Keywords: USB, MAX3420E, MAX3420, C, Enumeration Code

Nov 17, 2005

APPLICATION NOTE 3690

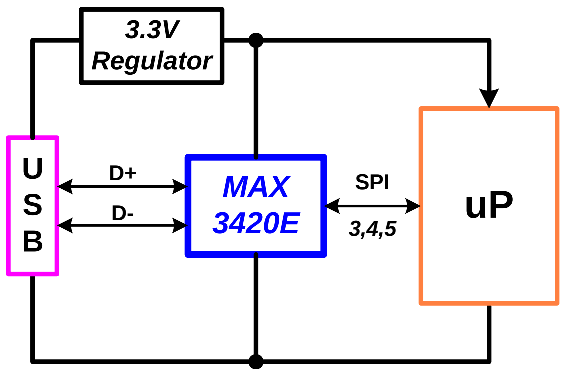
USB Enumeration Code (and More) for the MAX3420E

Abstract: *The MAX3420E USB controller allows designers to add USB peripheral functionality to any system. Because the MAX3420E provides an SPI interface to its register set rather than containing an onboard microprocessor, a set of MAX3420E C routines can be written to serve a wide variety of processors. This application note presents C code and explains all the functions to accomplish the basic USB operations. Basic USB transfers are explained to help understand the code.*

Download: Full Code Listing (including bug fix 1 and .h files) (ZIP, 11.6kb)

Table Of Contents

1. Introduction



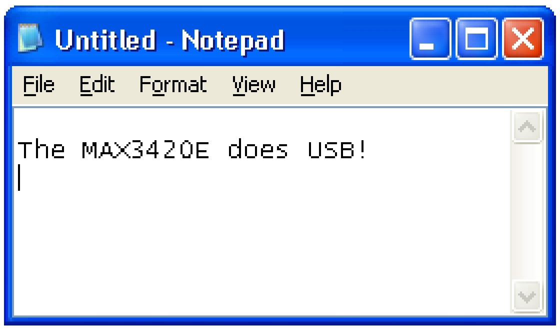
The MAX3420E USB controller allows designers to add USB peripheral functionality to any system. The MAX3420E provides an SPI interface to its register set rather than containing an onboard microprocessor. Consequently, a set of MAX3420E C routines can be written to serve a wide variety of processors. Code portability extends to input-output pins. For example, if you connect a push-button to one of the MAX3420E's GPI (General-Purpose Input) pins and write your C code to interrogate the button by reading the IOPINS register, the code will run unmodified on any processor regardless of how the processor implements its own IO pins.

Much of the programming overhead in a USB peripheral involves the process of enumeration. Specifically, the host detects a plugged-in peripheral, interrogates it to learn about its capabilities and requirements, and if all is well, configures it to bring it on-line. This application note presents a set of C functions that accomplish MAX3420E enumeration. The functions are applicable for any system using the MAX3420E.

While a device defines its *personality* by the data it supplies the host during enumeration, it defines its *functionality* from its application code. The code in this application note goes beyond enumeration; it adds application code to implement a simple keypad-emulator device that conforms to the standard USB HID (Human Interface Device) class. This approach provides a working application that allows you to test your port of the enumeration code with a PC. Because the code uses the standard USB HID class, it can be tested without needing to install a custom driver on the host PC.

1.1. What the Code Does

The code implements a one-button device that acts like a keyboard that conforms to the standard HID class. Plug the device into a USB port, press a button attached to the MAX3420E, and it types the message in *Figure 1* into any open window that accepts keyboard data. The code also handles USB bus reset and suspend-resume.



```
The MAX3420E does USB!
```

Figure 1. The application code types this text string.

USB firmware consists of two parts: USB protocol code and application code. The application code types the string in Figure 1. The code that accomplishes this is listed in *Figure 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
}
```

Figure 2. The application code for implementing a keyboard device.

When the USB host accepts a data packet from the MAX3420E over endpoint 3, the MAX3420E asserts the interrupt bit IN3BAVIRQ. This tells the SPI master that the endpoint 3 buffer (FIFO) is available for loading new data. The **do_IN3()** function in Figure 2 examines the flag **inhibit_send** to determine whether to send the key-up code of three zeros or three data bytes corresponding to the next keystroke. The main program loop checks a send button to update the **inhibit_send** flag. The function then checks a **send3zeros** flag to send the 'keys up' code between every keystroke.

That is all that there is to the application code. What about the remaining several pages of code in this note? The remaining code performs the overhead required by every USB peripheral. As such, it can be copied/pasted to serve as a framework for any USB peripheral code.

Here are the USB overhead operations performed by the code:

1. Recognize and respond to a USB bus reset.
2. Recognize and respond to a USB bus suspend event.
3. Perform device wakeup either by a host resume or a user-initiated RWU (Remote Wakeup) signal.
4. Recognize host CONTROL transfers and generate the appropriate responses.

Item 4 comprises the bulk of the overhead code. During enumeration the host sends multiple requests to the device asking for descriptors (table data) that define the device operation. Because decoding and responding to the descriptor requests are done exactly the same way for every peripheral device, you can use the code here for this task verbatim in your application. You only need to change a few table data (descriptor) items to give it your device’s personality.

1.2. About Portability

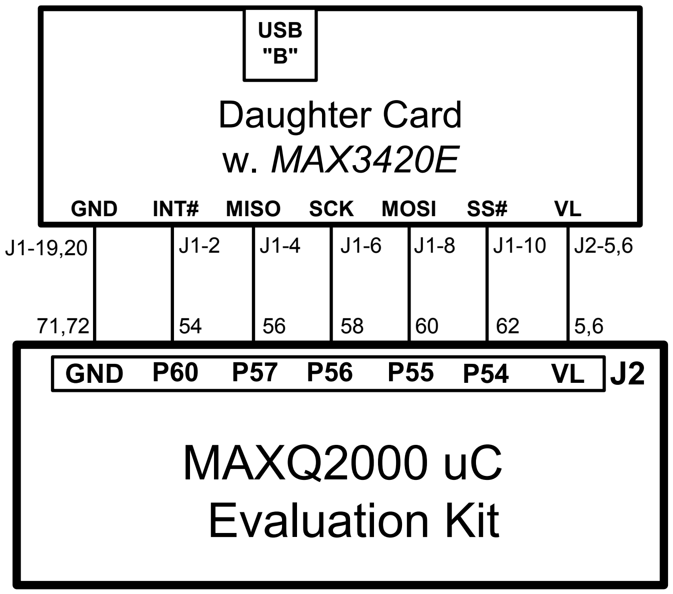


Figure 3. This code is written to run with a MAXQ2000 microcontroller evaluation kit.

Figure 3 shows the hardware used to run this example code. Because the MAX3420E can connect to any processor, this application note makes the code as portable as possible. Writing in C achieves most of this portability, but there is an unavoidable portion of the code that will depend on the processor and compiler that you use. The system-dependent portion of the code implements the SPI interface that talks to the MAX3420E. The code in this application note is written to minimize processor-to-processor differences in the following ways:

1. The code does not assume any particular microprocessor interrupt system. Every processor implements interrupts using different registers and vector mechanisms; compilers use different syntax to handle interrupt vectors. Therefore, this code runs a continuous loop that directly polls the MAX3420E INT output pin. In a typical system the INT pin would be wired to a microprocessor interrupt pin, and the code would be modified to be compatible with the method used by the processor and compiler to deal with interrupts.

   Note: A good debugging strategy in bringing up a new MAX3420E system uses the polled IRQ bit approach to get things working. Then as a last step, use the INT pin and the microprocessor interrupt mechanism.

2. Bit variables are handled as bit masks. Compilers handle bit variables in different ways. The most generic and C-portable method is to define bit masks (bmBitName) with only one bit set to 1, and use these bit masks with the C-standard logic operators. The *Figure 4* code, for example, shows how to test and clear the bit SUDAVIRQ (Setup Data Available Interrupt Request) in the EPIRQ (Endpoint Interrupt Request) register.

