

APPLICATION NOTE 3684

How to Use the DS2482 I²C 1-Wire® Master

Abstract: The DS2482 is an I²C to 1-Wire 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.

Introduction

A 1-Wire network consists of a single master and one or more slave devices. The 1-Wire master can be constructed with an IO pin of a microprocessor and manually-generated timing pulses. The [DS2482](#) I²C to 1-Wire bridge alleviates the design engineer from implementing the details of 1-Wire timing. See **Figure 1** for a simplified diagram of the DS2482 configuration. This document presents an efficient application programming interface (API) implementation of the basic and extended 1-Wire operations using the DS2482. The I²C communication for each 1-Wire operation is explained. With the exception of programming EPROM-based devices such as the [DS250x](#) series, 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 sheets, but does not replace them. The DS2482 is available in three configurations, a single-channel 1-Wire master (DS2482-100), a single channel 1-Wire master with low-power sleep mode (DS2482-101), and an eight-channel 1-Wire master (DS2482-800).

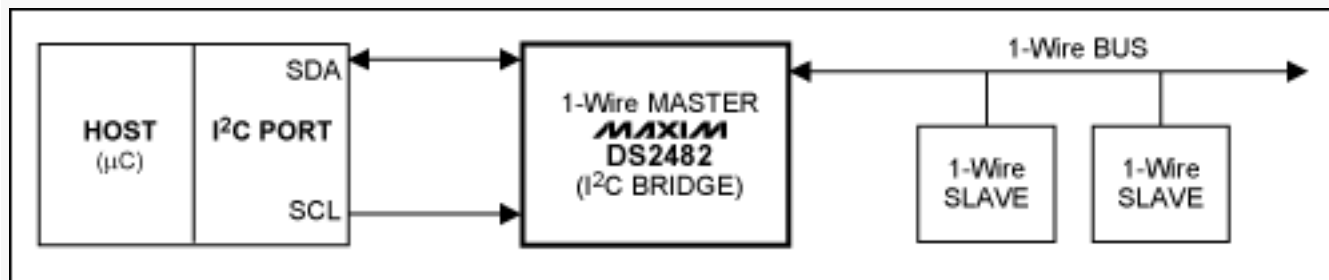


Figure 1. Simplified illustration of DS2482 function as a bridge for I²C communication and a 1-Wire network.

The 1-Wire Interface

There are a few basic 1-Wire functions, called primitives, which an application must have to perform any 1-Wire communication session. The first function (OWReset) resets all the 1-Wire slaves on the network, readying them for a command from the 1-Wire master. The second function (OWWriteBit) writes a bit from the 1-Wire master to the slaves, and the third function (OWReadBit) 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 network is just eight single bit writes.

The [1-Wire Search Algorithm](#) can also be constructed using these same three primitives. However, the DS2482 incorporates a search helper function called the 1-Wire triplet, which greatly reduces the communication overhead required for a search. Similarly there is a byte 1-Wire communication command that is more efficient than doing eight single-bit commands.

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 are 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 the presence pulse of 1-Wire slave devices.
OWWriteBit/OWReadBit	Sends to or receives from the 1-Wire network a single bit of data.
OWWriteByte/OWReadByte	Sends to or receives from the 1-Wire network a single byte of data.
OWBlock	Sends to and receives from the 1-Wire network multiple bytes of data.
OWSearch	Performs the 1-Wire Search Algorithm (see application note 187).
msDelay	Delays at least the specified number of milliseconds. Used for timing strong pullup operations.

Many 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 of their operating energy from the 1-Wire network. 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 a Secure Hash Algorithm (SHA-1) hash. The power for this action is supplied by enabling a stronger pullup on the 1-Wire network. Normal communication cannot occur during this power delivery. The DS2482 delivers power by setting the Strong Pullup (SPU) flag, which issues a strong pullup after the next byte/bit of 1-Wire communication. The DS2482-100 and DS2482-101 also have 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 network and optionally applies power delivery immediately after the bit is complete.
OWWriteBytePower	Sends a single byte of data to the 1-Wire network and applies power delivery immediately after the byte is complete.

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. Note that some of these I²C operations may be bundled into higher-level functions by host interfaces. The required operations can be seen in **Table 3**.

Table 3. Required I²C Host Operations

Operation	Description
I2C_start	I ² C start command.
I2C_rep_start	I ² C repeated start command.
I2C_stop	I ² C stop command.
I2C_write	Writes a byte to the I ² C bus. The byte to write is passed to the function.
I2C_read	Reads a byte from the I ² C bus. The byte read is returned from the function.

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, DS2482-101, and DS2482-800.

