Dr.Edgar Huckert V09-2019

Mail: edgar.huckert@huckert.com

Reading MIDI input under Windows

I have written many MIDI related programs under Windows and under Linux. When I bought yesterday a new small MIDI keyboard (M-Audio Keystation) I was advised to install a huge GUI based program (size >1 GB). After having installed this large program I had considerable trouble to understand it and to get input from the keyboard.

For normal users like me this GUI based program is a real overkill. I just wanted very basic MIDI input that I could transform later into input for a notation editor (my notation editor also accepts direct MIDI - but there also the handling is strange and complicated). I therefor started to write a basic MIDI input program based on a sample program I found on the internet.

Here is a list of the basic API calls and structures needed under Windows:

- *midiInOpen()*
- *midiInStart()*
- *midiInStop()*
- *midiInGetErrorText()*
- *midiInPrepareHeader()*
- *midiInProc() callback function*
- *MIDIHDR* structure
- *MIDIEVENT* structure

There is no synchronous API call waiting for MIDI input. You always have to write a callback routine that is called when MIDI input arrives. It is not important whether you use a classical MIDI interface (based on a serial protocol, not RS232!) or USB - these calls should work in any case.

My sample program written in C++ (it is rather C than C++) tests the basic MIDI input under Windows. It uses the Windows API functions mentioned above. The output is shown on the PC screen in hexadecimal notation. It you want to transform this output into a real (binary) MIDI file then a small interpreter program is needed that reads this input and generates a binary MIDI file. The most complicated task is converting the **absolute timer values** (the first column in the output) to **relative timer values** as needed in a MIDI file:

Here is a typical output (stored via redirection ">>" in a text file). It shows three MIDI keyboard actions:

- press and release key c1 (lines 1,2)
- press and release key d1 (lines 3,4)
- short action a keyboad wheel (the rest, controller events)

The channel (2nd nibble in 0x90 or 0x80) ist alway zero, i.e. MIDI channel 1 is used.

```
0x00002F69 0x90 0x30 0x49
0x00003034 0x80 0x30 0x48
0x000034F5 0x90 0x32 0x42
0x000035DF 0x80 0x32 0x46
0x00003FED 0xE0 0x00 0x3E
0x0000400C 0xE0 0x00 0x3C
0x0000400C 0xE0 0x00 0x3B
0x0000401B 0xE0 0x00 0x39
0x0000401B 0xE0 0x00 0x38
0x0000402B 0xE0 0x00 0x37
0x0000402B 0xE0 0x00 0x36
0x0000403B 0xE0 0x00 0x35
0x0000403B 0xE0 0x00 0x34
0x0000404A 0xE0 0x00 0x32
0x0000404A 0xE0 0x00 0x31
0x0000404A 0xE0 0x00 0x30
0x0000405A 0xE0 0x00 0x2F
0x0000405A 0xE0 0x00 0x2E
0x00004069 0xE0 0x00 0x2D
0x00004069 0xE0 0x00 0x2B
0x00004069 0xE0 0x00 0x2A
0x00004069 0xE0 0x00 0x29
0x00004069 0xE0 0x00 0x28
0x00004079 0xE0 0x00 0x27
0x00004079 0xE0 0x00 0x26
0x00004089 0xE0 0x00 0x24
```

