maxim integrated™

 Nov 17, 2005

キーワード: USB, MAX3420E, MAX3420, C, エニュメレーションコード

アプリケーションノート3690

MAX3420E の USB エニュメレーションコード(およびその他)

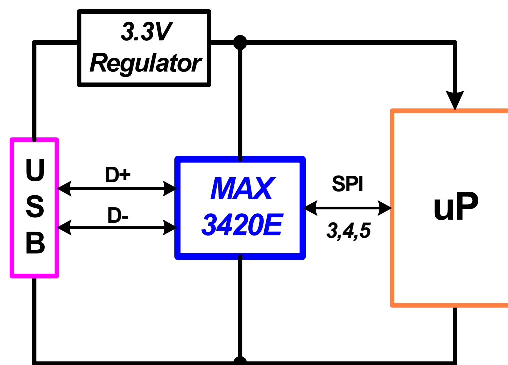
要約:MAX3420E USB コントローラを使えば、設計者はいずれのシステムにも USB 周辺機器の機能を追加することができます。MAX3420E は、IC 内にマイクロプロセッサを搭載する代わりに、内蔵するレジスタセットへの SPI インタフェースを装備しているため、MAX3420E の一連の C ルーチンを記述することで、さまざまなプロセッサに対応することができるようになります。このアプリケーションノートでは、C コードを提示し、USB の基本操作を行うためのあらゆる関数について説明します。コードを理解しやすくするために、USB の基本転送について説明しています。

ダウンロード:コードの全リスト(バグフィックス 1 と.h ファイルを含む) (ZIP、11.6kb)

目次

目次 2
1. はじめに 3
  1.1. コードの内容 3
  1.2. 移植性について 5
  1.3. wreg() and rreg() 6
2. 初期化 7
3. エニュメレーションのトラフィック 8
  3.1. ホスト要求の構造(CONTROL転送) 9
  3.2. USBのフロー制御とNAK 10
  3.3. STATUSハンドシェイクとACKSTATビット 10
  3.4. データフロー制御 11
4. プログラムのループ 12
  4.1. USBレジュームのチェック 13
  4.2. MAX3420EのIRQビットの処理 16
5. エニュメレーションの核心部 - SETUPデータの復号化 17
  記述子について 18
  5.1. set_featureとclear_feature 21
  5.2. get_status 22
  デバイスのステータスビット 22
  インタフェースのステータスビット 22
  エンドポイントのステータスビット 22
  5.3. set_interfaceとget_interface 23
  5.4. set_configurationとget_configuration 23
  5.5. set_address 24
  5.6. デバッグの補助 24

1. はじめに



MAX3420E USBコントローラを使えば、設計者はいずれのシステムにも USB 周辺機器の機能を追加することができます。MAX3420E は、IC 内にマイクロプロセッサを搭載する代わりに、内蔵するレジスタセットへの SPI インタフェースを装備しています。このため、MAX3420E の一連の C ルーチンを記述することで、さまざまなプロセッサに対応することができるようになります。コードの移植性は、入出力ピンにまで拡張されています。たとえば、押しボタンを MAX3420E の GPI (汎用入力)ピンの 1 つに接続し、IOPINS レジスタを読み出すことによってどのボタンが押されたかを判定する C コードを記述すれば、プロセッサがどのように専用 IO ピンを実装しているかどうかにかかわらず、コードを変更せずにそのままプロセッサ上で動作します。

USB 周辺機器でのプログラミングのオーバヘッドの大部分が、エニュメレーションのプロセスに関係しています。具体的には、ホストが、接続された周辺機器を検出し、この周辺機器に応答指令信号を送ってその能力と要件について確認します。すべてが問題なく完了すれば、ホストはこの周辺機器をオンラインに設定します。このアプリケーションノートでは、MAX3420E のエニュメレーションを遂行する一連の C 関数を紹介します。この C 関数は、MAX3420E を使用するいずれのシステムにも適用することができます。

デバイスは、エニュメレーションの間にホストに供給するデータによって、そのデバイスの*「特性」*を定義すると同時に、アプリケーションコードからデバイスの*「機能」*を定義します。このアプリケーションノートのコードは、エニュメレーションの範囲にとどまらず、標準の USB HID (ヒューマンインタフェースデバイス)クラスに準拠した簡単なキーパッドエミュレータのデバイスを組み込むためのアプリケーションコードも追加しています。これは実用的なアプリケーションであり、PC を用いてエニュメレーションコードのポートをテストすることができます。コードは標準の USB HID クラスを使用しているため、ホスト PC にカスタムのドライバをインストールしなくてもテストすることが可能です。

1.1. コードの内容

このアプリケーションノートのコードは、標準のHIDクラスに準拠した、キーボードのように機能する「1 ボタン」デバイスを実行します。デバイスをUSBポートに接続して、MAX3420Eに取り付けられたボタンを押すと、キーボードのデータを受け入れるウィンドウが開いて、*図 1*のメッセージが表示されます。このコードは、USBバスのリセットとサスペンド-レジュームも処理します。

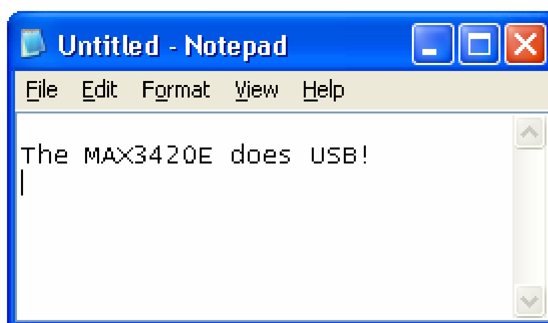


図 1. アプリケーションコードはこのテキスト文字列を表示します。

USBファームウェアは、USBプロトコルコードとアプリケーションコードの 2 つの部分から構成されています。アプリケーションコードは、図 1 の文字列を表示します。これを遂行するコードを*図 2*に示します。

```
void do IN3(void)
{
if (inhibit send==0x01)
   {
   wreg(rEP3INFIFO,0);         // send the "keys up" code
   wreg(rEP3INFIFO,0);
   wreg(rEP3INFIFO,0);
   }
else
  if (send3zeros==0x01)       // precede every keycode with the "no keys" code
   {
   wreg(rEP3INFIFO,0);         // send the "keys up" code
   wreg(rEP3INFIFO,0);
   wreg(rEP3INFIFO,0);
        send3zeros=0;          // next time through this function send the keycode
   }
  else
   {
   send3zeros=1;
   wreg(rEP3INFIFO,Message[msgidx++]);     // load the next keystroke (3 bytes)
   wreg(rEP3INFIFO,Message[msgidx++]);
   wreg(rEP3INFIFO,Message[msgidx++]);
   if(msgidx >= msglen)                    // check for message wrap
   {
      msgidx=0;
         L0 OFF
         inhibit send=1;      // send the string once per pushbutton press
      }
   }
wreg(rEP3INBC,3);             // arm it
}
```

図 2. キーボードデバイスを組み込むためのアプリケーションコード

USB ホストがエンドポイント 3 を介して MAX3420E からデータパケットを受け取ると、MAX3420E は割込みビット IN3BAVIRQ をアサートします。これによって、エンドポイント 3 のバッファ(FIFO)が新しいデータの読込みに利用可能であることを SPI マスタに通知します。図 2 の **do_IN3()**関数は、フラグ **inhibit_send** を調べて、3 つのゼロからなるキーアップコードを送信するか、あるいは次のキーストロークに対応する 3 データバイトを送信するかを決定します。メインプログラムのループは、送信ボタンをチェックして **inhibit_send** フラグを更新します。次に関数は、**send3zeros** フラグをチェックして、各キーストロークの間に「キーアップ」コードを送信します。

以上がアプリケーションコードのすべてです。それでは、このノートの残り数ページのコードは何を行っているのでしょうか。残りのコードは、各 USB 周辺機器が必要とするオーバヘッドを実行しています。このようにして、USB 周辺機器コードのフレームワークとしてコピー/ペーストすることができます。

コードが実行する USB のオーバヘッドの動作を以下に示します。

1. USB バスリセットを認識して、これに応答します。
2. USB バスサスペンドのイベントを認識して、これに応答します。
3. ホストのレジューム信号またはユーザが起動した RWU (リモートウェイクアップ)信号によって、デバイスのウェイクアップを実行します。
4. ホストの CONTROL 転送を認識して、適切な応答を生成します。

項目 4 に、オーバヘッドコードの大部分が含まれています。エニュメレーションの間、ホストは複数の要求をデバイスに送信し、デバイスの動作を定義した記述子(テーブルデータ)を要求します。記述子の要求を復号してこれに応答することは、あらゆる周辺機器でまったく同様に行われるため、このタスク用にここに記載したコードは、そのままアプリケーションで使用することができます。必要なことは、デバイスの特性を指定するための数項目のテーブルデータ(記述子)を変更することだけです。

