DALLAS SEMICONDUCTOR

アプリケーションノート187
1-Wire 検索アルゴリズム

japan.maxim-ic.com

はじめに

ダラスセミコンダクタの各 1-Wire®デバイスは読み取り専用メモリ(ROM)内に一意の 64 ビット登録番号を備え、その番号を使って 1-Wireネットワーク内の 1-Wireマスタは 1-Wireデバイスそれぞれのアドレスを指定します。1-Wire ネットワーク上のスレーブデバイスのROM番号が認識されていない場合は、検索アルゴリズムを使ってその番号を検出することができます。当ドキュメントでは検索アルゴリズムを詳しく説明し、迅速な統合のための実装例を紹介します。このアルゴリズムは、1-Wire インタフェースを特徴とする現行および将来の全デバイスに適用されます。

一意の 64 ビット ROM「登録」番号 図 1

MSB	64-bit 'Registration' ROM number	LSB
8-bit CRC MSB LSB	48-bit Serial Number MSB LSB	8-bit Family Code MSB LSB

検索アルゴリズム

検索アルゴリズムは、デバイス ROM 番号すなわちリーフが検出されるまで分岐を追跡するバリナリーツリー探索です。存在するリーフがすべて検出されるまで、以降の検索が残りの他の分岐パスをたどります。

検索アルゴリズムは、リセットおよびプレゼンスパルスシーケンスによりリセット中の 1-Wire のデバイスから始まります。これが成功すると、1-byte 検索コマンドが送信されます。検索コマンドは、1-Wire デバイスに検索を開始させます。

2 種類の検索コマンドがあります。標準的な検索コマンド(16 進法の F0)は、全デバイスを参加させ検索を実行します。アラームすなわち条件付き検索コマンド(16 進法の EC)は、ある程度アラーム状態にあるデバイスのみ検索を実行します。これにより、注意を要するデバイスに検索プールが迅速に対応します。

検索コマンドに従って、実際の検索は、参加する全デバイスが ROM 番号(別名、登録番号)の先頭ビット(最下位ビット)を同時送信することから始まります(図 1 参照)。すべての 1-Wire 通信と同様に、1-Wire マスタは、読み取りデータまたはスレーブデバイスへの書き込みデータを問わず全ビットを開始します。1-Wire の特性により、全デバイスが同時に応答すると、結果は送信ビットの論理 AND になります。デバイスが ROM 番号の先頭ビットを送信すると、マスタは次のビットを開始し、デバイスは先頭ビットの補数を送信します。以上の 2 ビットから、参加デバイスの ROM 番号の先頭ビットに関する情報を得ることができます(表 1 参照)。

ビット検索情報 表 1

ビット (true)	ビット (補数)	確認情報
0	0	参加 ROM 番号の現在のビット位置に 0 と 1 が両方ある。両方存在するのは不一致である。
0	1	参加 ROM 番号のビットに 0 のみがある。
1	0	参加 ROM 番号のビットに 1 のみがある。
1	1	検索に参加するデバイスなし。

次に、検索アルゴリズムに従って 1-Wire マスタは 1 ビットを参加デバイスに送り返す必要があります。参加するデバイスがそのビット値を備えている場合は、当デバイスは引き続き参加します。参加デバイスがビット値を備えていない場合は、次の 1-Wire リセットが検出されるまで、待機状態になります。次に、この「2 ビット読み取り」および「1 ビット書き込み」パターンが、ROM 番号の残り 63 ビットについて繰り返されます(表 2 参照)。こうして、検索アルゴリズムにより 1 台を除き残りの全デバイスが待機状態になります。1 つのパスの終了時に、最終デバイスの ROM 番号が認識されます。以降の検索パスの際には、その他のデバイス ROM 番号を検出するために別のパス (すなわち分岐) を通ります。当ドキュメントでは、ビット 1 (最下位ビット) ~ビット 64 (最上位ビット)を ROM 番号のビット位置と呼ぶことに注意してください。便宜上、ビット 0~63 の代わりにこの表記法を使って、後で対照するために不一致カウンタを 0 に初期設定することができます。