```
#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)
      }
```

Figure 4. This code uses the bit mask constant, bmSUDAVIRQ, to test and clear a MAX3420E register bit.

Note: The statement that clears the IRQ demonstrates a MAX3420E feature: to clear an IRQ bit, you write its register with the bit mask. This is because a 1 clears an IRQ register bit, and a 0 does nothing, leaving the remaining IRQ bits in the register alone.

3. The microprocessor- and compiler-specific code appears at the end of the listing. To customize the code for your particular processor and compiler, you only need to change this section.

4. The enumeration data is in its own file, EnumApp_enum_data.h. The following values require modification for your application:

   a. Vendor ID. This code uses the Maxim Vendor ID (VID) of 0x0B6A. Your code should substitute your own VID.
   b. Product ID. Replace 0x5346 with your own product ID.
   c. Serial Number (Device ID).
   d. String indices and strings unique to your application.
   e. Configuration and interface values to suit your device (e.g., power requirements, endpoint numbers and types, class-specific descriptors).

1.3. wreg() and rreg()

An SPI master controls the MAX3420E by writing and reading 21 registers over the SPI interface. The code in this application note uses the functions **wreg()** (write register) and **rreg()** (read register) to access the MAX3420E registers. Because this code uses a MAXQ2000 microcontroller, these functions manipulate MAXQ2000-specific registers to transfer data over the MAXQ2000's hardware SPI interface. A more general approach would be to 'bit bang' a microprocessor's general-purpose IO (GPIO) pins to implement the SPI interface. To port this example code to another processor, the **rreg()** and **wreg()** functions need to be rewritten to accommodate the processor's method for implementing an SPI master. The code listing for this application note includes a routine to help test and verify your versions of the **rreg()** and **wreg()**functions (*Figure 22*).

The example code also includes a multibyte function called **readbytes()** which demonstrates how to use the MAX3420E SPI burst mode. In burst mode the SPI master asserts the SS# (slave select) pin, sends the SPI command byte, reads or writes a series of bytes, and finally deasserts the SS# pin. The code also includes a **writebytes()** function for reference—it is not used by the example code.

2. Initialization

```
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
}
```

Figure 5. MAX3420E and program variable initialization.

Figure 5 shows MAX3420E initialization code. The first section initializes several variables used by the program. The variable **msgidx** is an offset into the text string typed by the application. The **SPI_Init()** function configures the microprocessor IO pins as an SPI port to talk to the MAX3420E. This example code uses a MAXQ2000 microcontroller, which contains a hardware SPI unit. Therefore, the SPI port initialization is done by writing various MAXQ2000 configuration registers. For a microprocessor without hardware SPI, this routine will simply set the directions and initial states for the GPIO pins used to implement the bit-banged SPI interface.

This application uses the MAX3430E full-duplex mode for the SPI port, which employs separate MOSI and MISO pins as shown in Figure 3 above. Since the MAX3420E power-on default is half-duplex, full-duplex operation must be configured by setting the FDUPSPI bit to 1 before reading the SPI port. The PINCTL register also contains a bit called INTLEVEL, which the firmware sets to 1 to configure the MAX3420E INT pin as active-low, level sensitive. Note that in this mode the INT pin is open-drain and must be pulled up to V_L, the system interface voltage. For more information on the MAX3420E interrupt system, see application note, *The MAX3420E Interrupt System* (www.maxim-ic.com/AN3661) on the Maxim website.

Next, the function resets the MAX3420E by setting then clearing its CHIPRES bit, and turns off the LEDs attached to the MAX3420E General-Purpose Output (GPO) pins. Finally, the function establishes the USB connection by setting the CONNECT and VBGATE bits in the USBCTL register. The CONNECT bit connects an internal 1500Ω pullup resistor between V_{CC} and D+; the VBGATE bit ensures that the pullup resistor is disconnected from D+ whenever V_{BUS} is not present on the

VBCOMP pin. This feature is important in a self-powered design (as is this one), because the peripheral must not power D+ in the absense of V_{BUS}.

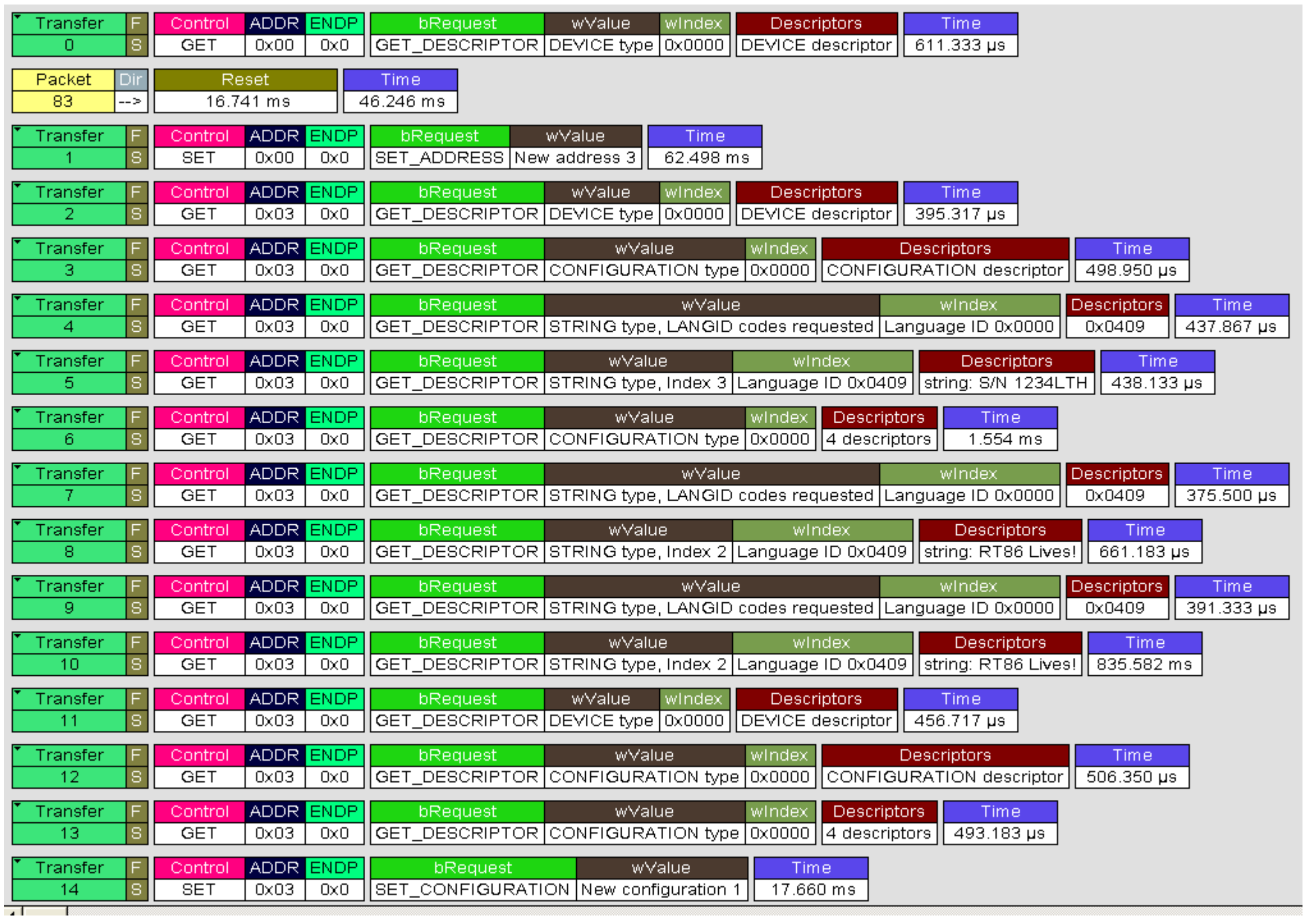
The macro ENABLE_IRQS is defined as follows:

```
#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
```

Enabling the interrupts in a macro allows the operation to be defined in one place and called in two functions: the initialization function and the interrupt function that detects the end of a bus reset. Because a bus reset clears these interrupt enable bits, they need to be reinitialized anytime the host issues a bus reset.

Note: Even though the interrupt enable bits associated with USB bus reset (URESIE and URESDNIE) are unaffected by a USB bus reset, it is still convenient to include them in the macro so the interrupt enables can be concentrated in one place in the code.

3. Enumeration Traffic



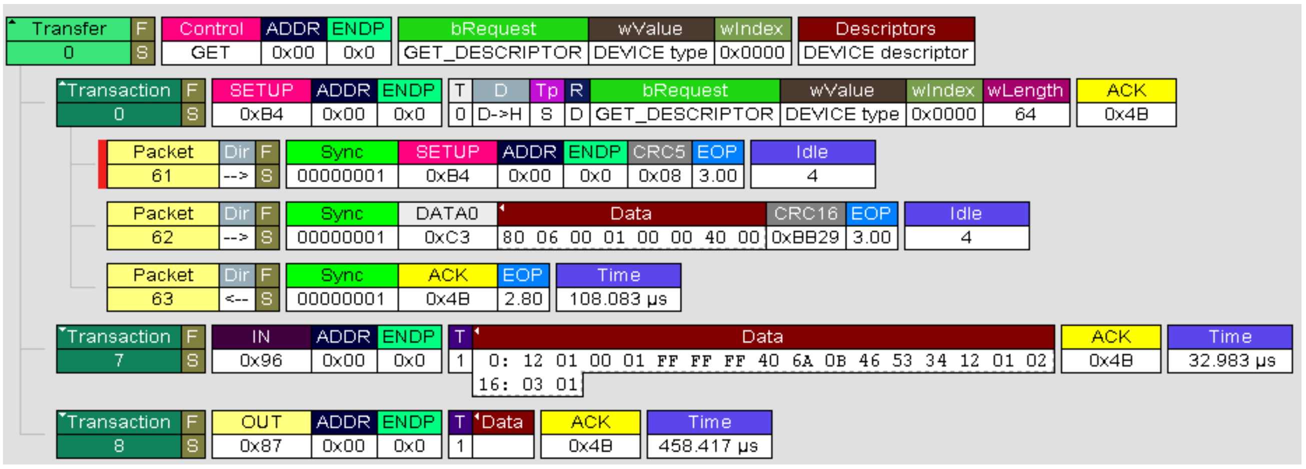
```
Transfer F | Control ADDR ENDP | bRequest          | wValue                               | wIndex             | Descriptors              | Time
0        S | GET     0x00 0x0  | GET_DESCRIPTOR    | DEVICE type                          | 0x0000             | DEVICE descriptor        | 611.333 µs
Packet 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
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
Transfer F | Control ADDR ENDP | bRequest          | wValue                               | Time
14       S | SET     0x03 0x0  | SET_CONFIGURATION | New configuration 1                  | 17.660 ms
```

Figure 6. Bus trace of host requests and the MAX3430E responses.

Figure 6 shows USB bus traffic as captured by a LeCroy (formerly CATC) USB bus analyzer. The traces show a PC enumerating the peripheral device that runs the C code in this application note. Before delving into the code, we will make some observations about Figure 6, which will help you understand just what the code does.

1. The host uses the default CONTROL endpoint EP0 (shown in the "ENDP" boxes) to send requests to the device. The host initially sends requests to address 0 (shown in the ADDR boxes) to communicate with a device to which it has not yet assigned a unique address.
2. Host requests can occur at any time, in any order. Therefore the firmware must always be alert to the SUDAVIRQ (Setup Data Available Interrupt Request).
3. The host begins by sending a Get_Descriptor-Device request (Transfer 0 in Figure 6). It does this to determine the maxPacketSize of the device's EP0 buffer (for the MAX3420E, it is 64 bytes). The host then resets the device by issuing a bus reset (Packet 83). This activates the MAX3420E USBRESIRQ (USB Bus Reset IRQ) and URESDNIRQ (USB Bus Reset Done IRQ) register bits.
4. In Transfer 1, the host assigns a unique address to the peripheral by using the Set_Address request. The assigned address depends on how many other USB devices are currently attached to the host. In this case, the address assigned to our peripheral device is 3. The MAX3420E handles this request by itself, loading the assigned address into the function address register, FNADDR (R19). Thereafter the MAX3420E responds only to requests directed to address 3. This address remains in force until the host does a bus reset or the device is disconnected. Notice that the peripheral address field (ADDR) in the bus trace changes from 0 to 3 after Transfer 1.
5. In transfers 2 through 13, the host asks for various descriptors. The device firmware needs to determine from the eight setup bytes which descriptor to send, use this information to access one of several character arrays (representing the descriptor arrays), and load those characters into an endpoint FIFO for transmission back to the host.

Figure 7 expands the Figure 6 Transfer 0 to show what is occurring at the packet level.