1.2.　移植性について

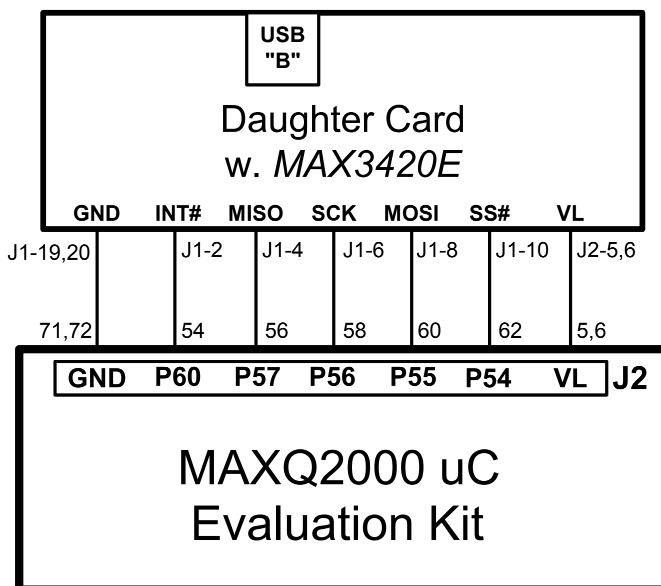


図 3. このコードは、MAXQ2000マイクロコントローラ評価キットで動作するように記述されています。

*図 3*は、このサンプルコードを実行するために使用するハードウェアを示しています。MAX3420Eは、いずれのプロセッサにも接続可能であるため、このアプリケーションノートでは、可能な限り移植性の高いコードを記述しています。Cで記述することで、この移植性をほとんど達成することが可能ですが、使用するプロセッサとコンパイラに依存した部分のコードについては、実現が困難な部分があります。システムに依存する部分のコードには、MAX3420Eとの交信をやりとりするSPIインタフェースを実行しています。このアプリケーションノートのコードは、プロセッサ間の相違をできるだけ少なくするため、以下を考慮して記述しています。

1. コードは、特定のマイクロプロセッサの割込みシステムを想定していません。各プロセッサは、いろいろなレジスタとベクトル構造を使用して割込みを実現しています。これに伴ってコンパイラは、さまざまな構文を使用して割込みベクトルを処理します。したがって、このコードでは、MAX3420E の INT 出力ピンをじかにポーリングする連続ループを実行しています。一般的なシステムでは、INT ピンをマイクロプロセッサの割込みピンに接続し、プロセッサとコンパイラが割込みの処理に使用する手法に合わせてコードを修正しています。

   注:新たに MAX3420E システムを開始するときの効果的なデバッグの方法は、最初に IRQ ビットのポーリング手法を使用して作業を進めるという方法です。最後のステップとして、INT ピンとマイクロプロセッサの割込み機構を使用してください。

2. ビット変数は、ビットマスクとして処理されます。コンパイラは、さまざまな方法でビット変数を処理します。最も一般的でC移植性の高い方法は、1 ビットだけを 1 にセットしたビットマスク(bmBitName)を定義し、これらのビットマスクとC標準の論理演算子を使用するというものです。たとえば、*図 4*のコードは、EPIRQ (エンドポイントの割込み要求)レジスタのビットSUDAVIRQ (SETUPデータ利用可能の割込み要求)のテストとクリアを行う方法を示しています。

```
#define rEPIRQ   11
#define bmSUDAVIRQ  0x20
unsigned char test;

test = rreg(rEPIRQ);     // read the register
if (test &  bmSUDAVIRQ) // test the IRQ bit
       {
       wreg(rEPIRQ,bmSUDAVIRQ)   // clear the IRQ
                                 // (do something)
       }
```

図 4. このコードは、ビットマスク定数 bmSUDAVIRQ を使用して、MAX3420E のレジスタビットのテストとクリアを行います。

注:IRQ をクリアするステートメントは、MAX3420E の機能を実際に示すものです。すなわち、IRQ ビットをクリアするために、ビットマスクを用いてレジスタに書き込んでいます。これは、1 によって IRQ のレジスタビットをクリアし、また 0 によって何も行わずにレジスタ内の残りの IRQ ビットをそのまま保持できるからです。

---

3. マイクロプロセッサとコンパイラに固有のコードは、リスト出力の最後に掲載されています。特定のマイクロプロセッサとコンパイラに合わせてコードをカスタマイズするには、このセクションを変更するだけです。

4. エニュメレーションのデータは、このデータ専用のファイルである EnumApp_enum_data.h にあります。お客様のアプリケーションでは、次の値を変更する必要があります。

   a. ベンダ ID。このアプリケーションノートのコードでは、マキシムのベンダ ID (VID)である 0x0B6A を使用しています。お客様のコードでは、お客様専用の VID に置き換えてください。
   b. 製品 ID。0x5346 をお客様の製品 ID に置き換えてください。
   c. シルアルナンバー(デバイス ID)
   d. お客様のアプリケーションに一意な文字列インデックスと文字列
   e. デバイスに適合する構成とインタフェースの値(たとえば、電源要件、エンドポイントの数とタイプ、クラス固有の記述子)。

1.3. wreg() and rreg()

SPIマスタは、SPIインタフェースを介して 21 のレジスタの書込みと読出しを行うことによって、MAX3420Eを制御しています。このアプリケーションノートのコードでは、関数**wreg()** (レジスタの書込み)と**rreg()** (レジスタの読出し)を使用して、MAX3420Eのレジスタにアクセスします。このコードではMAXQ2000 マイクロコントローラを使用しているため、これらの関数は、MAXQ2000 固有のレジスタを操作し、MAXQ2000 のハードウェアSPIインタフェースを介してデータを転送します。より一般的な手法では、マイクロプロセッサの汎用IO (GPIO)ピンを「ビットバンギング」して、SPIインタフェースを実現しています。このサンプルコードを別のプロセッサに移植するには、SPIマスタを実現するためのプロセッサの手法に対応した**rreg()**と**wreg()**の関数を記述する必要があります。このアプリケーションノートに記載したコードのリスト出力には、**rreg()**と**wreg()**の関数をテストしてそのバージョンを検証できるようにするためのルーチンが含まれています(*図 22*)。

サンプルコードには、MAX3420E SPI のバーストモードの使用方法を実際に示した **readbytes()**というマルチバイト関数も含まれています。バーストモードでは、SPI マスタは SS# (スレーブセレクト)ピンをアサートしてから、SPI コマンドバイトを送信して一連のバイトの読出しまたは書込みを行い、最後に SS#ピンをデアサートします。コードには、参考のため **writebytes()**関数も含まれていますが、サンプルコードでは使用しません。

2. 初期化

```
void initialize MAX(void)
{
ep3stall=0;       // EP3 inintially un-halted (no stall) (CH9 testing)
msgidx = 0;       // start of KB Message[]
msglen = sizeof(Message);       // so we can check for the end of the message
inhibit send = 0x01;    // 0 means send, 1 means inhibit sending
send3zeros=1;
msec timer=0;
blinktimer=0;
// software flags
configval=0;                    // at pwr on OR bus reset we're unconfigured
Suspended=0;
RWU enabled=0;                  // Set by host Set Feature(enable RWU) request
//
SPI Init();                     // set up MAXQ2000 to use its SPI port as a master
//
// Always set the FDUPSPI bit in the PINCTL register FIRST if you are using the SPI port in
// full duplex mode. This configures the port properly for subsequent SPI accesses.
//
wreg(rPINCTL,(bmFDUPSPI+bmINTLEVEL+gpxSOF)); // MAX3420: SPI=full-duplex, INT=neg level, GPX=SOF
Reset MAX();
wreg(rGPIO,0x00);                   // lites off (Active HIGH)
// This is a self-powered design, so the host could turn off Vbus while we are powered.
// Therefore set the VBGATE bit to have the MAX3420E automatically disconnect the D+
// pullup resistor in the absense of Vbus. Note: the VBCOMP pin must be connected to Vbus
// or pulled high for this code to work--a low on VBCOMP will prevent USB connection.
wreg(rUSBCTL,(bmCONNECT+bmVBGATE)); // VBGATE=1 disconnects D+ pullup if host turns off VBUS
ENABLE IRQS
wreg(rCPUCTL,bmIE);                 // Enable the INT pin
}
```

図 5. MAX3420E とプログラム変数の初期化

*図 5*は、MAX3420Eの初期化コードを示しています。最初のセクションで、プログラムが使用するいくつかの変数を初期化しています。変数**msgidx**は、アプリケーションが表示するテキスト文字列へのオフセットです。**SPI_Init()**関数は、MAX3420Eと交信するSPIポートとしてマイクロプロセッサのIOピンを設定しています。このサンプルコードでは、MAXQ2000 マイクロコントローラを使用し、ハードウェアSPIユニットを搭載しています。したがって、SPIポートの初期化は、MAXQ2000 のさまざまな設定レジスタに書き込むことによって行います。ハードウェアSPIを搭載していないマイクロプロセッサの場合、このルーチンは単に、GPIOピン(ビットバンギングしたSPIインタフェースを実現するために使用)の方向と初期状態を設定するだけです。

このアプリケーションでは、SPIポート用にMAX3430Eのフルデュープレクスモードを使用しており、上の図 3 で示したように独立したMOSIピンとMISOピンを利用しています。MAX3420Eの電源投入時のデフォルトはハーフデュープレクスであるため、SPIポートを読み出す前に、FDUPSPIビットを 1 に設定することによって、フルデュープレクス動作を設定する必要があります。PINCTLレジスタにもINTLEVELというビットが含まれています。ファームウェアは、このビットを 1 に設定することで、アクティブローのレベルセンスのピンとしてMAX3420EのINTピンを設定します。フルデュープレクスモードでは、INTピンはオープンドレインであるため、このピンをシステムインタフェースの電圧V_Lにプルアップする必要があることに留意してください。MAX3420Eの割込みシステムの詳細については、マキシムのウェブサイトにあるアプリケーションノート「*MAX3420Eの割込みシステム*」(japan.maxim-ic.com/AN3661)を参照してください。

