PortMidi

2.2.x

Generated by Doxygen 1.7.6.1

Sun Feb 19 2012 12:46:02

Contents

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

Module Documentation

3.1 Input/Output Devices Handling

Functions

- PMEXPORT const PmDeviceInfo * Pm_GetDeviceInfo (PmDeviceID id)

  Pm_GetDeviceInfo() returns a pointer to a PmDeviceInfo structure referring to the device specified by id.

- PMEXPORT PmError Pm_OpenInput (PortMidiStream **stream, PmDeviceID inputDevice, void *inputDriverInfo, int32_t bufferSize, PmTimeProcPtr time_proc, void *time_info)

  Pm_OpenInput() and Pm_OpenOutput() open devices.

- PMEXPORT PmError **Pm_OpenOutput** (PortMidiStream **stream, PmDeviceID outputDevice, void *outputDriverInfo, int32_t bufferSize, PmTimeProcPtr time_proc, void *time_info, int32_t latency)

3.1.1 Function Documentation

3.1.1.1 PMEXPORT const PmDeviceInfo* Pm_GetDeviceInfo (PmDeviceID *id*)

Pm_GetDeviceInfo() returns a pointer to a PmDeviceInfo structure referring to the device specified by id.

If id is out of range the function returns NULL.

The returned structure is owned by the PortMidi implementation and must not be manipulated or freed. The pointer is guaranteed to be valid between calls to Pm_Initialize() and Pm_Terminate().

Definition at line 183 of file portmidi.c.

3.1.1.2 PMEXPORT PmError Pm_OpenInput (PortMidiStream ** *stream,* PmDeviceID *inputDevice,* void * *inputDriverInfo,* int32_t *bufferSize,* PmTimeProcPtr *time_proc,* void * *time_info*)

Pm_OpenInput() and Pm_OpenOutput() open devices.

stream is the address of a PortMidiStream pointer which will receive a pointer to the newly opened stream.

inputDevice is the id of the device used for input (see PmDeviceID above).

inputDriverInfo is a pointer to an optional driver specific data structure containing additional information for device setup or handle processing. inputDriverInfo is never required for correct operation. If not used inputDriverInfo should be NULL.

outputDevice is the id of the device used for output (see PmDeviceID above.)

outputDriverInfo is a pointer to an optional driver specific data structure containing additional information for device setup or handle processing. outputDriverInfo is never required for correct operation. If not used outputDriverInfo should be NULL.

For input, the buffersize specifies the number of input events to be buffered waiting to be read using Pm_Read(). For output, buffersize specifies the number of output events to be buffered waiting for output. (In some cases -- see below -- PortMidi does not buffer output at all and merely passes data to a lower-level API, in which case buffersize is ignored.)

latency is the delay in milliseconds applied to timestamps to determine when the output should actually occur. (If latency is < 0, 0 is assumed.) If latency is zero, timestamps are ignored and all output is delivered immediately. If latency is greater than zero, output is delayed until the message timestamp plus the latency. (NOTE: the time is measured relative to the time source indicated by time_proc. Timestamps are absolute, not relative delays or offsets.) In some cases, PortMidi can obtain better timing than your application by passing timestamps along to the device driver or hardware. Latency may also help you to synchronize midi data to audio data by matching midi latency to the audio buffer latency.

time_proc is a pointer to a procedure that returns time in milliseconds. It may be NULL, in which case a default millisecond timebase (PortTime) is used. If the application wants to use PortTime, it should start the timer (call Pt_Start) before calling Pm_OpenInput or Pm_OpenOutput. If the application tries to start the timer *after* Pm_OpenInput or Pm_OpenOutput, it may get a ptAlreadyStarted error from Pt_Start, and the application's preferred time resolution and callback function will be ignored. time_proc result values are appended to incoming MIDI data, and time_proc times are used to schedule outgoing MIDI data (when latency is non-zero).

time_info is a pointer passed to time_proc.

Example: If I provide a timestamp of 5000, latency is 1, and time_proc returns 4990, then the desired output time will be when time_proc returns timestamp+latency = 5001. This will be 5001-4990 = 11ms from now.

return value: Upon success Pm_Open() returns PmNoError and places a pointer to a valid PortMidiStream in the stream argument. If a call to Pm_Open() fails a nonzero error code is returned (see PMError above) and the value of port is invalid.

Any stream that is successfully opened should eventually be closed by calling Pm_-Close().

Definition at line 669 of file portmidi.c.

References PM_FILT_ACTIVE.

3.2 Events and Filters Handling

Data Structures

- struct PmEvent

  All midi data comes in the form of PmEvent structures.