```
Transfer F | Control ADDR ENDP | bRequest       wValue      wIndex | Descriptors
0        S | GET     0x00 0x0  | GET_DESCRIPTOR DEVICE type 0x0000 | DEVICE descriptor

Transaction F | SETUP ADDR ENDP | T D    Tp R | bRequest       wValue      wIndex wLength | ACK
0           S | 0xB4  0x00 0x0  | 0 D->H S  D | GET_DESCRIPTOR DEVICE type 0x0000 64      | 0x4B

Packet Dir F | Sync     SETUP ADDR ENDP CRC5 EOP  | Idle
61     --> S | 00000001 0xB4  0x00 0x0  0x08 3.00 | 4

Packet Dir F | Sync     DATA0 Data                    CRC16  EOP  | Idle
62     --> S | 00000001 0xC3  80 06 00 01 00 00 40 00 0xBB29 3.00 | 4

Packet Dir F | Sync     ACK  EOP  | Time
63     <-- S | 00000001 0x4B 2.80 | 108.083 µs

Transaction F | IN   ADDR ENDP | T Data                                                  | ACK  | Time
7           S | 0x96 0x00 0x0  | 1  0: 12 01 00 01 FF FF FF 40 6A 0B 46 53 34 12 01 02 | 0x4B | 32.983 µs
                                  16: 03 01

Transaction F | OUT  ADDR ENDP | T Data | ACK  | Time
8           S | 0x87 0x00 0x0  | 1      | 0x4B | 458.417 µs
```

Figure 7. The Get_Descriptor-Device request expanded to the packet level.

3.1. Anatomy of a Host Request (CONTROL transfer)

The CONTROL transfer in Figure 7 comprises three stages, or transactions. The first stage is the SETUP transaction, in which the host sends SETUP packet 61 containing the device address and endpoint, followed by an 8-byte data packet 62 which tells the peripheral what it wants. The transaction ends when the device sends back ACK (Acknowledge) packet 63, telling the host that the transfer occurred without errors. As we will see in the code, the **do_SETUP()** function retrieves the eight setup bytes from a MAX3420E FIFO called SUDFIFO, which it accesses by reading register R4 eight times. The code then inspects the eight bytes to determine what to do.

The last two bytes in data packet 62 indicate a word value that represents the number of bytes the host wants the device to send back. USB is little-endian, so the host is asking for hexadecimal 0x0040 or 64 bytes. However, look at the data stage of this transfer (Transaction 7) where the MAX3420E sends back 18 bytes. The host asked for 64 bytes, but the code only sent back 18. What is happening?

The above transaction is a perfectly normal USB occurrence, governed by a simple rule—always send back the *smaller* of (a) the number of bytes that the host requests, and (b) the number of bytes you actually have. A Device Descriptor contains 18 bytes and the host asked for 64 bytes, so the MAX3420E firmware correctly sent back 18 bytes.

Some host requests do not contain a data stage. For example, the Set_Configuration request (Transfer 14 in Figure 6) contains the configuration value in the eight setup bytes. Thus no data stage is required.

All CONTROL transfers end with a STATUS stage (Transaction 8 in Figure 7) during which the host sends out an empty OUT packet (no data) just to give the peripheral a chance to provide some feedback. If the peripheral is busy processing the request, it answers with a NAK (Negative ACKnowledge). This tells the host to retry the STATUS stage at a later time, and to keep trying until it gets an ACK (Acknowledge) response.

3.2. USB Flow Control and NAKs

The USB architects wisely allowed for wide variation in the processing power of a USB peripheral. The USB host could be talking to a peripheral powered by a 100MHz 32-bit RISC or to an inexpensive mouse chip. To make USB independent of processing power in the peripheral device, the protocol allows the device to return a NAK (Negative Acknowledge) handshake whenever it is busy processing a request. When a device returns NAK, it is telling the host, “I am busy. Ask again later.”

In Transaction 8 (Figure 7) the device answers the OUT packet with an ACK handshake. If the device has not finished performing the operation requested by the host, it sends back a NAK response, telling the host to resend the OUT packet at a later time. The host keeps sending the OUT packets until it receives an ACK, at which time the status stage is acknowledged and the host can consider the transfer successfully completed.

3.3. STATUS Handshake and the ACKSTAT Bit

The MAX3420E handles this status handshake with a bit called ACKSTAT (R9 bit 6), which stands for ACKnowledge the STATUS Stage of a CONTROL transfer. The firmware decodes the Figure 7 request as Get_Descriptor-Device, looks up its device descriptor, and loads the 18 bytes into the EP0FIFO by writing R0 18 times. Then the firmware writes the EP0BC (Endpoint Zero Byte Count) register R5 with the number 18, which tells the MAX3420E how many bytes to send in Transaction 7. Finally, the firmware sets the ACKSTAT bit in R9, which instructs the MAX3420E to answer the STATUS stage (Transaction 8) with an ACK handshake. Because the ACKSTAT bit is used for every CONTROL transfer, the MAX3420E provides a shortcut for setting it. The first byte of every SPI transfer is a command byte, in which the controller connected to the MAX3420E sends a byte with the format shown in *Figure 8*.

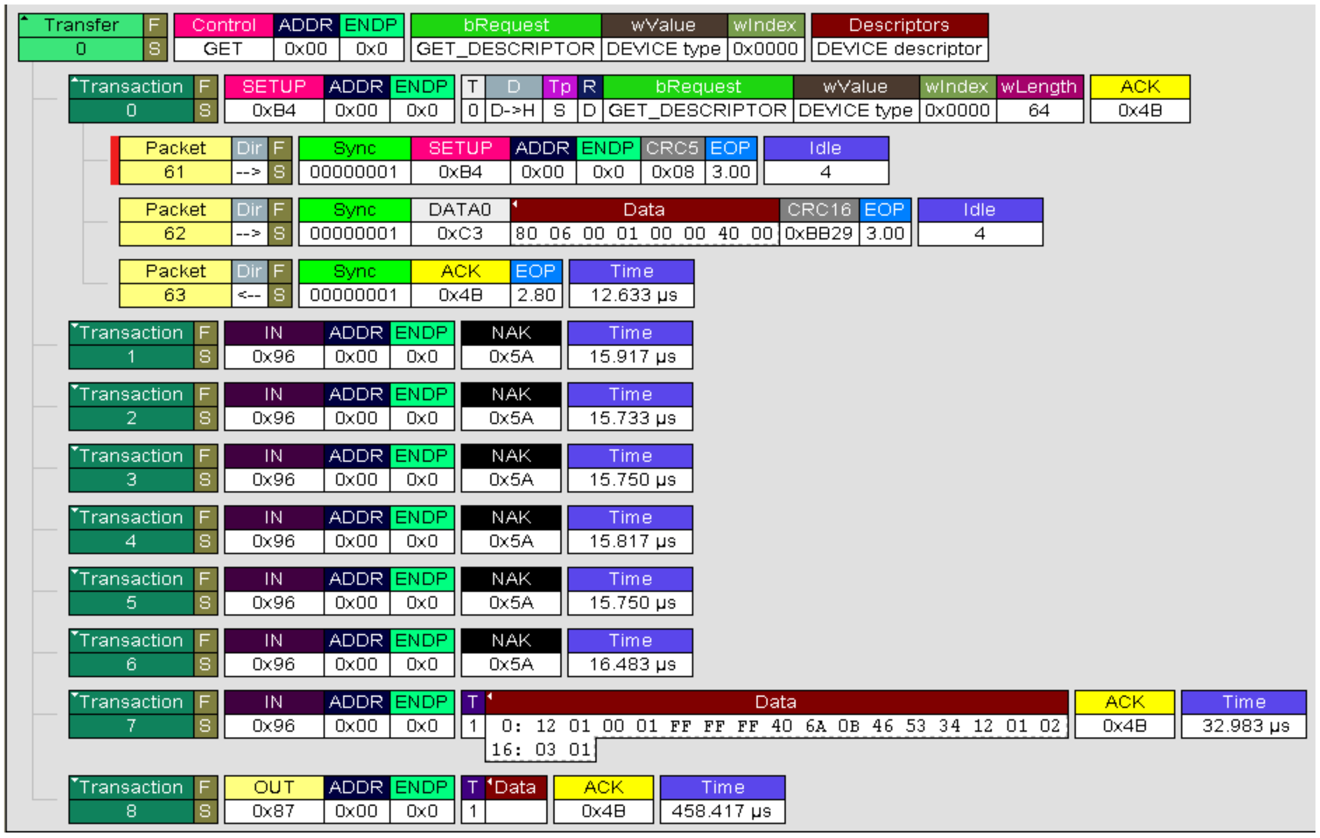
b7	b6	b5	b4	b3	b2	b1	b0
Reg4	Reg3	Reg2	Reg1	Reg0	0	DIR 1=wr 0=rd	**ACKSTAT**

Figure 8. The MAX3420E SPI command byte.

Bits 7 through 3 set the MAX3420E register address; bit 1 sets the direction, and bit 0 updates the ACKSTAT bit in R9. Therefore, by sending the register address 18 over the SPI bus with the ACKSTAT bit set, you accomplish two things at once: writing the EP0BC (byte count) register to 'arm' the data transfer, and also setting the ACKSTAT bit in R9. This is why there are two versions of the low-level functions that read and write bytes to the MAX3420E: **rreg**, **rregAS**, **wreg**, and **wregAS**. The AS functions do exactly what the non-AS ones do, but with the ACKSTAT bit set in the SPI command byte.

3.4. Data Flow Control

Another USB flow control mechanism is hidden in Figure 7. Sharp-eyed readers will have noticed the gap between Transaction 0 and Transaction 8. This is because Figure 7 used a display option to hide NAKs to make it clearer. (When debugging a USB design, it is usually good to hide NAKs since they clutter the display.)