次に、関数は、CHIPRESビットをセットした後にクリアすることによってMAX3420Eをリセットし、MAX3420Eの汎用出力(GPO)ピンに取り付けられているLEDを消灯します。最後に関数は、USBCTLレジスタのCONNECTビットとVBGATEビットをセットして、USB接続を確立します。CONNECTビットは、V_{CC}とD+との間に 1500Ωの内部プルアップ抵抗を接続します。一方、VBGATEビットは、VBCOMPピン上にV_{BUS}が存在しないときに必ずプルアップ

抵抗がD+から切断されるようにします。この機能は、(今回の場合のように)自己給電設計において重要となります。周辺機器は、V_{BUS}が存在しない場合に、D+に電力を供給してはならないからです。

マクロ ENABLE_IRQS は、次のように定義されます。

```
#define ENABLE_IRQS wreg(rEPIEN,(bmSUDAVIE+bmIN3BAVIE));
wreg(rUSBIEN,(bmURESIE+bmURESDNIE));
// Note: the SUSPEND IRQ will be enabled later, when the device is configured.
// This prevents repeated SUSPEND IRQ's
```

マクロで割込みをイネーブルにすることによって、動作を 1 箇所で定義し、さらに 2 つの関数(初期化関数およびバスリセットの終了を検出する割込み関数)内で呼び出すことが可能になります。これらの割込みイネーブルビットはバスリセットによってクリアされるため、ホストがバスリセットを発行するたびに再度初期化する必要があります。

注:USB バスリセットに関連する割込みイネーブルビット(URESIE と URESDNIE)は、USB バスリセットの影響を受けることはありませんが、マクロ内にこれらのビットを含めると、コード内の 1 箇所に割込みイネーブルを集めることができて便利です。

---

3. エニュメレーションのトラフィック

Transfer	F/S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time
0	S	GET	0x00	0x0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE descriptor	611.333 μs
Packet 83	Dir -->	Reset 16.741 ms							46.246 ms
1	S	SET	0x00	0x0	SET_ADDRESS	New address 3			62.498 ms
2	S	GET	0x03	0x0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE descriptor	395.317 μs
3	S	GET	0x03	0x0	GET_DESCRIPTOR	CONFIGURATION type	0x0000	CONFIGURATION descriptor	498.950 μs
4	S	GET	0x03	0x0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	0x0409	437.867 μs
5	S	GET	0x03	0x0	GET_DESCRIPTOR	STRING type, Index 3	Language ID 0x0409	string: S/N 1234LTH	438.133 μs
6	S	GET	0x03	0x0	GET_DESCRIPTOR	CONFIGURATION type	0x0000	4 descriptors	1.554 ms
7	S	GET	0x03	0x0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	0x0409	375.500 μs
8	S	GET	0x03	0x0	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	string: RT86 Lives!	661.183 μs
9	S	GET	0x03	0x0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	0x0409	391.333 μs
10	S	GET	0x03	0x0	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	string: RT86 Lives!	835.582 ms
11	S	GET	0x03	0x0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE descriptor	456.717 μs
12	S	GET	0x03	0x0	GET_DESCRIPTOR	CONFIGURATION type	0x0000	CONFIGURATION descriptor	506.350 μs
13	S	GET	0x03	0x0	GET_DESCRIPTOR	CONFIGURATION type	0x0000	4 descriptors	493.183 μs
14	S	SET	0x03	0x0	SET_CONFIGURATION	New configuration 1			17.660 ms

図 6. ホスト要求と MAX3430E 応答のバストレース

*図 6*は、LeCroy (以前のCATC) USBバスアナライザが取り込んだUSBバスのトラフィックを示しています。このトレースは、このアプリケーションノートのCコードを実行する周辺機器をPCがエニュメレートする様子を示しています。コードを詳しく調べる前に、図 6 を十分に考察すれば、コードの内容を容易に理解することができるようになります。

1. ホストは、デフォルトの CONTROL エンドポイントである EP0 (「ENDP」ボックスに表示)を使用して、デバイスに要求を送信します。最初に、ホストはアドレス 0 (ADDR ボックスに表示)に要求を送信して、まだ一意のアドレスが割り当てられていないデバイスと通信します。
2. ホスト要求は、任意の順序で任意の時点に行われる可能性があります。したがって、ファームウェアは常に、SUDAVIRQ (SETUP データ利用可能の割込み要求)に注意する必要があります。
3. ホストは、Get_Descriptor-Device 要求(図 6 の転送 0)を送信することで開始します。ホストはこの要求を送信して、デバイスの EP0 バッファの maxPacketSize を判断します(MAX3420E では 64 バイト)。次にホストは、バスリセット(パケット 83)を発行することでデバイスをリセットします。これによって、MAX3420E のレジスタビット USBRESIRQ (USB バスリセットの IRQ)と URESDNIRQ (USB バスリセット処理の IRQ)がアクティブになります。
4. 転送 1 では、ホストは Set_Address 要求を使用して、周辺機器に一意のアドレスを割り当てます。割り当てられるアドレスは、現在ホストに接続されている他の USB デバイスの数によって決まります。今回のケースでは、USB 周辺機器に割り当てられるアドレスは 3 です。MAX3420E は単独でこの要求を処理して、割り当てられたアドレスを関数のアドレスレジスタ FNADDR (R19)にロードします。これ以降、MAX3420E は、アドレス 3 宛ての要求にのみ応答します。このアドレスは、ホストがバスリセットを行うか、デバイスが切断されるまで有効な状態を維持します。転送 1 の後、バストレースの周辺機器のアドレスフィールド(ADDR)は、0 から 3 に変わった事に注目してください。
5. 転送 2～13 まで、ホストはさまざまな記述子を要求します。デバイスのファームウェアは、8 つのセットアップバイトから送信する記述子を判断し、この情報を利用して複数の文字配列(記述子の配列を表す)の 1 つにアクセスする必要があります。さらにデバイスは、これらの文字をエンドポイント FIFO にロードし、元のホストに返送する必要があります。

*図 7*は、図 6 の転送 0 を展開したものであり、パケットレベルで行われている内容を示しています。

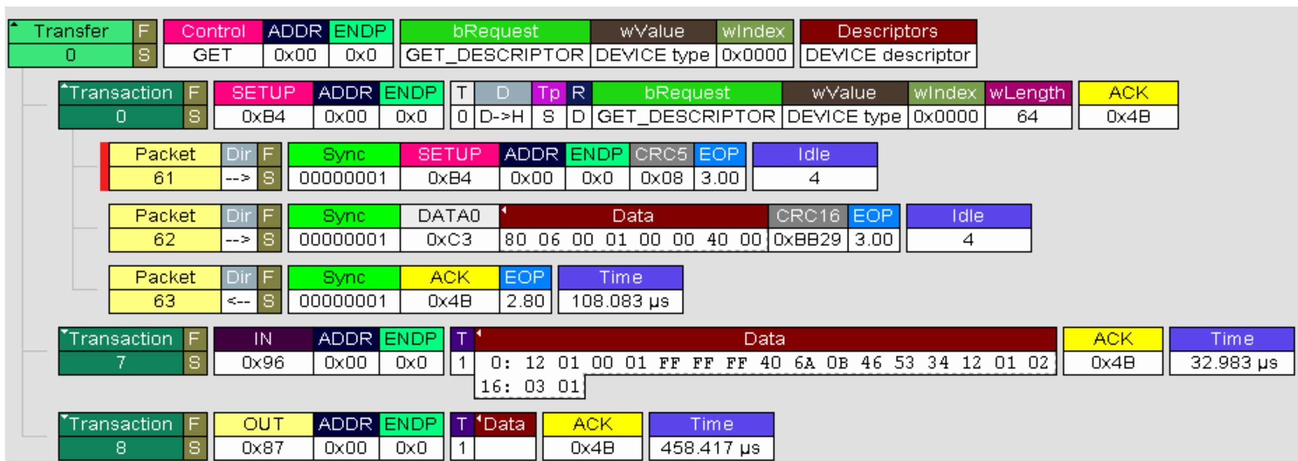


図 7. パケットレベルにまで展開した Get_Descriptor-Device 要求

3.1. ホスト要求の構造(CONTROL 転送)

図 7 の CONTROL 転送は、3 つのステージ(すなわちトランザクション)で構成されています。最初のステージは SETUP トランザクションです。このステージでは、ホストはデバイスのアドレスとエンドポイントを含んだ SETUP パケット 61 を送信した後、8 バイトのデータパケット 62 を送信して要求内容を周辺機器に通知します。トランザクションは、デバイスが ACK (肯定応答)パケット 63 を返送すると終了します(この ACK によって、エラーなしで転送が行われたことをホストに通知します)。コードを見ればわかるように、**do_SETUP()**関数は、SUDFIFO という

MAX3420E の FIFO から 8 つのセットアップバイトを取り出します。これは、レジスタ R4 を 8 回読み出すことでアクセスしています。次にコードは、この 8 バイトを検査して、実行すべき内容を決定します。

データパケット 62 の下位 2 バイトは、ホストがデバイスに返送してもらいたいバイトの数を表すワード値を示しています。USB はリトルエンディアンであるため、ホストは 16 進数の 0x0040 すなわち 64 バイトを要求しています。ところが、この転送のデータステージ(トランザクション 7)を見ると、MAX3420E は 18 バイトを返送しています。ホストは 64 バイトを要求しましたが、コードは 18 バイトしか返送しませんでした。どうしてでしょうか?