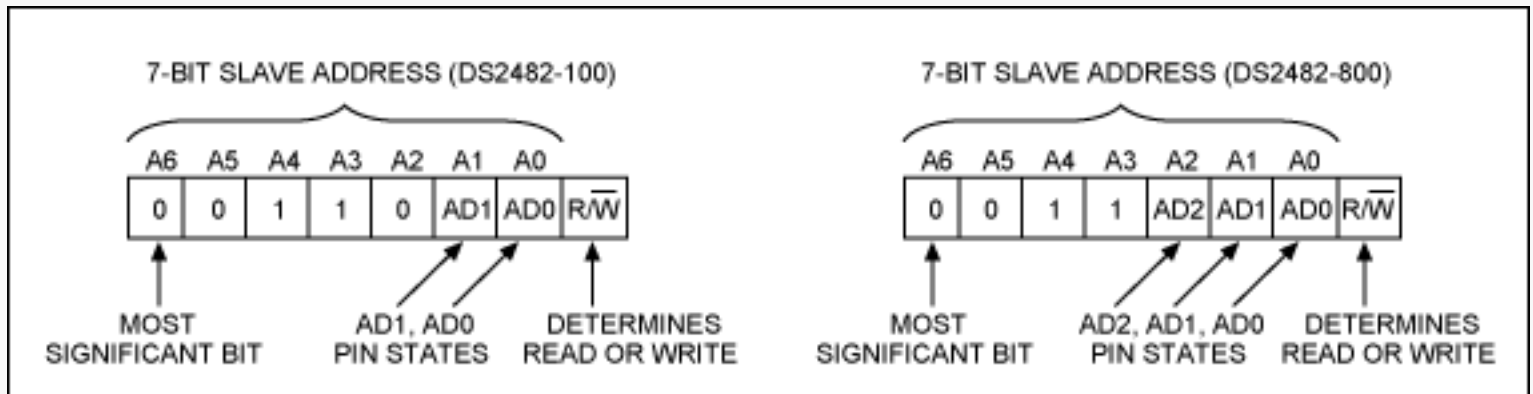


Figure 2. DS2482 I²C slave addresses.

DS2482 I²C Commands

The following legend from the DS2482 data sheet represents a shorthand 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
CHSL	Command 'Channel Select', C3h (DS2482-800 only)
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

Data Direction Codes



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).

DS2482 Configuration Operations

The following operations are used to set up and configure the DS2482. Some of these operations are called as subroutines by the 1-Wire operations.

DS2482 Detect

Example 1 shows the detect and configuration sequence in C. The default values written to the DS2482 include the 1-Wire speed (standard), the strong pullup (off), presence pulse masking (off), and the active pullup (on). This state is held in global variables so that it can be restored if the device needs to be reset back to this default state. Different default values may be preferred for different applications.

```

// DS2482 state
unsigned char I2C_address;
int c1WS, cSPU, cPPM, cAPU;
int short_detected;

//-----
// DS2428 Detect routine that sets the I2C address and then performs a
// device reset followed by writing the configuration byte to default values:
// 1-Wire speed (c1WS) = standard (0)
// Strong pullup (cSPU) = off (0)
// Presence pulse masking (cPPM) = off (0)
// Active pullup (cAPU) = on (CONFIG_APU = 0x01)
//
// Returns: TRUE if device was detected and written
//          FALSE device not detected or failure to write configuration byte
//
int DS2482_detect(unsigned char addr)
{
    // set global address
    I2C_address = addr;

    // reset the DS2482 ON selected address
    if (!DS2482_reset())
        return FALSE;

    // default configuration
    c1WS = FALSE;
    cSPU = FALSE;
    cPPM = FALSE;
    cAPU = CONFIG_APU;

    // write the default configuration setup
    if (!DS2482_write_config(c1WS | cSPU | cPPM | cAPU))
        return FALSE;

    return TRUE;
}

```

Example 1. DS2482 detect.

DS2482 Device Reset

Figure 3 shows the DS2482 device-reset I²C communication sequence. **Example 2** shows the DS2482 reset command sequence in C, 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 F0h.

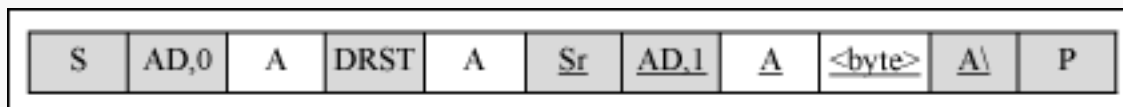


Figure 3. Device reset after power-up. This example includes an optional read access to verify the success of the command.

```

//-----
// Perform a device reset on the DS2482
//

```

```

// Returns: TRUE if device was reset
//          FALSE device not detected or failure to perform reset
//
int DS2482_reset()
{
    unsigned char status;

    // Device Reset
    // S AD,0 [A] DRST [A] Sr AD,1 [A] [SS] A\ P
    // [] indicates from slave
    // SS status byte to read to verify state

    I2C_start();
    I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
    I2C_write(CMD_DRST, EXPECT_ACK);
    I2C_rep_start();
    I2C_write(I2C_address | I2C_READ, EXPECT_ACK);
    status = I2C_read(NACK);
    I2C_stop();

    // check for failure due to incorrect read back of status
    return ((status & 0xF7) == 0x10);
}

```

Example 2. Device reset code.

DS2482 Write Configuration

Figure 4 shows the DS2482 write-configuration I²C communication example. **Example 3** shows the DS2482 write configuration command sequence in C. The command code for write configuration is D2h.

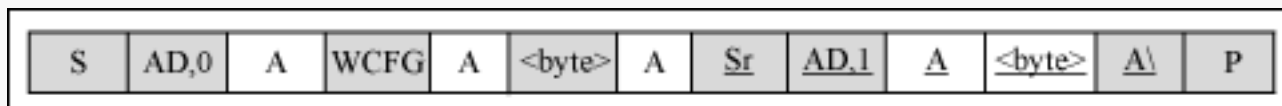


Figure 4. Write Configuration register. This example includes an optional read to verify the success of the command.

```

//-----
// Write the configuration register in the DS2482. The configuration
// options are provided in the lower nibble of the provided config byte.
// The upper nibble in bitwise inverted when written to the DS2482.
//
// Returns: TRUE: config written and response correct
//          FALSE: response incorrect
//
int DS2482_write_config(unsigned char config)
{
    unsigned char read_config;

    // Write configuration (Case A)
    // S AD,0 [A] WCFG [A] CF [A] Sr AD,1 [A] [CF] A\ P
    // [] indicates from slave
    // CF configuration byte to write

    I2C_start();
    I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
    I2C_write(CMD_WCFG, EXPECT_ACK);
    I2C_write(config | (~config << 4), EXPECT_ACK);

```

```

I2C_rep_start();
I2C_write(I2C_address | I2C_READ, EXPECT_ACK);
read_config = I2C_read(NACK);
I2C_stop();

// check for failure due to incorrect read back
if (config != read_config)
{
    // handle error
    // ...
    DS2482_reset();

    return FALSE;
}

return TRUE;
}

```

Example 3. DS2482 write configuration.