```
Transfer F | Control ADDR ENDP | bRequest       | wValue      | wIndex | Descriptors
0        S | GET     0x00 0x0  | GET_DESCRIPTOR | DEVICE type | 0x0000 | DEVICE descriptor

  Transaction F | SETUP ADDR ENDP | T D    Tp R | bRequest       | wValue      | wIndex wLength | ACK
  0           S | 0xB4  0x00 0x0  | 0 D->H S  D | GET_DESCRIPTOR | DEVICE type | 0x0000 64      | 0x4B

    Packet Dir F | Sync     | SETUP | ADDR ENDP CRC5 EOP  | Idle
    61     --> S | 00000001 | 0xB4  | 0x00 0x0  0x08 3.00 | 4

    Packet Dir F | Sync     | DATA0 | Data                    | CRC16  EOP  | Idle
    62     --> S | 00000001 | 0xC3  | 80 06 00 01 00 00 40 00 | 0xBB29 3.00 | 4

    Packet Dir F | Sync     | ACK  | EOP  | Time
    63     <-- S | 00000001 | 0x4B | 2.80 | 12.633 µs

  Transaction F | IN   ADDR ENDP | NAK  | Time
  1           S | 0x96 0x00 0x0  | 0x5A | 15.917 µs

  Transaction F | IN   ADDR ENDP | NAK  | Time
  2           S | 0x96 0x00 0x0  | 0x5A | 15.733 µs

  Transaction F | IN   ADDR ENDP | NAK  | Time
  3           S | 0x96 0x00 0x0  | 0x5A | 15.750 µs

  Transaction F | IN   ADDR ENDP | NAK  | Time
  4           S | 0x96 0x00 0x0  | 0x5A | 15.817 µs

  Transaction F | IN   ADDR ENDP | NAK  | Time
  5           S | 0x96 0x00 0x0  | 0x5A | 15.750 µs

  Transaction F | IN   ADDR ENDP | NAK  | Time
  6           S | 0x96 0x00 0x0  | 0x5A | 16.483 µs

  Transaction F | IN   ADDR ENDP | T | Data                                                | ACK  | Time
  7           S | 0x96 0x00 0x0  | 1 |  0: 12 01 00 01 FF FF FF 40 6A 0B 46 53 34 12 01 02 | 0x4B | 32.983 µs
                                 |   | 16: 03 01                                           |      |

  Transaction F | OUT  ADDR ENDP | T | Data | ACK  | Time
  8           S | 0x87 0x00 0x0  | 1 |      | 0x4B | 458.417 µs
```

Figure 9. Figure 7 with NAKs displayed.

Figure 9 shows that it took the firmware some time to understand the request, look up the right descriptor, stuff the EP0 FIFO with 18 bytes, and write the byte count. Actually, it took six NAKs worth of time. How did the MAX3420E know to return the NAK handshake while the firmware was doing all this? It is quite simple: the MAX3420E automatically returns the NAK handshake for an IN request to an endpoint until the firmware loads the byte count register for that endpoint. This arms the endpoint to transfer data and the MAX3420E responds to the next IN request (Transaction 7) with the requested data instead of a NAK handshake.

4. The Program Loop