上記のトランザクションは、USB ではまったく通常のことで、単純な規則によって規定されています。この規則とは、(a)ホストが要求するバイト数と、(b)実際にあるバイト数のうち、必ず*「小さい」*方のバイト数を返送するというものです。デバイス記述子には 18 バイトが記載されており、ホストは 64 バイトを要求しています。この結果、MAX3420E のファームウェアは、規則どおり正しく 18 バイトを返送しています。

ホスト要求の中にはデータステージがないものもあります。たとえば、Set_Configuration 要求(図 6 の転送 14)には、8 つのセットアップバイトに設定値が含まれているため、データステージは必要ありません。

すべての CONTROL 転送は、STATUS ステージ(図 7 のトランザクション 8)で終了します。STATUS ステージの間、ホストは、周辺機器が何らかのフィードバックを提供することができる機会を設けるためだけに、空の OUT パケット(データなし)を送信します。周辺機器が要求の処理中でビジー状態の場合、NAK (否定応答)で応答します。NAK は、STATUS ステージを後で再試行し、ACK (肯定)応答を得るまで続行することをホストに通知するものです。

3.2. USB のフロー制御と NAK

賢明なことに、USB 設計者は、USB 周辺機器の処理能力に大きなばらつきがあることを考慮に入れていました。USB ホストは、100MHz の 32 ビット RISC で駆動される周辺機器と交信することもできれば、安価なマウスチップと交信することも可能です。USB が周辺機器の処理能力に依存しないようにするため、プロトコルは、デバイスが要求の処理中でビジー状態のときは必ず NAK (否定応答)ハンドシェイクを返送することができるようにしています。NAK を返送するとき、デバイスはホストに対して「現在ビジー状態です。後で再度問い合わせてください」を通知します。

トランザクション 8 (図 7)では、デバイスは、ACK ハンドシェイクで OUT パケットに応答しています。デバイスは、ホストによって要求された動作の実行を完了していない場合、NAK 応答を返送して、後で OUT パケットを再送するようにホストに通知します。ホストは、ACK を受信するまで OUT パケットを送信し続けます。ホストが ACK を受信した時点で、ステータスステージは肯定応答され、ホストは、転送が正常に完了したものと見なすことができます。

3.3. STATUS ハンドシェイクと ACKSTAT ビット

MAX3420Eは、このステータスハンドシェイクをACKSTAT (R9 のビット 6)というビットで処理します。ACKSTATビットは、「CONTROL 転送の STATUS ステージに肯定応答する」という意味です。ファームウェアは、Get_Descriptor-Deviceとして図 7 の要求を復号してデバイスの記述子を調べ、R0 に 18 回書き込むことによってEP0FIFOに 18 バイトをロードします。次にファームウェアは、EP0BC (エンドポイントゼロのバイトカウント)のレジスタR5 に数字の 18 を書き込み、トランザクション 7 で送信するバイト数をMAX3420Eに通知します。最後にファームウェアは、R9 のACKSTATビットをセットして、STATUSステージ(トランザクション 8)にACKハンドシェイクで応答するようMAX3420Eに指示します。ACKSTATビットはあらゆるCONTROL転送で使用されるため、MAX3420Eにはこのビットをセットするためのショートカットが用意されています。各SPI転送の最初のバイトはコマンドバイトです。ここで、MAX3420Eに接続されているコントローラは、*図 8*に示す書式でバイトを送信します。

b7	b6	b5	b4	b3	b2	b1	b0
Reg4	Reg3	Reg2	Reg1	Reg0	0	DIR 1=wr 0=rd	**ACKSTAT**

図 8. MAX3420E の SPI コマンドバイト

ビット 7～3 は MAX3420E のレジスタアドレスを設定します。また、ビット 1 は方向をセットし、ビット 0 は R9 の ACKSTAT ビットを更新します。したがって、ACKSTAT ビットをセットした状態で、SPI バスを介してレジスタアドレス 18 を送信することで、同時に 2 つの処理を実現することができます。つまり、EP0BC (バイトカウント)レジスタに書き込んでデータ転送を｢準備｣すると同時に、R9 の ACKSTAT ビットもセットすることができるということです。これが、MAX3420E に対してバイトの読出し(**rreg**、**rregAS**)と書込み(**wreg**、**wregAS**)を行う低レベル関数が 2 種類ある理由です。SPI コマンドバイトで ACKSTAT ビットがセットされていれば、AS 関数は非 AS 関数とまったく同じ内容を実行します。

3.4. データフロー制御

図 7 には、USB のもう 1 つのフロー制御機構が隠されています。観察力の鋭い読者は、トランザクション 0 とトランザクション 8 のギャップに気付かれることでしょう。図 7 では、見やすくするため、表示オプションを使用して NAK を非表示にしているからです(NAK を表示すると画面が乱雑になるため、USB 設計をデバッグするときには通常、NAK を非表示にすることをお勧めします)。

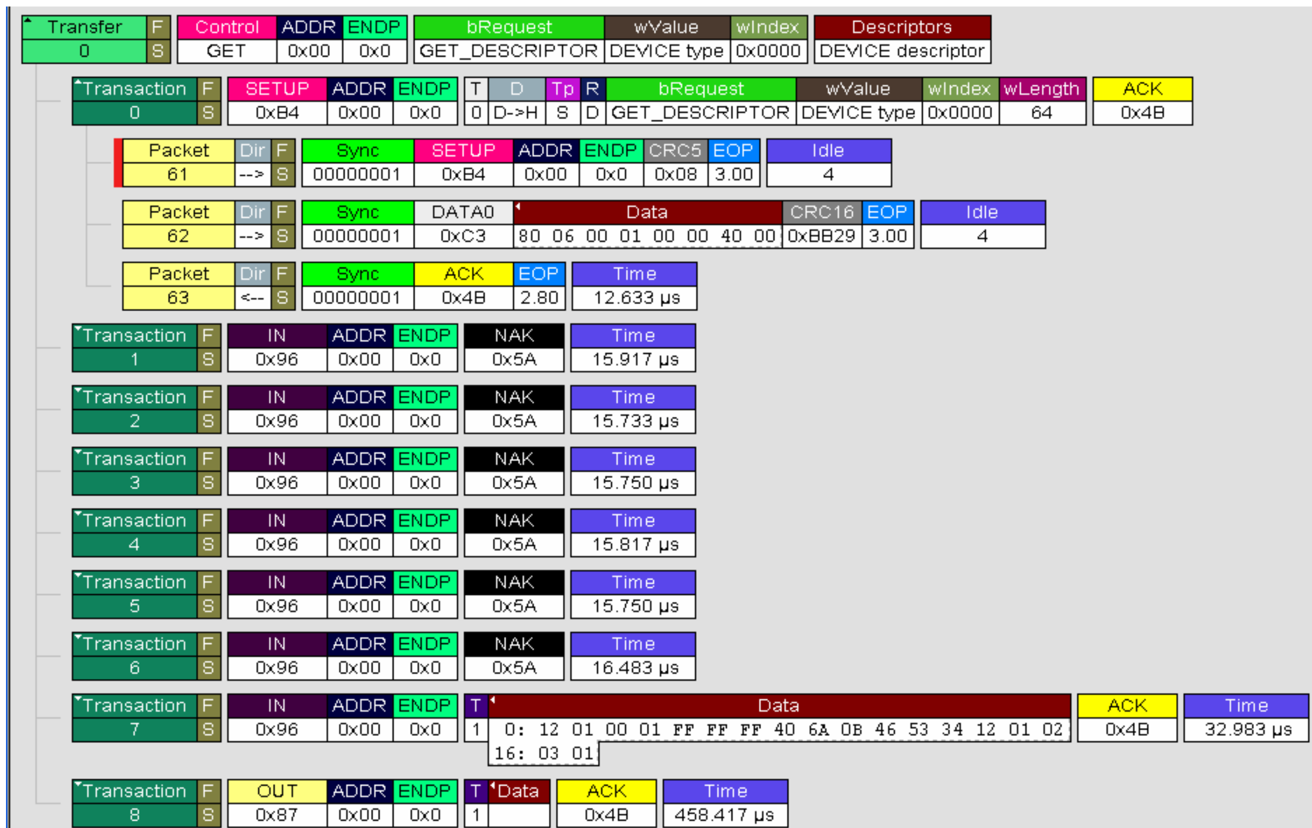


図 9. 図 7 で NAK を表示した場合

*図 9*は、ファームウェアが要求を理解し、適正な記述子を調べてEP0 FIFOに 18 バイトを挿入し、さらにバイトカウントを書き込むのにある程度の時間を要することを示しています。実際には、NAK6 回分の時間を要します。MAX3420Eは、ファームウェアがこれらのすべての処理を実行している間、NAKハンドシェイクを返送すべきことをどのようにして知るのでしょうか。答えは極めて簡単です。MAX3420Eは、ファームウェアがエンドポイントのバイトカウントにロードするまで、IN要求に対して自動的にそのエンドポイントへNAKハンドシェイクを返送していま