1-Wire マスタおよびスレーブ検索シーケンス 表 2

マスタ	スレーブ
1-Wire リセットの促進	プレゼンスパルスの作成。
検索コマンド(標準またはアラーム)の書き込み	各スレーブが検索に対応。.
ビット 1 の「AND」の読み取り	各スレーブが ROM 番号のビット 1 を送信。
補数ビット 1 の「AND」の読み取り	各スレーブが ROM 番号の補数ビット 1 を送信。
ビット 1 の方法を書き込む(アルゴリズムに準拠)	各スレーブがマスタ書き込みのビットを受信。読み取りビットが ROM 番号のビット 1 と同一でない場合は、待機状態になる。
⋮	⋮
ビット 64 の「AND」の読み取り	各スレーブが ROM 番号のビット 64 を送信。
補数ビット 64 の「AND」の読み取り	各スレーブが ROM 番号の補数ビット 64 を送信。
ビット 64 の方法を書き込む(アルゴリズムに準拠)	各スレーブがマスタ書き込みのビットを受信。読み取りビットが ROM 番号のビット 64 と同一でない場合は、待機状態になる。

表 1 をよく見ると、全参加デバイスがビット位置に同じ値を持っていると、通るべき分岐パスはたった 1 つしかないことが明らかです。参加デバイスがないという状態は、検索時に検出中のデバイスが 1-Wire から除去される場合に発生する異常な状態です。ビット位置に 0 と 1 がともにあるという状態は不一致と呼ばれ、以降の検索でデバイスを検出する手がかりとなります。検索アルゴリズムでは、最初のパスで不一致(ビット/補数 = 0/0)があると、「0」パスを通るように指示します。ただし、これは当アルゴリズムでは任意であることに注意してください。最初に「1」パスを通る別のアルゴリズムを作成することができます。最後の不一致のビット位置は、次の検索で用いるために記録されます。表 3 では、不一致が発生した際に以降の検索で通るパスを説明しています。

検索パスの方向 表 3

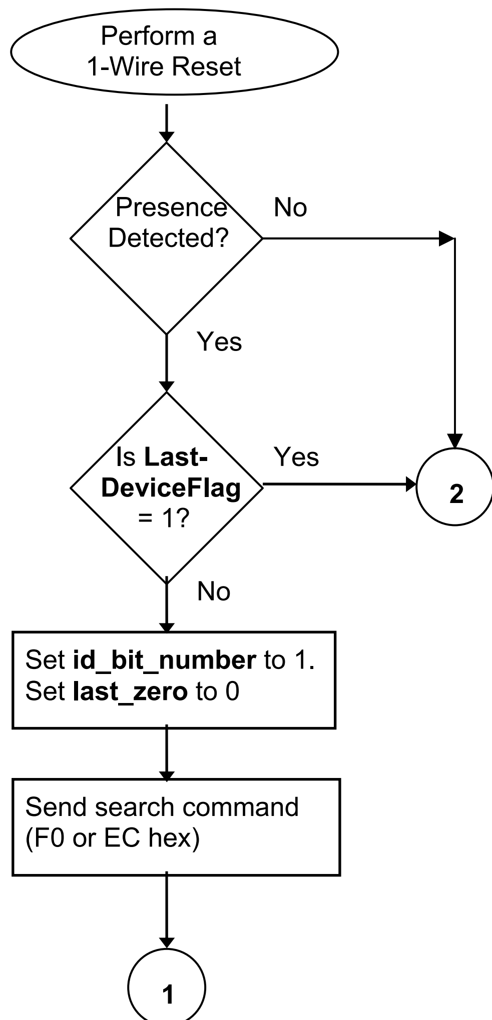
検索ビット位置および最後の不一致	**通るパス**
＝	「1」パスを通る
＜	(検出された最後の ROM 番号から)最後と同じパスを通る
＞	「0」パスを通る

