Appendix. Code listing

```
      //--------------------------------------------------------------------------
      // Copyright (C) 2005 Maxim Integrated Products, Inc. All Rights Reserved.
      //
      // Permission is hereby granted, free of charge, to any person obtaining a
      // copy of this software and associated documentation files (the "Software"),
      // to deal in the Software without restriction, including without limitation
      // the rights to use, copy, modify, merge, publish, distribute, sublicense,
      // and/or sell copies of the Software, and to permit persons to whom the
      // Software is furnished to do so, subject to the following conditions:
      //
      // The above copyright notice and this permission notice shall be included
      // in all copies or substantial portions of the Software.
      //
      // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
      // OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
      // MERCHANTABILITY,  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
      // IN NO EVENT SHALL MAXIM INTEGRATED PRODUCTS INC. BE LIABLE FOR ANY CLAIM, DAMAGES
      // OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
      // ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
      // OTHER DEALINGS IN THE SOFTWARE.
      //--------------------------------------------------------------------------
//
// PanicButton.C     10-4-04 LTH
// A simple but elegant USB HID (Human Interface Device). Simulates a two button keyboard.
// Runs on the MAXQ2000 development kit, but is easily modified for anything that either has an SPI port
// or can bit-bang one.
// When the pushbutton attached to the MAXQ3420E GP-OUTPUT-0 is pressed, send keystrokes to minimize all
// active windows and display only the desktop. When pressed again, the windows reappear.
// If the PC suspends ("Stand By"), AND it supports remote wakeup via USB, the button doubles as a wakeup button.
// (Many computers do NOT support USB remote wakeup)
// Tested with Windows XP Home Edition

#include <intrinsiSS.h>             // MAXQ2000 specific stuff
#include <iomaxq200x.h>             // ditto
#include "Qspi.h"                   // MAX3420E specific stuff (register & bit names, USB constants)
#include "Panic_Button_enum_data.h" // HID keyboard enumeration data
#define TIMEOUT 8000                              // time between pushbutton checks

#define SS_HI PO5 |= 0x10;          // Macros to set the MAXQ2000 SPI SS signal (PO5) high and low
#define SS_LO PO5 &= ~0x10;

#define IDLE 0              // IN3 state variables
#define RELEASE 1
#define WAIT 2

//Global variables
BYTE SUD[8];                        // my copy of setup data
BYTE configval;                     // From Set_Configuration, reported back in Get_Configuration
BYTE ep3stall;                              // Flag for EP3 Stall, set by Set_Feature, reported back in Get_Status
BYTE RWU_enabled;       // The host has enabled us for remote wakeup via the Set_Feature request
BYTE Suspended;         // "we are suspended" flag
BYTE state,button;      // state variable for the IN3 service routine, state of the button
//
// prototypes
void enable_USB_ints(void);
void do_SETUP(void);
void Do_IN3(void);
BYTE Check_INT(void);
void Reset_MAX3420E(WORD time);
void SPI_Init(void);
void wreg(BYTE reg, BYTE dat);
void wregAS(BYTE reg, BYTE dat);
BYTE rreg(BYTE reg);
BYTE rregAS(BYTE reg);
void readbytes(BYTE reg, BYTE N, BYTE *p);
void writebytes(BYTE reg, BYTE N, BYTE *p);

void enable_USB_ints(void)         // Call this at init time and after a USB bus reset
{
wreg(rEPIEN,(bmSUDAV|bmIN3BAV));      // Enable the SUDAV and IN3 interrupts
wreg(rUSBIEN,(bmUSBRES|bmUSBRESDN));  // NOTE: SUSPEND interrupt is enabled when the device is configured
}

// ***************************** MAIN *********************************
void main(void)
{
BYTE itest1,itest2;         // for polling the interrupt requests
WORD button_time;                   // time delay for polling the button state
ep3stall=0;                                 // EP3 inintially un-halted (no stall) (CH9 testing)
button_time = 0;
```

```
//
SPI_Init();                 // set up MAXQ2000 to use its SPI port as a master
wreg(rPINCTL,(bmFDUPSPI|bmPOSINT));   // 3420: INTLEVEL=0, POSINT=1
EIES1_bit.IT12 = 0;                   // MAXQ: 0=pos edge triggered IRQ
Reset_MAX3420E(1000);
enable_USB_ints();

// software flags
configval=0;                        // at pwr on OR bus reset we're unconfigured
Suspended=0;                // and not in suspend
RWU_enabled=0;              // and the host has not yet enabled us for remote wakeup
// EP3-IN. Load "key up" code (00 00 00) to initialize for HID
wreg(rEP3INFIFO,0);
wreg(rEP3INFIFO,0);
wreg(rEP3INFIFO,0);
wreg(rEP3INBC,3);                   // arm the first EP3-IN transfer