す。これによって、エンドポイントがデータを転送するための準備が完了します。MAX3420Eは、NAKハンドシェイクではなく、要求されたデータを送信して、次のIN要求(トランザクション 7)に応答します。

4. プログラムのループ

```
void main(void)
{
initialize_MAX();
while(1)    // endless loop
  {
  if(Suspended)
    check_for_resume();
  if (MAX_Int_Pending())
    service_irqs();
  msec_timer++;
  if(msec_timer==TWENTY_MSEC)
    {
    msec_timer=0;
    if((rreg(rGPIO) & 0x10) == 0) // Check the pushbutton on GPI-0
        {
        inhibit_send = 0x00;      // Tell the "do_IN3" function to send the text string
        L0_ON                     // Turn on the SEND light
        }
    blinktimer++;                 // blink the LOOP ACTIVE light every half second
    if(blinktimer==BLINKTIME)
        {
        blinktimer=0;
        L3_BLINK
        }
    }// msec_timer==ONE_MSEC
  } // while(1)
}// main
```

図 10. 永久に実行されるメインプログラムのループ

*図 10*の**main()**関数は、割込み駆動プログラムを模倣するように構成されているものの、特定プロセッサの割込みシステムには依存していません。MAX3420Eを初期化した後、main()関数は**while(1)**の無限ループに移行します。このループを通過するごとに、以下の 2 つの関数を呼び出します。

1. バスがサスペンド状態の場合、main()関数は **check_for_resume()**を呼び出して、ホストまたはユーザの起動したレジューム動作を検出します。
2. MAX3420E の割込みが保留状態の場合、main()関数は **service_irqs()**への呼出しを用いて割込みを処理します。
   このアプリケーションで可能な割込みを以下に示します。
   a. セットアップデータが到着した(SUDAVIRQ)。
   b. ホストが EP3-IN にデータの要求を送信して、キーボードデータを要求した。
   c. ホストが 3 ミリ秒間トラフィックを停止して、バスをサスペンド状態にした。
   d. ホストがバスリセットを起動した。
   e. ホストがバスリセット信号の送出を完了した。

MAX_Int_Pending()関数は、MAX3420E の INT ピンをポーリングし、ピンがローの場合に値 1 を返します。スマートな方法とはいえませんが、INT ピンをじかにポーリングすることによって、マイクロコントローラに依存しないコード

が可能になります。最終アプリケーションでは、コードが正しく動作することを検証した後、少数のステップでMAX3420E の INT ピンが接続されたマイクロコントローラの割込み機構を起動します。

メインプログラムのループは 20 ミリ秒ごとに実行され、以下を行います。

1. 「送信」押しボタンの状態を読み取り、ボタンが押されていれば、**inhibit_send** フラグをクリアします。
2. 0.5 秒ごとに、「ループアクティブ」ランプを点滅します。

コードは、**wreg** (書込みレジスタ)および **rreg** (読出しレジスタ)という 2 つの関数を用いて MAX3420E と通信します。これらのマイクロプロセッサに依存した関数は、全リスト出力の最後に掲載されています。レジスタ名の前には「r」が付加され(たとえば、rUSBIRQ)、ビットマスクの前には「bm」が付加されます(たとえば、bmBUSACTIRQ)。MAX3420E のレジスタとビットの等式は、インクルードされた MAX3040E.h ファイル内にあります。このファイルには、いくつかの便利なマクロも収録されています。コードは、たとえばマクロ L2_BLINK と L3_ON (すべて大文字で表示)を使用して、GPOUT ピンに取り付けられた LED を操作します。これらのマクロによって、回路が変わっても簡単にコードを修正することができるようになります。たとえば、アプリケーションがアクティブローの LED を使用している場合、必要なのはマクロの変更だけで、コードは変更せずにそのまま使用することが可能です。

0.5 秒の点滅タイマは、ソフトウェアループを使用して実現されているため、お客様の特定の実装内容に合わせてこのループを微調整することが必要となります。定数 TWENTY_MSEC と BLINKTIME は、プロセッサのクロック速度に対応して調整することができます。最終アプリケーションでは、マイクロコントローラのハードウェアタイマユニットを使用して、0.5 秒をカウントすることになると思われます。

4.1. USB レジュームのチェック

USB のサスペンド-レジュームについて
USBホストは、3 ミリ秒間、USB信号の通信を停止することによってデバイスをサスペンド状態にします。USB周辺機器は、このサスペンドの指示を検出して、V_{BUS}線から消費する電流がほとんどゼロの低電力状態に移行する必要があります。MAX3420Eは、SUSPIRQ (サスペンドIRQ)ビットをアクティブにして、ホストサスペンドの動作を示します。いったんサスペンド状態になると、デバイスは 2 つの方法でウェイクアップすることができます。1 番目は、ホストが単純にバスの信号送出をレジュームするというものです。これによって、MAX3420EのBUSACTIRQ (バスアクティブのIRQ)ビットが有効にします。2 番目は、デバイスがSIGRWUビットを使用してバス上のレジューム信号を駆動するというものです。ただし、周辺機器がリモートウェイクアップ信号を送出することが可能で、さらにホストがその周辺機器の信号送出機能を有効にしている場合に限ります。MAX3420Eは、RWUDNIRQ (リモートウェイクアップ信号処理のIRQ)という割込みビットをアサートして、RWU信号の送出が完了したことをSPIマスタに通知します。

注:バスパワーの周辺機器は、サスペンド状態において、MAX3420E をスリープ状態にすることでパワーダウン動作を実現しています。これは、ビット PWRDOWN = 1 をセットすることで行われます。これによって、MAX3420E のオンチップ発振器を停止します。このアプリケーションノートのコードでは、自己給電の周辺機器を実装しているため、この手順は不要です。

```
void check for resume(void)
{
  if(rreg(rUSBIRQ) & bmBUSACTIRQ) // THE HOST RESUMED BUS TRAFFIC
      {
      L2 OFF
      Suspended=0;                  // no longer suspended
      }
  else if(RWU enabled)                // Only if the host enabled RWU
      {
      if((rreg(rGPIO)&0x40)==0)       // See if the Remote Wakeup button was pressed
        {
        L2 OFF                        // turn off suspend light
        Suspended=0;                  // no longer suspended
        SETBIT(rUSBCTL,bmSIGRWU)      // signal RWU
        while ((rreg(rUSBIRQ)&bmRWUDNIRQ)==0) ; // spin until RWU signaling done
        CLRBIT(rUSBCTL,bmSIGRWU)      // remove the RESUME signal
        wreg(rUSBIRQ,bmRWUDNIRQ);     // clear the IRQ
        while((rreg(rGPIO)&0x40)==0) ;  // hang until RWU button released
        wreg(rUSBIRQ,bmBUSACTIRQ);    // wait for bus traffic -- clear the BUS Active IRQ
        while((rreg(rUSBIRQ) & bmBUSACTIRQ)==0) ; // & hang here until it's set again...
        }
      }
}
```

図 11. この関数は、2 つのソース(ホストと RWU 押しボタン)からの USB レジュームの有無をチェックします。

*図 11*の**check_for_resume()**関数は、サスペンド状態のデバイスをウェイクアップするための 2 つの方法についてテストします。

1. ホストが、バス上のトラフィックをレジュームする。
2. ユーザが、MAX3420E の GPIN2 ピンに接続された「リモートウェイクアップ」ボタンを押す。

最初の **if** ステートメントは、ホストのレジューム動作を処理します。MAX3420E がバスのトラフィックを検出すると、bmBUSACTIRQ (バスアクティブの IRQ)という IRQ ビットがアサートされます。このコードでは、単純にサスペンドランプ(L1)を消灯し、**サスペンド**フラグをクリアしているだけです。

else if ブロックは、リモートウェイクアップ(RWU)を処理します。リモートウェイクアップを有効にするには、以下のいくつかの条件を満たす必要があります。

1. *図 12*に示すように、デバイスがRWU信号の送出が可能であることを構成記述子の中で通知している。

```
unsigned char CD[]=  // CONFIGURATION Descriptor
   {0x09,   // bLength
   0x02,       // bDescriptorType = Config
   0x22,0x00,  // wTotalLength(L/H) = 34 bytes
   0x01,       // bNumInterfaces
   0x01,       // bConfigValue
   0x00,       // iConfiguration
   0xE0,       // bmAttributes. b7=1 b6=self-powered b5=RWU supported
   0x01,       // MaxPower is 2 ma
```

図 12. 周辺機器が、リモートウェイクアップ信号の送出が可能であることを構成記述子の bmAttributes バイトのビット 5 でホストに通知しています。

2. ホストが Set_Feature-RWU 要求を発行している。

Transfer	F	Interrupt	ADDR	ENDP	Bytes Transferred	Time
62	S	IN	0x03	0x3	3	32.000 ms

Transfer	F	Interrupt	ADDR	ENDP	Bytes Transferred	Time
63	S	IN	0x03	0x3	3	1.633 sec

