Keywords: 1-Wire search algorithm, unique register number (ID), ROM number

APPLICATION NOTE 187

1-Wire Search Algorithm

Mar 28, 2002

Abstract: Maxim's 1-Wire® devices each have a 64-bit unique registration number in read-only memory (ROM) that is used to address them individually by a 1-Wire master in a 1-Wire network. If the ROM numbers of the slave devices on the 1-Wire network are not known, then using a search algorithm can discover them. This document explains the search algorithm in detail and provides an example implementation for rapid integration. This algorithm is valid for all current and future devices that feature a 1-Wire interface.

Introduction

Maxim's 1-Wire devices each have a 64-bit unique registration number in read-only memory (ROM) (**Figure 1**) that is used to address them individually by a 1-Wire master in a 1-Wire network. If the ROM numbers of the slave devices on the 1-Wire network are not known, then they can be discovered by using a search algorithm. This document explains the search algorithm in detail and provides an example implementation for rapid integration. This algorithm is valid for all current and future devices that feature a 1-Wire interface.

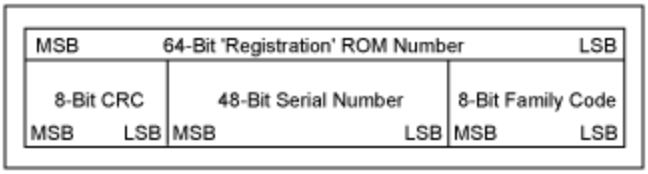


Figure 1. 64-bit unique ROM 'registration' number.

Search Algorithm

The search algorithm is a binary tree search where branches are followed until a device ROM number, or leaf, is found. Subsequent searches then take the other branch paths until all of the leaves present are discovered.

The search algorithm begins with the devices on the 1-Wire being reset using the reset and presence pulse sequence. If this is successful then the 1-byte search command is sent. The search command readies the 1-Wire devices to begin the search.

There are two types of search commands. The normal search command (F0 hex) performs a search

with all devices participating. The alarm or conditional search command (EC hex) performs a search with only the devices that are in some sort of alarm state. This reduces the search pool to quickly respond to devices that need attention.

Following the search command, the actual search begins with all participating devices simultaneously sending the first bit (least significant) in their ROM number (also called registration number). (See Figure 1.) As with all 1-Wire communication, the 1-Wire master starts every bit whether it is data to be read or written to the slave devices. Due to the characteristics of the 1-Wire, when all devices respond at the same time, this results in a logical AND of the bits sent. After the devices send the first bit of their ROM number, the master initiates the next bit and the devices then send the complement of the first bit. From these two bits, information can be derived about the first bit in the ROM numbers of the participating devices. (See **Table 1**.)

Table 1. Bit Search Information

Bit (true)	Bit (complement)	Information Known
0	0	There are both 0s and 1s in the current bit position of the participating ROM numbers. This is a discrepancy.
0	1	There are only 0s in the bit of the participating ROM numbers.
1	0	There are only 1s in the bit of the participating ROM numbers.
1	1	No devices participating in search.

According to the search algorithm, the 1-Wire master must then send a bit back to the participating devices. If the participating device has that bit value, it continues participating. If it does not have the bit value, it goes into a wait state until the next 1-Wire reset is detected. This 'read two bits' and 'write one bit' pattern is then repeated for the remaining 63 bits of the ROM number (see **Table 2**). In this way the search algorithm forces all but one device to go into this wait state. At the end of one pass, the ROM number of this last device is known. On subsequent passes of the search, a different path (or branch) is taken to find the other device ROM numbers. Note that this document refers to the bit position in the ROM number as bit 1 (least significant) to bit 64 (most significant). This convention was used instead of bit 0 to bit 63 for convenience, to allow initialization of discrepancy counters to 0 for later comparisons.

Table 2. 1-Wire Master and Slave Search Sequence

Master	Slave
1-Wire reset stimulus	Produce presence pulse.
Write search command (normal or alarm)	Each slave readies for search.
Read 'AND' of bit 1	Each slave sends bit 1 of its ROM number.
Read 'AND' of complement bit 1	Each slave sends complement bit 1 of its ROM number.
Write bit 1 direction (according to algorithm)	Each slave receives the bit written by master, if bit read is not the same as bit 1 of its ROM number then go into a wait state.

Read 'AND' of bit 64	Each slave sends bit 64 of its ROM number.
Read 'AND' of complement bit 64	Each slave sends complement bit 64 of its ROM number.
Write bit 64 direction (according to algorithm)	Each slave receives the bit written by master, if bit read is not the same as bit 64 of its ROM number then go into a wait state.

