The master then sends out each unique address one packet at a time and waits for a slave response or time-out.

The companion source code file to this Technical Note is TN0014-SC01.

Discussion

Figure 1 shows how this system is configured:

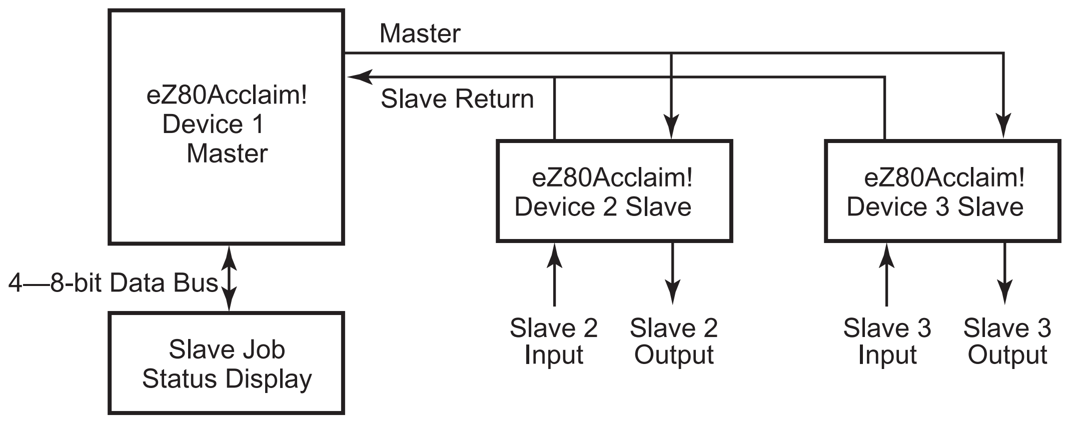

Figure 1. Multi-Drop Master/Slave Configuration

This system can be configured as a single-ended or differential interface. The single-ended configuration must be carefully considered so as to not damage the slave output drives. One consideration is to configure the slave outputs as open drain. A second is to use opto-couplers.

Both slave devices should power up and configure their serial ports in multi-drop mode. These slaves then wait and listen for their respective device addresses. The master device powers up and configures its serial port in multi-drop mode, then starts a main polling loop to each slave device for status. The master can also send down commands to a slave for it to perform. The eZ80Acclaim!™ device used for the next example is the eZ80F91 microcontroller. The core clock frequency used is 50 Mhz. The diagram in Figure 2 illustrates the code flow from the point of view of the master.

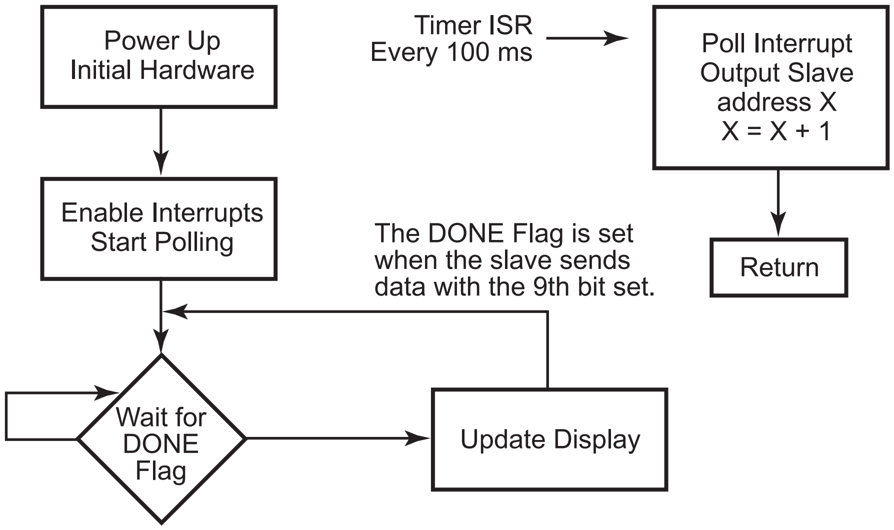

Figure 2. Multi-Drop Master/Slave Configuration

To talk to a slave device, the master sends out a command byte and a checksum byte, then waits for a return message and checksum from the slave device. The first byte sent from the master also includes a ninth bit set to tell the slave devices that this nine-bit byte is an address byte. This byte also encodes the command bytes/bits to that slave. The checksum byte is calculated by adding the command/address byte and all data bytes. The checksum byte is not included in the summation.