```
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
```

Figure 10. The main program loop executes forever.

The **main()** function in *Figure 10* is structured to mimic an interrupt-driven program, while still independent of any particular processor's interrupt system. After initializing the MAX3420E, it enters an endless **while(1)** loop. Every pass through the loop, it makes two function calls:

1. If the bus is suspended, it calls **check_for_resume()** to detect a host-initiated or user-initiated resume operation.
2. If any MAX3420E interrupts are pending, it services them with a call to **service_irqs()**. The possible interrupts for this application are:
    a. Setup Data arrived (SUDAVIRQ).
    b. The host requested keyboard data by sending a data request to EP3-IN.
    c. The host suspended the bus by stopping traffic for three milliseconds.
    d. The host initiated a bus reset.
    e. The host completed bus reset signaling.

The **MAX_Int_Pending()** function polls the MAX3420E INT pin and returns a 1 value if the pin is found to be low. Although not elegant, this direct polling of the INT pin makes the code independent of the microcontroller. In a final application, after the code is verified to run properly, it is a small step to activate the microcontroller interrupt mechanism to which the MAX3420E INT pin connects.

The main program loop executes every 20 milliseconds, and does the following:

1. Reads the state of the ‘send’ pushbutton, and clears the **inhibit_send** flag if the button is pressed.
2. Blinks a ‘loop active’ light every half-second.

The code communicates with the MAX3420E using two functions called **wreg** (write register) and **rreg** (read register). These microprocessor-dependent functions are included at the end of the full listing. The register names are prefixed with ‘r’ (e.g., rUSBIRQ) and the bit masks are prefixed with ‘bm’ (e.g., bmBUSACTIRQ). MAX3420E register and bit equates are in the included MAX3040E.h file. This file also includes some handy macros. The code uses, for example, macros (shown in all caps) L2_BLINK and L3_ON to manipulate the LEDs attached to the GPOUT pins. The macros allow easy modification of the code if the circuitry changes. For example, if your application uses active-low LEDs, you only need to change the macros while leaving the code alone.

The half-second blink timer is implemented using a software loop which needs fine-tuning for your particular implementation. The constants TWENTY_MSEC and BLINKTIME can be adjusted to account for your processor clock speed. In a final application you probably would use a microcontroller’s hardware timer unit to count off the half seconds.

4.1. Checking for USB Resume

About USB Suspend-Resume
A USB host suspends a device by stopping USB signaling for three milliseconds. A USB peripheral is required to sense this suspend indication and assume a low-power state than draws very little current from the V_{BUS} wire. The MAX3420E activates its SUSPIRQ (Suspend IRQ) bit to indicate a host-suspend operation. Once suspended, a device can wake up two ways. First, the host can simply resume bus signaling. This activates the MAX3420E BUSACTIRQ (Bus Active IRQ) bit. Second, if the peripheral is capable of signaling remote wakeup and the host has enabled it to do so, the device can drive a resume signal on the bus by using the SIGRWU bit. The MAX3420E asserts an interrupt bit called RWUDNIRQ (Remote Wakeup Done IRQ) to alert the SPI master that it has completed the RWU signaling.

Note: In the suspended state, a bus-powered peripheral completes power-down operations by putting the MAX3420E to sleep. It does this by setting the bit PWRDOWN = 1. This stops the MAX3420E on-chip oscillator. The code in this note implements a self-powered peripheral, and therefore, does not take this step.

```
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...
        }
      }
}
```

Figure 11. This function checks for USB resume from two sources: the host and RWU pushbutton.

The **check_for_resume()** function in *Figure 11* tests for two ways to wake up a suspended device:

1. The host resumes traffic on the bus.
2. The user presses the ‘Remote Wakeup’ button attached to the MAX3420E GPIN2 pin.

The first **if** statement handles a host-resume operation. The IRQ bit called bmBUSACTIRQ (Bus Active IRQ) asserts when the MAX3420E detects bus traffic. This code merely turns off the suspend light (L1), and clears the **suspended** flag.

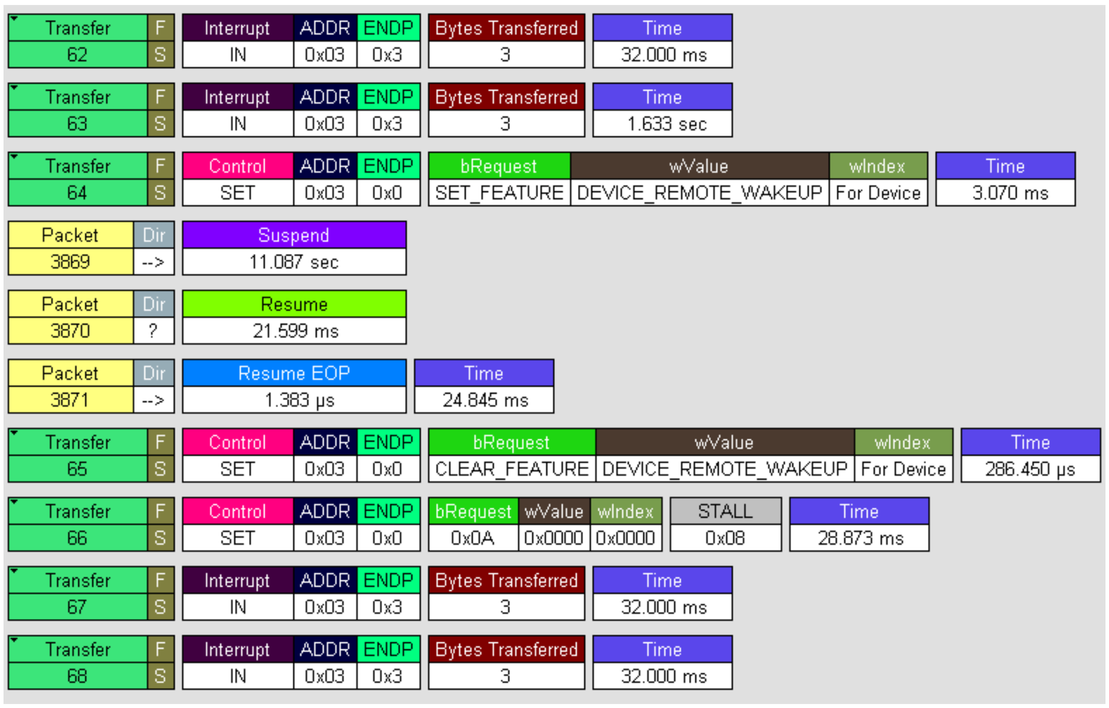
The **else if** block handles remote wakeup (RWU). For remote wakeup to be enabled, a few conditions need to be satisfied:

1. The device reports that it is capable of signaling RWU in its configuration descriptor, as shown in *Figure 12*.

```
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
```

Figure 12. The peripheral tells the host that it is capable of signaling remote wakeup in bit 5 of the bmAttributes byte of its CONFIGURATION descriptor.

2. The host issues a Set_Feature-RWU request.