On examination of Table 1, it is obvious that if all of the participating devices have the same value in a bit position then there is only one choice for the branch path to be taken. The condition where no devices are participating is an atypical situation that can arise if the device being discovered is removed from the 1- Wire during the search. If this situation arises then the search should be terminated and a new search could be done starting with a 1-Wire reset. A discrepancy, the condition where there are both 0s and 1s in the bit position, is the key to finding devices in the subsequent searches. The search algorithm specifies that on the first pass, when there is a discrepancy (bit/complement = 0/0), the '0' path is taken. Note that this is arbitrary for this particular algorithm. Another algorithm could be devised to use the '1' path first. The bit position for the last discrepancy is recorded for use in the next search. **Table 3** describes the paths that are taken on subsequent searches when a discrepancy occurs.

Table 3. Search Path Direction

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

The search algorithm also keeps track of the last discrepancy that occurs within the first eight bits of the algorithm. The first eight bits of the 64-bit registration number is a family code. As a result, the devices discovered during the search are grouped into family types. The last discrepancy within that family code can be used to selectively skip whole groups of 1-Wire devices. See the description of *ADVANCED SEARCH VARIATIONS* for doing selective searches. The 64-bit ROM number also contains an 8-bit cyclic-redundancy-check (CRC). This CRC value is verified to ensure that only correct ROM numbers are discovered. See Figure 1 for the layout of the ROM number.

The DS2480B Serial to 1-Wire Line Driver performs some of this same search algorithm in hardware. See the DS2480B data sheet and application note 192, *Using the DS2480B Serial 1-Wire Line Driver* for details. The DS2490 USB to 1-Wire bridge performs the entire search in hardware.

Figure 2 shows a flow chart of the search sequence. Note the *Reference* side bar that explains the terms used in the flow chart. These terms are also used in the source code appendix to this document.

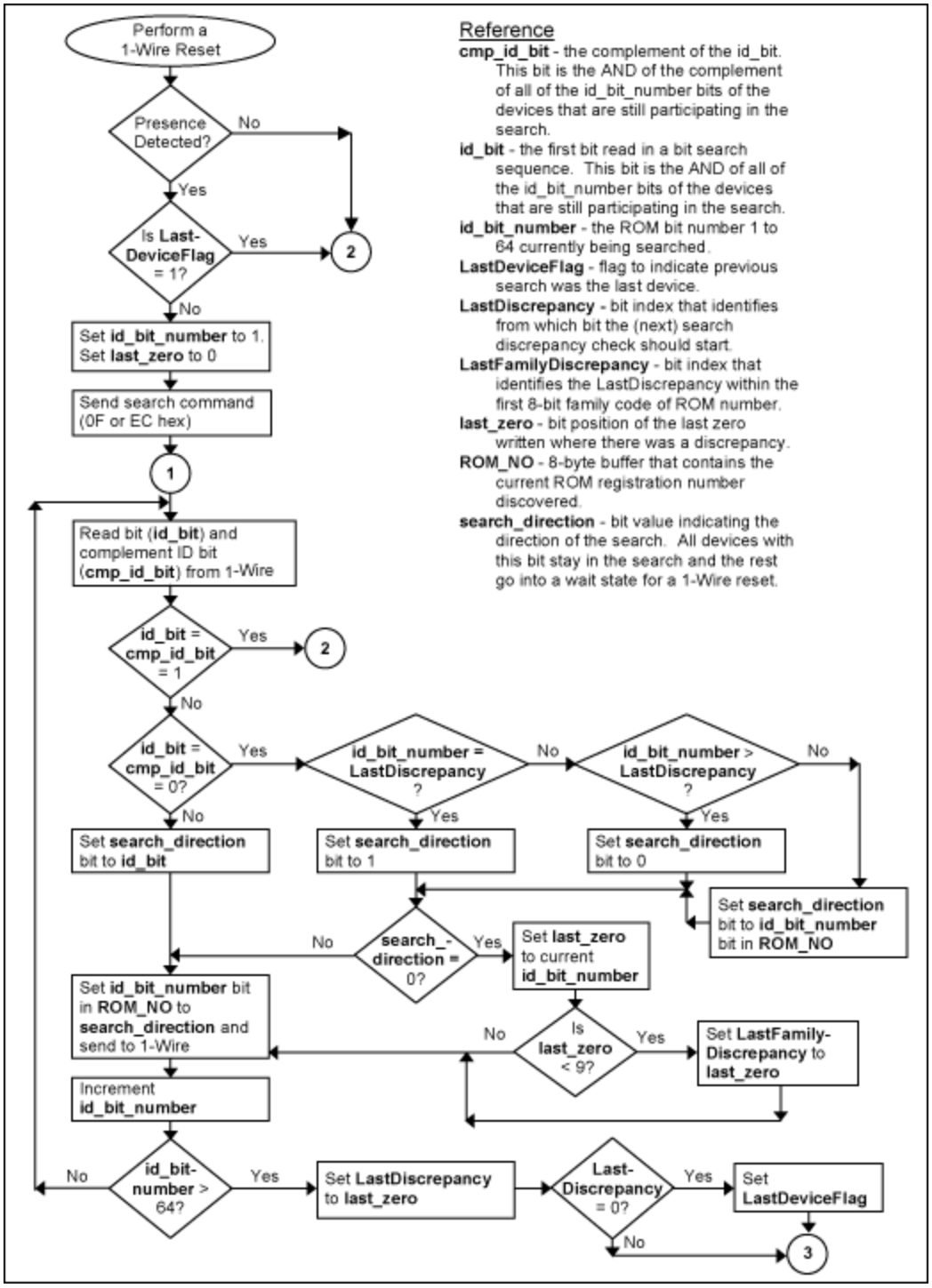


Figure 2. Search flow.