Defines

- #define PM_FILT_ACTIVE (1 << 0x0E)

  filter active sensing messages (0xFE):
- #define PM_FILT_SYSEX (1 << 0x00)

  filter system exclusive messages (0xF0):
- #define PM_FILT_CLOCK (1 << 0x08)

  filter MIDI clock message (0xF8)
- #define PM_FILT_PLAY ((1 << 0x0A) | (1 << 0x0C) | (1 << 0x0B))

  filter play messages (start 0xFA, stop 0xFC, continue 0xFB)
- #define PM_FILT_TICK (1 << 0x09)

  filter tick messages (0xF9)
- #define PM_FILT_FD (1 << 0x0D)

  filter undefined FD messages
- #define PM_FILT_UNDEFINED PM_FILT_FD

  filter undefined real-time messages
- #define PM_FILT_RESET (1 << 0x0F)

  filter reset messages (0xFF)
- #define PM_FILT_REALTIME

  filter all real-time messages
- #define PM_FILT_NOTE ((1 << 0x19) | (1 << 0x18))

  filter note-on and note-off (0x90-0x9F and 0x80-0x8F
- #define PM_FILT_CHANNEL_AFTERTOUCH (1 << 0x1D)

  filter channel aftertouch (most midi controllers use this) (0xD0-0xDF)
- #define PM_FILT_POLY_AFTERTOUCH (1 << 0x1A)

  per-note aftertouch (0xA0-0xAF)
- #define PM_FILT_AFTERTOUCH (PM_FILT_CHANNEL_AFTERTOUCH | PM_FILT_POLY_AFTERTOUCH)

  filter both channel and poly aftertouch
- #define PM_FILT_PROGRAM (1 << 0x1C)

  Program changes (0xC0-0xCF)
- #define PM_FILT_CONTROL (1 << 0x1B)

  Control Changes (CC's) (0xB0-0xBF)
- #define PM_FILT_PITCHBEND (1 << 0x1E)

  Pitch Bender (0xE0-0xEF.
- #define PM_FILT_MTC (1 << 0x01)

MIDI Time Code (0xF1)

- #define PM_FILT_SONG_POSITION (1 << 0x02)

  Song Position (0xF2)
- #define PM_FILT_SONG_SELECT (1 << 0x03)

  Song Select (0xF3)
- #define PM_FILT_TUNE (1 << 0x06)

  Tuning request (0xF6)
- #define PM_FILT_SYSTEMCOMMON (PM_FILT_MTC | PM_FILT_SONG_POSITION | PM_FILT_SONG_SELECT | PM_FILT_TUNE)

  All System Common messages (mtc, song position, song select, tune request)
- #define **Pm_Channel**(channel) (1<<(channel))
- #define Pm_Message(status, data1, data2)

  Pm_Message() encodes a short Midi message into a 32-bit word.
- #define **Pm_MessageStatus**(msg) ((msg) & 0xFF)
- #define **Pm_MessageData1**(msg) (((msg) >> 8) & 0xFF)
- #define **Pm_MessageData2**(msg) (((msg) >> 16) & 0xFF)

Typedefs

- typedef int32_t PmMessage

  see PmEvent

Functions

- PMEXPORT PmError **Pm_SetFilter** (PortMidiStream *stream, int32_t filters)
- PMEXPORT PmError Pm_SetChannelMask (PortMidiStream *stream, int mask)

  Pm_SetChannelMask() filters incoming messages based on channel.
- PMEXPORT PmError Pm_Abort (PortMidiStream *stream)

  Pm_Abort() terminates outgoing messages immediately The caller should immediately close the output port; this call may result in transmission of a partial midi message.
- PMEXPORT PmError Pm_Close (PortMidiStream *stream)

  Pm_Close() closes a midi stream, flushing any pending buffers.
- PmError Pm_Synchronize (PortMidiStream *stream)

  Pm_Synchronize() instructs PortMidi to (re)synchronize to the time_proc passed when the stream was opened.

3.2.1 Define Documentation

3.2.1.1 #define PM_FILT_REALTIME

Value:

```
(PM_FILT_ACTIVE | PM_FILT_SYSEX | PM_FILT_CLOCK | \
    PM_FILT_PLAY | PM_FILT_UNDEFINED | PM_FILT_RESET | PM_FILT_TICK)
```

filter all real-time messages

Definition at line 405 of file portmidi.h.

3.2.1.2 #define Pm_Message(*status, data1, data2*)

Value:

```
((((data2) << 16) & 0xFF0000) | \
          (((data1) << 8) & 0xFF00) | \
          ((status) & 0xFF))
```

Pm_Message() encodes a short Midi message into a 32-bit word.

If data1 and/or data2 are not present, use zero.

Pm_MessageStatus(), Pm_MessageData1(), and Pm_MessageData2() extract fields from a 32-bit midi message.

Definition at line 503 of file portmidi.h.