The following code fragment is an example of a data command from a master to a slave.

```
0x10        A ninth bit is set to tell slaves address/command byte
            The upper four bits are the slave address. The lower four
            bits are command bits.
            Command 0 is a get slave status.
0x10        The checksum byte for this simple get slave 1 status.
```

The master waits for data to return. When the slave sets the ninth bit, the interrupt routine sets the Done flag to signify to other parts of the code that a full slave message is ready. A time-out timer should be enabled to time-out slaves that are not present or are not ready. An example of a slave device sending back a status message is shown in the code fragment below.

```
0x01        The ninth bit is turned off; the slave tells the master
            that it is ready to poll status data 01.
0x01        The ninth bit is set, telling the master message complete
            with checksum 01.
```

On a time domain plot, the above transfer would resemble what is depicted in Figure 3.

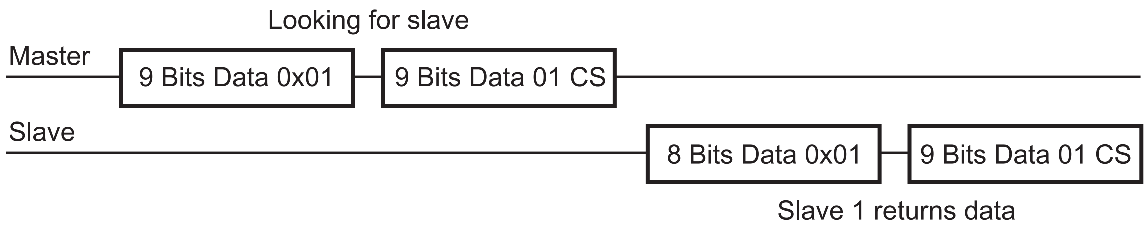

Figure 3. Multi-Drop Master/Slave Configuration

Source Code

The remainder of this TN Note lists the source code that performs this master/slave operation. This code runs within the master device. Make sure to include any other hardware or power-up routines required to bring your system up before entering "main".

```
// Define some variables

short power_on = 0;
short poll_even;
short powerup;
static volatile short byte_pos = 0;
volatile short done;
volatile int mdb_buff[36];
volatile polling_enabled;

void main(void)
{
  power_on=1;
  powerup=1;                      // Some power up flags
  poll_even = 1;                  // This bit controls what slave to poll.
                                  // 0 would be slave 1, a 1 would be slave 2.
  MDB_ACK_PENDING=0;              // This flag signals we are waiting for a
slave
                                  // to return a message
  byte_pos = 0;                   // This is a byte counter to tell how
                                  // many bytes the slave has sent us
  done = 0;                       // This tell us data is ready to read from a
                                  // slave device.
  mdb_buff[0]  = 0xff;            //
  mdb_buff[1]  = '/0';            // Buff to hold slave return bytes.
                                  // Start out with FF,NULL


// I2C_Lcd_position (LINE10, COL1);
// Routine to set the position of the cursor on the display
I2C_putstring ("Slave Status"); // Functions to display data on a
```

```
                              // LCD display Slave status

// Set up COM port 1 for 9600 baud

  init_com1();                // Int MDB com port
  init_timer1();              // 100ms Timer to poll MDB devices
  delay();                    // A little software delay

  _ei();                      // Turn on interrupt system

  powerup=0;                  // Tell everyone we are all powered up.
  polling_enabled=1;          // This turns on the ISR MDB polling routine

// At this point this "master" device will poll slave address 1 then
// slave address 2 and back to slave 1 every 100mS with the poll ISR
// routine.

  do {                        // Just do this forever
          if (done)           // Has a the slave device sent us data
                    {
          Update_Display();   // Yes update our display
          Done = 0;           // Just make sure to clear the done flag for
                              // next access
                    }

     } while(1);

} // end of main

/****************************************************************
 * Initialize timer1 to interrupt every 10ms
 *
 * 16 bit time constant is not big enough for 100ms interrupts,
 * so we will use additional intermediate counter to count
 * every 10 ticks.
 */

void init_timer1(void)
{
  ticks1 = 0x00;
  intermediate_ticks1 = 0x00;

  TMR_CTL1 = 0x00;
  TMR_RRL1 = 0xFF;            // setup timer to interrupt every 10ms
  TMR_RRH1 = 0x1F;
  TMR_CTL1 = 0x0e;            // timer0 = multipass, /16, interrupt enable
  TMR_CTL1 |= 0x01;           // enable timer
  TMR_IER1 = 0x01;
}
```

```
void init_com1(void)
{
  PC_ALT1 &= 0xf0;              // PD0 = uart0_tx, PD1 = uart0_rx
  PC_ALT2 |= 0x0F;
  UART_LCTL1=0x80;              // select dlab to access baud rate generators
  BRG_DLRL1=0x45;     /         / 9600  50M/(16*9600) = 325 = 145H
  BRG_DLRH1=0x01;
  UART_LCTL1=0x00;              // disable dlab
  UART_FCTL1=0xc7;              // clear tx fifo, clear rx fifo, fifo enable
  UART_LCTL1=0x1B;              // Say xmit 9bits, enable 9bit and set 8,1
  UART_MCTL1=0x20;              // Enable Multi drop mode
  UART_IER1=0x05;               // rx int enable, master int enable.
}

/*****************************************************************/
