Keywords: i2c, 1-wire, bridge, ds2482, line driver, guide, slew, waveforms, master

Dec 22, 2005

APPLICATION NOTE 3684

How to Use the DS2482 I²C 1-Wire® Master

Abstract: The DS2482 is an I²C bridge to the 1-Wire network protocol. As a bridge, the DS2482 allows any host with I²C communication to generate properly timed and slew-controlled 1-Wire waveforms. This Application Note is a user's guide for the DS2482 I²C 1-Wire Line Driver, and provides detailed communication sessions for common 1-Wire master operations.

1. Introduction

The 1-Wire communication protocol can be generated using the DS2482, which is a bridge for I²C communication to a 1-Wire network. This bridge allows any host with I²C to generate properly timed 1-Wire waveforms. See **Figure 1** for a simplified diagram of the DS2482 configuration. Implementing this protocol and navigating the available DS2482 commands can be time-consuming and confusing. This document presents an efficient implementation of the basic and extended 1-Wire operations using the DS2482. The construction of I²C input packets to handle 1-Wire communication is explained. These operations provide a complete foundation to perform all the functions for current and future 1-Wire devices. Abstracting the 1-Wire operations in this fashion leads to 1-Wire applications that are independent of the 1-Wire master type.

This document complements the DS2482 data sheet, but does not replace it. The DS2482 is available in two configurations, a single-channel 1-Wire master (DS2482-100) and an eight-channel 1-Wire master (DS2482-800).

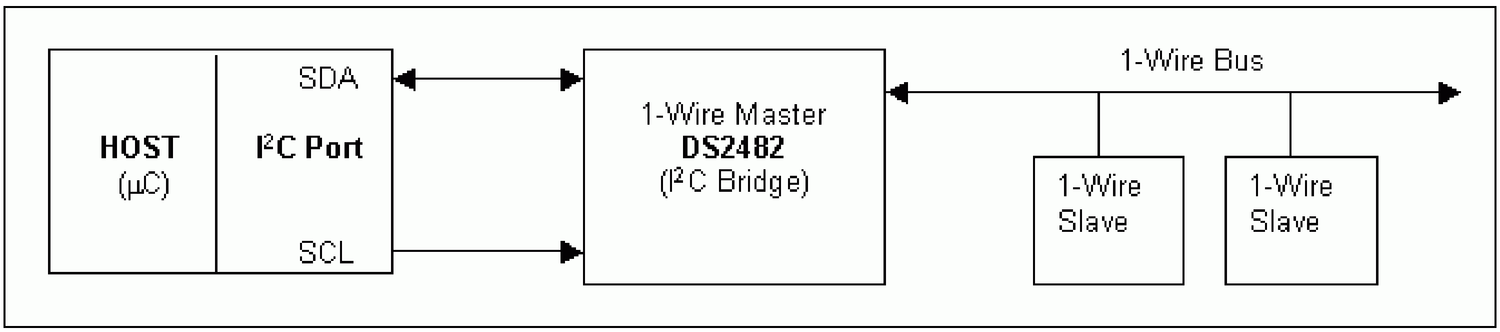

Figure 1. Simplified illustration of DS2482 function as a bridge for I²C communication and a 1-Wire network.

2. The 1-Wire Interface

There are a few basic 1-Wire functions, called primitives, which an application must have in order to perform any 1-Wire operation. This first function resets all the 1-Wire slaves on the bus, readying them for a command from the 1-Wire master. The second function writes a bit from the 1-Wire master to the slaves, and the third reads a bit from the 1-Wire slaves. Since the 1-Wire master must start all 1-Wire bit communication, a 'read' is technically a 'write' of a single bit with the result sampled. Almost all other 1-Wire operations can be constructed from these three operations. For example, a byte written to the 1-Wire bus is just eight single bit writes.

The 1-Wire Search Algorithm can also be constructed using these same three primitives. The DS2482 incorporates a search using the 1-Wire triplet command, which greatly reduces the communication required to do a search.

Table 1 shows the three basic primitives (OWReset, OWWriteBit/OWReadBit, and OWWriteByte/OWReadByte), along with three other useful functions (OWBlock, OWSearch, msDelay) that together make up a core set of basic 1-Wire operations. These operation names will be used throughout the remainder of this document.

Table 1. Basic 1-Wire Operations

Operation	Description
OWReset	Sends the 1-Wire reset stimulus and check for pulses of 1-Wire slave devices.
OWWriteBit/OWReadBit	Sends or receives a single bit of data to the 1-Wire bus.
OWWriteByte/OWReadByte	Sends or receives a single byte of data to the 1-Wire bus.
OWBlock	Sends and receives multiple bytes of data to and from the 1-Wire bus.
OWSearch	Performs the 1-Wire Search Algorithm (see Application Note 187 mentioned above).
msDelay	Delays at least the specified number of milliseconds.