3.2.2 Function Documentation

3.2.2.1 PMEXPORT PmError Pm_Abort (PortMidiStream * *stream*)

Pm_Abort() terminates outgoing messages immediately The caller should immediately close the output port; this call may result in transmission of a partial midi message.

There is no abort for Midi input because the user can simply ignore messages in the buffer and close an input device at any time.

Definition at line 906 of file portmidi.c.

3.2.2.2 PMEXPORT PmError Pm_Close (PortMidiStream * *stream*)

Pm_Close() closes a midi stream, flushing any pending buffers.

(PortMidi attempts to close open streams when the application exits -- this is particularly difficult under Windows.)

Definition at line 860 of file portmidi.c.

3.2.2.3 PMEXPORT PmError Pm_SetChannelMask (PortMidiStream * *stream,* int *mask*)

Pm_SetChannelMask() filters incoming messages based on channel.

The mask is a 16-bit bitfield corresponding to appropriate channels. The Pm_Channel macro can assist in calling this function. i.e. to set receive only input on channel 1, call with Pm_SetChannelMask(Pm_Channel(1)); Multiple channels should be OR'd together, like Pm_SetChannelMask(Pm_Channel(10) | Pm_Channel(11))

Note that channels are numbered 0 to 15 (not 1 to 16). Most synthesizer and interfaces number channels starting at 1, but PortMidi numbers channels starting at 0.

All channels are allowed by default

Definition at line 831 of file portmidi.c.

3.2.2.4 PmError Pm_Synchronize (PortMidiStream * *stream*)

Pm_Synchronize() instructs PortMidi to (re)synchronize to the time_proc passed when the stream was opened.

Typically, this is used when the stream must be opened before the time_proc reference is actually advancing. In this case, message timing may be erratic, but since timestamps of zero mean "send immediately," initialization messages with zero timestamps can be written without a functioning time reference and without problems. Before the first MIDI message with a non-zero timestamp is written to the stream, the time reference must begin to advance (for example, if the time_proc computes time based on audio samples, time might begin to advance when an audio stream becomes active). After time_proc return values become valid, and BEFORE writing the first non-zero timestamped MIDI message, call Pm_Synchronize() so that PortMidi can observe the difference between the current time_proc value and its MIDI stream time.

In the more normal case where time_proc values advance continuously, there is no need to call Pm_Synchronize. PortMidi will always synchronize at the first output message and periodically thereafter.

Definition at line 892 of file portmidi.c.

3.3 Reading and Writing Midi Messages

Functions

- PMEXPORT int Pm_Read (PortMidiStream *stream, PmEvent *buffer, int32_t length)

  Pm_Read() retrieves midi data into a buffer, and returns the number of events read.
- PMEXPORT PmError Pm_Poll (PortMidiStream *stream)

  Pm_Poll() tests whether input is available, returning TRUE, FALSE, or an error value.
- PMEXPORT PmError Pm_Write (PortMidiStream *stream, PmEvent *buffer, int32_t length)

  Pm_Write() writes midi data from a buffer.
- PMEXPORT PmError Pm_WriteShort (PortMidiStream *stream, PmTimestamp when, int32_t msg)

  Pm_WriteShort() writes a timestamped non-system-exclusive midi message.
- PMEXPORT PmError Pm_WriteSysEx (PortMidiStream *stream, PmTimestamp when, unsigned char *msg)

  Pm_WriteSysEx() writes a timestamped system-exclusive midi message.

3.3.1 Function Documentation

3.3.1.1 PMEXPORT int Pm_Read (PortMidiStream * *stream*, PmEvent * *buffer*, int32_t *length*)

Pm_Read() retrieves midi data into a buffer, and returns the number of events read.

Result is a non-negative number unless an error occurs, in which case a PmError value will be returned.

Buffer Overflow

The problem: if an input overflow occurs, data will be lost, ultimately because there is no flow control all the way back to the data source. When data is lost, the receiver should be notified and some sort of graceful recovery should take place, e.g. you shouldn't resume receiving in the middle of a long sysex message.

With a lock-free fifo, which is pretty much what we're stuck with to enable portability to the Mac, it's tricky for the producer and consumer to synchronously reset the buffer and resume normal operation.

Solution: the buffer managed by PortMidi will be flushed when an overflow occurs. - The consumer (Pm_Read()) gets an error message (pmBufferOverflow) and ordinary processing resumes as soon as a new message arrives. The remainder of a partial sysex message is not considered to be a "new message" and will be flushed as well.

Definition at line 357 of file portmidi.c.

3.3.1.2 PMEXPORT PmError Pm_Write (PortMidiStream * *stream,* PmEvent * *buffer,* int32_t *length*)

Pm_Write() writes midi data from a buffer.

This may contain:

- short messages or
- sysex messages that are converted into a sequence of PmEvent structures, e.g. sending data from a file or forwarding them from midi input.

