MAX3420E的USB枚举程序(及其他)

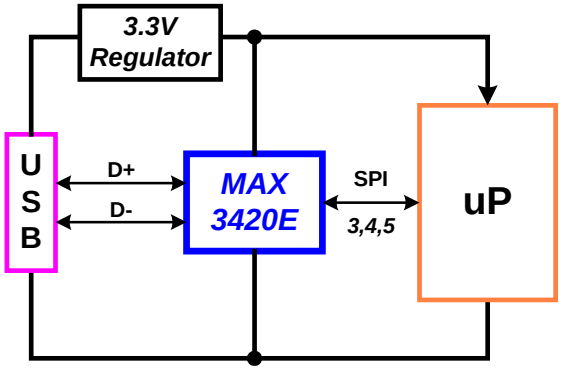
摘要：*MAX3420E USB控制器使设计人员能够给任何系统增加USB外设功能。MAX3420E提供了一个SPI接口用于访问寄存器组，而没有包含板上微处理器，因此可以编写一套适用于多种处理器的MAX3420E C程序。本应用笔记给出了C程序，对实现基本USB操作的所有函数进行了解释。*

下载：完整的程序清单 (包括缺陷修复1和 .h 文件) (ZIP, 11.6kb)。

目录

1. 引言

MAX3420E USB控制器使设计人员能够给任何系统增加USB外设功能。MAX3420E提供了一个SPI接口用于访问寄存器组，而没有包含板上微处理器。因此，可以编写一套适用于多种处理器的MAX3420E C程序。程序代码的可移植性可以扩展到输入输出引脚。例如，假设在MAX3420E的一个GPI (通用输入)引脚上连接了一个按钮，并编写了C代码，来读取IOPINS寄存器查询该按钮的状态，则无需修改代码即可在任何处理器上正确运行，与处理器控制自身IO引脚的方式无关。



USB外设的大部分编程工作是完成枚举过程。也就是说，主机探测外设插入，对外设进行查询以获取外设的性能和要求，如果一切正常，则配置该外设，使其能够在线工作。本应用笔记介绍了一组能够实现MAX3420E枚举过程的C函数。这些函数适用于使用MAX3420E的所有系统。

在枚举期间，设备通过向主机提供数据来说明自己的*身份特征*，并由其应用程序定义自己的*功能*。本应用笔记中的程序不仅仅涉及到枚举问题，还增加了应用程序代码来实现一个简单的键盘仿真设备，它符合标准USB HID (人机接口设备)类。该应用比较实用，使您能够配合PC来测试你的枚举代码部分。由于代码使用标准USB HID类，因此不必在主机PC上安装定制的驱动程序便能够进行测试。

1.1. 代码功能

代码实现了一个单按钮设备，符合标准HID类，完成类似键盘的功能。将该设备插到USB端口后，按下与MAX3420E连接的按钮，则会在任何能够接收键盘数据的打开窗口中输入*图1*所示的信息。代码还处理USB总线复位和挂起-恢复操作。

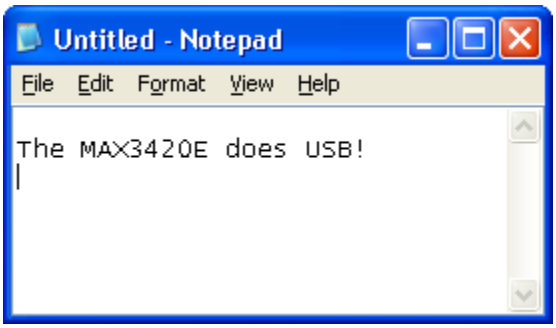


图1. 应用代码输入以上文本串

USB固件由两部分组成：USB协议代码和应用程序代码。应用程序代码输入图1所示的文本串。实现这一功能的代码见*图2*。

```
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
}
```

图2. 实现键盘设备的应用程序代码

当USB主机接收到来自MAX3420E端点3的数据包后，MAX3420E置位中断位IN3BAVIRQ，通知SPI主控制器端点3缓冲区(FIFO)已经准备好装入新的数据。图2中的**do_IN3()**函数检查标志**inhibit_send**，以确定是发送对应按钮弹起的三个零码，还是发送对应下一次按钮动作的三个数据字节。主程序循环检查发送按钮，以更新**inhibit_send**标志。然后，函数检查**send3zeros**标志，在击键动作之间发送“按钮弹起”字码。

这便是所有应用程序代码。那么本应用笔记中其他数页的代码完成什么功能呢? 其他代码完成每一个USB外设都必需完成的工作。而且，这些代码可以被复制/粘贴，作为任何USB外设代码的基本框架。

以下是代码所执行的USB开销操作：

1. 识别并响应USB总线复位。
2. 识别并响应USB总线挂起事件。
3. 通过主机恢复或者用户发出的RWU (远程唤醒)信号执行设备唤醒功能。
4. 识别主机的CONTROL传输，并产生适当的响应。

第4项是主要的代码开销。在枚举期间，主机向设备发送多个请求，询问说明设备工作的描述符(表数据)。由于每一个外设对描述符请求的解码和响应方式完全相同，你可以在其他应用程序中使用完成该功能的这些代码。你只需要改动少量的表数据(描述符)项，设置为特定设备的特性参数。

1.2. 关于可移植性

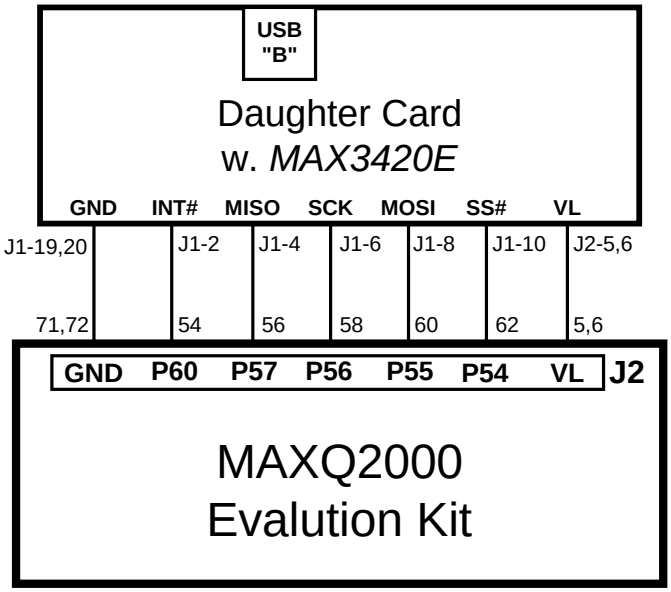


图3. 编写的这些代码运行于MAXQ2000微控制器评估板

*图3*给出了运行这些实例代码的硬件。由于MAX3420E可与任何处理器连接，本应用笔记的程序代码尽可能实现可移植性。采用C语言编写的程序最具可移植性，但该代码中仍有一部分不可避免地与你所采用的处理器和编译器相关。代码中与系统有关的部分包括与MAX3420E通信的SPI接口。本应用笔记中的代码通过以下措施来最大程度减少各处理器之间造成的代码差异。

