Wed Jun 10 13:43:55 PDT 2009
Craig Stuart Sapp <craig@ccrma.stanford.edu>
OS X CoreMIDI programming examples
==

```
testout.c       -- demonstration of how to output a MIDI message.
testout2.c      -- demonstration of how to send future MIDI output
                   data via a CoreMIDI buffer using time stamps.
testin.c        -- demonstration of how to read MIDI input data.
testin2.c       -- examination of the time stamps coming with MIDI input.
midimirror.c    -- example of how to do MIDI input and output in same program.
```

Links to primary documentation used to create the example programs:

MIDIServices.h Documentation:
   http://developer.apple.com/documentation/MusicAudio/Reference/CACoreMIDIRef/MIDIServices

CoreMIDI Documentation:
   https://developer.apple.com/documentation/MusicAudio/Reference/CACoreMIDIRef/MIDIServices

Nanosecond Timing in OS X used in MIDITimeStamp:
   http://developer.apple.com/qa/qa2004/qa1398.html

```
//
// Programmer:  Craig Stuart Sapp
// Date:        Mon Jun  8 14:54:42 PDT 2009
// Filename:    testout.c
// Syntax:      C; Apple OSX CoreMIDI
// $Smake:      gcc -o %b %f -framework CoreMIDI -framework CoreServices
//              note: CoreServices needed for GetMacOSSStatusErrorString().
//
// Description: This program plays a single MIDI note (middle C) on all
//              MIDI output ports which the program can find.
//
// Derived from "Audio and MIDI on Mac OS X" Preliminary Documentation,
// May 2001 Apple Computer, Inc. found in PDF form on the developer.apple.com
// website, as well as using links at the bottom of the file.
//

#include <CoreMIDI/CoreMIDI.h>    /* interface to MIDI in Macintosh OS X */
#include <unistd.h>               /* for sleep() function                */
#define MESSAGESIZE 3             /* byte count for MIDI note messages   */

void playPacketListOnAllDevices   (MIDIPortRef     midiout,
                                   const MIDIPacketList* pktlist);

////////////////////////////////////////////////////////////////////////

int main(void) {

   // Prepare MIDI Interface Client/Port for writing MIDI data:
   MIDIClientRef midiclient  = NULL;
   MIDIPortRef   midiout     = NULL;
   OSStatus status;
   if (status = MIDIClientCreate(CFSTR("TeStInG"), NULL, NULL, &midiclient)) {
       printf("Error trying to create MIDI Client structure: %d\n", status);
       printf("%s\n", GetMacOSStatusErrorString(status));
       exit(status);
   }
   if (status = MIDIOutputPortCreate(midiclient, CFSTR("OuTpUt"), &midiout)) {
       printf("Error trying to create MIDI output port: %d\n", status);
       printf("%s\n", GetMacOSStatusErrorString(status));
       exit(status);
   }

   // Prepare a MIDI Note-On message to send
   MIDITimeStamp timestamp = 0;   // 0 will mean play now.
   Byte buffer[1024];             // storage space for MIDI Packets (max 65536)
   MIDIPacketList *packetlist = (MIDIPacketList*)buffer;
   MIDIPacket *currentpacket = MIDIPacketListInit(packetlist);
   Byte noteon[MESSAGESIZE] = {0x90, 60, 90};
   currentpacket = MIDIPacketListAdd(packetlist, sizeof(buffer),
         currentpacket, timestamp, MESSAGESIZE, noteon);

   // send the MIDI data and wait for one second:
   playPacketListOnAllDevices(midiout, packetlist);
   sleep(1);

   // Prepare a MIDI Note-Off message to send
   currentpacket = MIDIPacketListInit(packetlist);
   Byte noteoff[MESSAGESIZE] = {0x90, 60, 0};
   currentpacket = MIDIPacketListAdd(packetlist, sizeof(buffer),
         currentpacket, timestamp, MESSAGESIZE, noteoff);

   // send the MIDI data and exit immediately
   playPacketListOnAllDevices(midiout, packetlist);
```

```
   if (status = MIDIPortDispose(midiout)) {
      printf("Error trying to close MIDI output port: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   midiout = NULL;

   if (status = MIDIClientDispose(midiclient)) {
      printf("Error trying to close MIDI client: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   midiclient = NULL;

   return 0;
}


////////////////////////////////////////////////////////////////////////

//////////////////////////////
//
// playPacketOnAllDevices -- play the list of MIDI packets
//    on all MIDI output devices which the computer knows about.
//    (Send the MIDI message(s) to all MIDI out ports).
//

void playPacketListOnAllDevices(MIDIPortRef midiout,
      const MIDIPacketList* pktlist) {
   // send MIDI message to all MIDI output devices connected to computer:
   ItemCount nDests = MIDIGetNumberOfDestinations();
   ItemCount iDest;
   OSStatus status;
   MIDIEndpointRef dest;
   for(iDest=0; iDest<nDests; iDest++) {
      dest = MIDIGetDestination(iDest);
      if (status = MIDISend(midiout, dest, pktlist)) {
          printf("Problem sendint MIDI data on port %d\n", iDest);
          printf("%s\n", GetMacOSStatusErrorString(status));
          exit(status);
      }
   }
}


////////////////////////////////////////////////////////////////////////
//
// NOTES
//
/*

Crapple functions used in this program:

/// TYPEDEFS ///////////////////////////////////////////////////////////

UInt16    => unsigned short int
UInt32    => unsigned long int
UInt64    => unsigned long long int
Byte      => char
ByteCount => int
OSStatus  => int
Boolean   => char

MIDITimeStamp => UInt64
```

```
   A host clock time representing the time of an event, as returned by
   mach_absolute_time() or UpTime().  Since MIDI applications will tend to
   do a fair amount of math with the times of events, it is more
   convenient to use a UInt64 than an AbsoluteTime.  See CoreAudio/HostTime.h

struct MIDIPacket { MIDITimeStamp timeStamp; UInt16 length; Byte data[256]; };
      timeStamp = The time at which the events occurred (if receiving MIDI),