```
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
```

Figure 13. Bus trace for host suspend-resume.

Figure 13 is a USB bus trace showing a PC entering its suspend state as a result of the user pressing the sleep button on the PC keyboard. In transfers 62 and 63, the host sends its periodic INTERRUPT-IN requests, and the keyboard (our keypad emulator application) returns the required three bytes. The host prepares for USB suspend by first sending the peripheral a Set_Feature request with the feature selector set to DEVICE_REMOTE_WAKEUP in transfer 64. If the device had not previously reported that it is capable of signaling a device remote wakeup, the host would never send this request.

11.087 seconds later (packet 3869) the PC user hits a keyboard key or moves the mouse, or the user presses the remote wakeup button on the peripheral device. This wakes up the PC, which then proceeds to wake up the peripheral in packet 3870. The host clears the remote wakeup feature in transfer 65. Transfer 66 is a HID request called Set_Idle. If the device does not support this feature (the code here does not need this feature), the correct response is the STALL handshake. Finally, starting with transfer 67, the host continues sending IN packets to endpoint 3 to periodically request keypad data.

Note: A STALL handshake does not actually stall operation of the USB peripheral. The USB architects could have chosen a less foreboding sounding name for this handshake.

4.2. Servicing the MAX3420E IRQ Bits

```
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
    }
}
```

Figure 14. This function checks the IRQ bits needed by this application.

The **service_irqs()** function in *Figure 14* checks four MAX3420E IRQ bits for a USB event that requires service. These four events and their interrupt request bit names are:

1.	**bmSUDAVIRQ**	A SETUP packet arrived. The host controls a device by sending SETUP packets.
2.	**bmIN3BAVIRQ**	The host asked for another data packet from the HID keyboard. It uses endpoint 3-IN for this request. BAV means Buffer Available.
3.	**bmURESIRQ**	The host started signaling a bus reset.
4.	**bmUSBRESDNIRQ**	The host finished signaling a bus reset.

Although the MAX3420E has a total of 14 IRQ bits, only these four are needed in the main loop to run the keypad-emulator application.

The function first reads the two MAX3420E IRQ registers and saves their values in the variables **itest1** and **itest2**. Then it checks **itest1** for one of two endpoint interrupts: endpoint 0 for SETUP data and endpoint 3-IN for a request to send keyboard data. If satisfied, these tests call the service function,

do_SETUP() or **do_IN3()**. Note that while the **do_SETUP()** branch clears the requesting IRQ bit, the **do_IN3()** branch does not. To send IN data, you clear the IRQ bit by writing the IN endpoint's byte count register, never by directly clearing the bit. This mechanism prevents a competing race situation where you could be loading IN FIFO data at the same time the USB IN transfer is occurring.

The remaining checks inspect the **itest2** value for USB signaling events. The MAX3420E asserts the SUSPIRQ bit when it detects the lack of bus activity for three milliseconds. If the SUSPIRQ bit is set, the function sets the **suspended** flag to tell the main loop that the host has suspended the bus. The suspend code also turns off the blinking light and turns on a suspend light.

Note: The suspend light is fine for a self-powered device. If, however, your application is bus powered (drawing power from the V_{BUS} wire), you probably do not want an LED adding current to the limited suspend current budget.

The last two checks handle USB bus reset. These code sections clear the IRQ bits and turn a bus reset light on and off. Note that MAX3420E code should always include a test for a USB bus reset. This is because the MAX3420E clears most of its interrupt enable register bits during a USB bus reset. Therefore, the code should be alert to a bus reset. When the reset is complete (signaled by the USBRESDN IRQ), it should re-enable the interrupts that it is using for the application.

5. The Heart of Enumeration—Decoding the SETUP Data

Figure 15 is the function call that handles a SETUP transfer request. The main loop calls this function whenever the MAX3420E asserts the SUDAVIRQ bit.

```
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)
```

Figure 15. First enumeration step is to decode the request type.

The **readbytes** function takes three arguments: first, a MAX3420E register from which to read data (in this case, the Setup Data FIFO "SUDFIFO"); second, the number of bytes to read; and third, a pointer to a byte array to store the bytes. The switch statement looks at the first SETUP byte (bmRequestType) to determine the request type. Most enumeration requests are standard requests. If the device implements a standard USB class (such as HID), there may also be requests unique to that class. Vendor requests are custom requests defined by the peripheral maker. The default statement shows how USB decoding functions should end. If no legal case value is found, the correct peripheral response is to send the STALL handshake instead of ACK or NAK to indicate that it failed to recognize the request.

In this example code, the **class_request** and **vendor_request** functions are empty. They are included here just to show where to put the hooks if you need to implement either of these functions.

The function shown in *Figure 16* services the standard USB requests.

```
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
   }
}
```

Figure 16. Second step is to figure out which request.

The **std_request()** function checks for valid descriptor types in the bRequest byte of the SETUP data packet. The ‘SR_’ names in the switch statement correspond to the nomenclature in Chapter 9 of the USB Specification, which defines the USB standard requests and data formats. The bulk of the USB compliance testing involves checking this part of your code to ensure that your device properly responds to the requests defined in Chapter 9.

The SET_FEATURE and CLEAR_FEATURE requests are served by a single **feature()** function that takes an argument indicating which operation, set (1) or clear (0), is required. The SET_ADDRESS request does nothing but set the ACKSTAT bit by doing a dummy read to the REVISION register. This is because the MAX3420E hardware automatically handles the Set_Address request.

The remainder of the enumeration code consists of the seven functions called in Figure 16. The first one, **send_descriptor()**, is the big one. The others are quite simple.

About Descriptors

Our application uses the following USB descriptor types.

- Device
- Configuration
    - Interface
        - (HID)
        - Endpoint
- String

- (Report)

The include file EnumApp_enum_data.h contains the various descriptors used to give our peripheral device its personality. A USB device can have multiple configurations and interfaces, but our application uses only one of each. The descriptors in parenthesis above are unique to the HID class, and would be omitted by a non-HID device.

Note: A configuration descriptor contains interface and endpoint descriptors. The host never asks for an interface or endpoint descriptor by name. It knows that it will retrieve these descriptors as part of the configuration descriptor.