1. 代码没有使用任何特定的微处理器中断系统。每一种处理器采用不同的寄存器和向量机制来实现中断；编译器使用不同的语法来处理中断向量。因此，该代码程序采用连续循环方式，直接轮询MAX3420E的INT输出引脚。而通常情况下，MAX3420E的INT引脚会接至微处理器的中断引脚，需要修改相应代码以适应特定处理器和编译器处理中断的方法。

   注意：开发MAX3420E新系统时，较好的调试策略是采用轮询IRQ位的方法。到了最后一步，再使用INT引脚和微处理器的中断机制。

2. 位变量采用位掩码方式处理。各种编译器采用不同的方式处理位变量。最常用的C语言移植方法是定义只有一位置1的位掩码(bmBitName)，以C语言标准逻辑运算符来使用这些位掩码。例如，*图4*中的代码显示了怎样测试和清除EPIRQ (端点中断请求)寄存器中的SUDAVIRQ位(设置数据准备好中断请求)。

```
#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)
      }
```

图4. 本代码使用位掩码常量bmSUDAVIRQ来测试和清除MAX3420E的寄存器位

注意：清除IRQ的语句展示了MAX3420E的一个特性：要清除一个IRQ位，你需要将位掩码写入寄存器。这是因为写1将清除IRQ寄存器位，而写0不进行任何操作，不影响寄存器中的其他IRQ位。

3. 程序清单的最后部分是与微处理器和编译器相关的代码。为使代码适应你的特定处理器和编译器，只需要修改这一部分即可。

4. 枚举数据在单独的EnumApp_enum_data.h文件中。你的应用程序需要修改以下数值：

   a. 供应商ID。本代码使用了Maxim的供应商ID (VID) 0x0B6A。你的代码应以自己的VID进行替换。
   b. 产品ID。以自己的产品ID替换0x5346。
   c. 序列号(设备ID)。
   d. 你的应用程序的特定字串索引和字串。
   e. 适合你设备的配置和接口数值(例如，功耗要求、端点数量和类型、类专用描述符等)。

1.3. wreg()和rreg()

SPI主控制器通过SPI接口读写21个寄存器来控制MAX3420E。本应用笔记中的代码使用函数**wreg()** (写寄存器)和**rreg()** (读寄存器)访问MAX3420E的寄存器。由于代码使用MAXQ2000微控制器，这些函数控制MAXQ2000的专用寄存器，通过MAXQ2000的硬件SPI接口来传送数据。更常用的方法是利用微处理器的通用IO (GPIO)引脚，以“位模拟”方式实现SPI接口。如果要将实例代码移植到另一处理器，则需要重新编写**rreg()**和**wreg()**函数，以适应处理器实现SPI主控制器的方法。本应用笔记的程序清单包括一个帮助测试和验证你的**rreg()**和**wreg()**函数版本的程序(*图22*)。

该实例代码还包括一个名为**readbytes()**的多字节函数，演示了怎样使用MAX3420E SPI的突发模式。在突发模式下，SPI主控制器选中SS# (从机选择)引脚，发送SPI命令字节，读取或者写入多个字节，最后解除SS#引脚选择。代码还包括一个**writebytes()**函数供参考—实例代码中并未使用该函数。

2. 初始化

```
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
}
```

图5. MAX3420E和程序变量初始化

*图5*所示为MAX3420E的初始化代码。第一部分初始化程序所使用的几个变量。变量**msgidx**是应用程序输入的文本串的偏移量。**SPI_Init()**函数将微处理器IO引脚配置为SPI端口，与MAX3420E进行通信。实例代码使用MAXQ2000微控制器，它包含硬件SPI单元。因此，通过写多个MAXQ2000配置寄存器来初始化SPI端口。对于没有硬件SPI的微处理器，该程序可简单地设置所用GPIO引脚的方向和初始状态，来实现位模拟SPI接口。

本应用采用MAX3420E SPI端口的全双工模式，使用上面图3所示的分离MOSI和MISO引脚。由于MAX3420E的上电缺省模式是半双工模式，在读取SPI端口之前，必须通过将FDUPSPI位置1来配置全双工模式。PINCTL寄存器还含有一个INTLEVEL位，固件将其设置为1，以配置MAX3420E INT引脚为低电平有效。注意，在这种模式下，INT引脚为开漏极，必须上拉至系统接口电压V_L。关于MAX3420E中断系统的详细信息，请参考Maxim网站的应用笔记：*MAX3420E中断系统* (www.maxim-ic.com.cn/AN3661)。

接下来，函数置位然后清除CHIPRES位，复位MAX3420E，关闭连接在MAX3420E通用输出(GPO)引脚上的LED。最后，函数置位USBCTL寄存器中的CONNECT和VBGATE位，建立USB连接。CONNECT位连接V_{CC}和D+之间的1500Ω内部上拉电阻；VBGATE位确保VBCOMP引脚上没有V_{BUS}时，该上拉电阻从D+上断开。这一特性在自供电设计(例如本设计)中非常重要，因为没有V_{BUS}时，外设一定不能对D+供电。