state = IDLE;              // initialize the 'Do_IN3' function state machine

wreg(rUSBCTL,bmCONNECT);    // since VBGATE=0, CONNECT is not conditional on Vbus present
wreg(rCPUCTL,bmIE);                 // Enable interrupt pin

while(1)                                                    // endless loop
      {
             do
             {
             button_time++;
             if (button_time == TIMEOUT)
                    {
                    button_time = 0;
                    button = (~rreg(rGPIO)) & 0x10;       // Button state. GPO-0,complement to make active high
                    if(Suspended)                                                 // check the remote wakeup button
      only if suspended
                           {
                           if (button)                                            // pushbutton doubles as remote
      wakeup button
                              {
                              SETBIT(rUSBCTL,bmRESUME)   // signal resume
                                   while ((rreg(rUSBIRQ)&bmRSUMDN)==0) ; // spin until resume signaling done
                                   CLRBIT(rUSBCTL,bmRESUME)      // remove the RESUME signal
                                   wreg(rUSBIRQ,bmRSUMDN);              // clear the IRQ
                                   Suspended=0;                                  // stop checking the button
                                   }
                           }

                    } // if (button_time == TIMEOUT)
             } // do
             while (Check_INT() == 0);                          // if no pending interrupts, just check the button
             itest1 = rreg(rEPIEN) & rreg(rEPIRQ);        // only consider the enabled ones
             itest2 = rreg(rUSBIEN) & rreg(rUSBIRQ);      // ditto

// Something is pending.
// Data section (either setup data has arrived or EP3-IN is available for another key to be sent)
      if(itest1 & bmSUDAV)
             {
             wreg(rEPIRQ,bmSUDAV);                 // clear the SUDAV IRQ
             do_SETUP();
             }
      else if(itest1 & bmIN3BAV)
             {
             Do_IN3();
             }
// SUSPEND-RESUME Section
      else if(itest2 & bmSUSPEND)           // HOST suspended bus for 3 msec
             {
             wreg(rUSBIRQ,bmSUSPEND);              // clear the IRQ
             CLRBIT(rUSBIEN,bmSUSPEND);            // de-activate the SUSPEND IRQ so we don't keep getting it during
      suspend
             wreg(rUSBIRQ,bmBUSACT);                      // clear remnants of bus activity
             SETBIT(rUSBIEN,bmBUSACT);             // enable 'resume' interrupt
             Suspended=1;                                          // so main loop can check for a "resume" button
      press
             }
      else if(itest2 & bmBUSACT)                   // This indicates that the host has resumed USB traffic
             {
             wreg(rUSBIRQ,bmBUSACT);                      // clear the IRQ
             CLRBIT(rUSBIEN,bmBUSACT);             // disable 'resume' interrupt
             SETBIT(rUSBIEN,bmSUSPEND);            // enable SUSPEND interrupt
             Suspended=0;                                          // flag to stop checking the RESUME button
             }

// USB Bus Reset Section
      else if(itest2 & bmUSBRES)
```

```
            wreg(rUSBIRQ,bmUSBRES);                     // clear the IRQ
      else if(itest2 & bmUSBRESDN)
            {
            wreg(rUSBIRQ,bmUSBRESDN);
            enable_USB_ints();                          // start over
            }
      } // end while(1)
} // end main

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 at b6&b5
      {
      case 0x00:     std_request();        break;
      case 0x20:     class_request();      break;
      case 0x40:     vendor_request();     break;
      default:              STALL_EP0
      }
}

// The IN3 FIFO is available. State machine states go from IDLE -> SEND -> RELEASE -> IDLE
// Uses the two global variables 'button' (1=pressed), and 'state', which is initialized to IDLE
void Do_IN3(void)
{
  switch(state)
  {
  case IDLE:
    if (button)
      {
      wreg(rEP3INFIFO,0x08);  // "Windows" prefix key
      wreg(rEP3INFIFO,0);
      wreg(rEP3INFIFO,0x07);  // "D" key
      wreg(rEP3INBC,3);                 // arm it
      state = RELEASE;              // next state sends the "keys up" code
      }
      break;    // else do nothing (and the SIE will NAK)
//
  case RELEASE:
      {
      wreg(rEP3INFIFO,0x00);        // key up
      wreg(rEP3INFIFO,0x00);
      wreg(rEP3INFIFO,0x00);        // key up
      wreg(rEP3INBC,3);                             // arm it
      state = WAIT;                 // next state waits for the PB to be unpressed