また、検索アルゴリズムは、アルゴリズムの先頭の 8 ビット内で発生する最後の不一致を追跡します。64 ビット登録番号の先頭の 8 ビットは、ファミリコードです。このため、検索時に検出されたデバイスは、ファミリタイプに分類されます。このファミリコード内の最後の不一致を使って、1-Wire デバイスの全グループを抜粋してにスキップすることができます。抜粋した検索を実行するには、*Advanced Search Variations*(詳細検索バリエーション)の説明を参照してください。また、64 ビット ROM 番号には、8 ビット巡回冗長検査(CRC)も含まれています。正しい ROM 番号のみが検出されているかを確認するために、この CRC の値が検証されます。ROM 番号の配置については、図 1 を参照してください。

シリアルから 1-WireへのラインドライバDS2480Bでは、ハードウェアで同じ当検索アルゴリズムの一部を実行します。詳細については、*DS2480Bのデータシートおよびアプリケーションノート 192:DS2480Bシリアル 1-Wireラインドライバの使用*(http://pdfserv.maxim-ic.com/jp/an/app192J.pdf)を参照してください。USBから 1-Wireへの変換チップDS2490 では、ハードウェアで検索全体を実行します。

図 2 では、検索シーケンスのフローチャートを紹介しています。このフローチャートで使われる用語を説明する*リファレンス*サイドバーに注意してください。また、こうした用語は当ドキュメントのソースコード付録でも使われています。

検索フロー 図 2 (次ページに続く)



リファレンス

cmp_id_bit - id_bit の補数。当ビットは、検索にまだ参加しているデバイスの全 id_bit_number ビットの AND および補数である。

id_bit - ビット検索シーケンスで読み取られた最初のビット。当ビットは、検索にまだ参加しているデバイスの全 id_bit_number ビットの AND である。