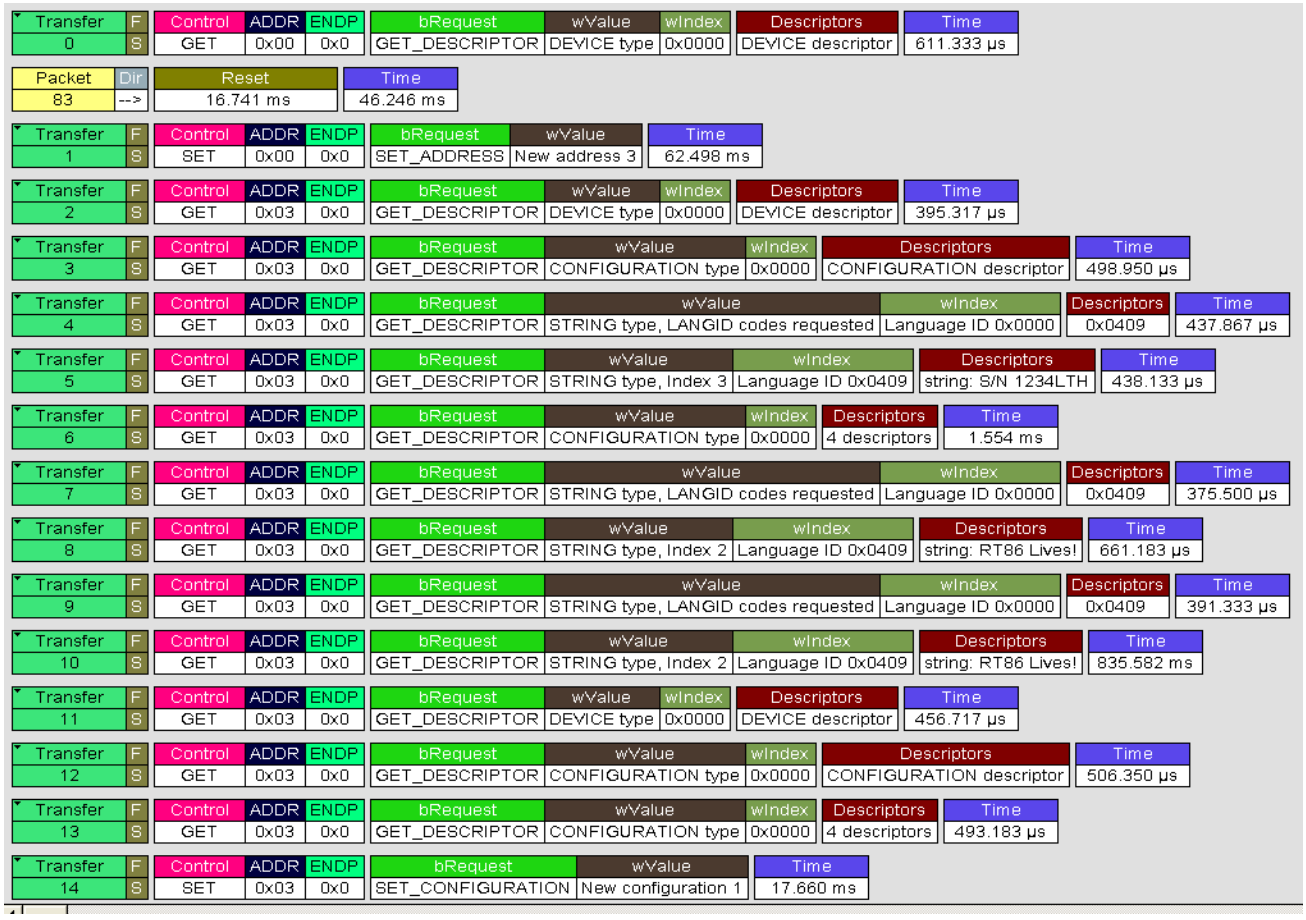
宏 ENABLE_IRQS 的定义如下：

```
#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
```

使能宏内的中断后，在一个地方定义的操作可以被两个函数调用：初始化函数和检测总线复位完成的中断函数。由于总线复位会清除这些中断使能位，因此无论何时主机启动总线复位都需要重新进行初始化。

注意：尽管与USB总线复位(URESIE和URESDNIE)相关的中断使能位不受USB总线复位的影响，最好还是将它们包含在宏中，这样，中断使能可以集中放在代码的一个位置。

3. 枚举处理过程



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

图6. 主机请求和 MAX3420E 响应的总线处理过程

*图6*给出了采用LeCroy (以前的CATC) USB总线分析仪捕获的USB总线处理过程。总线处理过程显示了PC对外设(运行本应用笔记中的C程序)进行的枚举过程。在研究代码之前，我们先观察一下图6，这有助于理解代码是如何工作的。

1. 主机使用缺省的CONTROL端点EP0 (显示在“ENDP”框中)向设备发送请求。主机最初向地址0 (显示在ADDR框中)发送请求，与还没有分配地址的设备进行通信。
2. 主机可以在任意时间、以任意顺序来发出请求。因此，固件必须始终关注SUDAVIRQ (设置数据准备好中断请求)。
3. 主机先发送Get_Descriptor-Device请求(图6中的Transfer 0)，以确定设备EP0缓冲区的数据包最大长度(MAX3420E是64字节)。然后，主机发送总线复位(数据包83)，对设备进行复位。这样便设置了MAX3420E的USBRESIRQ (USB总线复位IRQ)和URESDNIRQ (USB总线复位完成IRQ)寄存器位。
4. 在Transfer 1中，主机使用Set_Address请求，给外设分配一个特定的地址。分配的地址取决于主机当前连接的其他USB设备数量。在本例中，分配给外设的地址是3。MAX3420E自己处理这一请求，将分配的地址装入功能地址寄存器FNADDR (R19)。这样，MAX3420E只响应指向地址3的请求。在主机进行总线复位或者设备断开之前，这一地址始终保持有效。注意，Transfer 1之后，总线处理过程中的外设地址域(ADDR)由0变成了3。
5. 在Transfer 2至13中，主机请求各个描述符。设备固件需要从8个设置字节中确定发送哪个描述符，使用该信息来访问几个字符数组之一(代表描述符数组)，将这些字符装入一个端点FIFO，准备回送给主机。