Extended 1-Wire functions (such as overdrive communication functions) are not covered in the basic operations in the table above. Some 1-Wire slave devices can operate at two different communication speeds: standard and overdrive. All devices support the standard speed; overdrive is approximately 10 times faster than standard. The DS2482 supports both 1-Wire speeds.

1-Wire devices normally derive some, or all their operating energy from the 1-Wire bus. Some devices, however, require additional power delivery at a particular place in the protocol. For example, a device may need to do a temperature conversion or compute an SHA-1 hash. The power for this action is supplied by enabling a stronger pullup on the 1-Wire bus. Normal communication cannot occur during this power delivery. The DS2482 delivers power by setting the Strong Pullup (SPU) flag, which will issue a strong pullup after the next byte/bit of 1-Wire communication. The DS2482-100 has an external pin (PCTLZ) to control a supplemental high-current strong pullup.

Table 2 lists the extended 1-Wire operations for 1-Wire speed, power delivery, and programming pulse.

Table 2. Extended 1-Wire Operations

Operation	Description
OWSpeed	Sets the 1-Wire communication speed, either standard or overdrive. Note that this only changes the communication speed of the 1-Wire master; the 1-Wire slave device must be instructed to make the switch when going from normal to overdrive. The 1-Wire slave will always revert to standard speed when it encounters a standard-speed 1-Wire reset.
OWLevel	Sets the 1-Wire power level (normal or power delivery).
OWReadBitPower	Reads a single bit of data from the 1-Wire bus and optionally applies power delivery immediately after the bit is complete.
OWWriteBytePower	Sends a single byte of data to the 1-Wire bus and applies power delivery immediately after the byte is complete.

3. Host Configuration

The host of the DS2482 must have an I²C communication port. Configuration of the host is not covered by this document. The host must, however, provide standard interface I²C operations. The required operations can be seen in **Table 3**.

Table 3. Required I²C Host Operations

Operation	Description
InitI2C	Sets the I²C communication speed and selects the DS2482 device. The I2C_clock_delay is the time between clock pulses for I²C communication. The DS2482_slave_address is the I²C address for the DS2482.
I2CBus_write	Writes an I²C byte to the selected DS2482. The byte is passed to the function to write.
I2CBus_write_packet	Writes a packet of I²C bytes to the selected DS2482. The buffer of bytes along with the length of the buffer is passed to the function.
I2CBus_read	Reads an I²C byte from the DS2482. The byte that was read is returned.

3.1. DS2482 Configuration

Before any 1-Wire operations can be attempted, the host must set up and synchronize with the DS2482 I²C 1-Wire line driver. To communicate with the DS2482, the slave address must be known. **Figure 2** shows the slave address for the DS2482-100 and DS2482-800.

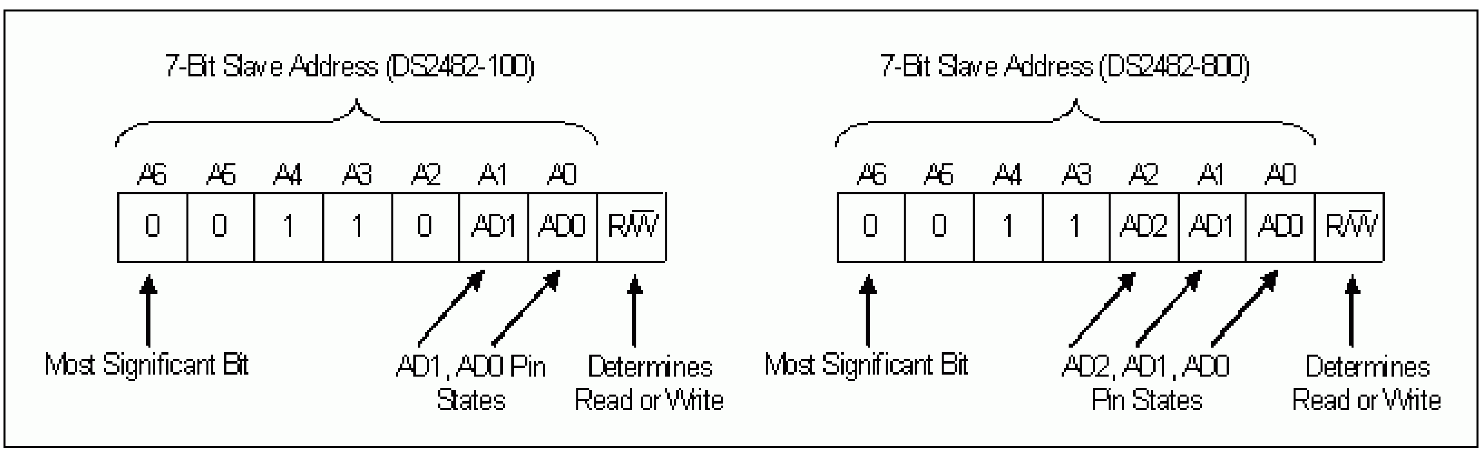

Figure 2. DS2482 I²C slave addresses.