Use Pm_WriteSysEx() to write a sysex message stored as a contiguous array of bytes.

Sysex data may contain embedded real-time messages.

Definition at line 448 of file portmidi.c.

Referenced by Pm_WriteShort(), and Pm_WriteSysEx().

3.3.1.3 PMEXPORT PmError Pm_WriteShort (PortMidiStream * *stream,* PmTimestamp *when,* int32_t *msg*)

Pm_WriteShort() writes a timestamped non-system-exclusive midi message.

Messages are delivered in order as received, and timestamps must be non-decreasing. (But timestamps are ignored if the stream was opened with latency = 0.)

Definition at line 581 of file portmidi.c.

References Pm_Write().

Chapter 4

Data Structure Documentation

4.1 PmDeviceInfo Struct Reference

Data Fields

- int structVersion

  this internal structure version
- const char * interf

  underlying MIDI API, e.g.
- const char * name

  device name, e.g.
- int input

  true iff input is available
- int output

  true iff output is available
- int opened

  used by generic PortMidi code to do error checking on arguments

4.1.1 Detailed Description

Definition at line 206 of file portmidi.h.

4.1.2 Field Documentation

4.1.2.1 const char* PmDeviceInfo::interf

underlying MIDI API, e.g.

MMSystem or DirectX

Definition at line 208 of file portmidi.h.

4.1.2.2 const char* PmDeviceInfo::name

device name, e.g.

USB MidiSport 1x1

Definition at line 209 of file portmidi.h.

The documentation for this struct was generated from the following file:

- portmidi.h

4.2 PmEvent Struct Reference

All midi data comes in the form of PmEvent structures.

`#include <portmidi.h>`

Data Fields

- PmMessage **message**
- PmTimestamp **timestamp**

4.2.1 Detailed Description

All midi data comes in the form of PmEvent structures.

A sysex message is encoded as a sequence of PmEvent structures, with each structure carrying 4 bytes of the message, i.e. only the first PmEvent carries the status byte.

Note that MIDI allows nested messages: the so-called "real-time" MIDI messages can be inserted into the MIDI byte stream at any location, including within a sysex message. MIDI real-time messages are one-byte messages used mainly for timing (see the MIDI spec). PortMidi retains the order of non-real-time MIDI messages on both input and output, but it does not specify exactly how real-time messages are processed. This is particulary problematic for MIDI input, because the input parser must either prepare to buffer an unlimited number of sysex message bytes or to buffer an unlimited number of real-time messages that arrive embedded in a long sysex message. To simplify things, the input parser is allowed to pass real-time MIDI messages embedded within a sysex message, and it is up to the client to detect, process, and remove these messages as they arrive.

When receiving sysex messages, the sysex message is terminated by either an EOX status byte (anywhere in the 4 byte messages) or by a non-real-time status byte in the low order byte of the message. If you get a non-real-time status byte but there was no EOX byte, it means the sysex message was somehow truncated. This is not considered an error; e.g., a missing EOX can result from the user disconnecting a MIDI cable during sysex transmission.

A real-time message can occur within a sysex message. A real-time message will always occupy a full PmEvent with the status byte in the low-order byte of the PmEvent

message field. (This implies that the byte-order of sysex bytes and real-time message bytes may not be preserved -- for example, if a real-time message arrives after 3 bytes of a sysex message, the real-time message will be delivered first. The first word of the sysex message will be delivered only after the 4th byte arrives, filling the 4-byte PmEvent message field.

The timestamp field is observed when the output port is opened with a non-zero latency. A timestamp of zero means "use the current time", which in turn means to deliver the message with a delay of latency (the latency parameter used when opening the output port.) Do not expect PortMidi to sort data according to timestamps -- messages should be sent in the correct order, and timestamps MUST be non-decreasing. See also "-Example" for Pm_OpenOutput() above.

A sysex message will generally fill many PmEvent structures. On output to a PortMidi-Stream with non-zero latency, the first timestamp on sysex message data will determine the time to begin sending the message. PortMidi implementations may ignore timestamps for the remainder of the sysex message.

On input, the timestamp ideally denotes the arrival time of the status byte of the message. The first timestamp on sysex message data will be valid. Subsequent timestamps may denote when message bytes were actually received, or they may be simply copies of the first timestamp.

Timestamps for nested messages: If a real-time message arrives in the middle of some other message, it is enqueued immediately with the timestamp corresponding to its arrival time. The interrupted non-real-time message or 4-byte packet of sysex data will be enqueued later. The timestamp of interrupted data will be equal to that of the interrupting real-time message to insure that timestamps are non-decreasing.

Definition at line 577 of file portmidi.h.

The documentation for this struct was generated from the following file:

- portmidi.h