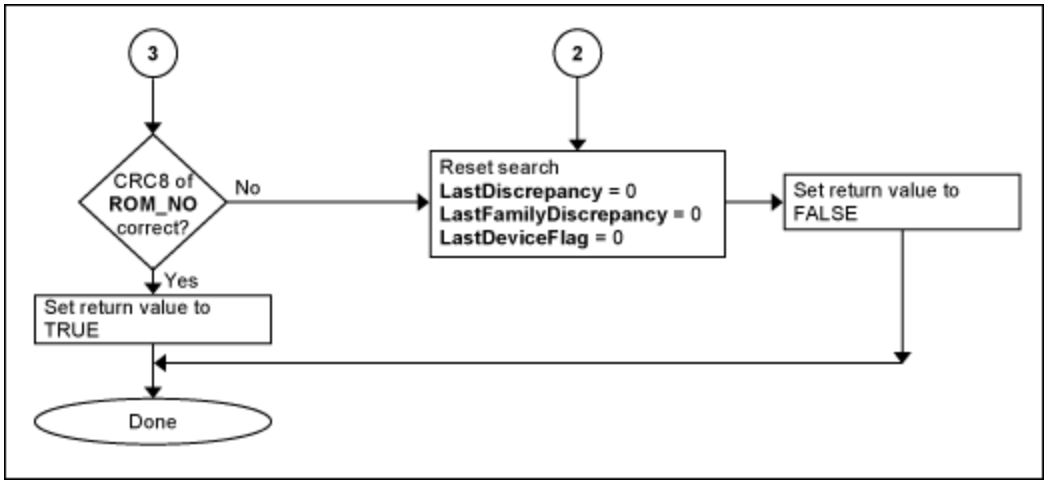


Figure 2. Search flow part II.

There are two basic types of operations that can be performed by using the search algorithm by manipulating the LastDiscrepancy, LastFamilyDiscrepancy, LastDeviceFlag, and ROM_NO register values (see **Table 4**). These operations concern basic discovery of the ROM numbers of 1-Wire devices.