3.2. DS2482 I²C Commands

The following legend comes from the DS2482 data sheet and represents a short-hand notation to describe the I²C communication sequences with the device. As we proceed, we will repeat these communication sequences and provide additional explanation and C code examples for implementing the basic and extended 1-Wire operations.

I²C Communication Sequences—Legend

SYMBOL	DESCRIPTION
S	START Condition
AD, 0	Select DS2482 for Write Access
AD, 1	Select DS2482 for Read Access
Sr	Repeated START Condition
P	STOP Condition
A	Acknowledged
A\	Not acknowledged
(Idle)	Bus not busy
<byte>	Transfer of one byte
DRST	Command 'Device Reset', F0h
WCFG	Command 'Write Configuration', D2h
SRP	Command 'Set Read Pointer', E1h
1WRS	Command '1-Wire Reset', B4h
1WWB	Command '1-Wire Write Byte', A5h
1WRB	Command '1-Wire Read Byte', 96h
1WSB	Command '1-Wire Single Bit', 87h
1WT	Command '1-Wire Triplet', 78h

3.3. Data Direction Codes

Master-to-Slave	Slave-to-Master

The data direction codes found in many of the Figures in this document show communication either from the master to the slave (grey) or vice-versa, from the slave to the master (white). By looking at the shading of each code, the communication direction can be established.

4. Device Reset

Figure 3 is the Device Reset I²C communication example. Reset **Example 1** shows the DS2482 reset command, which performs a global reset of the device state-machine logic and terminates any ongoing 1-Wire communication. The command code for the device reset is 0xF0.

S	AD,0	A	DRST	A	Sr	AD,1	A	<byte>	A\	P

Figure 3. Device reset after power-up. This example includes an optional read access to verify the success of the command.

```
// initialize the I2C port
if(!InitI2C(I2C_clock_delay,DS2482_slave_address))
{
   // Report an error that occurred
}

// reset the DS2482
// DRST is 0xF0
if(!I2CBus_write(DRST))
{
   // Report an error that occurred
}
```

Example 1. Reset device code.

5. DS2482 1-Wire Operations

These are the commands sent to the DS2482 that affect 1-Wire communication.

5.1. OWReset

The Reset command (0xB4) generates a 1-Wire Reset/Presence Detect at the 1-Wire line. The state of the 1-Wire line is sampled and reported through the Presence-Pulse Detect (PPD) and the Short Detected (SD) fields in the status register. **Figure 4** shows I²C communication for the 1-Wire Reset command. **Example 2** shows the command sent and status register checked for a presence pulse.

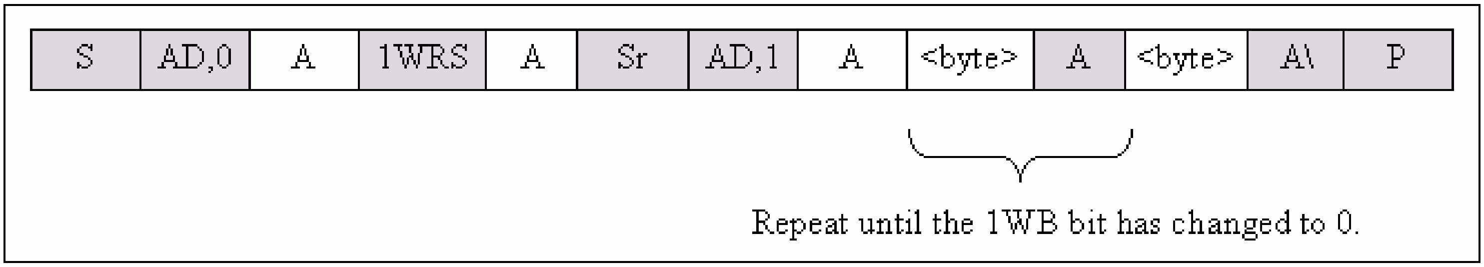

Figure 4. 1-Wire reset. Begins or ends 1-Wire communication. 1-Wire Idle (1WB = 0), Busy polling until the 1-Wire command is completed, then read the result.

```
// OWReset
//
// Resets the 1-Wire using I2C through the DS2482.
//
// returns Success or Failure
//
uchar OWReset()
{
   uchar buffer;
   uchar test;

   // reset the 1-Wire line
   // resetOneWireCommand is 0xB4
   if(!I2CBus_write(resetOneWireCommand))
   {
      // Report an error that occurred
   }

   for(;;)// checking if 1-Wire busy
   {
      // checking LSB of status register
      // to see if 1-Wire is busy.
      test = I2CBus_read()| 0xFE;
      if(test == 0xFE)
      {break;}
   }

   // checking for presence pulse detect
   test = I2CBus_read() | 0xFC;
   if(test == 0xFE)
   {
      return Success;   // Presence Pulse found
   }
   else
   {
      return Failure;   // No presence pulse
   }
}
```

Example 2. OWReset code.