DS2482 Channel Select

Figure 5 shows the DS2482-800 channel-select I²C communication example. The valid channels are 0 to 7. Note that this operation does not apply to the DS2482-100 or DS2482-101. **Example 4** shows the DS2482-800 channel-select command sequence in C. The command code for channel select is C3h. After the channel is selected, all 1-Wire operations will be performed on that channel.

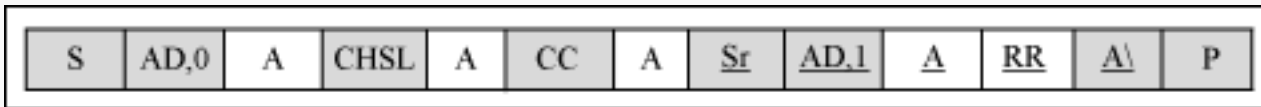


Figure 5. Write Channel Selection register. This example includes an optional read to verify the success of the command.

```

//-----
// Select the 1-Wire channel on a DS2482-800.
//
// Returns: TRUE if channel selected
//          FALSE device not detected or failure to perform select
//
int DS2482_channel_select(int channel)
{
    unsigned char ch, ch_read, check;

    // Channel Select (Case A)
    // S AD,0 [A] CHSL [A] CC [A] Sr AD,1 [A] [RR] A\ P
    // [] indicates from slave
    // CC channel value
    // RR channel read back

    I2C_start();
    I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
    I2C_write(CMD_CHSL, EXPECT_ACK);

    switch (channel)
    {
        default: case 0: ch = 0xF0; ch_read = 0xB8; break;
        case 1: ch = 0xE1; ch_read = 0xB1; break;
        case 2: ch = 0xD2; ch_read = 0xAA; break;
    }
}

```

```

    case 3: ch = 0xC3; ch_read = 0xA3; break;
    case 4: ch = 0xB4; ch_read = 0x9C; break;
    case 5: ch = 0xA5; ch_read = 0x95; break;
    case 6: ch = 0x96; ch_read = 0x8E; break;
    case 7: ch = 0x87; ch_read = 0x87; break;
};

I2C_write(ch, EXPECT_ACK);
I2C_rep_start();
I2C_write(I2C_address | I2C_READ, EXPECT_ACK);
check = I2C_read(NACK);
I2C_stop();

// check for failure due to incorrect read back of channel
return (check == ch_read);
}

```

Example 4. DS2482-800 channel select.

DS2482 1-Wire Operations

OWReset

The 1-Wire Reset command (B4h) generates a 1-Wire reset on the 1-Wire network and samples for a 1-Wire device presence pulse. The state of the sample is reported through the Presence-Pulse Detect (PPD) and the Short Detected (SD) fields in the status register. **Figure 6** shows I²C communication for the 1-Wire Reset command. **Example 5** shows the command sent and status register checked for a presence pulse in C.

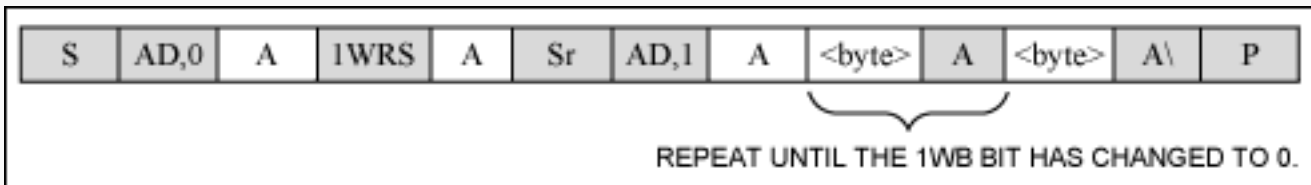


Figure 6. 1-Wire reset. Begins or ends 1-Wire communication. 1-Wire Idle (1WB = 0), busy polling until the 1-Wire command is completed, then read the result.

```

//-----
// Reset all of the devices on the 1-Wire Net and return the result.
//
// Returns: TRUE(1):  presence pulse(s) detected, device(s) reset
//           FALSE(0): no presence pulses detected
//
int OWReset(void)
{
    unsigned char status;
    int poll_count = 0;

    // 1-Wire reset (Case B)
    //  S AD,0 [A] 1WRS [A] Sr AD,1 [A] [Status] A [Status] A\ P
    //                                     \-----/
    //                                     Repeat until 1WB bit has changed to 0
    //  [] indicates from slave

    I2C_start();
    I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
    I2C_write(CMD_1WRS, EXPECT_ACK);

```

```

I2C_rep_start();
I2C_write(I2C_address | I2C_READ, EXPECT_ACK);

// loop checking 1WB bit for completion of 1-Wire operation
// abort if poll limit reached
status = I2C_read(ACK);
do
{
    status = I2C_read(status & STATUS_1WB);
}
while ((status & STATUS_1WB) && (poll_count++ < POLL_LIMIT));

I2C_stop();

// check for failure due to poll limit reached
if (poll_count >= POLL_LIMIT)
{
    // handle error
    // ...
    DS2482_reset();
    return FALSE;
}

// check for short condition
if (status & STATUS_SD)
    short_detected = TRUE;
else
    short_detected = FALSE;

// check for presence detect
if (status & STATUS_PPD)
    return TRUE;
else
    return FALSE;
}

```

Example 5. OWRreset code.