      }
      break;
  case WAIT:
    if (!button)
      state = IDLE;
    break;
  default:  state = IDLE;
  } // end switch
}

// ********************************************************************************************

void std_request(void)
{
BYTE dum;                                 // dummy byte to read register with AckStat bit set
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:                       dum=rregAS(rFNADDR);  break;
      default:  STALL_EP0
      }
}

void set_configuration(void)
{
BYTE dumval;
configval=SUD[wValueL];                         // Config value is here
      if (configval != 0)
            SETBIT(rUSBIEN,bmSUSPEND);    // enable the suspend interrupt
dumval=rregAS(rFNADDR);                         // dummy read to set the ACKSTAT bit
}
```

```
void get_configuration(void)
{
wreg(rEP0FIFO,configval);
wregAS(rEP0BC,1);
}
```

```
// ********************************************************************************************
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
}

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

      default:      STALL_EP0                   // the host messed up
      }
}

// ********************************************************************************************
// FUNCTION: Set/Get_Feature. Call as feature(1) for Set_Feature or feature(0) for Clear_Feature.
// There are two set/clear feature requests:
//          To a DEVICE:   Remote Wakeup (RWU). When RWU is enabled, enable the SUSPEND interrupt.
//                                      (Otherwise, SUSPEND will trigger if not plugged in).
//    To an ENDPOINT:       stall (EP3 only for this app)
//
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 += bmEP3IN;      // set only this bit
                              ep3stall=1;
                              }
                        else                                            // clear_feature
                              {
                              mask &= !bmEP3IN;     // clear only this bit
                              ep3stall=0;
                              }
                  wreg(rEPSTALLS,(mask|bmACKSTAT)); // Don't use the wregAS for this--already 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
}

void send_descriptor(void)
{
WORD reqlen,sendlen,desclen;
BYTE *pDdata;                                                           // pointer to ROM Descriptor data to send

// NOTE This function assumes all packets are 64 or fewer bytes
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 I/F, HID, report and EP descriptors
                        pDdata = CD;
                        break;
      case    GD_STRING:
                            switch     (SUD[2])             // wValueL is string index
                                {
                                case 0:
                                  desclen = STR0[0];
                                  pDdata = STR0;
                                  break;
                                case 1:
                                  desclen = STR1[0];
                                  pDdata = STR1;
                                  break;
                                case 2:
                                  desclen = STR2[0];
                                  pDdata = STR2;
                                  break;
                                case 3:
                                  desclen = STR3[0];
                                  pDdata = STR3;
                                } // end switch
                            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/avaiable
      writebytes(rEP0FIFO,sendlen,pDdata);
      wregAS(rEP0BC,sendlen);                               // load the EP0BC to arm the EP0-IN transfer ('sendlen'
      bytes requested) & ACKSTAT
      }
else STALL_EP0
}


void class_request(void)
{
STALL_EP0           // the only one we should get is Set_Idle, which we don't support and therefore must STALL
}

void vendor_request(void)
{}

// ****************************************************************************
// These functions are normally saved in a separate file to make the code more modular.
// They are included here to have all the code in one listing, better for an article.
// ****************************************************************************
void SPI_Init(void)
{
// MAXQ2000 SPI port
```

```
  CKCN = 0x00;              // system clock divisor is 1
  SS_HI                     // SS# high
  PD5 |= 0x070;             // Set SPI output pins (SS, SCLK, DOUT) as output.
  PD5 &= ~0x080;            // Set SPI input pin (DIN) as input.
  SPICK = 0x00;             // fastest SPI clock--div by 2
  SPICF = 0x00;             // mode(0,0), 8 bit data
  SPICN_bit.MSTM = 1;       // Set Q2000 as the master.