5.2. OWWriteBit / OWReadBit

The 1-Wire bit command (0x87) generates a single 1-Wire bit time slot. **Figure 5** shows the I²C communication code for the 1-Wire Single Bit command cases. **Figure 6** is the bit allocation byte where if V is 1b, then a write-one time slot is generated; if V is 0b, a write-zero time slot is generated. **Example 3** shows OWWriteBit code and **Example 4** shows OWReadBit code.

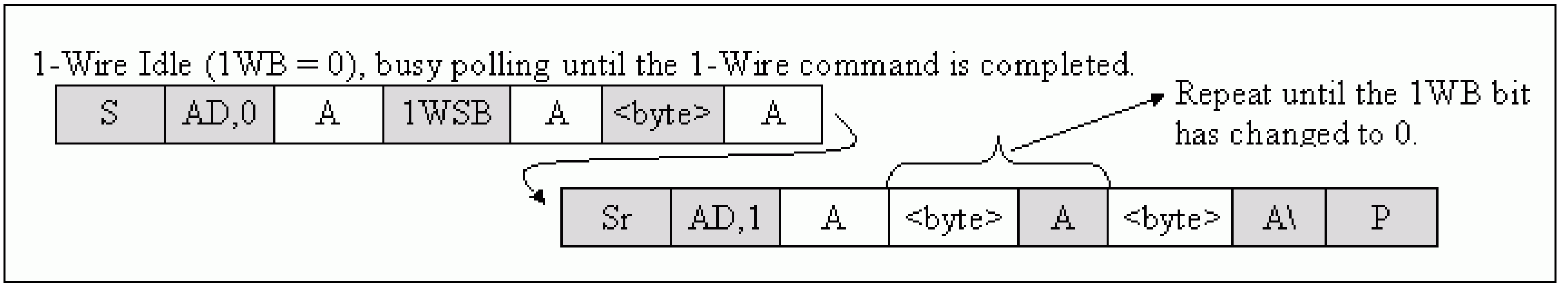

Figure 5. 1-Wire Single Bit. Generates a single time slot on the 1-Wire line. When 1WB has changed from 1 to 0, the Status register holds the valid result of the 1-Wire Single Bit command.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
V	x	x	x	x	x	x	x

x = do not care

Figure 6. 1-Wire Single Bit. Generates a single time slot on the 1-Wire line.

```
// OWWriteBit - Writes a single bit to 1-Wire using the DS2482.
//
// value - bit to be written to the 1-Wire (0 or 1)
//
// Return Success or Failure
//
uchar OWWriteBit(uchar value)
{
   uchar buff[3];
   uchar test;

   buff[0] = onewireBitCommand;   // 1-Wire bit command

   if(value)
   {
      buff[1] = 0xFF;
   }
   else
   {
      buff[1] = 0x7F;
   }

   if(!I2CBus_write(&buff[0],2))
   {
      return FAILURE;
   }

   // checking if 1-Wire busy
   // Check here to make sure the 1-Wire isn't
   // busy so other commands don't have to check
   // before proceeding.
   for(;;)
   {
      if(!I2CBus_read(&buff[0],1))
      {
         return FAILURE;
      }

      test = buff[0] | 0xFE;

          if(test == 0xFE)
          {
             break;
          }
   }

   return SUCCESS;
}
```

Example 3. OWWriteBit code.

```
// OWReadBit
//
// Returns 0 or 1 for the bit read.
uchar OWReadBit()
{
   uchar buff[3];

   OWWriteBit(1);

   buff[0] = setReadPointerCommand;
   buff[1] = statusRegister;

   if(!I2CBus_write(&buff[0],2))
   {
      // Report an error that occurred
   }
   else if(!I2CBus_read(&buff[2],1))
   {
        // Report an error that occurred
   }

   if(buff[2] & 0x20)
   {
      return 1;
   }
   else
   {
      return 0;
   }
}
```

Example 4. OWReadBit code.