id_bit_number - 現在検索中の ROM ビット番号 1～64。

LastDeviceFlag - 前の検索が最後のデバイスであったことを示すフラグ。

LastDiscrepancy - (次の**)**検索不一致チェックがどのビットから開始するかを識別するビット索引。

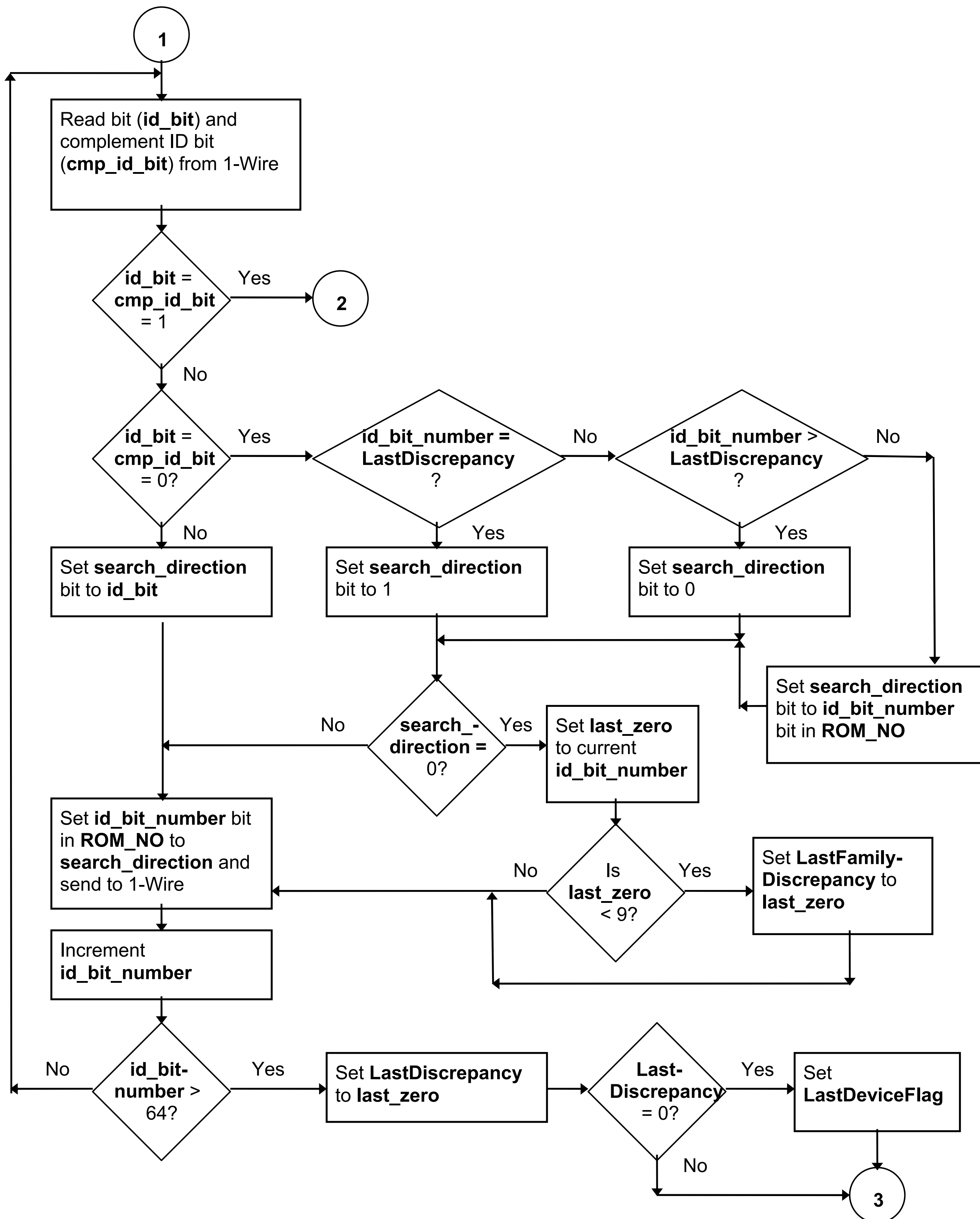
LastFamilyDiscrepancy - ROM 番号の先頭の 8 ビットファミリコード内の LastDiscrepancy を識別するビット索引。

last_zero - 不一致があった場合、書き込まれる最後のゼロのビット位置。

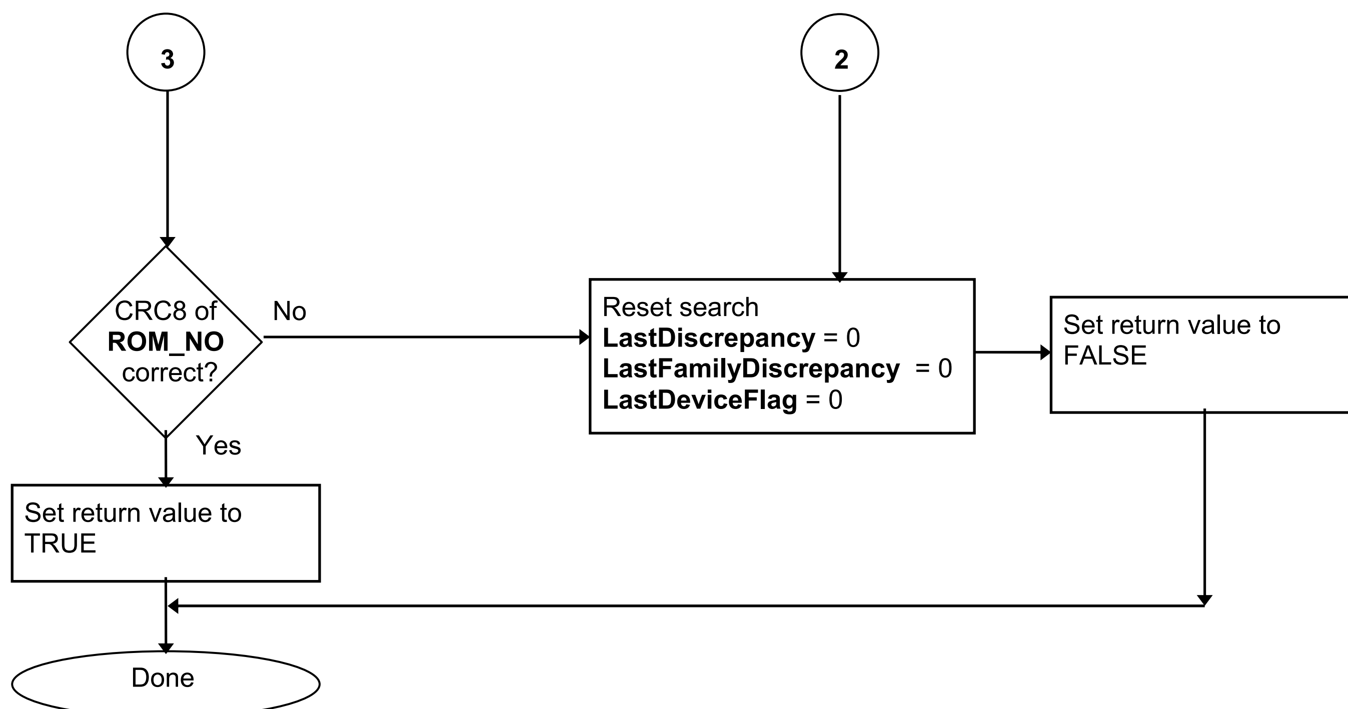
ROM_NO - 検出された現在の ROM 登録番号を含む 8 バイトバッファ。