*图7*展开了图6中的Transfer 0，展示了数据包层次的处理过程。

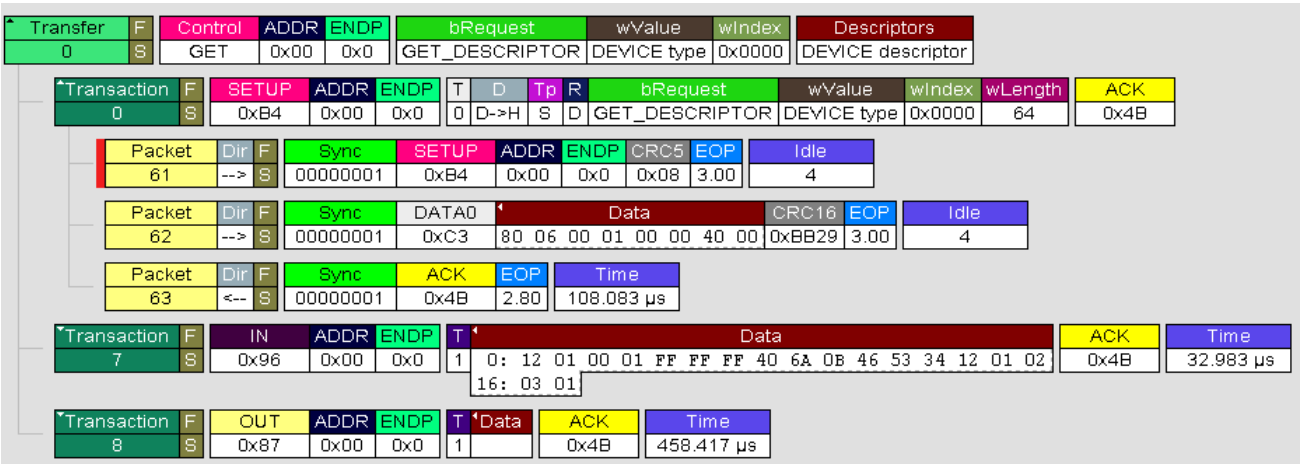


图 7. 将Get_Descriptor-Device请求展开到数据包层次

3.1. 主机请求剖析(CONTROL 传输)

图7中的CONTROL传输由三个阶段或者三个处理过程组成。第一阶段是SETUP处理过程，主机发送SETUP数据包61，它含有设备地址和端点，其后是一个8字节数据包62，告诉外设它需要什么。当设备回送ACK (应答)数据包63，告诉主机传输过程没有错误后，该处理过程结束。正如我们在代码中所看到的，**do_SETUP()**函数从称作SUDFIFO的MAX3420E FIFO中取回8个设置字节，这是通过8次读取寄存器R4实现的。程序分析这8个字节，以确定怎样处理。

数据包62中的最后两个字节指明主机希望设备回送字节的数量。USB采用little-endian格式，因此主机要求十六进制数值0x0040，或者64个字节。但是，在此次传输(Transaction 7)的数据阶段，MAX3420E回送了18个字节。主机要求64个字节，但是程序仅回传了18个字节。到底怎么回事？

以上处理过程是很正常的USB事件，这是由一个简单的规则决定的—总是在(a)主机请求的字节数，以及(b)实际具有的字节数之间取*较小*的字节数目回送。一个设备描述符包括18个字节，而主机要求64个字节，因此MAX3420E固件正确地回送了18个字节。

某些主机请求并不含有数据阶段。例如，Set_Configuration请求(图6中的Transfer 14)含有8个设置字节配置数值。因此不需要数据阶段。

所有的CONTROL传输结束于STATUS阶段(图7中的Transaction 8)，在此期间主机送出一个空的OUT数据包(没有数据)，使外设有机会提供反馈信息。如果外设正忙于处理请求，则以NAK (非应答)进行应答，告诉主机稍后重试STATUS阶段，在得到ACK (应答)响应之前主机将持续进行尝试。

3.2. USB流量控制和NAK

USB架构可巧妙地支持处理能力各不相同的USB外设。USB主机能够与100MHz 32位RISC外设通信，也可以和廉价的鼠标芯片通信。为了使USB与外设的处理能力无关，协议允许设备忙于处理请求时返回一个NAK (非应答)握手包。当设备返回NAK时，告诉主机“我现在忙，稍后再试。”

在Transaction 8 (图7)中，设备以ACK握手包来应答OUT数据包。如果设备还没有完成主机请求的操作，它回送一个NAK应答，告诉主机稍后再次发送OUT数据包。主机在接收到ACK之前将持续发送OUT数据包，收到ACK之后，表示状态阶段已被确认，主机便可认为成功地完成了传输。

3.3. STATUS握手包和ACKSTAT位

MAX3420E采用一个ACKSTAT位(R9的第6位)来处理这一状态握手包，这一位代表确认CONTROL传输的STATUS阶段。固件将图7中的请求解码为Get_Descriptor-Device，查找其设备描述符，通过进行18次R0写操作，将18个字节装入EP0FIFO。然后，固件把数值18写入EP0BC (端点0字节计数)寄存器R5，告诉MAX3420E在Transaction 7中要发送多少字节。最后，固件置位R9中的ACKSTAT位，通知MAX3420E以ACK握手包应答STATUS阶段(Transaction 8)。由于ACKSTAT位用于每一次CONTROL传输过程，MAX3420E为该设置提供了一种捷径。每一次SPI传输的第一个字节是命令字节，连接MAX3420E的控制器发送一个格式如*图8*所示的字节。

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

图8. MAX3420E SPI命令字节

第7位到第3位设置MAX3420E的寄存器地址；第1位设置方向，第0位更新R9中的ACKSTAT位。因此，通过SPI总线向寄存器地址发送18并置位ACKSTAT，能够同时完成两件事情：写入EP0BC (字节计数)寄存器，“准备好”数据传输，同时也置位了R9中的ACKSTAT位。这就是为什么有两个版本的底层函数对MAX3420E进行读写操作的原因：**rreg**、**rregAS**、**wreg**和**wregAS**。AS函数除了置位SPI命令字节中的ACKSTAT位外，其他功能与非AS函数完全相同。

3.4. 数据流量控制

另一个USB流控制机制隐藏在图7中。眼尖的读者可能已经注意到了Transaction 0和Transaction 8之间的空白。这是因为图7使用了隐藏NAK的显示选项，目的是使显示更清晰。(当调试一个USB设计时，最好隐藏这些看起来比较凌乱的NAK。)