5.3. OWWriteByte

The 1-Wire write byte command (0xA5) writes a single data byte to the 1-Wire line. 1-Wire activity must have ended before the DS2482 can process this command. **Figure 7** shows the I²C write 1-Wire byte case. Code **Example 5** checks 1-Wire activity before issuing the write byte command.

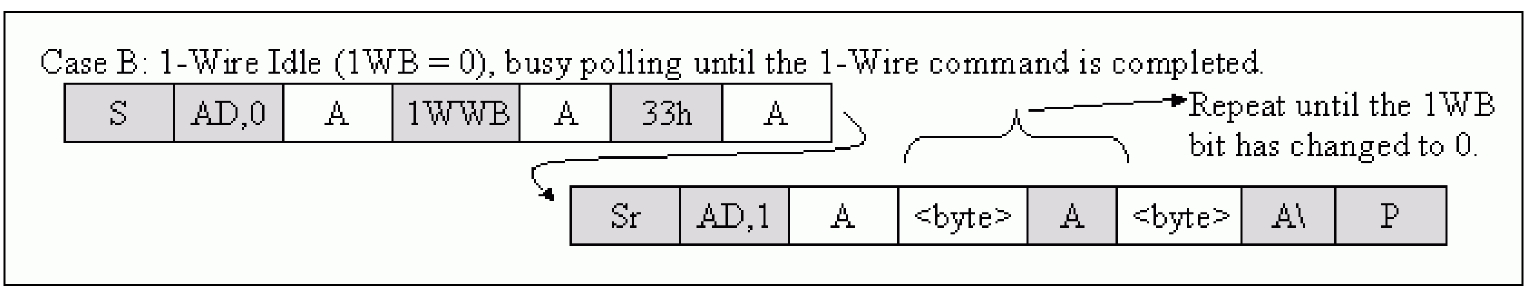

Figure 7. 1-Wire Write Byte. Sends a command code to the 1-Wire line. When 1WB has changed from 1 to 0, the 1-Wire Write Byte command is completed.

```
// OWWriteByte
//
// Writes a 1-Wire byte using I2C commands sent to the DS2482.
//
// wrByte - The byte to be written to the 1-Wire.
//
void OWWriteByte(uchar wrByte)
{
   uchar buffer[2];
   uchar test;

   // set the read pointer to the status
   // register to check 1-Wire busy
   buffer[0] = setReadPointerCommand;  // 0xE1
   buffer[1] = statusRegister;         // 0xF0

   if(!I2CBus_write(buffer,2))
```

```
   {
      // Report an error that occurred
   }

   // checking if 1-Wire busy
   for(;;)
   {
      test = I2CBus_read() | 0xFE;

      if(test == 0xFE)
      {
           break;
      }
   }

   buffer[0] = writeByteCommand;  // 0xA5
   buffer[1] = wrByte;

   if(!I2CBus_write_packet(buffer,2))
   {
      // Report an error that occurred
   }

   // checking if 1-Wire busy
   for(;;)
   {
      test = I2CBus_read() | 0xFE;

      if(test == 0xFE)
      {
           break;
      }
   }
}
```

Example 5. OWWriteByte code.

5.4. OWReadByte

The 1-Wire read byte command (0x96) reads a single data byte to the 1-Wire line. 1-Wire activity must have ended before the DS2482 can process this command. **Figure 8** shows the I²C case. Code for a 1-Wire Read Byte Command can be found in Code **Example 6**. The 1-Wire activity is checked before issuing the read byte command.