```
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
}}
```

Figure 17. This function decodes and sends the requested descriptor.

The file EnumApp_enum_data.h contains byte arrays for the various descriptors that give the USB device its personality. The **send_descriptor** function in *Figure 17* checks the SETUP bytes in the SUD[8] array to determine the requested descriptor type and length. It loads the pointer *pData with the address of the requested descriptor, figures out how many bytes to send, and sends them.

The **send_descriptor** function uses two variables to determine the number of bytes to send: the *requested* length, **reqlen**; and the *actual* descriptor length, **desclen,** which it retrieves from the descriptor tables. The function starts by setting **desclen = 0**. If **desclen** is still zero after all the descriptor-type checks are complete, a valid descriptor type was not found and the function sends the STALL handshake.

The requested lengths are found in the **wLengthL** and **wLengthH** bytes of the SUD array. After loading this 16-bit value into the **reqlen** variable, the function uses a switch statement to check for various values of **wValueH,** which indicates the descriptor type.

The GD_CONFIGURATION test has an important comment. This function assumes that all descriptors fit into one packet, and therefore can be handled with one call to the **send_descriptor** function. The MAX3420E implements a 64-byte FIFO for endpoint 0, which is the maximum size allowed for a full-speed device. Therefore, all descriptor data fits into a single packet since no descriptor is larger than 64 bytes. A more complex device could contain descriptors that exceed 64 bytes. In that case this routine should be modified both to use the full 16-bit length value (as shown in the comment), and to make multiple calls to the **send_descriptor** function.

The rest of the function is straightforward, as the case statements determine the descriptor type, set a pointer to the desired descriptor data, and load the **desclen** variable with the descriptor length. What may not be obvious is that the descriptor length values are found in various places in the descriptor tables. Device and string descriptors contain their length in the first byte of the descriptor. The device descriptor length, for example, is in the byte DD[0]. The configuration descriptor has its length in the second and third bytes, and this length includes the summed lengths of the configuration descriptor, the HID descriptor (if present), and all the endpoint descriptors.

Coding is somewhat convoluted in the HID descriptors. The CONFIGURATION descriptor contains the 9-byte HID descriptor at CD[18]. Inside the HID descriptor is the length of the REPORT descriptor used by the HID class device. (REPORTS are data messages sent and received by HID peripherals.) So when the Figure 17 function is asked to provide the REPORT descriptor, it furnishes the address at RepD (the report descriptor address) and the length from CD[25], that is the report descriptor length inside the HID descriptor which is inside the CONFIGURATION descriptor.

This is a little confusing, but it is all in the USB and HID specs. Once you understand it, you never need to do it again.

Following the switch statement, the function sends the proper descriptor or the STALL handshake if it did not recognize one of the defined USB descriptors. It sets the **sendlen** variable to the smaller of the requested and available lengths, writes this number of bytes into the EP0FIFO, and, finally, loads the byte count into the EP0BC register using the **wregAS()** function.

Loading the byte count does the following:

1. Arms EP0 to transfer the data when the host sends this endpoint the next IN token.
2. Sets the ACKSTAT bit to tell the MAX3420E to answer the next control transfer handshake with ACK, indicating that it has finished servicing the request.

5.1. Set/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
}
```

Figure 18. Set Feature and Clear Feature requests.

The *Figure 18* function handles both Set_Feature and Clear_Feature requests. The calling routine sets the **sc** argument to 1 for set and 0 for clear.

The host sends feature requests that apply to the device or to an endpoint. For a full-speed device, one feature request is defined for each recipient:

- **Endpoint:** Halt
- **Device:** Remote Wakeup

The function first checks for the valid combination of setup bytes that define an endpoint halt request. When the host halts an endpoint, the peripheral is required to return the STALL handshake for any request to that endpoint until the host clears the halt condition. To accomplish this, the MAX3420E has a register called EPSTALLS, which includes bits for each MAX3420E endpoint. The only action required for an endpoint halt is to set one of these MAX3420E STALL bits.

The host removes an endpoint halt condition by sending the Clear_Feature (Endpoint Halt) request. In this case the firmware first clears that endpoint's STALL bit, which restores the endpoint to normal operation, and then clears that endpoint's data toggle to DATA0. The MAX3420E register CLRTOGS allows any endpoint's toggle bit to be set to zero. Note that this is the only time that the firmware needs to be concerned with an endpoint toggle bit. The MAX3420E automatically takes care of the data toggles and verifications during normal USB transfers.

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
   }
}
```

Figure 19. Get Status request.

The **get_status** function in *Figure 19* first decodes that portion of the USB peripheral to which the request is directed: device, interface, or endpoint.

Device Status Bits

- Currently self-powered. Because our device is self-powered, the LSB of the status byte is always 1.
- Currently enabled for remote wakeup (RWU). The code maintains a flag for **RWU_enabled**, which is set by a Set_Feature request, cleared by a Clear_Feature request, and which becomes the returned RWU status bit.

Interface Status Bits

There are no currently defined interface status bits. The function simply returns two zero bytes.

Endpoint Status Bits

One endpoint status bit is defined, endpoint halt. The host halts an endpoint by sending the Set_Feature(Endpoint halt) request, and clears an endpoint halt by sending the Clear_Feature(Endpoint halt) request. The endpoint status request simply returns the internal flag, **ep3stall,** in the first byte, and a zero second byte. Since only one data endpoint is used in this design, the function checks the wIndexL field for endpoint 3-IN (0x83) and, if it does not find it, stalls the request.

Note: The endpoint number in wIndexL contains a direction bit in its MSB. 1 is IN; 0 is OUT. Therefore, EP3-IN is 0x83, not 0x03. This little detail provided the author many happy debugging hours.

5.3. Set Interface and 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
}
```

Figure 20. Set Interface and Get Interface requests.

As *Figure 20* shows, the **set_interface** and **get_interface** functions are simple for this application because this device reports only a single interface (index 0) and a single alternate setting value (= 0). In a more complex design, the code would save the alternate setting value and reconfigure the endpoints to match the alternate setting any time the host changes it with a **set_interface** request. Then the **get_interface** function would return the current alternate setting value rather than zero.

5.4. Set Configuration and 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);
}
```

Figure 21. Set Configuration and Get Configuration requests.

The host configures a device using the Set_Configuration request with a value of 1. The *Figure 21* code updates the variable **configval** to the value sent by the Set_Configuration request, and returns this value for the Get_Configuration request. When the device is configured, the code enables the SUSPEND interrupt to start checking for a bus suspend. If the suspend interrupt were enabled when the device is initialized, an unplugged or unconfigured device would cause the MAX3420E to repeatedly assert the SUSPEND IRQ.

5.5. Set_Address

This is the simplest request of all: no code. The MAX3420E takes care of this. It automatically updates an internal FNADDR register with the new address, and subsequently responds only to requests directed to that address. All the code needs to do is set the ACKSTAT bit to terminate the request.

5.6. A Debugging Aid

The code in *Figure 22* is a debug aid to check your versions of the **rreg()** and **wreg ()** functions that read and write the MAX3420E registers over the SPI port. For more debugging help, check the Maxim website for the application note, *Bringing Up a MAX3420E System* (www.maxim-ic.com/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
  }
}
```

Figure 22. Call this function and step through it to verify your ***rreg*** *and* ***wreg*** *functions.*