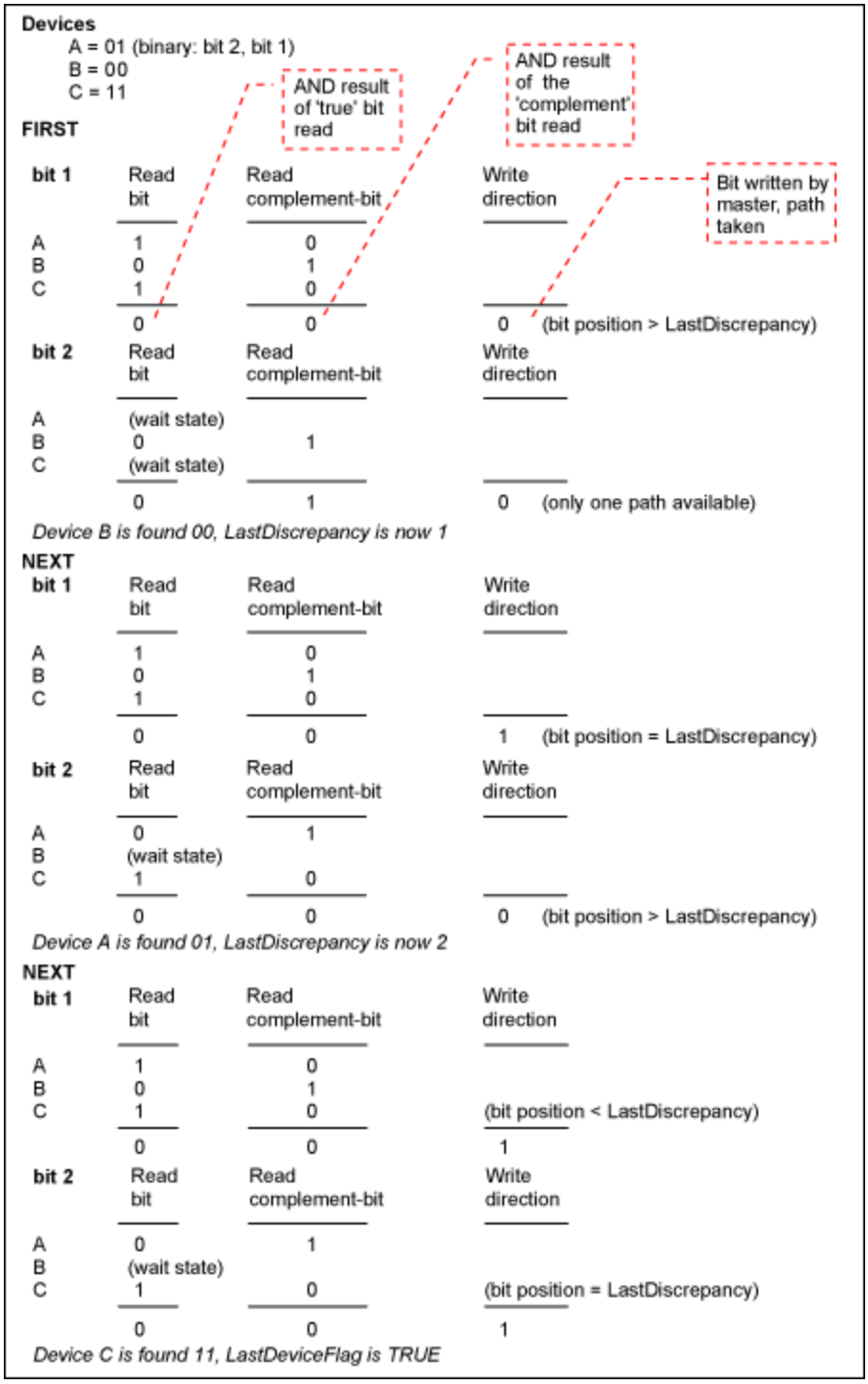
First

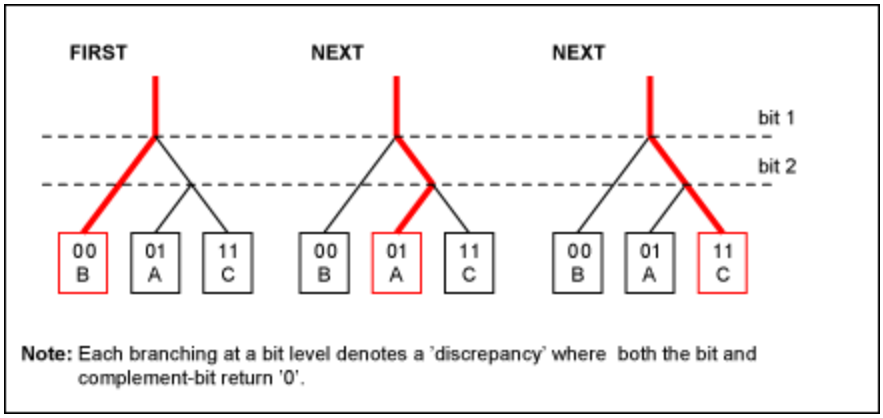
The 'FIRST' operation is to search on the 1-Wire for the first device. This is performed by setting LastDiscrepancy, LastFamilyDiscrepancy, and LastDeviceFlag to zero and then doing the search. The resulting ROM number can then be read from the ROM_NO register. If no devices are present on the 1-Wire the reset sequence does not detect a presence and the search is aborted.

Next

The 'NEXT' operation is to search on the 1-Wire for the next device. This search is usually performed after a 'FIRST' operation or another 'NEXT' operation. It is performed by leaving the state unchanged from the previous search and performing another search. The resulting ROM number can then be read from the ROM_NO register. If the previous search was the last device on the 1-Wire then the result is FALSE and the condition is set to execute a 'FIRST' with the next call of the search algorithm.

Figure 3 (a, b, c) goes through a simple search example with three devices. For illustration, this example assumes devices with a 2-bit ROM number only.


Devices
A = 01 (binary: bit 2, bit 1)
B = 00
C = 11
FIRST
AND result of 'true' bit read
AND result of the 'complement' bit read
Bit written by master, path taken
bit 1
Read bit
Read complement-bit
Write direction
A 1 0
B 0 1
C 1 0
0 0 0 (bit position > LastDiscrepancy)
bit 2
Read bit
Read complement-bit
Write direction
A (wait state)
B 0 1
C (wait state)
0 1 0 (only one path available)
Device B is found 00, LastDiscrepancy is now 1
NEXT
bit 1
Read bit
Read complement-bit
Write direction
A 1 0
B 0 1
C 1 0
0 0 1 (bit position = LastDiscrepancy)
bit 2
Read bit
Read complement-bit
Write direction
A 0 1
B (wait state)
C 1 0
0 0 0 (bit position > LastDiscrepancy)
Device A is found 01, LastDiscrepancy is now 2
NEXT
bit 1
Read bit
Read complement-bit
Write direction
A 1 0
B 0 1
C 1 0 (bit position < LastDiscrepancy)
0 0 1
bit 2
Read bit
Read complement-bit
Write direction
A 0 1
B (wait state)
C 1 0 (bit position = LastDiscrepancy)
0 0 1
Device C is found 11, LastDeviceFlag is TRUE



Note: Each branching at a bit level denotes a 'discrepancy' where both the bit and complement-bit return '0'.