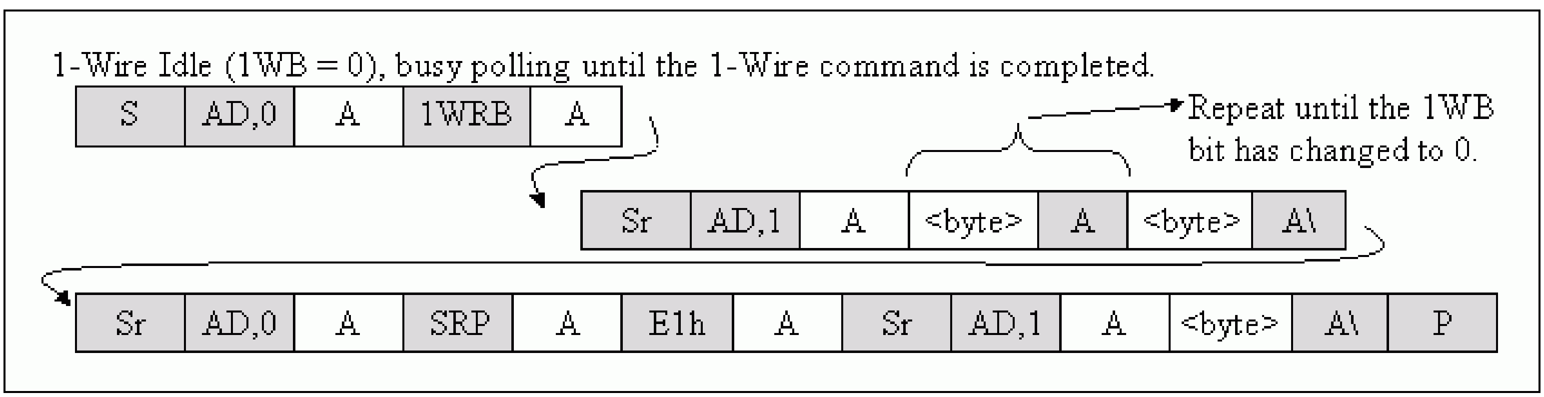

Figure 8. 1-Wire Read Byte. Reads a byte from the 1-Wire line. Poll the Status register until the 1WB bit has changed from 1 to 0. Then set the read pointer to the Read Data register (code E1h) and access the device again to read the data byte obtained from the 1-Wire line.

```
// OWReadByte
//
// Reads a 1-Wire byte using I2C commands to the DS2482.
//
// returns the byte read
//
uchar OWReadByte
{
   uchar buffer[2];
   uchar test;
```

```
   // set the read pointer to the status
   // register to check 1-Wire busy
   buffer[0] = setReadPointerCommand;  // 0xE1
   buffer[1] = statusRegister;         // 0xF0

   if(!I2CBus_write_packet(buffer,2))
   {
      // Report an error that occurred
   }

   // checking if 1-Wire busy
   for(;;)
   {
      test = I2CBus_read() | 0xFE;

      if(test == 0xFE)
      {
           break;
      }
   }

   // readByteCommand is 0x96
   if(!I2CBus_write(readByteCommand))
   {
      // Report an error that occurred
   }

   buffer[0] = setReadPointerCommand;  // 0xE1
   buffer[1] = readDataRegister;       // 0xE1

   if(!I2CBus_write_packet(buffer,2))
   {
      // Report an error that occurred
   }
   // gets the byte that was read
   else
   {
      buffer[2] = I2CBus_read();
   }

   return buffer[2];
}
```

Example 6. OWReadByte code.

5.5. OWBlock

The OWBlock operation is just calling the byte operations since a block of data cannot be transferred without using the byte commands. **Example 7** shows a code example of OWBlock.

```
// OWBlock – writes/reads a block of data
// block – block of data
// return – Success or failure of the operation.
uchar OWBlock (uchar *block, uchar len)
{
   uchar buffer[2];
   int i;

   for(i=0;i<len;i++)
   {
      owWriteByte(block[i]);

      buffer[0] = setReadPointerCommand;    // The read pointer value is 0xE1
      buffer[1] = readDataRegister;         // The read data register value is 0xE1

      if(!I2CBus_write(&buffer[0],2))
      {
         return FAILURE;
      }
      else
```

```
      block[i] = I2CBus_read();
   }

   return SUCCESS;
}
```

Example 7. OWBlock code.

5.6 OWSearch / 1-WIRE Triplet Command

The Triplet command (0x78) generates three time slots, two read time slots, and one write time slot on the 1-Wire line. The direction byte (DIR) determines the type of write time slot (**Figure 9**). **Example 8** illustrates the 1-Wire Triplet command using the search command with only one device attached. For an explanation of the 1-Wire search algorithm, see Application Note 187 (cited above) which shows the I²C setup for a 1-Wire Triplet command.