  SPICN_bit.SPIEN = 1;      // Enable SPI
// MAX3420E INT pin is tied to MAXQ2000 P60; make it an input
  PD6 &= ~0x01;             // PD6.0=0 (turn off output)
}

void Reset_MAX3420E(WORD time)
{
WORD k;
wreg(rUSBCTL,0x20);                 // chip reset
for(k=0; k<time; k++);              // a delay
wreg(rUSBCTL,0x00);                 // remove the reset
}

BYTE Check_INT(void)        // returns 0 if nothing pending, nonzero if something pending
{
  if(EIF1_bit.IE12)         // Test the IRQ Flag (P60 pin feeds the Int12 IRQ flipflop)
    {
    EIF1_bit.IE12 = 0;      // It's set--clear it
    return(1);              // show an IRQ is active
    }
  else return(0);           // flag=0: no IRQ active
}

// Read a register, return its value.
BYTE rreg(BYTE reg)
{
BYTE dum;
  SS_LO
  SPIB = reg<<3;                // reg number w. dir=0 (IN)
  while(SPICN_bit.STBY);        // loop if data still being sent
  dum = SPIB;                   // read and toss the input byte
  SPIB=0x00;                    // data is don't care, we're clocking in MISO bits
  while(SPICN_bit.STBY);        // loop if data still being sent
  SS_HI
  return(SPIB);
}

// Same as rreg, but also set the AckStat bit in the command byte.
BYTE rregAS(BYTE reg)
{
BYTE dum;
  SS_LO
  SPIB = (reg<<3)+1;            // reg number w. dir=0 (IN) and ACKSTAT=1
  while(SPICN_bit.STBY);        // loop if data still being sent
  dum = SPIB;                   // read and toss the input byte
  SPIB=0xFF;                    // data is don't care, we're clocking in MISO bits
  while(SPICN_bit.STBY);        // loop if data still being sent
  SS_HI
  return(SPIB);
}
void wreg(BYTE reg, BYTE dat)
{
  SS_LO                         // Set SS# low
  SPIB = (reg<<3)+2;            // send the register number with the DIR bit (b1) set to WRITE
  while(SPICN_bit.STBY);        // loop if data still being sent
  SPIB = dat;                   // send the data
  while(SPICN_bit.STBY);        // loop if data still being sent
  SS_HI                         // set SS# high
}

// Write a MAX3420E register with the "ACK STATUS" bit set in the command byte
void wregAS(BYTE reg, BYTE dat)
{
  SS_LO                         // Set SS# low
  SPIB = (reg<<3)+3;            // reg number with DIR=1 (write) and ACKSTAT=1
  while(SPICN_bit.STBY);        // loop if data still being sent
  SPIB = dat;                   // send the data
  while(SPICN_bit.STBY);        // loop if data still being sent
  SS_HI                         // set SS# high
}

void readbytes(BYTE reg, BYTE N, BYTE *p)
{
BYTE j;
  SS_LO
  SPIB = reg<<3;            // write bit b1=0 to command a read operation
```

```
  while(SPICN_bit.STBY);    // loop if data still being sent
  j = SPIB;                 // NECESSARY TO RE-ENABLE THE INPUT BUFFER in BYTE MODE
  for(j=0; j<N; j++)
    {
    SPIB = 0x00;            // dummy value to get the next read byte
    while(SPICN_bit.STBY);  // loop if data still being received
    *p = SPIB;              // store it in the data array
    p++;                    // bump the pointer
    }
  SS_HI
}
void writebytes(BYTE reg, BYTE N, BYTE *p)
{
BYTE j,wd;
  SS_LO
  SPIB = (reg<<3)+2;        // write bit b1=1 to command a write operation
  while(SPICN_bit.STBY);    // loop if data still being sent
  for(j=0; j<N; j++)
    {
    wd = *p;                // write the array value
    SPIB = wd;
    while(SPICN_bit.STBY);  // loop if data still being received
    p++;                    // bump the pointer
    }
  SS_HI
}

// Panic_Button_Enum_Data.h
// Enumeration tables for a HID keyboard device

unsigned char DD[]=                 // DEVICE Descriptor
    {0x12,                  // bLength = 18d
    0x01,                           // bDescriptorType = Device (1)
      0x00,0x01,                    // bcdUSB(L/H) USB spec rev (BCD)
      0xFF,0xFF,0xFF,       // bDeviceClass, bDeviceSubClass, bDeviceProtocol
      0x40,                         // bMaxPacketSize0 EP0 is 64 bytes
      0x6A,0x0B,                    // idVendor(L/H)--Maxim is 0B6A
      0x46,0x53,                    // idProduct(L/H)--5346
      0x34,0x12,                    // bcdDevice--1234
      1,2,3,                        // iManufacturer, iProduct, iSerialNumber