And here ist the source code. I have compiled and tested it with Digital Mars C++ and GNU C++ (g++). Hints for compilation are in the comments at the start of the program source:

```
// module MidiInput.cpp
// Dr.E.Huckert 03-2007

// This tests the very basic functions for MIDI input under Windows
// No additional library (except winmm.lib) is  needed,
// no GUI here (black command line window)

// Derived from: http://www.borg.com/~jglatt/tech/lowmidi.htm
// 07-2019, retested, embellished, added comments
//          works with an M-Audio Keystation 61 MK3 keyboard, USB interface
//          command line option -v added (switch on logging)

// Compile with Digital Mars under Windows 7
//   dmc -mn MidiInp.cpp winmm.lib
// Compile with GNU g++ under Windows:
//   g++ -o MidiInp.exe MidiInp.cpp -lwinmm

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <mmsystem.h>

#define BUFFERSIZE 200

unsigned char SysXBuffer[256];
int           logging = 0;    // can be changed via option -v
```

```
// ------------------------------------------------------------
// Prints on stdout without line terminators!
void log(const char *buf)
{
  printf("%s", buf);
  fflush(stdout);
}   // end log()

/* A flag to indicate whether I'm currently receiving a SysX message */
unsigned char SysXFlag = 0;

// ------------------------------------------------------------
// callback routine for MIDI input
// see parameter in midiInOpen()
void CALLBACK midiCallback(HMIDIIN handle,
                           UINT uMsg,
                           DWORD dwInstance,
                           DWORD dwParam1,
                           DWORD dwParam2)
{
   LPMIDIHDR      lpMIDIHeader;
   unsigned char *   ptr;
   TCHAR         buffer[80];
   unsigned char    bytes;

   /* Determine why Windows called me */
   switch (uMsg)
   {
      /* Received some regular MIDI message */
      case MIM_DATA:
      {
         DWORD type = dwParam1 &  0x000000FF;
         if (::logging)
           log("received=MIM_DATA\r\n");
         /* Display the time stamp, and the bytes.
            (Note: I always display 3 bytes even for
            Midi messages that have less) */
         sprintf(&buffer[0], "0x%08X 0x%02X 0x%02X 0x%02X\r\n",
                             dwParam2,
                             type,
                             (dwParam1>>8) & 0x000000FF,
                             (dwParam1>>16) & 0x000000FF);
         log(&buffer[0]);
         break;
      }
      /* Received all or part of some System Exclusive message */
      case MIM_LONGDATA:
      {
        if (::logging)
          log("received=MIM_LONGDATA\r\n");
         /* If this application is ready to close down, then don't
midiInAddBuffer() again */
         if (!(SysXFlag & 0x80))
         {
            /*   Assign address of MIDIHDR to a LPMIDIHDR variable. Makes it
easier to access the
                 field that contains the pointer to our block of MIDI events */
            lpMIDIHeader = (LPMIDIHDR)dwParam1;
            /* Get address of the MIDI event that caused this call */
            ptr = (unsigned char *)(lpMIDIHeader->lpData);
            /* Is this the first block of System Exclusive bytes? */
            if (!SysXFlag)
            {
```

```
               /* Print out a noticeable heading as well as the timestamp of the
first block.
                  (But note that other, subsequent blocks will have their own
time stamps). */
               printf("*************** System Exclusive *************\r\n0x%08X
", dwParam2);
               /* Indicate we've begun handling a particular System Exclusive
message */
               SysXFlag |= 0x01;
            }
            /* Is this the last block (ie, the end of System Exclusive byte is
here in the buffer)? */
            if (*(ptr + (lpMIDIHeader->dwBytesRecorded - 1)) == 0xF7)
            {
               /* Indicate we're done handling this particular System Exclusive
message */
               SysXFlag &= (~0x01);
            }
            /* Display the bytes -- 16 per line */
            bytes = 16;
            while((lpMIDIHeader->dwBytesRecorded--))
            {
               if (!(--bytes))
               {
                  sprintf(&buffer[0], "0x%02X\r\n", *(ptr)++);
                  bytes = 16;
               }
               else
                  sprintf(&buffer[0], "0x%02X ", *(ptr)++);
               log(&buffer[0]);
            }
            /* Was this the last block of System Exclusive bytes? */
            if (! SysXFlag)
            {
               /* Print out a noticeable ending */
               log("\r\n*****************************************\r\n");