OWWriteBit/OWReadBit

The 1-Wire bit command (0x87) generates a single 1-Wire bit time slot. **Figure 7** shows the I²C communication code for the 1-Wire single bit command cases. **Figure 8** is the bit allocation byte where V is bit to send. **Example 6** shows the OWWriteBit, OWReadBit, and OWTouchBit code.

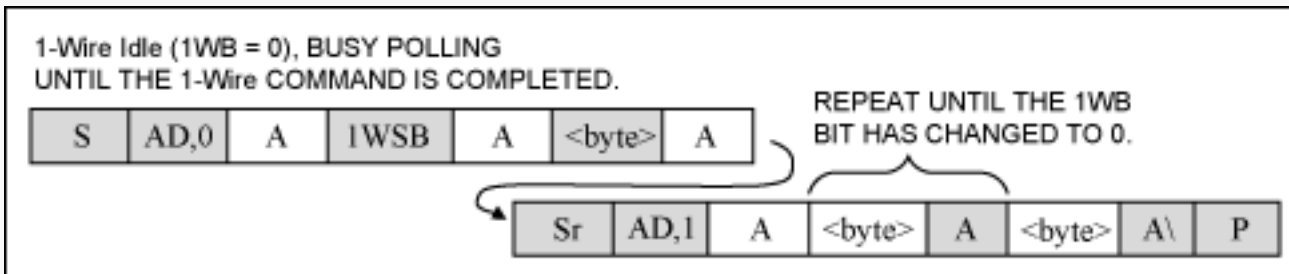


Figure 7. 1-Wire Single Bit. Generates a single time slot on the 1-Wire line. When 1WB has changed from 1 to 0, the Status register holds the valid result of the 1-Wire single bit command.

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 8. 1-Wire single data byte.

```
//-----
// Send 1 bit of communication to the 1-Wire Net.
// The parameter 'sendbit' least significant bit is used.
//
// 'sendbit' - 1 bit to send (least significant byte)
//
void OWWriteBit(unsigned char sendbit)
{
    OWTouchBit(sendbit);
}

//-----
// Reads 1 bit of communication from the 1-Wire Net and returns the
// result
//
// Returns: 1 bit read from 1-Wire Net
//
unsigned char OWReadBit(void)
{
    return OWTouchBit(0x01);
}

//-----
// Send 1 bit of communication to the 1-Wire Net and return the
// result 1 bit read from the 1-Wire Net. The parameter 'sendbit'
// least significant bit is used and the least significant bit
// of the result is the return bit.
//
// 'sendbit' - the least significant bit is the bit to send
//
// Returns: 0: 0 bit read from sendbit
//          1: 1 bit read from sendbit
//
unsigned char OWTouchBit(unsigned char sendbit)
{
    unsigned char status;
    int poll_count = 0;

    // 1-Wire bit (Case B)
    // S AD,0 [A] 1WSB [A] BB [A] Sr AD,1 [A] [Status] A [Status] A \ P
    //                                     \-----/
    //                                     Repeat until 1WB bit has changed to 0
    // [] indicates from slave
    // BB indicates byte containing bit value in msbit

    I2C_start();
    I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
    I2C_write(CMD_1WSB, EXPECT_ACK);
    I2C_write(sendbit ? 0x80 : 0x00, EXPECT_ACK);
    I2C_rep_start();
    I2C_write(I2C_address | I2C_READ, EXPECT_ACK);
```

```

// loop checking 1WB bit for completion of 1-Wire operation
// abort if poll limit reached
status = I2C_read(ACK);
do
{
    status = I2C_read(status & STATUS_1WB);
}
while ((status & STATUS_1WB) && (poll_count++ < POLL_LIMIT));

I2C_stop();

// check for failure due to poll limit reached
if (poll_count >= POLL_LIMIT)
{
    // handle error
    // ...
    DS2482_reset();
    return 0;
}

// return bit state
if (status & STATUS_SBR)
    return 1;
else
    return 0;
}

```

Example 6. 1-Wire single bit code.

OWWriteByte

The 1-Wire write byte command (A5h) writes a single data byte to the 1-Wire bus. 1-Wire activity must have ended before the DS2482 can process this command. **Figure 9** shows the I²C write 1-Wire byte sequence. Code **Example 7** checks that the 1-Wire activity completes before returning from this operation.