search_direction - 検索方向を示すビット値。当ビットを持つ全デバイスは検索に参加し、残りのデバイスは 1-Wire リセット用に待機状態になる。

検索フロー 図 2 (続き)



検索フロー 図 2 (続き)



LastDiscrepancy、LastFamilyDiscrepancy、LastDeviceFlag、および ROM_NO レジスタの操作により検索アルゴリズムを用いて、実行することができる 2 種類の基本的なオペレーションがあります(表 4 参照)。これらのオペレーションは、1-Wire デバイスの ROM 番号の基本的検出に関するものです。

FIRST

「FIRST」オペレーションとは、1-Wire 上の最初のデバイスの検索です。このオペレーションを実行すると、LastDiscrepancy、LastFamilyDiscrepancy、および LastDeviceFlag をゼロに設定して、検索を実行します。次に、ROM 番号の結果は、ROM_NO レジスタから読み取ることができます。1-Wire 上にデバイスが存在しない場合は、リセットシーケンスは存在を検出せず、検索が終了します。

NEXT

「NEXT」オペレーションとは、1-Wire 上の次のデバイスの検索です。この検索は、「FIRST」オペレーションまたは別の「NEXT」オペレーションの後に通常実行されます。このオペレーションを実行すると、状態を前回の検索と同じにして、新たな検索を実行します。次に、ROM 番号の結果は、ROM_NO レジスタから読み取ることができます。前回の検索が 1-Wire 上の最後のデバイスの場合は、その結果は FALSE で、検索アルゴリズムの次の呼び出しにより、「FIRST」を実行するように状態が設定されます。