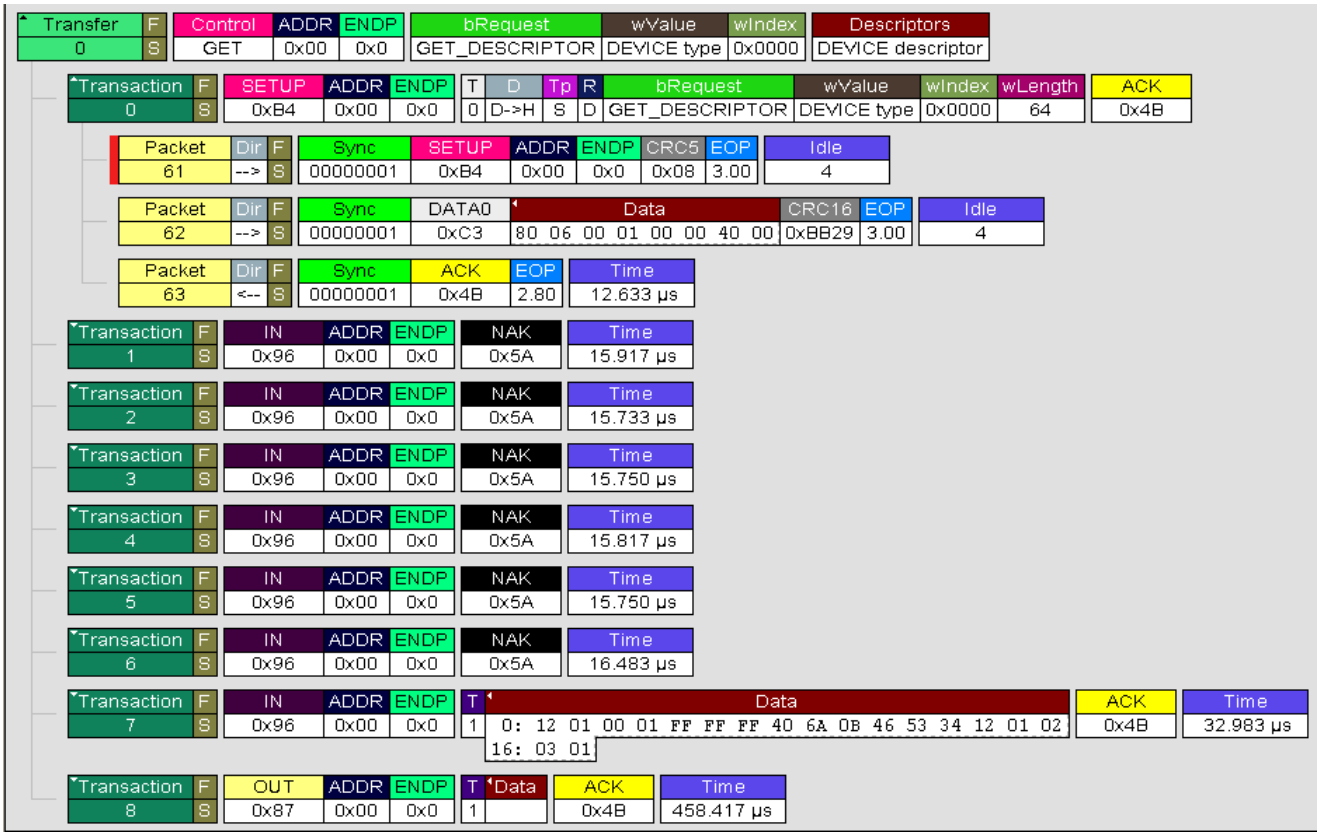


图9. 显示图7中的多个NAK

*图9*显示了固件需要一定的时间来领会请求，查找合适的描述符，将18个字节装入到EP0 FIFO中，写入字节计数。实际上，这花费了6个NAK的时间。当固件处理这些工作时，MAX3420E是怎么知道返回NAK握手包的？这很简单：MAX3420E对于一个端点的IN请求自动返回NAK握手包，直到固件装入该端点的字节计数寄存器为止。这时端点准备好发送数据，MAX3420E以所请求的数据而不是NAK握手包来响应下一个IN请求(Transaction 7)。

4. 程序循环

```
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
```

图10. 始终执行主程序循环

*图10*中的**main()**函数结构模仿一个中断驱动程序，同时又与任何特定的处理器中断系统无关。初始化MAX3420E后，它进入**while(1)**无穷循环。每循环一次，进行两次函数调用：

1. 如果总线被挂起，它调用**check_for_resume()**来检测主机或者用户发起的恢复操作。
2. 如果有MAX3420E中断还没有处理，它调用**service_irqs()**来进行处理。
   本应用中可能出现的中断是：
   a. 设置数据到达(SUDAVIRQ)。
   b. 主机向EP3-IN发送一个数据请求，请求键盘数据。
   c. 主机停止工作3毫秒，挂起总线。
   d. 主机启动总线复位。
   e. 主机完成总线复位信令。

MAX_Int_Pending()函数查询MAX3420E的INT引脚，如果发现该引脚为低电平，则返回数值1。尽管这种方法有些笨拙，但是直接查询INT引脚使程序与微控制器不再相关。在最终应用中，确定程序运行正确后，还需要简单的步骤来启动与MAX3420E INT引脚连接的微控制器中断机制。

主程序循环每20毫秒执行一次，进行以下操作：

1. 读取“发送”按钮的状态，如果按下了按钮，则清除**inhibit_send**标志。
2. 每半秒钟闪烁一次“循环运行中”指示灯。

程序采用**wreg** (写寄存器)和**rreg** (读寄存器)两个函数与MAX3420E进行通信。最后的程序清单列出了这些与具体微处理器有关的函数。寄存器名称前面冠以“r”(例如，rUSBIRQ)，位掩码冠以“bm”(例如，bmBUSACTIRQ)。MAX3420E寄存器和位的定义等式包含在MAX3040E.h文件中。该文件还含有一些使用方便的宏。例如，程序使用宏(全部大写显示) L2_BLINK和L3_ON来控制连接在GPOUT引脚上的LED。如果电路改动了，可以通过宏来方便地修改程序。例如，如果你的应用使用低电平有效的LED，那么只需要修改宏，而其他代码保持不变。

采用软件循环来实现半秒闪烁定时，要根据你的具体应用来进行精确调整。可以调整常量TWENTY_MSEC和BLINKTIME来适应你的处理器时钟速率。在最终应用中，你可能会使用微控制器的硬件定时器单元来实现半秒计时。