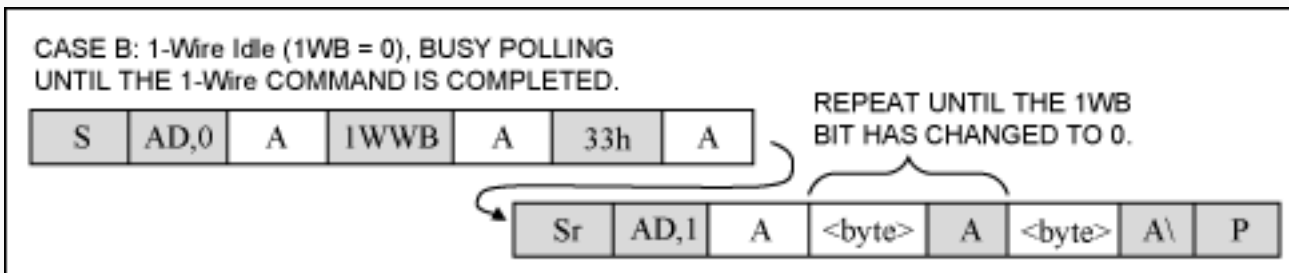


Figure 9. 1-Wire Write Byte. Sends a command code to the 1-Wire line. When 1WB has changed from 1 to 0, the 1-Wire Write Byte command is completed.

```

//-----
// Send 8 bits of communication to the 1-Wire Net and verify that the
// 8 bits read from the 1-Wire Net are the same (write operation).
// The parameter 'sendbyte' least significant 8 bits are used.
//
// 'sendbyte' - 8 bits to send (least significant byte)
//
// Returns:  TRUE: bytes written and echo was the same
//           FALSE: echo was not the same
//
void OWWriteByte(unsigned char sendbyte)

```

```

{
  unsigned char status;
  int poll_count = 0;

  // 1-Wire Write Byte (Case B)
  //   S AD,0 [A] 1WWB [A] DD [A] Sr AD,1 [A] [Status] A [Status] A\ P
  //                                     \-----/
  //                                     Repeat until 1WB bit has changed to 0
  // [] indicates from slave
  // DD data to write

  I2C_start();
  I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
  I2C_write(CMD_1WWB, EXPECT_ACK);
  I2C_write(sendbyte, EXPECT_ACK);
  I2C_rep_start();
  I2C_write(I2C_address | I2C_READ, EXPECT_ACK);

  // loop checking 1WB bit for completion of 1-Wire operation
  // abort if poll limit reached
  status = I2C_read(ACK);
  do
  {
    status = I2C_read(status & STATUS_1WB);
  }
  while ((status & STATUS_1WB) && (poll_count++ < POLL_LIMIT));

  I2C_stop();

  // check for failure due to poll limit reached
  if (poll_count >= POLL_LIMIT)
  {
    // handle error
    // ...
    DS2482_reset();
  }
}

```

Example 7. OWWriteByte code.

OWReadByte

The 1-Wire read byte command (96h) reads a single byte from the 1-Wire network. 1-Wire activity must have ended before the DS2482 can process this command. **Figure 10** shows the I²C sequence. Code for a 1-Wire Read Byte Command can be found in Code **Example 8**. The 1-Wire activity is checked after issuing the read byte command to verify that it completes and the result is returned.

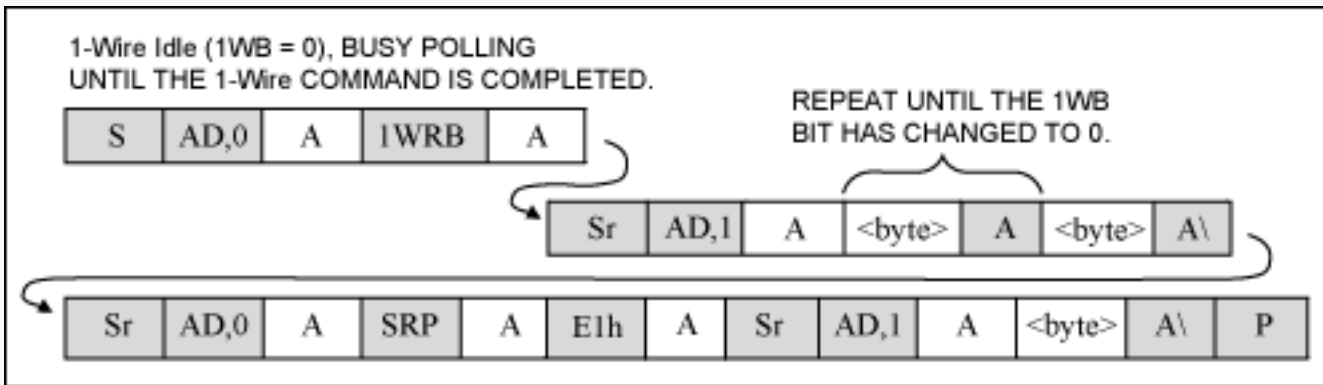


Figure 10. 1-Wire Read Byte. Reads a byte from the 1-Wire network. Poll the Status register until the 1WB bit has changed from 1 to 0. Then set the read pointer to the Read Data register (code E1h) and access the device again to read the data byte obtained from the 1-Wire network.

```
//-----
// Send 8 bits of read communication to the 1-Wire Net and return the
// result 8 bits read from the 1-Wire Net.
//
// Returns: 8 bits read from 1-Wire Net
//
unsigned char OWReadByte(void)
{
    unsigned char data, status;
    int poll_count = 0;

    // 1-Wire Read Bytes (Case C)
    // S AD,0 [A] 1WRB [A] Sr AD,1 [A] [Status] A [Status] A\
    //                                     \-----/
    //                                     Repeat until 1WB bit has changed to 0
    // Sr AD,0 [A] SRP [A] E1 [A] Sr AD,1 [A] DD A\ P
    //
    // [] indicates from slave
    // DD data read

    I2C_start();
    I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
    I2C_write(CMD_1WRB, EXPECT_ACK);
    I2C_rep_start();
    I2C_write(I2C_address | I2C_READ, EXPECT_ACK);

    // loop checking 1WB bit for completion of 1-Wire operation
    // abort if poll limit reached
    status = I2C_read(ACK);
    do
    {
        status = I2C_read(status & STATUS_1WB);
    }
    while ((status & STATUS_1WB) && (poll_count++ < POLL_LIMIT));

    // check for failure due to poll limit reached
    if (poll_count >= POLL_LIMIT)
    {
        // handle error
        // ...
        DS2482_reset();
        return 0;
    }
}
```

```
I2C_rep_start();
I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
I2C_write(CMD_SRP, EXPECT_ACK);
I2C_write(0xE1, EXPECT_ACK);
I2C_rep_start();
I2C_write(I2C_address | I2C_READ, EXPECT_ACK);
data = I2C_read(NACK);
I2C_stop();

return data;
}
```

Example 8. OWReadByte code.