図 3 (a、b、c)では、3 つのデバイスの簡単な検索例を紹介しています。説明用にこの例では 2 ビット ROM 番号付きデバイスのみを想定しています。

検索例 図 3a

Devices

A = 01 (binary: bit 2, bit 1)
B = 00
C = 11

AND result of 'true' bit read

AND result of the 'complement' bit read

Bit written by master, path taken

FIRST

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

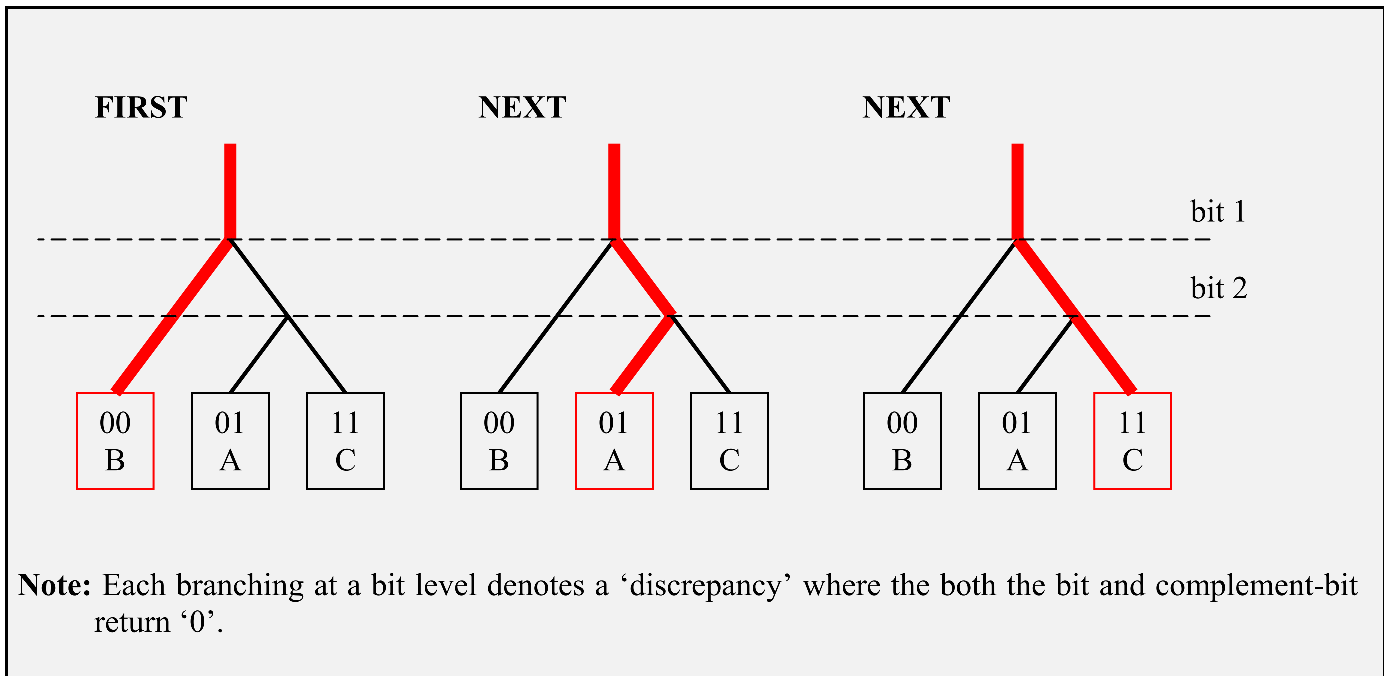
NEXT

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

検索例 ツリーグラフ 図 3b



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

検索例 疑似コード 図 3c

(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, F0 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** > **LastDescrepancy** 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
- **LastDescrpancy** = **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, F0 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** = **LastDescrepancy** 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** > **LastDescrepancy** 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
- **LastDescrpancy** = **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, F0 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** < **LastDescrepancy** 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** = **LastDescrepancy** 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
- **LastDescrpancy** = **last_zero** which is 0 so **LastDeviceFlag** = TRUE

NEXT

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

詳細検索バリエーション

同じ状態情報、すなわち LastDiscrepancy、LastFamilyDiscrepancy、LastDeviceFlag、および ROM_NO による 3 つの詳細検索バリエーションがあります。これらのバリエーションでは、対象または非対象の特定ファミリタイプと、デバイス存在の検証に対処します(表 4 参照)。

VERIFY

「VERIFY」オペレーションでは、認識済みの ROM 番号を備えたデバイスが 1-Wire に現在接続されているか検証します。当オペレーションを実行すると、ROM 番号を提供し、その番号に基づいて対象を絞った検索を実行し、その番号が存在しているかを検証します。まず、ROM_NO レジスタを認識済みの ROM 番号に設定します。次に、LastDiscrepancy を 64 (16 進法の 40)、LastDeviceFlag を 0 に設定します。検索オペレーションを実行し、*ROM_NO* の結果を読み取ります。検索が成功し、ROM_NO が検索された ROM 番号のままである場合は、そのデバイスは現在 1-Wire 上に存在します。