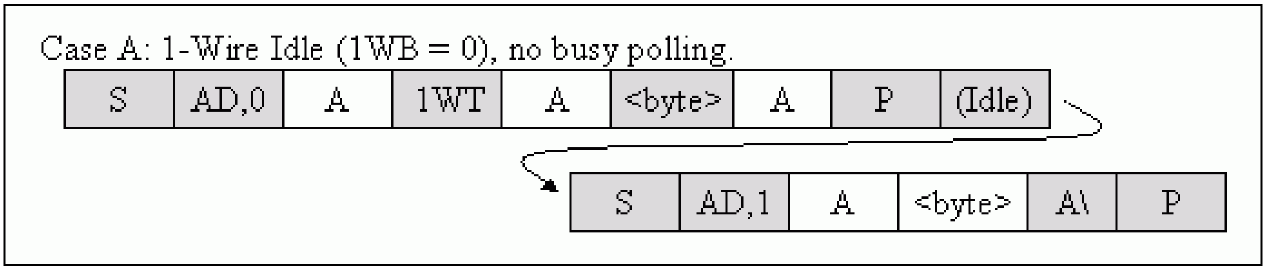

Figure 9. 1-Wire Triplet. Performs a Search ROM function on the 1-Wire line. The idle time is needed for the 1-Wire function to complete. Then access the device in read mode to get the result from the 1-Wire Triplet command.

```
// Global value for the current serial number
uchar SearchSerialNum[8];

// oneWireSearch
//
// Does a 1-Wire search using the 1-Wire Triplet command.
//
// resetSearch – Reset the search(1) or not(0).
// lastDevice  - If the last device has been found(1) or not(0).
// deviceAddress – The returned serial number.
//
// returns SUCCES or FAILURE
//
uchar OWSearch(uchar resetSearch, uchar *lastDevice, uchar *deviceAddress)
{
   uchar retVal = FAILURE;
   uchar bit_number = 1;
   uchar last_zero = 0;
   uchar serial_byte_number = 0;
   uchar serial_byte_mask = 1;
   uchar firstBit, secondBit, dir;
   uchar i = 0;

   if(resetSearch)
   {
      lastDevice = 0;
         LastDiscrepancy = 0;
   }

   // if the last call was not the last one
   if (!(*lastDevice))
   {
      // reset the 1-wire
      // if there are no parts on 1-wire, return FALSE
      if(!OWReset())
      {
         // reset the search
         lastDevice = 0;
         LastDiscrepancy = 0;
         return FAILURE;
      }

      // Issue the Search ROM command
```

```
OWWireByte(0xF0);

// loop to do the search
do
{
   if (bit_number < LastDiscrepancy)
   {
      if(SearchSerialNum[serial_byte_number] & serial_byte_mask)
         dir = 1;
      else
         dir = 0;
   }
   else
   {
      // if equal to last pick 1, if not then pick 0
      if(bit_number==LastDiscrepancy)
         dir = 1;
      else
         dir = 0;
   }

   if(!owTriplet(&dir, &firstBit, &secondBit))
   {
      return FAILURE;
   }

   // if 0 was picked then record its position in LastZero
   if (firstBit==0 && secondBit==0 && dir == 0)
   {
      last_zero = bit_number;
      }

   // check for no devices on 1-wire
   if (firstBit==1 && secondBit==1)
      break;

   // set or clear the bit in the SerialNum byte serial_byte_number
   // with mask serial_byte_mask
   if (dir == 1)
         SearchSerialNum[serial_byte_number] |= serial_byte_mask;
   else
      SearchSerialNum[serial_byte_number] &= ~serial_byte_mask;

   // increment the byte counter bit_number
   // and shift the mask serial_byte_mask
   bit_number++;
   serial_byte_mask <<= 1;

   // if the mask is 0 then go to new SerialNum[portnum] byte serial_byte_number
   // and reset mask
   if (serial_byte_mask == 0)
   {
      serial_byte_number++;
      serial_byte_mask = 1;
   }
}
while(serial_byte_number < 8);  // loop until through all SerialNum[portnum]

retVal = FAILURE;
// if the search was successful then
if (bit_number == 65)//|| crcl))
{
   // search successful so set LastDiscrepancy,LastDevice
   LastDiscrepancy = last_zero;
   if(LastDiscrepancy==0)
      *lastDevice = SUCCESS;
   else
          *lastDevice = FAILURE;

   for(i=0; i<8; i++)
      {
      deviceAddress[i] = SearchSerialNum[i];
      }
```

```
            return SUCCESS;
         }
      }

      // if no device found then reset counters so next 'next' will be
      // like a first
      if (!retVal || !SearchSerialNum[0])
      {
         LastDiscrepancy = 0;
         *lastDevice = FAILURE;
         retVal = FAILURE;
      }

      return retVal;
   }

// oneTriplet
//
// Uses the 1-Wire Triplet command.
//
// dir – Returns the direction that was chosen (1) or (0).
// firstBit  - Returns the first bit of the search (1) or (0).
// secondBit – Returns the complement of the first bit (1) or (0).
//
// returns SUCCES or FAILURE
//
uchar owTriplet(uchar *dir, uchar *firstBit, uchar *secondBit)
{
   uchar buff[3];
   uchar test;

   buff[0] = 0x78;

   if(*dir>0)
      *dir = (uchar)0xFF;

   buff[1] = *dir;

   if(!I2CBus_write(&buff[0],2))
   {
      lcd_putchar('f');
   }

   if(!I2CBus_read(&buff[2],1))
   {
      return FAILURE;
   }
   else
   {
      test = buff[2] & 0x20;
         if(test == 0x20)
            *firstBit = 1;
         else
            *firstBit = 0;

         test = buff[2] & 0x40;
         if(test == 0x40)
            *secondBit = 1;
         else
            *secondBit = 0;

         test = buff[2] & 0x80;
         if(test == 0x80)
            *dir = 1;
         else
            *dir = 0;

      return SUCCESS;
   }

   return FAILURE;
}
```

Example 8. OWSearch code.