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Time
64	S	SET	0x03	0x0	SET_FEATURE	DEVICE_REMOTE_WAKEUP	For Device	3.070 ms

Packet	Dir	Suspend
3869	-->	11.087 sec

Packet	Dir	Resume
3870	?	21.599 ms

Packet	Dir	Resume EOP	Time
3871	-->	1.383 µs	24.845 ms

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Time
65	S	SET	0x03	0x0	CLEAR_FEATURE	DEVICE_REMOTE_WAKEUP	For Device	286.450 µs

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	STALL	Time
66	S	SET	0x03	0x0	0x0A	0x0000	0x0000	0x08	28.873 ms

Transfer	F	Interrupt	ADDR	ENDP	Bytes Transferred	Time
67	S	IN	0x03	0x3	3	32.000 ms

Transfer	F	Interrupt	ADDR	ENDP	Bytes Transferred	Time
68	S	IN	0x03	0x3	3	32.000 ms

図 13. ホストのサスペンド-レジュームのバストレース

*図 13*は、ユーザがPCキーボードのスリープボタンを押した結果として、PCがサスペンド状態に移行したことを示すUSBバストレースです。転送 62 と 63 では、ホストが周期的にINTERRUPT-IN要求を送信し、キーボード(今回のケースではキーパッドエミュレータのアプリケーション)は要求された 3 バイトを返送しています。転送 64 では、ホストは、機能選択をDEVICE_REMOTE_WAKEUPに設定した状態で、最初にSET_FEATURE要求を周辺機器に送信することでUSBサスペンドの準備を行っています。デバイスが、デバイスのリモートウェイクアップ信号の送出が可能であることを事前に通知していない場合、ホストがこの要求を送信することはありません。

11.087 秒後に(パケット 3869)、PC ユーザは、キーボードのキーを押すかマウスを移動しています。あるいは、ユーザは USB 周辺機器にあるリモートウェイクアップのボタンを押しています。これによって、PC はウェイクアップし、続いてパケット 3870 で周辺機器をウェイクアップします。ホストは、転送 65 で、リモートウェイクアップ機能をクリアします。転送 66 は、Set_Idle と呼ばれる HID 要求です。デバイスがこの機能に対応していない場合(今回のコードではこの機能を必要としません)、正しい応答は STALL ハンドシェイクになります。最後に、転送 67 をはじめとして、ホストはエンドポイント 3 に IN パケットを送信し続けて、定期的にキーパッドのデータを要求します。

注:STALL ハンドシェイクは、実際には USB 周辺機器の動作を「stall (停止)」していません。USB の設計者は、このハンドシェイクにもっと適切な名前を選ぶことができたのではないかと思われます。

4.2. MAX3420E の IRQ ビットの処理

```
void service_irqs(void)
{
BYTE itest1,itest2;
itest1 = rreg(rEPIRQ);              // Check the EPIRQ bits
itest2 = rreg(rUSBIRQ);             // Check the USBIRQ bits
if(itest1 & bmSUDAVIRQ)
    {
     wreg(rEPIRQ,bmSUDAVIRQ);       // clear the SUDAV IRQ
     do_SETUP();
    }
if(itest1 & bmIN3BAVIRQ)
    do_IN3();
if((configval != 0) && (itest2&bmSUSPIRQ))  // HOST suspended bus for 3 msec
    {
    wreg(rUSBIRQ,(bmSUSPIRQ+bmBUSACTIRQ));  // clear the IRQ and bus activity IRQ
    L2_ON                          // turn on the SUSPEND light
    L3_OFF                         // turn off blinking light (in case it's on)
    Suspended=1;                   // signal the main loop
    }
if(rreg(rUSBIRQ)& bmURESIRQ)
    {
    L1_ON                          // turn the BUS RESET light on
    wreg(rUSBIRQ,bmURESIRQ);       // clear the IRQ
    }
if(rreg(rUSBIRQ) & bmURESDNIRQ)
    {
    L1_OFF                         // turn the BUS RESET light off
    wreg(rUSBIRQ,bmURESDNIRQ);     // clear the IRQ bit
    Suspended=0;                   // in case we were suspended
    ENABLE_IRQS                    // ...because a bus reset clears the IE bits
    }
}
```

図 14. この関数は、アプリケーションが必要とする IRQ ビットをチェックします。

*図 14*の**service_irqs()**関数は、サービスを必要とするUSBイベントの 4 つのMAX3420E IRQビットをチェックしています。これら 4 つのイベントと割込み要求ビットの名前を以下に示します。

1. **bmSUDAVIRQ** SETUP パケットが到着しました。ホストは SETUP パケットを送信してデバイスを制御しています。
2. **bmIN3BAVIRQ** ホストは、HID キーボードに対して別のデータパケットを要求しました。ホストは、この要求のためにエンドポイント 3-IN を使用します。BAV は、Buffer Available (バッファ利用可能)を意味しています。
3. **bmURESIRQ** ホストがバスリセットの信号送出を開始しました。
4. **bmUSBRESDNIRQ** ホストがバスリセットの信号送出を終了しました。

MAX3420E には全体で 14 の IRQ ビットがありますが、メインループでキーパッドエミュレータのアプリケーションを実行するのに必要な IRQ ビットは、上記の 4 つだけです。

関数は最初に、MAX3420E の 2 つの IRQ レジスタを読み出して、変数 **itest1** と **itest2** にその値を保存します。次に、**itest1** を調べて、2 つのエンドポイント割込み(SETUP データのためのエンドポイント 0 とキーボードデータの送

信要求のためのエンドポイント 3-IN)のいずれかの有無をチェックします。割込みが見つかれば、これらのテストはサービス関数 **do_SETUP()**または **do_IN3()**を呼び出します。**do_SETUP()**分岐は要求 IRQ ビットをクリアしますが、**do_IN3()**分岐はこのビットをクリアしないことに留意してください。IN データを送信するには、ビットをじかにクリアするのではなく、IN エンドポイントのバイトカウントレジスタに書き込むことによって IRQ ビットをクリアします。この機構によって、USB の IN 転送と IN FIFO データをロードが同時に起きるような競合する状況が回避されます。

残りのチェックでは、**itest2** を検査して USB の信号送出イベントの有無を確認しています。MAX3420E は、バスが 3 ミリ秒の間、動作していないことを検出すると、SUSPIRQ ビットをアサートします。SUSPIRQ ビットがセットされると、関数は **suspended** フラグをセットして、ホストがバスをサスペンド状態にしたことをメインループに通知します。またサスペンドのコードは点滅ランプを消灯し、サスペンドランプを点灯します。

注：サスペンドランプは、自己給電デバイスの場合には問題ありませんが、アプリケーションがバスパワー(V_{BUS}線から電力を得る)の場合には、サスペンド電流のバジェットが限られているため、LED電流を加えたくないだろうと思われます。

最後の 2 つのチェックは、USB のバスリセットを処理します。このコードセクションは、IRQ ビットをクリアし、またバスリセットのランプを点灯および消灯します。MAX3420E のコードには、必ず USB バスリセットのテストを含めるようにしてください。MAX3420E は、USB バスリセットの間に、割込みイネーブルレジスタのビットのほとんどをクリアするからです。したがって、コードは、常にバスリセットに注目する必要があります。リセットが完了すると(USBRESDN IRQ が信号を送出)、アプリケーションに使用している割込みを再イネーブルにする必要があります。

5. エニュメレーションの核心部 - SETUP データの復号化

図 15 は、SETUP転送要求を処理する関数呼出しです。メインループは、MAX3420EがSUDAVIRQビットをアサートすると必ずこの関数を呼び出します。

```
void do SETUP(void)
{
readbytes(rSUDFIFO,8,SUD);           // got a SETUP packet. Read 8 SETUP bytes
switch(SUD[bmRequestType]&0x60)      // Parse the SETUP packet. For request type, look only at b6&b5
    {
    case 0x00: std request();    break;
    case 0x20: class request();  break;  // just a stub in this program
    case 0x40: vendor request(); break;  // just a stub in this program
    default:STALL EP0                  // unrecognized request type
    }
}
void do SETUP(void)
```

図 15. エニュメレーションの最初のステップは要求のタイプを復号することです。

readbytes 関数は、以下の 3 つの引数をとります。1 番目は、データの読出し先である MAX3420E のレジスタです(今回のケースでは、セットアップデータ FIFO の「SUDFIFO」)。2 番目は、読み出すバイトの数です。3 番目は、バイトを格納するバイト配列へのポインタです。switch ステートメントは、最初の SETUP バイト(bmRequestType)を調べて、要求のタイプを判断します。エニュメレーションの要求の多くは、標準的な要求です。デバイスが標準の USB クラス(HID など)を実装していれば、このクラスに特有の要求が存在する場合もあります。ベンダ要求は、周辺機器のメーカが定義するカスタムの要求です。default ステートメントは、USB の復号関数を終了する方法を示しています。正当な case 値が見つからない場合、ACK や NAK の代わりに STALL ハンドシェイクを送信し、要求を認識できなかったことを示すことが適切な周辺機器の応答です。

このサンプルコードでは、関数 **class_request** と **vendor_request** は空の状態です。これらの関数は単に、関数のいずれかを実装する必要がある場合にそれらの挿入場所を示すために記されています。