TARGET SETUP

「TARGET SETUP」オペレーションとは、特定のファミリタイプを最初に検出するために検索状態を事前設定する方法です。各 1-Wire デバイスは、ROM 番号内に組み込まれた 1 バイトの*ファミリコード*を備えています (図 1 参照)。このファミリコードにより、デバイスが実行可能なオペレーションを 1-Wire マスタが識別することができます。1-Wire 上に複数のデバイスがある場合は、関心のあるデバイスファミリのみに検索の的を絞るのが一般的です。特定ファミリを対象にするには、ROM_NO 登録の先頭のバイトに要求するファミリコードのバイトを設定し、ROM_NO レジスタの残りにゼロを入れます。次に LastDiscrepancy を 64 (16 進法の 40)に、LastDeviceFlag および LastFamilyDiscrepancy の両方を 0 に設定します。検索アルゴリズムが以後実行されると、要求するファミリタイプの最初のデバイスが検出され、ROM_NO 登録に配置されます。要求するファミリのデバイスが 1-Wire 上に現在存在しない場合は、別のタイプが検出され、ROM_NO のファミリコードを検索後に検証する必要があります。

FAMILY SKIP SETUP

「FAMILY SKIP SETUP」オペレーションでは、前回の検索で検出されたファミリコードを持つデバイスをすべてスキップするように、検索状態を設定します。当オペレーションは、検索後にのみ実行することができます。当オペレーションを実行すると、LastFamilyDiscrepancy を LastDiscrepancy にコピーし、LastDeviceFlag を除去します。次に、次回の検索で現在のファミリコードの後に来るデバイスを検出します。現在のファミリコードグループが検索における最後のグループの場合は、検索は LastDeviceFlag セットとともに元に戻ります。

検索バリエーション状態セットアップ 表 4

	LastDiscrepancy	**LastFamily-Discrepancy**	**LastDeviceFlag**	**ROM_NO**
FIRST	0	0	0	result
NEXT	leave unchanged	leave unchanged	leave unchanged	result
VERIFY	64	don't care	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

終わりに

こうした提供される検索アルゴリズムにより、1-Wire デバイスの任意のグループからそれぞれ一意の ROM 番号を検出することができます。このアルゴリズムは、マルチドロップ 1-Wire アプリケーションに不可欠です。ROM 番号を入手して、各 1-Wire デバイスをオペレーション用にそれぞれ選択することができます。また、本ドキュメントでは、特定の 1-Wire デバイスタイプの検出やスキップを行う検索バリエーションについても説明しました。検索の「C」コード実装例と全検索バリエーションについては、*付録*を参照してください。

付録

図 4 では、各検索バリエーションの機能とともに、検索アルゴリズムの「C」コード実装を紹介しています。「FAMILY SKIP SETUP」および「TARGET SETUP」機能は実際に検索をせず、次の「NEXT」が希望のタイプのスキップや検出を行うように、検索登録を単に設定するだけです。低レベルの 1-Wire 機能は、TMEX API を呼び出すことにより実装されます。こうした呼び出しはテスト用で、プラットフォーム専用の呼び出しと取り替えることができます。TMEX API および他の 1-Wire API については、*アプリケーションノート 155* を参照してください。(http://pdfserv.maxim-ic.com/jp/an/AN155J.pdf)

以下のコード例のTMEX API テスト実装を次のリンクからダウンロードすることができます。
ftp://ftp.dalsemi.com/pub/auto_id/public/an187.zip

検索「C」コード例 図 4 (次ページに続く)

```
// 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
}
```

改訂履歴

01/30/02 Version 1.0 – Initial release
05/16/03 Version 1.1 – Corrections: Search ROM commands corrected to F0 hex.