OWBlock

The OWBlock operation performs a group of 1-Wire byte operations. This is useful when transferring blocks of data on the 1-Wire network. **Example 9** shows a code example of OWBlock.

```
//-----  
// The 'OWBlock' transfers a block of data to and from the  
// 1-Wire Net. The result is returned in the same buffer.  
//  
// 'tran_buf' - pointer to a block of unsigned  
//             chars of length 'tran_len' that will be sent  
//             to the 1-Wire Net  
// 'tran_len' - length in bytes to transfer  
//  
void OWBlock(unsigned char *tran_buf, int tran_len)  
{  
    int i;  
  
    for (i = 0; i < tran_len; i++)  
        tran_buf[i] = OWTouchByte(tran_buf[i]);  
}  
  
//-----  
// Send 8 bits of communication to the 1-Wire Net and return the  
// result 8 bits read from the 1-Wire Net. The parameter 'sendbyte'  
// least significant 8 bits are used and the least significant 8 bits  
// of the result are the return byte.  
//  
// 'sendbyte' - 8 bits to send (least significant byte)  
//  
// Returns: 8 bits read from sendbyte  
//  
unsigned char OWTouchByte(unsigned char sendbyte)  
{  
    if (sendbyte == 0xFF)  
        return OWReadByte();  
    else  
    {  
        OWWriteByte(sendbyte);  
        return sendbyte;  
    }  
}
```

Example 9. OWBlock code.

OWSearch/1-Wire Triplet Command

The 1-Wire search is used to discover the 64-bit unique registration number of the devices on a 1-Wire network. The unique registration number is often referred to as the ROM number in the 1-Wire data sheets since it is in Read-Only Memory. The search begins with a 1-Wire reset followed by a search command. The search command that all 1-Wire devices respond to is F0h. After the search command, the 1-Wire master does a binary search to find one device. The binary search looks at each of the 64 bits by first reading the bit, reading the complement of the bit, and then writing a bit to indicate which devices remain in the search. These three single-bit sequences are referred to as a triplet. The DS2482 has a shortcut command to do a 1-Wire search more efficiently by performing this triplet.

The Triplet command (78h) generates three time slots, two read time slots, and one write time slot on the 1-

Wire network. The direction byte (DIR) in the Status register determines the type of write time slot (**Figure 11**). **Example 10** illustrates a complete 1-Wire search using the 1-Wire triplet command. The example also has functions to set the search so it finds the first and next devices. Calling OWFirst followed by repeated calls to OWNext will find all of the devices on the 1-Wire network. For an explanation of the 1-Wire search algorithm, see application note 187, "[1-Wire Search Algorithm](#)."