(for simplicity the family discrepancy register and tracking has been left out of this example)

FIRST

- **LastDiscrepancy** = **LastDeviceFlag** = 0
- Do 1-Wire reset and wait for presence pulse, if no presence pulse then done
- **id_bit_number** = 1, **last_zero** = 0
- Send search command, 0F hex
- Read first bit **id_bit**: 1 (Device A) AND 0 (Device B) AND 1 (Device C) = 0
- Read complement of first bit **cmp_id_bit**: 0 (Device A) AND 1 (Device B) AND 0 (Device C) = 0
- Since **id_bit_number** > **LastDiscrepancy** then **search_direction** = 0, **last_zero** = 1
- Send **search_direction** bit of 0, both Devices A and C go into wait state
- Increment **id_bit_number** to 2
- Read second bit **id_bit**: 0 (Device B) = 0
- Read complement of second bit **cmp_id_bit**: 1 (Device B) = 1
- Since bit and complement are different then **search_direction** = **id_bit**
- Send **search_direction** bit of 0, Device B is discovered with **ROM_NO** of '00' and is now selected
- **LastDiscrepancy** = **last_zero**

NEXT

- Do 1-Wire reset and wait for presence pulse, if no presence pulse then done
- **id_bit_number** = 1, **last_zero** = 0
- Send search command, 0F hex
- Read first bit **id_bit**: 1 (Device A) AND 0 (Device B) AND 1 (Device C) = 0
- Read complement of first bit **cmp_id_bit**: 0 (Device A) AND 1 (Device B) AND 0 (Device C) = 0
- Since **id_bit_number** = **LastDiscrepancy** then **search_direction** = 1
- Send **search_direction** bit of 1, Devices B goes into wait state
- Increment **id_bit_number** to 2
- Read second bit **id_bit**: 0 (Device A) AND 1 (Device C) = 0
- Read complement of second bit **cmp_id_bit**: 1 (Device A) AND 0 (Device C) = 0
- Since **id_bit_number** > **LastDiscrepancy** then **search_direction** = 0, **last_zero** = 2
- Send **search_direction** bit of 0, Devices C goes into wait state
- Device A is discovered with **ROM_NO** of '01' and is now selected
- **LastDiscrepancy** = **last_zero**

NEXT

- Do 1-Wire reset and wait for presence pulse, if no presence pulse then done
- **id_bit_number** = 1, **last_zero** = 0
- Send search command, 0F hex
- Read first bit **id_bit**: 1 (Device A) AND 0 (Device B) AND 1 (Device C) = 0
- Read complement of first bit **cmp_id_bit**: 0 (Device A) AND 1 (Device B) AND 0 (Device C) = 0
- Since **id_bit_number** < **LastDiscrepancy** then **search_direction** = **ROM_NO** (first bit) = 1
- Send **search_direction** bit of 1, Devices B goes into wait state
- Increment **id_bit_number** to 2
- Read second bit **id_bit**: 0 (Device A) AND 1 (Device C) = 0
- Read complement of second bit **cmp_id_bit**: 1 (Device A) AND 0 (Device C) = 0
- Since **id_bit_number** = **LastDiscrepancy** then **search_direction** = 1
- Send **search_direction** bit of 1, Devices A goes into wait state
- Device C is discovered with **ROM_NO** of '11' and is now selected
- **LastDiscrepancy** = **last_zero** which is 0 so **LastDeviceFlag** = TRUE

NEXT

- **LastDeviceFlag** is true so return FALSE
- **LastDiscrepancy** = **LastDeviceFlag** = 0

Figure 3. Search Example.

Advanced Search Variations

There are three advanced search variations using the same state information, namely LastDiscrepancy, LastFamilyDiscrepancy, LastDeviceFlag, and ROM_NO. These variations allow specific family types to be

targeted or skipped and device present verification (see **Table 4**).

Verify

The 'VERIFY' operation verifies if a device with a known ROM number is currently connected to the 1-Wire. It is accomplished by supplying the ROM number and doing a targeted search on that number to verify it is present. First, set the ROM_NO register to the known ROM number. Then set the LastDiscrepancy to 64 (40 hex) and the LastDeviceFlag to 0. Perform the search operation and then read the ROM_NO result. If the search was successful and the ROM_NO remains the ROM number that was being searched for, then the device is currently on the 1-Wire.

Target Setup

The 'TARGET SETUP' operation is a way to preset the search state to first find a particular family type. Each 1-Wire device has a one byte *family* code embedded within the ROM number (see Figure 1). This family code allows the 1-Wire master to know what operations this device is capable of. If there are multiple devices on the 1-Wire it is common practice to target a search to only the family of devices that are of interest. To target a particular family, set the desired family code byte into the first byte of the ROM_NO register and fill the rest of the ROM_NO register with zeros. Then set the LastDiscrepancy to 64 (40 hex) and both LastDeviceFlag and LastFamilyDiscrepancy to 0. When the search algorithm is next performed the first device of the desired family type is discovered and placed in the ROM_NO register. Note that if no devices of the desired family are currently on the 1-Wire, then another type will be found, so the family code in the resulting ROM_NO must be verified after the search.