4.1. 检查USB恢复

USB 挂起-恢复

USB主机停止发送USB信号3毫秒将使设备进入挂起状态。USB外设需要检测这一挂起指示，以进入低功耗状态，仅从V_{BUS}上吸取很小的电流。MAX3420E设置其SUSPIRQ (挂起IRQ)位，指示主机的挂起操作。一旦进入挂起状态后，可以有两种方法来唤醒设备。第一种，主机直接恢复总线信令。这样会置位MAX3420E的BUSACTIRQ (总线工作IRQ)位。第二种，如果外设能够发送远程唤醒信号，并且主机已使能了这一操作，那么设备可以利用SIGRWU位在总线上发送恢复信号。MAX3420E置位RWUDNIRQ (远程唤醒完成IRQ)中断位，告诉SPI主控制器它已经完成RWU信令。

注意：在挂起状态下，总线供电的外设将MAX3420E置为休眠模式，完成关断操作。通过设置PWRDOWN = 1来完成这一步。这使MAX3420E的片内振荡器停止工作。本应用笔记中的程序实现了一个自供电外设，因此没有这一步操作。

```
void check for resume(void)
{
  if(rreq(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((rreq(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 ((rreq(rUSBIRQ)&bmRWUDNIRQ)==0) ; // spin until RWU signaling done
        CLRBIT(rUSBCTL,bmSIGRWU)       // remove the RESUME signal
        wreq(rUSBIRQ,bmRWUDNIRQ);      // clear the IRQ
        while((rreq(rGPIO)&0x40)==0) ;  // hang until RWU button released
        wreq(rUSBIRQ,bmBUSACTIRQ);     // wait for bus traffic -- clear the BUS Active IRQ
        while((rreq(rUSBIRQ) & bmBUSACTIRQ)==0) ; // & hang here until it's set again...
        }
      }
}
```

图11. 此函数检查USB的两种恢复根源：主机和RWU按钮

*图11*中的**check_for_resume()**函数测试两种唤醒挂起设备的途径：

1. 主机恢复总线上的通信过程。
2. 用户按下与MAX3420E GPIN2引脚相连的“远程唤醒”按钮。

第一条**if**语句处理主机恢复工作。当MAX3420E探测到总线工作时，名为bmBUSACTIRQ (总线工作IRQ)的IRQ位置位。这些代码只是关断挂起指示灯(L1)，并清除**suspended**标志。

else if 部分处理远程唤醒 (RWU)。要使能远程唤醒，必须满足几个条件：

1. 如*图12*所示，设备在其配置描述符中报告它能够发送RWU信号。

```
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
```

图12. 外设借助其CONFIGURATION描述符中bmAttributes字节的第5位，告诉主机它能够发送远程唤醒信号

2. 主机发出Set_Feature-RWU请求。

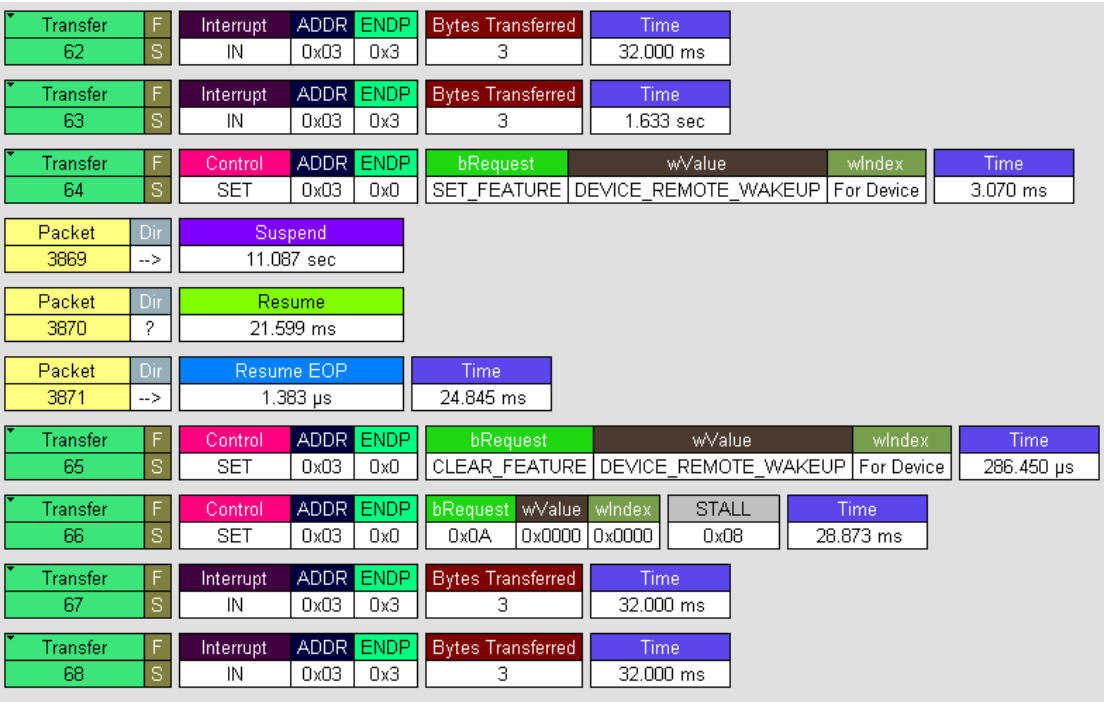


图13. 主机挂起-恢复的总线处理过程

如*图13*的USB总线处理过程所示，由于用户按下PC键盘上的休眠按钮，PC进入了挂起状态。在Transfers 62和63中，主机发送周期性INTERRUPT-IN请求，小键盘(我们的键盘仿真应用程序)相应返回需要的三个字节数据。USB主机将Transfer 64中的特性选择设置为DEVICE_REMOTE_WAKEUP，首先向外设发送一个Set_Feature请求，为USB挂起做准备。如果设备以前没有报告过它能够发送设备远程唤醒信号，主机将不会发送这一请求。

11.087秒之后(数据包3869)，PC用户按下了键盘按键或者移动了鼠标，或者按下了外设的远程唤醒按钮。这样就唤醒了PC，然后在数据包3870中唤醒外设。主机在Transfer 65中清除远程唤醒特性。Transfer 66是名为Set_Idle的HID请求。如果设备不支持这一特性(这里的程序不需要该特性)，正确的响应是STALL握手包。最后，从Transfer 67开始，主机继续向端点3周期性地发送IN数据包，请求键盘数据。