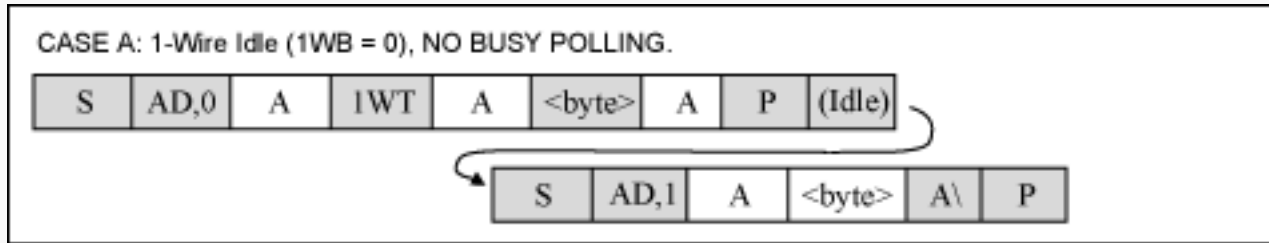


Figure 11. 1-Wire Triplet. Performs a Search ROM function on the 1-Wire network. The idle time is needed for the 1-Wire function to complete. Then access the device in read mode to get the result from the 1-Wire triplet command.

```
// Search state
unsigned char ROM_NO[8];
int LastDiscrepancy;
int LastFamilyDiscrepancy;
int LastDeviceFlag;
unsigned char crc8;

//-----
// Find the 'first' devices on the 1-Wire network
// Return TRUE : device found, ROM number in ROM_NO buffer
//          FALSE : no device present
//
int OWFirst()
{
    // reset the search state
    LastDiscrepancy = 0;
    LastDeviceFlag = FALSE;
    LastFamilyDiscrepancy = 0;

    return OWSearch();
}

//-----
// Find the 'next' devices on the 1-Wire network
// Return TRUE : device found, ROM number in ROM_NO buffer
//          FALSE : device not found, end of search
//
int OWNext()
{
    // leave the search state alone
    return OWSearch();
}

//-----
// The 'OWSearch' function does a general search. This function
// continues from the previous search state. The search state
// can be reset by using the 'OWFirst' function.
// This function contains one parameter 'alarm_only'.
// When 'alarm_only' is TRUE (1) the find alarm command
// 0xEC is sent instead of the normal search command 0xF0.
// Using the find alarm command 0xEC will limit the search to only
// 1-Wire devices that are in an 'alarm' state.
```

```

//
// Returns:   TRUE (1) : when a 1-Wire device was found and its
//             Serial Number placed in the global ROM
//             FALSE (0): when no new device was found.  Either the
//             last search was the last device or there
//             are no devices on the 1-Wire Net.
//
//
int OWSearch()
{
    int id_bit_number;
    int last_zero, rom_byte_number, search_result;
    int id_bit, cmp_id_bit;
    unsigned char rom_byte_mask, search_direction, status;

    // initialize for search
    id_bit_number = 1;
    last_zero = 0;
    rom_byte_number = 0;
    rom_byte_mask = 1;
    search_result = FALSE;
    crc8 = 0;

    // if the last call was not the last one
    if (!LastDeviceFlag)
    {
        // 1-Wire reset
        if (!OWReset())
        {
            // reset the search
            LastDiscrepancy = 0;
            LastDeviceFlag = FALSE;
            LastFamilyDiscrepancy = 0;
            return FALSE;
        }

        // issue the search command
        OWWriteByte(0xF0);

        // loop to do the search
        do
        {
            // if this discrepancy is before the Last Discrepancy
            // on a previous next then pick the same as last time
            if (id_bit_number < LastDiscrepancy)
            {
                if ((ROM_NO[rom_byte_number] & rom_byte_mask) > 0)
                    search_direction = 1;
                else
                    search_direction = 0;
            }
            else
            {
                // if equal to last pick 1, if not then pick 0
                if (id_bit_number == LastDiscrepancy)
                    search_direction = 1;
                else
                    search_direction = 0;
            }

            // Perform a triple operation on the DS2482 which will perform
            // 2 read bits and 1 write bit

```

```

status = DS2482_search_triplet(search_direction);

// check bit results in status byte
id_bit = ((status & STATUS_SBR) == STATUS_SBR);
cmp_id_bit = ((status & STATUS_TSB) == STATUS_TSB);
search_direction =
    ((status & STATUS_DIR) == STATUS_DIR) ? (byte)1 : (byte)0;

// check for no devices on 1-Wire
if ((id_bit) && (cmp_id_bit))
    break;
else
{
    if ((!id_bit) && (!cmp_id_bit) && (search_direction == 0))
    {
        last_zero = id_bit_number;

        // check for Last discrepancy in family
        if (last_zero < 9)
            LastFamilyDiscrepancy = last_zero;
    }

    // set or clear the bit in the ROM byte rom_byte_number
    // with mask rom_byte_mask
    if (search_direction == 1)
        ROM_NO[rom_byte_number] |= rom_byte_mask;
    else
        ROM_NO[rom_byte_number] &= (byte)~rom_byte_mask;

    // increment the byte counter id_bit_number
    // and shift the mask rom_byte_mask
    id_bit_number++;
    rom_byte_mask <<= 1;

    // if the mask is 0 then go to new SerialNum byte rom_byte_number
    // and reset mask
    if (rom_byte_mask == 0)
    {
        calc_crc8(ROM_NO[rom_byte_number]); // accumulate the CRC
        rom_byte_number++;
        rom_byte_mask = 1;
    }
}
}
while(rom_byte_number < 8); // loop until through all ROM bytes 0-7

// if the search was successful then
if (!(id_bit_number < 65) || (crc8 != 0))
{
    // search successful so set LastDiscrepancy,LastDeviceFlag
    // search_result
    LastDiscrepancy = last_zero;

    // check for last device
    if (LastDiscrepancy == 0)
        LastDeviceFlag = TRUE;

    search_result = TRUE;
}
}

```

```

// if no device found then reset counters so next
// 'search' will be like a first

if (!search_result || (ROM_NO[0] == 0))
{
    LastDiscrepancy = 0;
    LastDeviceFlag = FALSE;
    LastFamilyDiscrepancy = 0;
    search_result = FALSE;
}

return search_result;
}

//-----
// Use the DS2482 help command '1-Wire triplet' to perform one bit of a
//1-Wire search.
//This command does two read bits and one write bit. The write bit
// is either the default direction (all device have same bit) or in case of
// a discrepancy, the 'search_direction' parameter is used.
//
// Returns - The DS2482 status byte result from the triplet command
//
unsigned char DS2482_search_triplet(int search_direction)
{
    unsigned char status;
    int poll_count = 0;

    // 1-Wire Triplet (Case B)
    //   S AD,0 [A] 1WT [A] SS [A] Sr AD,1 [A] [Status] A [Status] A\ P
    //                                     \-----/
    //                                     Repeat until 1WB bit has changed to 0
    // [] indicates from slave
    // SS indicates byte containing search direction bit value in msbit

    I2C_start();
    I2C_write(I2C_address | I2C_WRITE, EXPECT_ACK);
    I2C_write(CMD_1WT, EXPECT_ACK);
    I2C_write(search_direction ? 0x80 : 0x00, EXPECT_ACK);
    I2C_rep_start();
    I2C_write(I2C_address | I2C_READ, EXPECT_ACK);

    // loop checking 1WB bit for completion of 1-Wire operation
    // abort if poll limit reached
    status = I2C_read(ACK);
    do
    {
        status = I2C_read(status & STATUS_1WB);
    }
    while ((status & STATUS_1WB) && (poll_count++ < POLL_LIMIT));

    I2C_stop();

    // check for failure due to poll limit reached
    if (poll_count >= POLL_LIMIT)
    {
        // handle error
        // ...
        DS2482_reset();
        return 0;
    }
}

```

```

}

// return status byte
return status;
}

```

Example 10. OWSearch code.