      1};                           // bNumConfigurations

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
// INTERFACE Descriptor
      0x09,                         // length = 9
      0x04,                         // type = IF
      0x00,                         // IF #0
      0x00,                         // bAlternate Setting
      0x01,                         // bNum Endpoints
      0x03,                         // bInterfaceClass = HID
      0x00,0x00,            // bInterfaceSubClass, bInterfaceProtocol
      0x00,                         // iInterface
// HID Descriptor--It's at CD[18]
      0x09,                         // bLength
      0x21,                         // bDescriptorType = HID
      0x10,0x01,            // bcdHID(L/H) Rev 1.1
      0x00,                         // bCountryCode (none)
      0x01,                         // bNumDescriptors (one report descriptor)
      0x22,                         // bDescriptorType    (report)
      43,0x00,                      // CD[25]: wDescriptorLength(L/H) (report descriptor size is 43 bytes)
// Endpoint Descriptor
      0x07,                         // bLength
      0x05,                         // bDescriptorType (Endpoint)
      0x83,                         // bEndpointAddress (EP3-IN)
      0x03,                         // bmAttributes       (interrupt)
      0x40,0x00,            // wMaxPacketSize (64)
      0xFF};                // bInterval (poll every 255 msec)


unsigned char RepD[]=       // Report descriptor
      {
      0x05,                         // bDescriptorType (report)
      0x01,                         // Usage Page (generic desktop)
      0x09,0x06,                    // Usage
      0xA1,0x01,                    // Collection
```

```
      0x05,0x07,                    //   Usage Page 7 (Keyboard/Keypad)
      0x19,0xE0,                    //       Usage Minimum = 224
      0x29,0xE7,                    //   Usage Maximum = 231
      0x15,0x00,                    //   Logical Minimum = 0
      0x25,0x01,                    //   Logical Maximum = 1
      0x75,0x01,                    //       Report Size = 1
      0x95,0x08,                    //   Report Count = 8
      0x81,0x02,                    //  Input(Data,Variable,Absolute) FIRST byte is key modifier
      0x95,0x01,                    //       Report Count = 1
      0x75,0x08,                    //   Report Size = 8
      0x81,0x01,                    //  Input(Constant) SECOND byte is 00
      0x19,0x00,                    //   Usage Minimum = 0
      0x29,0x65,                    //   Usage Maximum = 101
      0x15,0x00,                    //   Logical Minimum = 0,
      0x25,0x65,                    //   Logical Maximum = 101
      0x75,0x08,                    //   Report Size = 8
      0x95,0x01,                    //   Report Count = 1
      0x81,0x00,                    //  Input(Data,Variable,Array) THIRD byte is keystroke
      0xC0};                        // End Collection
      unsigned char STR0[]=  // Language string
      {0x04,                 // bLength
      0x03,                         // bDescriptorType = string
      0x09,0x04};            // wLANGID(L/H)

unsigned char STR1[]=        // Manufacturer ID
      {12,                          // bLength
      0x03,                         // bDescriptorType = string
      'M',0,                 // love that Unicode!
      'a',0,
      'x',0,
      'i',0,
      'm',0};

unsigned char STR2[]=        // Product ID
      {52,                          // bLength
      0x03,                         // bDescriptorType = string
      'M',0,
      'A',0,
      'X',0,
      '3',0,
      '4',0,
      '2',0,
      '0',0,
      'E',0,
      ' ',0,
      'U',0,
      'S',0,
      'B',0,
      ' ',0,
      'P',0,
      'a',0,
      'n',0,
      'i',0,
      'c',0,
      ' ',0,
      'B',0,
      'u',0,
      't',0,
      't',0,
      'o',0,
      'n',0};

unsigned char STR3[]=        // Serial Number
      {24,                          // bLength
      0x03,                         // bDescriptorType = string
      'S',0,
      '/',0,
      'N',0,
      ' ',0,
      '1',0,
      '2',0,
      '3',0,
      '4',0,
      'L',0,
      'T',0,
      'H',0};
```