Family Skip Setup

The 'FAMILY SKIP SETUP' operation sets the search state to skip all of the devices that have the family code that was found in the previous search. This operation can only be performed after a search. It is accomplished by copying the LastFamilyDiscrepancy into the LastDiscrepancy and clearing out the LastDeviceFlag. The next search then finds devices that come after the current family code. If the current family code group is the last group in the search then the search returns with the LastDeviceFlag set.

Table 4. Search Variations State Setup

	LastDiscrepancy	LastFamily-Discrepancy	LastDeviceFlag	ROM_NO
FIRST	0	0	0	Result
NEXT	Leave unchanged	Leave unchanged	Leave unchanged	Result
VERIFY	64	0	0	Set with ROM to verify, check if same after search
TARGET SETUP	64	0	0	Set first byte to family code, set rest to zeros

FAMILY SKIP SETUP	Copy from LastFamilyDiscrepancy	0	0	Leave unchanged

Conclusion

The supplied search algorithm allows the discovery of the individually unique ROM numbers from any given group of 1-Wire devices. This is essential to any multidrop 1-Wire application. With the ROM numbers in hand, each 1-Wire device can be selected individually for operations. This document also discusses search variations to find or skip particular 1-Wire device types. See the *Appendix* for a 'C' code example implementation of the search and all of the search variations.

Appendix

Figure 4 shows a 'C' code implementation of the search algorithm along with a function for each search variation. The FamilySkipSetup and TargetSetup functions do not actually do a search, they just set up the search registers so the next 'Nextí skips or finds the desired type. Note that the low-level 1-Wire functions are implemented with calls to the TMEX API. These calls are for test purposes and can be replaced with platform-specific calls. See application note 155, "*1-Wire® Software Resource Guide Device Description*" for a description of the TMEX API and other 1-Wire APIs.

The TMEX API test implementation of the following code example can be downloaded from the Maxim website.

```
// TMEX API TEST BUILD DECLARATIONS
#define TMEXUTIL
#include "ibtmexcw.h"
long session_handle;
// END TMEX API TEST BUILD DECLARATIONS

// definitions
#define FALSE 0
#define TRUE  1

// method declarations
int  OWFirst();
int  OWNext();
int  OWVerify();
void OWTargetSetup(unsigned char family_code);
void OWFamilySkipSetup();
int  OWReset();
void OWWriteByte(unsigned char byte_value);
void OWWriteBit(unsigned char bit_value);
unsigned char OWReadBit();
int  OWSearch();
unsigned char docrc8(unsigned char value);

// global search state
```

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

//--------------------------------------------------------------------------
// Find the 'first' devices on the 1-Wire bus
// 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 bus
// 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();
}

//--------------------------------------------------------------------------
// Perform the 1-Wire Search Algorithm on the 1-Wire bus using the existing
// search state.
// Return TRUE  : device found, ROM number in ROM_NO buffer
//        FALSE : device not found, end of search
//
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;

   // initialize for search
   id_bit_number = 1;
   last_zero = 0;
   rom_byte_number = 0;
```

```
   rom_byte_mask = 1;
   search_result = 0;
   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
      {
         // read a bit and its complement
         id_bit = OWReadBit();
         cmp_id_bit = OWReadBit();

         // check for no devices on 1-wire
         if ((id_bit == 1) && (cmp_id_bit == 1))
            break;
         else
         {
            // all devices coupled have 0 or 1
            if (id_bit != cmp_id_bit)
               search_direction = id_bit;  // bit write value for search
            else
            {
               // if this discrepancy if before the Last Discrepancy
               // on a previous next then pick the same as last time
               if (id_bit_number < LastDiscrepancy)
                  search_direction = ((ROM_NO[rom_byte_number] &
rom_byte_mask) > 0);
               else
                  // if equal to last pick 1, if not then pick 0
                  search_direction = (id_bit_number == LastDiscrepancy);

               // if 0 was picked then record its position in LastZero
               if (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] &= ~rom_byte_mask;

            // serial number search direction write bit
            OWWriteBit(search_direction);

            // 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)
            {
                docrc8(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])
   {
      LastDiscrepancy = 0;
      LastDeviceFlag = FALSE;
      LastFamilyDiscrepancy = 0;
      search_result = FALSE;
   }

   return search_result;
}

//--------------------------------------------------------------------------
// Verify the device with the ROM number in ROM_NO buffer is present.
// Return TRUE  : device verified present
//        FALSE : device not present
//
int OWVerify()
{
   unsigned char rom_backup[8];
   int i,rslt,ld_backup,ldf_backup,lfd_backup;