*図 16*に示す関数は、標準のUSB要求を処理します。

```
void std request(void)
{
switch(SUD[bRequest])
   {
   case  SR GET DESCRIPTOR:   send descriptor();    break;
   case  SR SET FEATURE:      feature(1);           break;
   case  SR CLEAR FEATURE:       feature(0);           break;
   case  SR GET STATUS:          get status();         break;
   case  SR SET INTERFACE:       set interface();      break;
   case  SR GET INTERFACE:       get interface();      break;
   case  SR GET CONFIGURATION:   get configuration();  break;
   case  SR SET CONFIGURATION:set configuration();  break;
   case  SR SET ADDRESS:      rregAS(rREVISION);    break;
   default:  STALL EP0
   }
}
```

図 16. 2 番目のステップは、どの要求であるかを明らかにすることです。

std_request()関数は、SETUP データパケットの bRequest バイト内で有効な記述子タイプチェックします。switch ステートメントの「SR_」名は、USB の標準要求とデータ書式を定義した「USB 仕様」の第 9 章の命名に対応しています。USB 適合テストの大部分は、コードのこの部分をチェックして、第 9 章で定義された要求にデバイスが適切に応答することを確認するためのものです。

SET_FEATURE と CLEAR_FEATURE の要求は、1 つの **feature()**関数だけで処理します。この関数は、セット(1)またはクリア(0)のいずれの動作が必要であるかを示す引数をとります。SET_ADDRESS 要求は、REVISION レジスタへのダミー読出しを行って ACKSTAT ビットをセットする以外は何も行いません。MAX3420E のハードウェアが自動的に Set_Address 要求を処理するからです。

エニュメレーションコードの残りは、図 16 で呼び出されている 7 つの関数で構成されています。最初の関数 **send_descriptor()**は複雑ですが、残りの関数は極めて単純なものです。

記述子について

今回のアプリケーションでは、以下の USB 記述子タイプを使用しています。

- デバイス
- 構成
  - インタフェース
    - (HID)
    - エンドポイント
- 文字列
- (レポート)

インクルードしたファイル EnumApp_enum_data.h には、周辺機器に「特性」を与えるために使われるさまざまな記述子が含まれています。USB デバイスでは、複数の構成とインタフェースを持つことが可能ですが、今回のアプリケーションでは、それぞれ 1 つだけを使用しています。上述の括弧内の記述子は、HID クラスに独自なもので、HID 以外のデバイスでは省略されます。

注：構成記述子は、インタフェース記述子とエンドポイント記述子で構成されています。ホストは、インタフェース記述子やエンドポイント記述子を名指しで要求することはありません。ホストは、構成記述子の一部としてこれらの記述子を取得することができることを承知しています。

```
void send descriptor(void)
{
WORD reqlen,sendlen,desclen;
BYTE *pDdata;              // pointer to ROM Descriptor data to send
//
// NOTE This function assumes all descriptors are 64 or fewer bytes and can be sent in a single
packet
//
desclen = 0;             // check for zero as error condition (no case statements satisfied)
reqlen = SUD[wLengthL] + 256*SUD[wLengthH];  // 16-bit
   switch (SUD[wValueH])      // wValueH is descriptor type
   {
   case  GD DEVICE:
         desclen = DD[0];  // descriptor length
         pDdata = DD;
         break;
   case  GD CONFIGURATION:
         desclen = CD[2];  // Config descriptor includes interface, HID, report and ep descriptors
         pDdata = CD;
         break;
   case  GD STRING:
         desclen = strDesc[SUD[wValueL]][0];   // wValueL=string index, array[0] is the length
         pDdata = strDesc[SUD[wValueL]];       // point to first array element
         break;
   case  GD HID:
         desclen = CD[18];
         pDdata = &CD[18];
         break;
   case  GD REPORT:
         desclen = CD[25];
         pDdata = RepD;
        break;
   }  // end switch on descriptor type
//
if (desclen!=0)                // one of the case statements above filled in a value
   {
   sendlen = (reqlen <= desclen) ? reqlen : desclen; // send the smaller of requested and avaiable
        writebytes(rEP0FIFO,sendlen,pDdata);
   wregAS(rEP0BC,sendlen);   // load EP0BC to arm the EP0-IN transfer & ACKSTAT
   }
else STALL EP0                 // none of the descriptor types match
}}
```

図 17. この関数は、要求された記述子を復号して送信します。

ファイルEnumApp_enum_data.hには、USBデバイスに特性を与えるさまざまな記述子のバイト配列が含まれています。*図 17*の**send_descriptor**関数は、SUD[8]配列のSETUPバイトをチェックして、要求された記述子のタイプと長さを判断します。この関数は、要求された記述子のアドレスをポインタ*pDataにロードし、送信するバイト数を判断してこれらを送信します。

send_descriptor 関数は、送信するバイト数(要求された長さ **reqlen**)と、記述子テーブルから取得した記述子の実際の長さ **desclen** の 2 つを使用します。この関数は、**desclen = 0** を設定することで開始します。すべての記述子タイプのチェックが終了した後もなお **desclen** がゼロの場合、有効な記述子タイプが見つからなかったということで、関数は STALL ハンドシェイクを送信します。

要求された長さは、SUD 配列の **wLengthL** と **wLengthH** のバイトにあります。この 16 ビットの値を **reqlen** 変数にロードした後、関数は switch ステートメントを使用して、記述子タイプを示す、さまざまな **wValueH** の値をチェックします。

GD_CONFIGURATION テストには、重要なコメントが記載されています。この関数は、すべての記述子が 1 つのパケットに収まること、したがって、**send_descriptor** 関数への 1 つの呼出しで記述子を処理することができるものと想定しています。MAX3420E は、エンドポイント 0 用に 64 バイトの FIFO を実装しています。これは、フルスピードのデバイスに許される最大サイズです。64 バイトより大きな記述子はないため、すべての記述子データはただ 1 つのパケットに収まります。さらに複雑なデバイスであれば、64 バイトを超える記述子を格納することができます。この場合、全 16 ビット長の値を使用することができるよう(コメントに示す)、かつ **send_descriptor** 関数への複数の呼出しを行えるようにルーチンを修正する必要があります。

残りの関数は、単純で、case ステートメントによって記述子タイプを判断し、目的の記述子データへのポインタをセットし、さらに **desclen** 変数に記述子の長さをロードしています。不明瞭なことに、記述子の長さの値は記述子テーブル内のさまざまな場所に存在します。デバイス記述子と文字列記述子には、記述子の最初のバイトに長さが記載されています。たとえば、デバイス記述子の長さは、バイト DD[0]にあります。構成記述子の長さは、2 番目と 3 番目のバイトにあります。この長さには、構成記述子、HID 記述子(存在する場合)、および全エンドポイント記述子の長さの合計が含まれています。

HID 記述子では、コーディングはやや複雑になります。構成記述子は CD[18]に 9 バイトの HID 記述子を含んでいます。HID 記述子の内部には、HID クラスのデバイスによって使用されているレポート記述子の長さがあります(レポートは、HID 周辺機器が送受信するデータメッセージです)。したがって、図 17 の関数は、レポート記述子を提供するよう求められると、RepD (レポート記述子のアドレス)にアドレスとし、CD[25]から長さを提供します。これが、構成記述子の内部にある HID 記述子内のレポート記述子の長さです。

少しわかりにくいですが、これが USB と HID 仕様のすべてです。一度理解すると忘れることはありません。

switch ステートメントに続いて、関数は、適切な記述子を送信するか、あるいは定義された記述子のうちのいずれかを認識することができなかった場合には STALL ハンドシェイクを送信します。関数は、要求された長さと利用可能な長さのうちの小さい方の値に **sendlen** 変数を設定し、このバイト数を EP0FIFO に書き込みます。最後に、**wregAS()**関数を使用して、バイトカウントを EP0BC レジスタにロードします。

バイトカウントのロードでは、以下が実行されます。

1. EP0 を準備し、ホストがこのエンドポイントに次の IN トークンを送信したときにデータを転送することができるようにします。
2. ACKSTAT ビットをセットし、次の CONTROL 転送のハンドシェイクに ACK で応答するように MAX3420E に指示します。これによって、要求の処理が完了したことを示します。

5.1. set_feature と clear_feature

```
void feature(BYTE sc)
{
BYTE mask;
  if((SUD[bmRequestType]==0x02)  // dir=h->p, recipient = ENDPOINT
  &&  (SUD[wValueL]==0x00) // wValueL is feature selector, 00 is EP Halt
  &&  (SUD[wIndexL]==0x83))// wIndexL is endpoint number IN3=83
      {
      mask=rreg(rEPSTALLS);   // read existing bits
      if(sc==1)               // set_feature
        {
        mask += bmSTLEP3IN;       // Halt EP3IN
        ep3stall=1;
        }
      else                        // clear_feature
        {
        mask &= ~bmSTLEP3IN;      // UnHalt EP3IN
        ep3stall=0;
        wreg(rCLRTOGS,bmCTGEP3IN);  // clear the EP3 data toggle
        }
      wreg(rEPSTALLS,(mask|bmACKSTAT)); // Don't use wregAS--directly writing the ACKSTAT bit
      }
  else if ((SUD[bmRequestType]==0x00)  // dir=h->p, recipient = DEVICE
           &&  (SUD[wValueL]==0x01))// wValueL is feature selector, 01 is Device_Remote_Wakeup
            {
            RWU_enabled = sc<<1; // =2 for set, =0 for clear feature. The shift puts it in the
get_status bit position.
            rregAS(rFNADDR);     // dummy read to set ACKSTAT
            }
  else STALL_EP0
}
```