/*****************************************************************
 * This is the timer ISR that gets called every 10ms.
 */

#pragma interrupt
void isr_timer1(void)
{
  unsigned char temp;
  unsigned int delay;

  temp = TMR_CTL1;              //read to clear pending int
  temp = TMR_IIR1;

  intermediate_ticks1++;
  if(intermediate_ticks1 >= 10  //100mS
  {
  intermediate_ticks1 = 0;      // Reset big loop counter for
                                // next time
  ticks1++;                     // count this one
  if (polling_enabled)          // Is our Slave polling system on
  {
  if (poll_even)                // Yes then check are we reading
                                // Slave 1 or 2
  {
  poll_even = 0;                // reading slave 2 reset to read
                                // slave 1 next time
  byte_pos = 0;                 // Reset the serial data byte
                                // counter
  MDB_ACK_PENDING = 1;
  done = 0;                     // Reset the done flag
  UART_LCTL1=0x1B;              // Say xmit 9bits, enable 9bit
                                // and set 8,1
  UART_MCTL1=0x20;              // Enable Multi drop mode
  putc(0x10, uart1tx);          // Slave address 1 , command "0"
                                // get status
  putc(0x10, uart1tx);          // Check sum byte
  }
  else
  {
  poll_even = 1;
            byte_pos = 0;
  polling_bill=1;
  MDB_ACK_PENDING = 1;
  done = 0;
  UART_LCTL1=0x1B;     // Say xmit 9bits, enable 9bit and set 8,1
  UART_MCTL1=0x20;     // Enable Multi drop mode
  putc(0x20, uart1tx); // Slave address 2 , command "0" get status

  putc(0x20, uart1tx);//Check sum byte
  } //end poll even
  }  // end polling enable
  } // end if(intermediate_ticks1 >= 10)
} // end void isr_timer1(void)


/*****************************************************************
 * All this ISR should do is put the data into our internal fifos
 *
 */
#pragma interrupt
void isr_uart1(void)
{
     short temp;

     temp = UART_LSR1;

   if ( temp & 0x04 )           // If this is true then the received byte is a
                                // "address"
                                // or nine bit byte.
            {
  mdb_buff[byte_pos] = UART_RBR1;// Save the Data in our rec. buff
   byte_pos++;         // Ready for next byte to store
   done = 1;           // Tell others we have a command
                       // string ready for a slave.
  }
  if ( temp & 0x01)    // If this is true the we have
                       // just plan old 8 bit data
  {
  mdb_buff[byte_pos] = UART_RBR1;// Save the Data in our rec. buff
  byte_pos++;          // Ready for next byte to store
  }

  while( UART_LSR1 & 0x20) {    // TX int

  if( ! fifo_empty(uart1tx->fifo) )
{                               // and we still have stuff to
```

```
                              // send ...
  UART_THR1=fifo_get(uart1tx->fifo);// send it.
  }
  else
{                             // otherwise ...
  UART_IER1&=0xfd;            // disable tx interrupts
  break;
  }
  }
}

/*****************************************************************
 * Display the slave status
 *
 */
void Update_Display(void)
{
  polling_enabled=0;          // Stop polling until we get this
                              // done.

  byte_pos = byte_pos-1;      // Remove one count, ISR sets up
                              // for next byte.

  for (i=0; i<byte_pos; i++)  // Display all slave bytes
  {
  *buffy='\0';
  c = mdb_buff[i];            // read byte from buffer
  bin_to_ascii(c, buffy);     // convert to ascii
  I2C_putstring(buffy);       // output to display
  if (i == 9)                 // This display can only have 10
                              // char. On a line
   I2C_Lcd_position(col1,row4);// go to next line
  }
  byte_pos = 0;               // Reset byte counter
  polling_enabled=1;          // Reenable polling
}// end update_display

/*****************************************************************/
// This section converts the binary value c into it's ascii representation
//     in hex. It will append the two ascii characters at the end of *buff
//     If you want to save it at the start of buff, make buff[0]='\0';
//     This function will also null terminate the string.

void bin_to_ascii(unsigned char c, char *buff) {

  while(*buff)
            {
            buff++;
            }

  if( ((c >> 4) & 0x0f) <= 9) {
```

```
            *buff++ = ((c >> 4) & 0x0f) + '0';
  } else {
            *buff++ = ((c >> 4) & 0x0f) + 'A' - 10;
  }

  if( (c & 0x0f) <= 9) {
            *buff++ = (c & 0x0f) + '0';
  } else {
            *buff++ = (c & 0x0f) + 'A' - 10;
  }

  *buff='\0';
}
```

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Headquarters
532 Race Street
San Jose, CA 95126
Telephone: 408.558.8500
Fax: 408.558.8300
www.zilog.com

Information Integrity

Document Disclaimer