   // keep a backup copy of the current state
   for (i = 0; i < 8; i++)
      rom_backup[i] = ROM_NO[i];
   ld_backup = LastDiscrepancy;
   ldf_backup = LastDeviceFlag;
   lfd_backup = LastFamilyDiscrepancy;

   // set search to find the same device
   LastDiscrepancy = 64;
   LastDeviceFlag = FALSE;

   if (OWSearch())
   {
      // check if same device found
      rslt = TRUE;
      for (i = 0; i < 8; i++)
      {
         if (rom_backup[i] != ROM_NO[i])
         {
            rslt = FALSE;
            break;
         }
      }
   }
   else
```

```
     rslt = FALSE;

   // restore the search state
   for (i = 0; i < 8; i++)
      ROM_NO[i] = rom_backup[i];
   LastDiscrepancy = ld_backup;
   LastDeviceFlag = ldf_backup;
   LastFamilyDiscrepancy = lfd_backup;

   // return the result of the verify
   return rslt;
}

//--------------------------------------------------------------------------
// Setup the search to find the device type 'family_code' on the next call
// to OWNext() if it is present.
//
void OWTargetSetup(unsigned char family_code)
{
   int i;

   // set the search state to find SearchFamily type devices
   ROM_NO[0] = family_code;
   for (i = 1; i < 8; i++)
      ROM_NO[i] = 0;
   LastDiscrepancy = 64;
   LastFamilyDiscrepancy = 0;
   LastDeviceFlag = FALSE;
}

//--------------------------------------------------------------------------
// Setup the search to skip the current device type on the next call
// to OWNext().
//
void OWFamilySkipSetup()
{
   // set the Last discrepancy to last family discrepancy
   LastDiscrepancy = LastFamilyDiscrepancy;
   LastFamilyDiscrepancy = 0;

   // check for end of list
   if (LastDiscrepancy == 0)
      LastDeviceFlag = TRUE;
}

//--------------------------------------------------------------------------
// 1-Wire Functions to be implemented for a particular platform
//--------------------------------------------------------------------------
```

```
//---------------------------------------------------------------------------
// Reset the 1-Wire bus and return the presence of any device
// Return TRUE  : device present
//        FALSE : no device present
//
int OWReset()
{
   // platform specific
   // TMEX API TEST BUILD
   return (TMTouchReset(session_handle) == 1);
}

//---------------------------------------------------------------------------
// Send 8 bits of data to the 1-Wire bus
//
void OWWriteByte(unsigned char byte_value)
{
   // platform specific

   // TMEX API TEST BUILD
   TMTouchByte(session_handle,byte_value);
}

//---------------------------------------------------------------------------
// Send 1 bit of data to teh 1-Wire bus
//
void OWWriteBit(unsigned char bit_value)
{
   // platform specific

   // TMEX API TEST BUILD
   TMTouchBit(session_handle,(short)bit_value);
}

//---------------------------------------------------------------------------
// Read 1 bit of data from the 1-Wire bus
// Return 1 : bit read is 1
//        0 : bit read is 0
//
unsigned char OWReadBit()
{
   // platform specific

   // TMEX API TEST BUILD
   return (unsigned char)TMTouchBit(session_handle,0x01);

}

// TEST BUILD
```

```
static unsigned char dscrc_table[] = {
        0, 94,188,226, 97, 63,221,131,194,156,126, 32,163,253, 31, 65,
      157,195, 33,127,252,162, 64, 30, 95,  1,227,189, 62, 96,130,220,
       35,125,159,193, 66, 28,254,160,225,191, 93,  3,128,222, 60, 98,
      190,224,  2, 92,223,129, 99, 61,124, 34,192,158, 29, 67,161,255,
       70, 24,250,164, 39,121,155,197,132,218, 56,102,229,187, 89,  7,
      219,133,103, 57,186,228,  6, 88, 25, 71,165,251,120, 38,196,154,
      101, 59,217,135,  4, 90,184,230,167,249, 27, 69,198,152,122, 36,
      248,166, 68, 26,153,199, 37,123, 58,100,134,216, 91,  5,231,185,
      140,210, 48,110,237,179, 81, 15, 78, 16,242,172, 47,113,147,205,
       17, 79,173,243,112, 46,204,146,211,141,111, 49,178,236, 14, 80,
      175,241, 19, 77,206,144,114, 44,109, 51,209,143, 12, 82,176,238,
       50,108,142,208, 83, 13,239,177,240,174, 76, 18,145,207, 45,115,
      202,148,118, 40,171,245, 23, 73,  8, 86,180,234,105, 55,213,139,
       87,  9,235,181, 54,104,138,212,149,203, 41,119,244,170, 72, 22,
      233,183, 85, 11,136,214, 52,106, 43,117,151,201, 74, 20,246,168,
      116, 42,200,150, 21, 75,169,247,182,232, 10, 84,215,137,107, 53};

//--------------------------------------------------------------------------
// Calculate the CRC8 of the byte value provided with the current
// global 'crc8' value.
// Returns current global crc8 value
//
unsigned char docrc8(unsigned char value)
{
   // See Application Note 27