6. Extended 1-WIRE Operations

6.1. OWSpeed

Example 9 shows how to change the speed of the 1-Wire bus using the DS2482. Overdrive or standard speeds are available.

```
// OWSpeed – changes the 1-Wire speed to normal or overdrive.
//          A Overdrive match rom or overdrive skip rom will be needed.
//
// speed – overdrive (Overdrive) or standard (Standard) speed.
// state_config – The current configuration byte settings.
//
// return – success or failure of the operation.
//
uchar OWSpeed(uchar speed, uchar state_config)
{
   uchar buffer[2];

   buffer[0] = writeConfigCommand;

   if(speed == Overdrive)
      buffer[1] = (state_config | 0x08) & 0x7F;
   else
      buffer[1] = (state_config | 0x80) & 0xF7;

   if(!I2CBus_write_packet(buffer,2))
   {
      return FAILURE;
   }

   return SUCCESS;
}
```

Example 9. OWSpeed code.

6.2. OWLevel

Example 10 shows how to change the level of the 1-Wire bus using the DS2482. Normal or power-delivery modes are available.

```
// OWLevel
//
// level – Normal or Power Delivery mode
// state_config – The current configuration settings
//
// Returns if the operation was Successful or not.
//
uchar OWLevel(uchar level, uchar state_config)
{
   uchar buffer[2];

   buffer[0] = writeConfigCommand;

   // NORMAL = 0, POWER_DELIV = 1
   if(level == NORMAL)
      buffer[1] = (state_config | 0x01) & 0xEF;
   else
      buffer[1] = (state_config | 0x10) & 0xFE;

   if(!I2CBus_write_packet(buffer,2))
   {
      return FAILURE;
   }

   return SUCCESS;
}
```

Example 10. OWLevel code.

6.3. OWReadBitPower

Example 11 shows the code used for OWReadBitPower, which reads a 1-Wire bit and implements power delivery. When the Strong Pullup (SPU) bit in the configuration register is enabled, the DS2482 actively pulls the 1-Wire line high after the next bit or byte communication.

```
// OWReadBitPower
//
// config_byte – current configuration settings
// delay – ms delay used before disabling active pullup
//
// Returns the bit information read.
//
uchar OWReadBitPower(uchar config_byte)
{
   uchar buffer[2];
   uchar return_bit;

   buffer[0] = writeConfigCommand;
   buffer[1] = (config_byte | 0x04) & 0xBF;
   // Sets strong pullup active so after the next byte or bit
   // strong pullup will be active

   if(!I2CBus_write_packet(buffer,2))
   {
      Error;
   }

   return OWReadBit();
}
```

Example 11. OWReadBitPower code.

6.4. OWWriteBytePower

Example 12 shows the code used for OWWriteBytePower, which writes a 1-Wire byte and implements power delivery. When the Strong Pullup (SPU) bit in the configuration register is enabled, the DS2482 actively pulls the 1-Wire line high after the next bit or byte communication.

```
// OWWriteBytePower
//
// config_byte – current configuration settings.
// wrbyte – byte to be written before the strong pullup is active
// delay – ms delay used before disabling active pullup
//
// Returns failure or success of the operation.
//
uchar OWWriteBytePower(uchar config_byte, uchar wrbyte)
{
   uchar buffer[2];

   buffer[0] = writeConfigCommand;
   buffer[1] = (config_byte | 0x04) & 0xBF;
   // Sets strong pullup active so after the next byte or bit
   // strong pullup will be active

   if(!I2CBus_write_packet(buffer,2))
   {
      return FAILURE;
   }

   OWWriteByte(wrbyte);

   return SUCCESS;
}
```

Example 12. OWWriteBytePower code.

Conclusion

The DS2482 has successfully been tested to convert I²C commands to 1-Wire communication. This document has presented a complete 1-Wire interface solution using the DS2482 I²C 1-Wire Line Driver. The code examples are easily implemented on any host system with an I²C communications port. A complete C implementation is also available for download.

1-Wire is a registered trademark of Dallas Semiconductor Corp.

Dallas Semiconductor is a wholly owned subsidiary of Maxim Integrated Products, Inc.

Application Note 3684: www.maxim-ic.com/an3684

More Information
For technical support: www.maxim-ic.com/support
For samples: www.maxim-ic.com/samples
Other questions and comments: www.maxim-ic.com/contact

Related Parts

DS2482-100: QuickView -- Full (PDF) Data Sheet -- Free Samples

DS2482-800: QuickView -- Full (PDF) Data Sheet -- Free Samples

AN3684, AN 3684, APP3684, Appnote3684, Appnote 3684