Application Note 120
Communicating Through the 1-Wire® Master

www.dalsemi.com

INTRODUCTION

The DS1WM 1-Wire Master was created to facilitate host CPU communication with devices over a 1-Wire bus without concern for bit timing. This application note steps through a theoretical situation that will utilize every function of the 1-Wire Master for example purposes. It is assumed the reader has knowledge of the DS18B20 thermal sensor, the DS1WM 1-Wire Master, and Dallas Semiconductor's 1-Wire protocol. For more detailed information see [1] Book of iButton Standards, [2] DS1WM Datasheet, [3] DS18B20 Datasheet, [4] Application Note 119 – Embedding the 1-Wire Master.

EXAMPLE CIRCUIT Figure 1

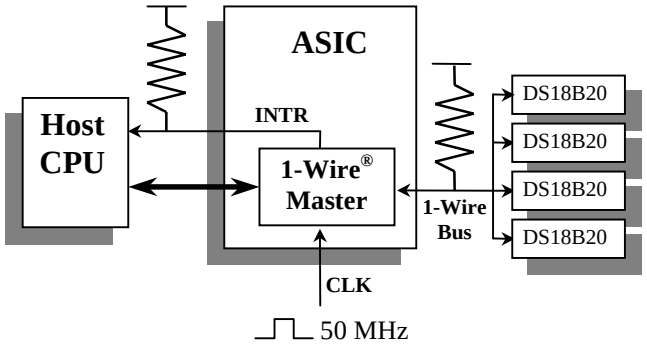

Figure 1 shows the circuit configuration this example will refer to. The 1-Wire Master has been designed into a customer ASIC and the host CPU will use it to order four thermal sensors to convert temperature and report their values. The 1-Wire bus and the INTR line each have a standard 5k pull-up resistor attached. A 50 MHz system clock drives the 1-Wire Master's timing through the CLK pin. The 1-Wire Master has been memory mapped into port address space of the CPU.

OVERVIEW

The code for this example is written in C. The main function is called GetTemperatures. It will initialize the 1-Wire Master, find all devices on the 1-Wire bus, ask those devices to measure temperature, and then store those temperature values. If no devices are present on the bus the function returns 1, it returns 0 otherwise. All other functions are described in detail below. The constant BASE in this example refers to the base address where the 1-Wire Master has been memory mapped. TEMPS and ROMS are global variables seen by all.

```
//individual Serial #s and readings
#define DEVICES 4
int ROMS[DEVICES][8];
int TEMPS[DEVICES];

int GetTemperatures(int BASE)
{
```

```
  // init. the 1-Wire Master
  Initialize(BASE);
  // exit if no devices can be found
  if(Reset(BASE)) return(1);

  // find all individual Serial#s
  FindROMs(BASE);

  // Convert temp. and read devices
  ConvertT(BASE);
  ReadTemps(BASE);

  return(0);
}
```

OPERATION

The host begins by initializing the DS1WM for proper 1-Wire bus timing by writing to the Clock Divisor Register. The value for a 50 MHz clock input is 0x0Fh (see the DS1WM datasheet). The host initializes INTR line by writing 0x3Dh to the Interrupt Enable Register. This makes the INTR pin active low and generates interrupts on data transmitted and reset complete.

```
void Initialize(int BASE)
{
  // Divide the clock
  outp(BASE+4,0x0F);

  // Generate INTs on reset and send
  outp(BASE+3,0x09);
}
```

The host must then determine if there are any devices present on the 1-Wire bus. To do this it issues a reset by writing 0x01 to the Command Register and waits for an interrupt to be generated. Upon an interrupt, the host reads the Interrupt Register and verifies that a slave device generated a presence detect by reading bit 2 to be low. If bit 2 is high, the CPU determines there is a bus error or no devices present and takes appropriate action. The WaitforInterrupt() function is not defined in this paper. It is a way for the main program to accomplish other tasks while waiting and can be defined completely by the user.

```
int Reset(int BASE)
{
  outp(BASE,0x01); // send reset
  WaitforInterrupt();

  if(inp(BASE+2) & 0x02)
    return(1); //no presence found
  else
    return(0); //presence found
}
```

The host must learn the individual ROM codes for each of the devices on the 1-Wire bus. This is accomplished by running a Search ROM algorithm. The host sends the Search ROM command to the slaves and places the 1-Wire Master in Search ROM Accelerator mode. The host transmits the 16 byte search value based on the last ROM value found. These 16 bytes are 0x00 for the first run. The 16 bytes returned contain the new ROM code and are also used to generate the next 16 bytes to transmit.