Extended 1-WIRE Operations

OWSpeed

Example 11 shows how to change the speed of the 1-Wire network using the DS2482. All 1-Wire devices default to standard communication speed. Some devices can then negotiate to move to overdrive speed using the device Overdrive-Match-ROM or Overdrive-Skip-ROM commands. Once the device is in overdrive speed, all 1-Wire communication can proceed in overdrive. A standard-speed 1-Wire reset brings the devices back to standard speed.

```

//-----
// Set the 1-Wire Net communication speed.
//
// 'new_speed' - new speed defined as
//             MODE_STANDARD  0x00
//             MODE_OVERDRIVE 0x01
//
// Returns:  current 1-Wire Net speed
//
int OWSpeed(int new_speed)
{
    // set the speed
    if (new_speed == MODE_OVERDRIVE)
        c1WS = CONFIG_1WS;
    else
        c1WS = FALSE;

    // write the new config
    DS2482_write_config(c1WS | cSPU | cPPM | cAPU);

    return new_speed;
}

```

Example 11. OWSpeed code.

OWLevel

Example 12 shows how to change the level of the 1-Wire network using the DS2482. The DS2482 enables strong pullup only after a 1-Wire communication bit or byte. Consequently the OWLevel code can only return the 1-Wire network back to standard pullup. To enable strong pullup use the OWWriteBytePower or OWReadBitPower operations.

```

//-----
// Set the 1-Wire Net line level pullup to normal. The DS2482 only
// allows enabling strong pullup on a bit or byte event. Consequently this
// function only allows the MODE_STANDARD argument. To enable strong pullup

```

```

// use OWWriteBytePower or OWReadBitPower.
//
// 'new_level' - new level defined as
//             MODE_STANDARD    0x00
//
// Returns:   current 1-Wire Net level
//
int OWLevel(int new_level)
{
    // function only will turn back to non-strong pullup
    if (new_level != MODE_STANDARD)
        return MODE_STRONG;

    // clear the strong pullup bit in the global config state
    cSPU = FALSE;

    // write the new config
    DS2482_write_config(c1WS | cSPU | cPPM | cAPU);

    return MODE_STANDARD;
}

```

Example 12. OWLevel code.

OWReadBitPower

Example 13 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 network high after the next bit or byte communication. This operation verifies that the bit read is the expected response. If the response is not correct then the 1-Wire level is returned to the standard pullup state.

```

//-----
// Send 1 bit of communication to the 1-Wire Net and verify that the
// response matches the 'applyPowerResponse' bit and apply power delivery
// to the 1-Wire net. Note that some implementations may apply the power
// first and then turn it off if the response is incorrect.
//
// 'applyPowerResponse' - 1 bit response to check, if correct then start
//                       power delivery
//
// Returns:   TRUE: bit written and response correct, strong pullup now on
//           FALSE: response incorrect
//
int OWReadBitPower(int applyPowerResponse)
{
    unsigned char rdbit;

    // set strong pullup enable
    cSPU = CONFIG_SPU;

    // write the new config
    if (!DS2482_write_config(c1WS | cSPU | cPPM | cAPU))
        return FALSE;

    // perform read bit
    rdbit = OWReadBit();

    // check if response was correct, if not then turn off strong pullup

```

```

if (rdbit != applyPowerResponse)
{
    OWLevel(MODE_STANDARD);
    return FALSE;
}

return TRUE;
}

```

Example 13. OWReadBitPower code.

OWWriteBytePower

Example 14 shows the code used for OWWriteBytePower, which writes a 1-Wire byte and implements strong pullup power delivery. When the Strong Pullup (SPU) bit in the configuration register is enabled, the DS2482 actively pulls the 1-Wire network high after the next bit or byte communication. The OWLevel function can then be called to return the 1-Wire network to standard pullup.

```

//-----
// Send 8 bits of communication to the 1-Wire Net and verify that the
// 8 bits read from the 1-Wire Net are the same (write operation).
// The parameter 'sendbyte' least significant 8 bits are used. After the
// 8 bits are sent change the level of the 1-Wire net.
//
// 'sendbyte' - 8 bits to send (least significant bit)
//
// Returns:  TRUE: bytes written and echo was the same, strong pullup now on
//           FALSE: echo was not the same
//
int OWWriteBytePower(int sendbyte)
{
    // set strong pullup enable
    cSPU = CONFIG_SPU;

    // write the new config
    if (!DS2482_write_config(c1WS | cSPU | cPPM | cAPU))
        return FALSE;

    // perform write byte
    OWWriteByte(sendbyte);

    return TRUE;
}

```

Example 14. OWWriteBytePower code.

Conclusion

The DS2482 successfully bridges the I²C bus to a 1-Wire network. This document has presented a complete 1-Wire application interface solution using the family of DS2482 I²C 1-Wire Line Drivers. With this interface one can communicate with all 1-Wire devices. The code examples are easily implemented on any host system with an I²C communications port. A complete C implementation is also available for [download](#). The test platform for this code was the [CMAXQUSB evaluation kit](#).

Revision History

11/2005 Version 1.0—Initial release.

9/2008 Version 2.0—Changed code examples to use lower level I²C communication calls. Revised text to match.

1-Wire is a registered trademark 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

Automatic Updates

Would you like to be automatically notified when new application notes are published in your areas of interest?

[Sign up for EE-Mail™](#).

Related Parts

DS2482-100: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

DS2482-101: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

DS2482-800: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

AN3684, AN 3684, APP3684, Appnote3684, Appnote 3684

Copyright © by Maxim Integrated Products

Additional legal notices: www.maxim-ic.com/legal