図 18. set_feature と clear_feature の要求

*図 18*の関数は、set_featureとclear_featureの両方の要求を処理します。この呼出しルーチンは、**sc**引数を 1 (セットの場合)または 0 (クリアの場合)に設定します。

ホストは、デバイスまたはエンドポイントに適用される機能要求を送信します。フルスピードのデバイスの場合、受信者ごとに 1 つの機能要求が定義されます。

- **エンドポイント**: 停止
- **デバイス**: リモートウェイクアップ

関数は最初に、エンドポイントの停止要求を定義したセットアップバイトの有効な組み合わせをチェックします。ホストがエンドポイントを停止すると、周辺機器は、ホストが停止状態をクリアするまで、そのエンドポイントに対するいかなる要求についても STALL ハンドシェイクを返送する必要があります。これを実現するため、MAX3420E には EPSTALLS というレジスタがあります。このレジスタには、MAX3420E の各エンドポイント用のビットが含まれています。エンドポイントの停止に必要な唯一の動作は、これらの MAX3420E STALL ビットの 1 つをセットすることです。

ホストは、clear_feature (エンドポイントの停止)要求を送信して、エンドポイントの停止状態を解除します。今回のケースでは、ファームウェアは最初に、エンドポイントの STALL ビットをクリアし、エンドポイントを通常動作に戻してから、このエンドポイントのデータトグルを DATA0 にクリアします。MAX3420E のレジスタ CLRTOGS を使えば、いずれのエンドポイントのトグルビットもゼロに設定することができます。ただし、ファームウェアがエンドポイントのトグルビットを取り扱う必要があるのはこの時だけです。MAX3420E は、通常の USB 転送の間に、自動的にデータトグルと検証を処理します。

5.2. get_status

```
void get_status(void)
{
BYTE testbyte;
testbyte=SUD[bmRequestType];
switch(testbyte)
   {
   case 0x80:        // directed to DEVICE
      wreg(rEP0FIFO,RWU_enabled+1); // first byte is 000000rs
                                 // where r=enabled for RWU and s=self-powered.
      wreg(rEP0FIFO,0x00);       // second byte is always 0
      wregAS(rEP0BC,2); break;// load byte count, arm the IN transfer,
                                 // ACK the status stage of the CTL transfer
   case 0x81:        // directed to INTERFACE
      wreg(rEP0FIFO,0x00);       // this one is easy--two zero bytes
      wreg(rEP0FIFO,0x00);
      wregAS(rEP0BC,2); break;// load byte count, arm the IN transfer,
                                 // ACK the status stage of the CTL transfer
   case 0x82:        // directed to ENDPOINT
      if(SUD[wIndexL]==0x83)     // We only reported ep3, so it's the only one
                                 // the host can stall. IN3=83
                  {
                  wreg(rEP0FIFO,ep3stall);// first byte is 0000000h where h is the halt bit
                  wreg(rEP0FIFO,0x00);       // second byte is always 0
                  wregAS(rEP0BC,2); break;// load byte count, arm the IN transfer,
                                             // ACK the status stage of the CTL transfer
                  }
      else  STALL_EP0      // Host tried to stall an invalid endpoint (not 3)
   default:       STALL_EP0    // don't understand the request
   }
}
```

図 19. get_status 要求

*図 19*の**get_status**関数は最初に、要求先のUSB周辺機器の部分(デバイス、インタフェース、またはエンドポイント)を復号します。

デバイスのステータスビット

- 現在、自己給電の状態です。今回のデバイスは自己給電であるため、ステータスバイトの LSB は常に 1 です。
- 現在、リモートウェイクアップ(RWU)が有効です。コードは、**RWU_enabled** のフラグを維持しています。このフラグは、set_feature 要求でセットされ、clear_feature 要求でクリアされて、リターンされる RWU ステータスビットになります。

インタフェースのステータスビット

現在、インタフェースのステータスビットは定義されていません。関数は単純に 2 つのゼロバイトを返します。

エンドポイントのステータスビット

1 個のエンドポイントのステータスビットは、「エンドポイントの停止」に定義されています。ホストは、set_feature (エンドポイントの停止)要求を送信することで、エンドポイントを停止し、clear_feature (エンドポイントの停止)要求を送信することで、エンドポイントの停止をクリアします。エンドポイントのステータス要求は、最初のバイトで内部フラグ **ep3stall** を返し、次のバイトでゼロを返します。この設計では、1 つのデータエンドポイントしか使用していないため、関数は、wIndexL フィールドを調べてエンドポイント 3-IN (0x83)の有無をチェックし、見つからなければ要求を停止します。

注:wIndexL のエンドポイント番号には、方向ビット(1 は IN、0 は OUT)が MSB に含まれています。したがって、EP3-IN は 0x03 ではなく 0x83 です。この些細な事項によって、著者はデバッグ処理をたっぷりと「楽しむ」ことができました。

5.3. set_interface と get_interface

```
void set_interface(void)// All we accept are Interface=0 and AlternateSetting=0,
                              // otherwise send STALL
{
BYTE dumval;
if((SUD[wValueL]==0)     // wValueL=Alternate Setting index
  &&(SUD[wIndexL]==0))     // wIndexL=Interface index
   dumval=rregAS(rFNADDR); // dummy read to set the ACKSTAT bit
else STALL_EP0
}

void get_interface(void)// Check for Interface=0, always report AlternateSetting=0
{
if(SUD[wIndexL]==0)      // wIndexL=Interface index
  {
  wreg(rEP0FIFO,0);      // AS=0
  wregAS(rEP0BC,1);      // send one byte, ACKSTAT
  }
else STALL_EP0
}
```

図 20. set_interface と get_interface の要求

図 20 に示すように、このアプリケーションにおける**set_interface**と**get_interface**の関数は単純です。これは、このデバイスが、1 つのインタフェース(インデックス 0)と 1 つの代替設定値(= 0)だけを報告しているからです。より複雑な設計では、コードは代替設定値を保存しており、ホストが代替設定値を**set_interface**要求で変更するたびに、この代替設定に一致するようにエンドポイントを再設定しています。次に、**get_interface**関数がゼロではなく現在の代替設定値を返します。

5.4. set_configuration と get_configuration

```
void set_configuration(void)
{
configval=SUD[wValueL];           // Store the config value
if(configval != 0)                // If we are configured,
  SETBIT(rUSBIEN,bmSUSPIE);       // start looking for SUSPEND interrupts
rregAS(rFNADDR);                  // dummy read to set the ACKSTAT bit
}

void get_configuration(void)
{
wreg(rEP0FIFO,configval);         // Send the config value
wregAS(rEP0BC,1);
}
```

図 21. set_configuration と get_configuration の要求

ホストは、set_configuration要求(値が 1)を使用してデバイスを設定します。*図 21*のコードは、set_configuration要求によって送信された値に変数**configval**を更新し、get_configuration要求でこの値を返します。デバイスが設定さ

れたら、コードはSUSPEND割込みを有効にして、バスサスペンドのチェックを開始します。デバイスを初期化するときにサスペンド割込みが有効であった場合、接続されていないデバイスや設定されていないデバイスによって、MAX3420Eは、繰り返しSUSPEND IRQをアサートすることになります。

5.5. set_address

set_address にはコードがないため、最も簡単な要求です。MAX3420E がこれを処理します。MAX3420E は、内部 FNADDR レジスタを自動的に新しいアドレスで更新した後、このアドレス宛ての要求にのみ応答します。コードで必要なことは、ACKSTAT ビットをセットして要求を終了させることだけです。

5.6. デバッグの補助

*図 22*のコードはデバッグの補助で、SPIポートを介してMAX3420Eのレジスタの読出しと書込みを行う関数**rreg()**と**wreg()**のバージョンをチェックします。デバッグ処理の詳細については、マキシムのウェブサイト上のアプリケーションノート「*MAX3420Eシステムの立上げ*」(japan.maxim-ic.com/AN3663)を参照してください。

```
//
// Diagnostic Aid:
// Call this function from main() to verify operation of your SPI port.
//
void test_SPI(void)          // Use this to check your versions of the rreg and wreg functions
{
BYTE j,wr,rd;
SPI_Init();                  // Configure and initialize the uP's SPI port
wreg(rPINCTL,bmFDUPSPI);     // MAX3420: SPI=full-duplex
wreg(rUSBCTL,bmCHIPRES);     // reset the MAX3420E
wreg(rUSBCTL,0);             // remove the reset
wr=0x01;                     // initial register write value
for(j=0; j<8; j++)
  {
  wreg(rUSBIEN,wr);
  rd = rreg(rUSBIEN);
  wr <<= 1;         // Put a breakpoint here. Values of 'rd' should be 01,02,04,08,10,20,40,80
  }
}
```

図 22. この関数を呼び出して 1 ステップずつ実行することによって ***rreg*** *関数と* ***wreg*** *関数を検証します。*