This process is repeated until serial numbers duplicate to find all devices. In this case it known that there are only four devices and since only enough memory has been allocated for four ROM codes, the loop is completed only four times. The RecoverROM function will generate the new data to transmit and extract the new ROM code from the data just received. It must be called once with a NULL pointer for the received data to initialize. The entire RecoverROM function is contained at the end of this application note. The Search ROM Accelerator process is described in detail in the DS1WM datasheet.

```
int FindROMs(int BASE)
{
  int loop;
  int dev;
  int TData[16];
  int RData[16];

  // reset RecoverROM and generate the
  // starting TData
  RecoverROM(NULL,TData,NULL);

  //run once for each device
  for(dev=0;dev<4;dev++)
  {
    outp(BASE,0x01); // send reset
    WaitforInterrupt();
    outp(BASE+1,0xF0); // send SeachROM
    WaitforInterrupt();

    // enter Accelerator mode
    outp(BASE,0x01);

    // transmit the TDATA and receive
    // the RDATA.
    for(loop=0;loop<16;loop++)
    {
      outp(BASE+1,TData[loop]);
      WaitforInterrupt();
      inp(BASE+1,RData[loop]);
    }

    //decode recovered ROM and generate
    //next Search value
    RecoverROM(RDATA,TData,ROMS[dev]);
  }
}
```

The unique serial numbers found will be used later to read out individual data from the slave devices. They are not needed to write a global command to all devices. The host orders all slaves to perform a temperature conversion by writing 0x44h. This command can be sent to all four devices simultaneously by following the skip ROM command, 0xCC, directly after a bus reset.

```
int ConvertT(int BASE)
{
  outp(BASE,0x01); // send reset
  WaitforInterrupt();

  outp(BASE+1,0xCC); // skip ROM
  WaitforInterrupt();

  outp(BASE+1,0x44); // convert Temp.
  WaitforInterrupt();
```

```
}
```

DS18B20s convert temperature fast enough that no wait time is necessary. The host must now individually address each device to read its temperature value. After resetting the bus, the host issues a Match ROM command followed by the 64-bit serial number then a Read Scratchpad command. The host then reads 2 bytes of temperature data. Remember, to read data on the 1-Wire bus, the host must transmit 0xFFh first. The 2 bytes of data are then combined to produce the complete 16 bit temperature value. This process is repeated for each device.

```
int ReadTemps(int BASE)
{
  int dev,loop;
  int LSB,MSB;

  for(dev=0;dev<4;dev++)
  {
    outp(BASE,0x01); // send reset
    WaitforInterrupt();
    outp(BASE+1,0x55); // match ROM
    WaitforInterrupt();

    // send 8 bytes of ROM code
    for(loop=0;loop<8;loop++)
    {
      outp(BASE+1,ROMS[dev][loop]);
      WaitforInterrupt();
    }

    outp(BASE+1,0xBE); // read memory
    WaitforInterrupt();

    outp(BASE+1,0xFF); // read LSB
    WaitforInterrupt();
    LSB = inp(BASE+1);

    outp(BASE+1,0xFF); // read MSB
    WaitforInterrupt();
    MSB = inp(BASE+1);

    TEMPS[dev] = MSB<<8 + LSB;
  }
}
```

The process is now complete. GetTemperatures has either returned 1 signaling no devices were found on the bus, or it updated ROMS and TEMPS with the values found from each of the four devices.

If the devices' ROM codes were known ahead of time the entire Search ROM process could have been skipped. If there was only one device on the bus, its serial number does not need to be known, a Skip ROM command could have been used for every transaction.

If no interrupt line is available, the WaitforInterrupt function could be written to poll the Interrupt Register for command completion. This action, however, ties up the CPU waiting for the 1-Wire Master to finish its operation.

RecoverROM SOURCE CODE

The source code provided on the following two pages can be included in the user's code to generate the 16 byte transmit values and extract the latest ROM code from the 16 byte received values during the Search ROM process.

The function requires pointers to a 16 byte receive data array, a 16 transmit data array (which it will write to), and an 8-byte ROM code array (which it will also write to). The function returns a 0 if there are still unknown devices on the 1-Wire bus or 1 if all devices on the bus have been found. This feature was not used in the above example.