   // TEST BUILD
   crc8 = dscrc_table[crc8 ^ value];
   return crc8;
}

//--------------------------------------------------------------------------
// TEST BUILD MAIN
//
int main(short argc, char **argv)
{
   short PortType=5,PortNum=1;
   int rslt,i,cnt;

   // TMEX API SETUP
   // get a session
   session_handle = TMExtendedStartSession(PortNum,PortType,NULL);
   if (session_handle <= 0)
   {
      printf("No session, %d\n",session_handle);
      exit(0);
   }
```

```
// setup the port
rslt = TMSetup(session_handle);
if (rslt != 1)
{
   printf("Fail setup, %d\n",rslt);
   exit(0);
}
// END TMEX API SETUP

// find ALL devices
printf("\nFIND ALL\n");
cnt = 0;
rslt = OWFirst();
while (rslt)
{
   // print device found
   for (i = 7; i >= 0; i--)
      printf("%02X", ROM_NO[i]);
   printf("  %d\n",++cnt);

   rslt = OWNext();
}

// find only 0x1A
printf("\nFIND ONLY 0x1A\n");
cnt = 0;
OWTargetSetup(0x1A);
while (OWNext())
{
   // check for incorrect type
   if (ROM_NO[0] != 0x1A)
      break;

   // print device found
   for (i = 7; i >= 0; i--)
      printf("%02X", ROM_NO[i]);
   printf("  %d\n",++cnt);
}

// find all but 0x04, 0x1A, 0x23, and 0x01
printf("\nFIND ALL EXCEPT 0x10, 0x04, 0x0A, 0x1A, 0x23, 0x01\n");
cnt = 0;
rslt = OWFirst();
while (rslt)
{
   // check for incorrect type
   if ((ROM_NO[0] == 0x04) || (ROM_NO[0] == 0x1A) ||
       (ROM_NO[0] == 0x01) || (ROM_NO[0] == 0x23) ||
       (ROM_NO[0] == 0x0A) || (ROM_NO[0] == 0x10))
```

```
          OWFamilySkipSetup();
      else
      {
         // print device found
         for (i = 7; i >= 0; i--)
            printf("%02X", ROM_NO[i]);
         printf("  %d\n",++cnt);
      }

      rslt = OWNext();
   }

   // TMEX API CLEANUP
   // release the session
   TMEndSession(session_handle);
   // END TMEX API CLEANUP
}
```

Revision History

01/30/02 Version 1.0—Initial release

05/16/03 Version 1.1—Corrections: Search ROM commands corrected to F0 hex.

Related Parts		
DS18B20	Programmable Resolution 1-Wire Digital Thermometer	Free Samples
DS18S20	1-Wire Parasite-Power Digital Thermometer	Free Samples
DS1904	iButton RTC	Free Samples
DS1920	iButton Temperature Logger	
DS1921G	Thermochron iButton Device	
DS1963S	iButton Monetary Device with SHA-1 Function	
DS1971	iButton 256-Bit EEPROM	
DS1973	iButton 4Kb EEPROM	Free Samples
DS1982	iButton 1Kb Add-Only	Free Samples
DS1985	iButton 16Kb Add-Only	Free Samples
DS1990A	iButton Serial Number	Free Samples
DS1992	iButton 1Kb/4Kb Memory	Free Samples
DS1993	iButton 1Kb/4Kb Memory	Free Samples
DS1995	iButton 16Kb Memory	Free Samples

DS1996	iButton 64Kb Memory	Free Samples
DS2401	Silicon Serial Number	Free Samples
DS2406	Dual Addressable Switch Plus 1Kb Memory	Free Samples
DS2408	1-Wire 8-Channel Addressable Switch	Free Samples
DS2411	Silicon Serial Number with V_{CC} Input	Free Samples
DS2411	Silicon Serial Number with V_{CC} Input	Free Samples
DS2417	1-Wire Time Chip With Interrupt	Free Samples
DS2431	1024-Bit 1-Wire EEPROM	Free Samples
DS2432	1Kb Protected 1-Wire EEPROM with SHA-1 Engine	Free Samples
DS2433	4Kb 1-Wire EEPROM	
DS2438	Smart Battery Monitor	Free Samples
DS2450	1-Wire Quad A/D Converter	
DS2502	1Kb Add-Only Memory	Free Samples
DS2502	1Kb Add-Only Memory	Free Samples
DS2505	16Kb Add-Only Memory	Free Samples
DS2506	64Kb Add-Only Memory	
DS2740	High-Precision Coulomb Counter	Free Samples
DS2762	High-Precision Li+ Battery Monitor with Alerts	Free Samples
MAX31826	1-Wire Digital Temperature Sensor with 1Kb Lockable EEPROM	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

Application Note 187: http://www.maximintegrated.com/an187
APPLICATION NOTE 187, AN187, AN 187, APP187, Appnote187, Appnote 187