注意：STALL握手包实际上并没有停止USB外设的工作。USB的设计师们本可以为该握手包选择一个更通俗的名称。

4.2. 处理 MAX3420E IRQ 位

```
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
    }
}
```

图14. 该函数检查应用程序需要的IRQ位

*图14*中的**service_irqs()**函数检查MAX3420E的4个IRQ位，处理需要服务的USB事件。这4个事件及其中断请求位的名称是：

1. **bmSUDAVIRQ** SETUP 包到达。主机通过发送 SETUP 包来控制设备。
2. **bmIN3BAVIRQ** 主机请求来自 HID 键盘的另一数据包。它使用端点 3-IN 实现这一请求。BAV 是指 Buffer Available。
3. **bmURESIRQ** 主机开始发送总线复位信号。
4. **bmUSBRESDNIRQ** 主机完成了总线复位过程。

尽管MAX3420E共有14个IRQ位，主循环中只需要这4个来运行键盘仿真应用程序。

函数首先读取MAX3420E的两个IRQ寄存器，将其数值保存在变量**itest1**和**itest2**中。然后，它检查**itest1**确定两个端点中断：SETUP数据端点0和请求发送键盘数据的端点3-IN。如果满足测试条件，将调用服务函数**do_SETUP()**或者**do_IN3()**。注意，尽管**do_SETUP()**分支清除了请求

IRQ位，而**do_IN3()**分支并不清除。要发送IN数据，通过写入IN端点的字节计数寄存器来清除IRQ位，一定不要直接清除该位。这一机制避免了正在装入IN FIFO数据时出现USB IN传输的竞争情况。

其余代码检查**itest2**的数值，确定USB信号事件。MAX3420E检测到总线已经有3ms没有工作后，置位SUSPIRQ位。如果SUSPIRQ位置位，函数设置**suspended**标志，告诉主循环主机已经使总线进入挂起状态。挂起程序还关闭闪烁灯，点亮挂起指示灯。

注意：挂起指示灯比较适用于自供电设备。但如果你的应用是总线供电的(从V_{BUS}线吸取功率)，可能不希望在有限的挂起电流预算中再增加LED电流。

最后两项检查处理USB总线复位。这些代码清除IRQ位，点亮或者关闭总线复位指示灯。注意，MAX3420E的代码一定要含有USB总线复位测试。这是因为MAX3420E在USB总线复位期间会清除大部分中断使能寄存器位。因此，程序应该始终密切注意总线复位事件。复位完成后(由USBRESDN IRQ指示)，应重新使能应用中使用的中断。

5. 枚举的核心—解码 SETUP 数据

*图15*是处理SETUP传输请求的调用函数。只要MAX3420E置位SUDAVIRQ位，主循环便调用该函数。

```
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)
```

图15. 枚举的第一步是解码请求类型

Readbytes函数带有3个参数：第1个，从哪个MAX3420E寄存器中读取数据(在本例中是设置数据FIFO“SUDFIFO”)；第2个，读取的字节数；第3个，指向字节数组的指针，数组用于存储读取的字节。Switch语句查看第一个SETUP字节(bmRequestType)，确定请求类型。大多数枚举请求都是标准请求类型。如果设备采用了标准USB类(例如HID)，也可能出现该类专有的请求。供应商请求是外设生产商规定的定制请求。Default语句显示了USB解码函数应怎样结束。如果没有找到合法的数值，正确的外设响应将发送STALL握手包，而不是ACK或者NAK包，指示没有能够识别该请求。

在实例代码中，**class_request**和**vendor_request**函数是空的。包含这些函数是为了显示当您需要采用这些函数时，在哪里放置它们。

*图16*中的函数处理标准USB请求。

```
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
   }
}
```

图16. 第二步是完成请求

std_request()函数检查SETUP数据包中bRequest字节的有效描述符类型。Switch语句中的“SR_”对应于USB规范第9章中的命名，该规范定义了USB标准请求和数据格式。USB兼容测试的主要部分涉及到检查您的这一段程序，确保您的设备能够正确地响应第9章所定义的请求。

SET_FEATURE和CLEAR_FEATURE请求由同一个**feature()**函数进行处理，该函数采用一个参数来指示需要进行设置(1)还是清除(0)操作。SET_ADDRESS请求不进行任何操作，仅通过无其他用意的读REVISION寄存器操作来置位ACKSTAT位。这是因为MAX3420E硬件自动处理Set_Address请求。

枚举程序的其他部分由图16中的7个函数调用构成。第一个**send_descriptor()**函数比较复杂，其他函数比较简单。

描述符

我们的应用采用以下 USB 描述符类型。

- 设备
- 配置
  - 接口
    - (HID)
    - 端点
- 字串
- (报告)

包含文件EnumApp_enum_data.h含有用于说明外设的各种描述符。USB设备可能会有多种配置和接口，但是我们的应用仅使用了其中一种。上面圆括号中的描述符仅针对HID类，非HID设备将忽略它。

注意：配置描述符包括接口和端点描述符。主机从来不会按名称来请求接口或者端点描述符。它知道能够在配置描述符中找到这些描述符。

```
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);
   wreqAS(rEP0BC,sendlen);   // load EP0BC to arm the EP0-IN transfer & ACKSTAT
   }
else STALL EP0                // none of the descriptor types match
}}
```

图17. 该函数解码并发送所请求的描述符

文件EnumApp_enum_data.h含有说明USB设备特性的各种描述符的字节数组。*图17*给出的**send_descriptor**函数检查SUD[8]数组中的SETUP字节，确定请求的描述符类型和长度。它将所请求的描述符的地址装入指针*pData，确定要发送多少字节并进行发送。

send_descriptor函数使用两个变量来确定要发送的字节数：*请求的*长度**reqlen**和*实际的*描述符长度**desclen** (从描述符表中找到)。该函数从设置**desclen = 0**开始。如果完成了所有的描述符类型检查后，**desclen**仍然是零，那么没有找到一个有效的描述符类型，函数发送STALL握手包。

可以在SUD数组的**wLengthL**和**wLengthH**字节中找到所请求的长度。将该16位数值赋值给**reqlen**变量后，函数使用一个switch语句来检查**wValueH**的数值，从而确定描述符类型。