This function has no error checking. If used in a condition where no devices are present, the ROM codes found will be incorrect.

```
//////////////////////////////////////////////////////////////////////////////
// RecoverROM performs two functions. Given 16 bytes of receive data taken from
// the 1-Wire Master during a Search ROM function, it will extract the ROM code
// found into an 8 byte array and it will generate the next 16 bytes to be trans-
// mitted during the next Search ROM.
// RecoverROM must be initialized by sending a NULL pointer in ReceivedData. It
// will write 16 bytes of zeros into TransmitData and clear the discrepancy tree.
// The discrepancy tree keeps track of which ROM discrepancies have already been
// explored.
// RecoverROM also returns a value telling whether there are any more ROM codes to
// be found. If a zero is returned, there are still discrepancies. If a one is
// returned all ROMs on the bus have been found. Running RecoverROM again in this
// case will result in repeating ROM codes already found
//////////////////////////////////////////////////////////////////////////////

int RecoverROM(int* ReceiveData, int* TransmitData, int* ROMCode)
{
       int loop;
       int result;
       int TROM[64];  // the transmit value being generated
       int RROM[64];  // the ROM recovered from the received data
       int RDIS[64];  // the discrepancy bits in the received data

       static int TREE[64]; // used to keep track of which discrepancy bits have
                            // already been flipped.

       // If receivedata is NULL, this is the first run. Transmit data should be all
       // zeros, and the discrepancy tree must also be reset.

       if(ReceiveData == NULL)
       {
              for(loop = 0; loop < 64; loop++) TREE[loop] = 0;
              for(loop = 0; loop < 16; loop++) TransmitData[loop] = 0;
              return 1;
       }

       // de-interleave the received data into the new ROM code and the discrepancy bits

       for(loop = 0; loop < 16; loop++)
       {
              if((ReceiveData[loop] & 0x02) == 0x00) RROM[loop*4]   = 0; else RROM[loop*4  ] = 1;
              if((ReceiveData[loop] & 0x08) == 0x00) RROM[loop*4+1] = 0; else RROM[loop*4+1] = 1;
              if((ReceiveData[loop] & 0x20) == 0x00) RROM[loop*4+2] = 0; else RROM[loop*4+2] = 1;
              if((ReceiveData[loop] & 0x80) == 0x00) RROM[loop*4+3] = 0; else RROM[loop*4+3] = 1;

              if((ReceiveData[loop] & 0x01) == 0x00) RDIS[loop*4]   = 0; else RDIS[loop*4  ] = 1;
              if((ReceiveData[loop] & 0x04) == 0x00) RDIS[loop*4+1] = 0; else RDIS[loop*4+1] = 1;
              if((ReceiveData[loop] & 0x10) == 0x00) RDIS[loop*4+2] = 0; else RDIS[loop*4+2] = 1;
              if((ReceiveData[loop] & 0x40) == 0x00) RDIS[loop*4+3] = 0; else RDIS[loop*4+3] = 1;
       }

       // initialize the transmit ROM to the recovered ROM

       for(loop = 0; loop < 64; loop++) TROM[loop] = RROM[loop];

       // work through the new transmit ROM backwards setting every bit to 0 until the
```

```
        // most significant discrepancy bit which has not yet been flipped is found.
        // The transmit ROM bit at that location must be flipped.

        for(loop = 63; loop >= 0; loop--)
        {
                // This is a new discrepancy bit. Set the indicator in the tree, flip the
                // transmit bit, and then break from the loop.

                if((TREE[loop] == 0) && (RDIS[loop] == 1) && (TROM[loop] == 0))
                {
                        TREE[loop] = 1;
                        TROM[loop] = 1;
                        break;
                }
                if((TREE[loop] == 0) && (RDIS[loop] == 1) && (TROM[loop] == 1))
                {
                        TREE[loop] = 1;
                        TROM[loop] = 0;
                        break;
                }

                // This bit has already been flipped, remove it from the tree and continue
                // setting the transmit bits to zero.

                if((TREE[loop] == 1) && (RDIS[loop] == 1)) TREE[loop] = 0;
                TROM[loop] = 0;
        }
        result = loop; // if loop made it to -1, there are no more discrepancy bits
                        // and the search can end.

        // Convert the individual transmit ROM bit into a 16 byte format
        // every other bit is don't care.

        for(loop = 0; loop < 16; loop++)
        {
                TransmitData[loop] = (TROM[loop*4]<<1) +
                                     (TROM[loop*4+1]<<3) +
                                     (TROM[loop*4+2]<<5) +
                                     (TROM[loop*4+3]<<7);
        }

        // Convert the individual recovered ROM bits into an 8 byte format

        for(loop = 0; loop < 8; loop++)
        {
                ROMCode[loop] = (RROM[loop*8]) +
                                (RROM[loop*8+1]<<1) +
                                (RROM[loop*8+2]<<2) +
                                (RROM[loop*8+3]<<3) +
                                (RROM[loop*8+4]<<4) +
                                (RROM[loop*8+5]<<5) +
                                (RROM[loop*8+6]<<6) +
                                (RROM[loop*8+7]<<7);
        }

        if(result == -1) return 1;    // There are no DIS bits that haven't been flipped
                                      // Tell the main loop the seach is over
        return 0; // else continue
}
```

REFERENCES:

[1] **Book of iButton Standards**, Dallas Semiconductor, online at http://www.ibutton.com/iButtons/standard.pdf
[2] **DS1WM Datasheet**, Dallas Semiconductor, online at http://www.dalsemi.com/DocControl/PDFs/1WM.pdf
[3] **DS18B20 Datasheet**, Dallas Semiconductor, online at http://www.dalsemi.com/DocControl/PDFs/18b20.pdf
[4] **Application Note 119 – Embedding the 1-Wire Master**, Dallas Semiconductor, online at http://www.dalsemi.com/DocControl/PDFs/app119.pdf