            }
            /* Queue the MIDIHDR for more input */
            midiInAddBuffer(handle, lpMIDIHeader, sizeof(MIDIHDR));
         }
         break;
      }
      /* Process these messages if you desire */
      case MIM_OPEN:
      {
        if (::logging)
          log("msg received=MIM_OPEN\r\n");
        break;
      }
      case MIM_CLOSE:
      {
        if (::logging)
          log("msg received=MIM_CLOSE\r\n");
        break;
      }
      case MIM_ERROR:
      {
        if (::logging)
          log("msg received=MIM_ERROR\r\n");
        break;
      }
      case MIM_LONGERROR:
      {
        if (::logging)
```

```
          log("msg received=MIM_LONGERROR\r\n");
        break;
      }
      case MIM_MOREDATA:
      {
        if (::logging)
          log("msg received=MIM_MOREDATA\r\n");
        break;
      }
  }
}   // end  midiCallback()

/********************** PrintMidiInErrorMsg() ************************
 * Retrieves and displays an error message for the passed MIDI In error
 * number. It does this using midiInGetErrorText().
 *********************************************************************/
void PrintMidiInErrorMsg(unsigned long err)
{
   char   buffer[BUFFERSIZE];
   if (!(err = midiInGetErrorText(err, &buffer[0], BUFFERSIZE)))
   {
      printf("%s\r\n", &buffer[0]);
   }
   else if (err == MMSYSERR_BADERRNUM)
   {
      printf("Strange error number returned!\r\n");
   }
   else if (err == MMSYSERR_INVALPARAM)
   {
      printf("Specified pointer is invalid!\r\n");
   }
   else
   {
      printf("Unable to allocate/lock memory!\r\n");
   }
}   // end PrintMidiInErrorMsg()

// ----------------------------------------------------------------
void usage()
{
  log("usage: midiinp [-v]\r\n");
}   // end usage()

// ----------------------------------------------------------------
int main(int argc, char *argv[])
{
   HMIDIIN         handle;
   MIDIHDR         midiHdr;
   unsigned long   err;

   usage();

   // evaluated the command line
   for (int n=1; n < argc; n++)
   {
      if (::strcmp(argv[n], "-v") == 0)
        ::logging = 1;

   }

   /* Open default MIDI In device */
   if (!(err = midiInOpen(&handle,
                          0,
                          (DWORD)midiCallback, // implementation see above
```

```
                          0,
                          CALLBACK_FUNCTION)))
   {
      /* Store pointer to our input buffer for System Exclusive messages in
MIDIHDR */
      midiHdr.lpData = (char *)(LPBYTE)&SysXBuffer[0];
      /* Store its size in the MIDIHDR */
      midiHdr.dwBufferLength = sizeof(SysXBuffer);
      /* Flags must be set to 0 */
      midiHdr.dwFlags = 0;
      /* Prepare the buffer and MIDIHDR */
      err = midiInPrepareHeader(handle,
                                &midiHdr,
                                sizeof(MIDIHDR));
      if (!err)
      {
         /* Queue MIDI input buffer */
         err = midiInAddBuffer(handle,
                               &midiHdr,
                               sizeof(MIDIHDR));
         if (!err)
         {
            /* Start recording Midi */
            err = midiInStart(handle);
            if (!err)
            {
               /* Wait for user to abort recording */
               printf("Press any key (on PC keyboard) to stop recording...\r\n\
n");
               _getch();
               /* We need to set a flag to tell our callback midiCallback()
                  not to do any more midiInAddBuffer(), because when we
                  call midiInReset() below, Windows will send a final
                  MIM_LONGDATA message to that callback. If we were to
                  allow midiCallback() to midiInAddBuffer() again, we'd
                  never get the driver to finish with our midiHdr
               */
               SysXFlag |= 0x80;
               printf("\r\nRecording stopped!\n");
            }
            /* Stop recording */
            midiInReset(handle);
         }
      }   // end ! err

      /* If there was an error above, then print a message */
      if (err)
        PrintMidiInErrorMsg(err);
      /* Close the MIDI In device */
      while ((err = midiInClose(handle)) == MIDIERR_STILLPLAYING)
        Sleep(0);
      if (err)
        PrintMidiInErrorMsg(err);
      /* Unprepare the buffer and MIDIHDR. Unpreparing a buffer that has not
been prepared is ok */
       midiInUnprepareHeader(handle,
                             &midiHdr,
                             sizeof(MIDIHDR));
   }
   else
   {
      printf("Error opening the default MIDI In Device!\r\n");
      PrintMidiInErrorMsg(err);
   }
```

```
   return(0);
}   // end main()
```