GD_CONFIGURATION测试含有一个重要的注释。该函数假定所有的描述符都包含在一个数据包中，因此可以使用一个**send_descriptor**函数调用来进行处理。MAX3420E为端点0提供了一个64字节FIFO，这也是全速设备支持的最大长度。任何描述符的长度不会超过64字节，因此，所有的描述符数据都能够装到一个数据包中。更复杂的设备可能含有长度超过64字节的描述符。在这种情况下，应该修改该代码程序，使用整个16位长度值(如注释中所述)，并且多次调用**send_descriptor**函数。

函数的其他部分比较直观，由case语句确定描述符类型，设置指向所需描述符的指针，将描述符长度装入**desclen**变量。有一点可能不太明朗，描述符长度值出现在描述符表中的不同位置。在描述符的第一个字节中含有设备和字串描述符的长度。例如，设备描述符的长度位于字节DD[0]中。配置描述符的长度在第2和第3个字节中，该长度是配置描述符、HID描述符(如果有)以及全部端点描述符的总长度。

HID描述符的编码比较复杂。CONFIGURATION描述符在CD[18]中包含了9个字节的HID描述符。HID描述符中包含HID类设备使用的REPORT描述符长度(REPORTS是HID外设发送和接收到的数据信息)。因此，当要求图17中的函数提供REPORT描述符时，它提供RepD (报告描述符地址)的地址以及来自CD[25]的长度，这就是HID描述符中包含的报告描述符长度，而HID描述符又位于CONFIGURATION描述符内。

这有些混乱，但都是USB和HID规范所要求的。一旦您理解了之后，就再也不需要重新作这些处理了。

在switch语句之后，函数发送正确的描述符，如果没有识别到任何定义的USB描述符，则发送STALL握手包。它将**sendlen**变量设置为请求长度和实际长度二者中的较小数值，并把这一长度字节写入EP0FIFO，最后，使用函数**wregAS()**将字节计数值装入EP0BC寄存器。

装入字节计数完成以下工作：

1. 当主机向该端点发送下一个IN令牌时， EP0准备好进行数据发送。
2. 设置ACKSTAT位，告诉MAX3420E以ACK应答下一个控制传输握手包，表明它已经完成了请求处理。

5.1. 设置/清除特性

```
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
}
```

图18. 设置特性和清除特性请求

*图18*中的函数同时处理Set_Feature和Clear_Feature请求。调用程序设置**sc**参数为1来设置特性，设置为0来清除特性。

主机发送针对设备或者端点的特性请求。对于全速设备，为每个接收方定义了一种特性：

- **端点:** 停用
- **设备:** 远程唤醒

函数首先检查定义端点停用请求的有效设置字节组合。当主机停用一个端点时，要求外设在主机清除停止状态之前，以STALL握手包响应所有针对该端点的请求。为了做到这一点，MAX3420E提供一个EPSTALLS寄存器，该寄存器含有对应每一个MAX3420E端点的位。端点停用所需要的唯一工作是对这些MAX3420E STALL位之一进行置位。

主机发送Clear_Feature (端点停用)请求，取消端点停用条件。在本例中，固件首先清除该端点的STALL位，恢复端点的正常工作，然后清除该端点的数据触发为DATA0。MAX3420E的CLRTOGS寄存器支持任何端点的触发位清零。注意，这是端点触发位需要固件参与的唯一情况。在正常的USB传输过程中，MAX3420E自动处理数据触发和验证。

5.2. 获取状态

```
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
   }
}
```

图19. 获取状态请求

*图19*中的**get_status**函数首先解码请求所指向的USB外设部分：设备、接口和端点。

设备状态位

- 当前的自供电状态。由于我们的设备采用自供电方式，状态字节的LSB保持为1。
- 当前远程唤醒(RWU)的使能状态。程序提供一个**RWU_enabled**标志，由Set_Feature请求置位，由Clear_Feature请求清零，最终反映返回的RWU状态位。

接口状态位

当前没有定义接口状态位。函数只是返回两个零值字节。

端点状态位

定义了一个端点停用的端点状态位。主机发送Set_Feature (端点停用)请求停用一个端点，发送Clear_Feature (端点停用)请求清除端点停用。端点状态请求在第一个字节中直接返回内部标志**ep3stall**，并返回第二个零值字节。由于在该设计中只用到一个数据端点，函数检查wIndexL域搜寻端点3-IN (0x83)，如果没有找到，则停止请求。

注意：wIndexL中端点号的MSB表示方向位。1是IN；0是OUT。因此，EP3-IN是0x83，而不是0x03。这个小小的细节对作者当初的调试工作大有帮助。

5.3. 设置接口和获取接口设置

```
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
}
```

图20. 设置接口和获取接口设置请求

如*图20*所示，本应用中的**set_interface**和**get_interface**函数比较简单，这是因为设备仅报告一个接口(接口号为0)和一个备用设置值(= 0)。在更复杂的设计中，程序将保存备用设置值和重新配置端点，当主机以**set_interface**请求改变设置时，可匹配备用设置。然后，**get_interface**函数可返回当前的备用设置值，而不是零。

5.4. 设置配置和获取配置

```
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);
}
```

图21. 设置配置和获取配置请求

主机利用Set_Configuration请求，以数值1来配置设备。*图21*中的程序以Set_Configuration请求发送的数值更新变量**configval**，并为Get_Configuration请求返回该数值。当设备配置完毕后，程序使能SUSPEND中断，开始检查总线挂起状态。如果设备初始化时使能了挂起中断，拔掉的或者没有配置的设备将导致MAX3420E重复置位SUSPEND IRQ。

5.5. Set_Address

这是所有请求中最简单的请求：没有代码。MAX3420E对其进行处理。它以新的地址自动更新内部FNADDR寄存器，接下来，设备仅仅响应指向该地址的请求。程序需要做的所有工作就是设置ACKSTAT位来终止该请求。

5.6. 调试助手

*图22*中的程序是检验你的**rreg()**和**wreg()**函数版本的调试助手，这些函数通过SPI端口来读取和写入MAX3420E寄存器。关于更详细的调试帮助信息，请参考Maxim公司网站上的应用笔记：*MAX3420E系统调试* (www.maxim-ic.com.cn/AN3663)。

```
//
// 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
  }
}
```

*图22. 调用该函数，单步执行以验证你的**rreg**和**wreg**函数*