                  or the time at which the events are to be played (if sending
                  MIDI).  Zero means "now" when sending MIDI data.  The time
                  stamp applies to the first MIDI byte in the packet.
      length    = The number of valid MIDI bytes which follow in data[].
                  It may be larger than 256 bytes if the packet is dynamically
                  allocated.
      data      = A variable-length stream of MIDI messages. Running status
                  is not allowed.  In the case of system-exclusive messages,
                  a packet may only contain a single message, or portion
                  of one, with no other MIDI events.  The MIDI messages in
                  the packet must always be complete, except for
                  system-exclusive messages.  data[] is declared to be 256
                  bytes in length so clients don't have to create custom data
                  structures in simple situations.

struct MIDIPacketList { UInt32  numPackets; MIDIPacket packet[1]; };
      numPackets = The number of MIDIPackets in the list.
      packet     = An open-ended array of variable-length MIDIPackets.
   The timestamps in the packet list must be in ascending order.
   Note that the packets in the list, while defined as an array, may
   *not* be accessed as an array, since they are vairable-length.  To
   iterate through the packets in a packet list, use a loop such as:
      MIDIPacket *packet = &packetList->packet[0];
      for (int i=0; i<packetList->numPackets; i++) {
         ...
         packet = MIDIPacketNext(packet);
      }

struct MIDISysexSendRequest {
   MIDIEndpointRef destination;
   const Byte     *data;
   UInt32          bytesToSend;
   Boolean         complete;
   Byte            reserved[3];  // to fill up 4-byte boundary, I suppose.
   MIDICompetionProc completionProc;
   void *completionRefCon;
};
      destination      = The endpoint to which the event is to be sent.
      data             = Initially, a pointer to the sys-ex event to be
                         sent.  MIDISendSysex will advance this pointer
                         as bytes are sent.
      bytesToSend      = Initially, the number of bytes to be sent.
                         MIDISendSysex will decrement this counter
                         as bytes are sent.
      complete         = The client may set this to true at any time
                         to abort transmission.  The implementation
                         sets this to true when all bytes have been sent.
      completionProc   = Called when all bytes ahve been sent, or after the
                         client has set complete to true.
      completionRefCon = Passed as a refCon to completionProc.
   This data structure represents a request to send a single system-exclusive
   MIDI event to a MIDI destination asynchronously.


/// MIDI OPENING/CLOSING FUNCTIONS ////////////////////////////////////

OSStatus MIDIClientCreate(CFStringRef name, MIDINotifyProc notifyProc,
```

```
                            void* notifyRefCon, MIDIClientRef* outClient);
      name         = The client's name.
      notifyProc   = An optional (may be NULL) callback function through
                     which the client will receive notifications of changes
                     to the system.
      notifyRefCon = An optional (may be NULL) refCon passed back to
                     notifyRefCon.
      outClient    = On successful return, points to the newly-created
                     MIDIClientRef.
   All clients must be created and disposed on the same thread.
   Note that notifyProc will always be called on the run loop
   which was current when MIDIClientCreate was first called.


OSStatus MIDIOutputPortCreate(MidiClientRef midiclient, CFStringRef portName,
                              MIDIPortRef *outPort);
      client   = The client to own the newly-created port.
      portName = The name of the port.
      outPort  = On successful return, points to the newly-created MIDIPort.
   Creates an output port through which the client may send outgoing
   MIDI messages to any MIDI destination.  Output ports provied a
   mechanism for MIDI merging.  CoreMIDI assumes that each output port
   will be responsible for sending only a single MIDI stream to each
   destination, although a single port may address all of the destinations
   in the system.  Multiple output ports are only necessary when an
   application is capable of directing multiple simultaneous MIDI streams
   to the same destination.


OSStatus MIDIPortDispose(MIDIPortRef port);
      port = The port to dispose.
   It is not usually necessary to call this function.  When an application's
   MIDIClient's are automatically disposed at termination, or explicitly,
   via MIDIClientDispose, the client's ports are automatically disposed at
   that time.

OSS MIDIClientDispose(MIDIClientRef client);
      client = The client to dispose
   Not an essential function to call; CoreMIDI framework will
   automatically dispose all MIDIClients when an application terminates.


/// MIDI PACKET FUNCTIONS ////////////////////////////////////////////

MIDIPacket* MIDIPacketListInit(MIDIPacketList* packetList);
      packetList = The packet list to be initialized.
   Returns a pointer to the first MIDIPacket in the packet list.

MIDIPacket* MIDIPacketListAdd(MIDIPacketList* packetList, ByteCount listSize,
                                  MIDIPacket* curPacket, MIDITimeStamp time,
                                  ByteCount nData, const Byte* data);
      packetList = The packet list to which the event is to be added.
      listSize   = The size, in bytes, of the packet list.
      curPacket  = A packet pointer returned by a previous call to
                   MIDIPacketListInit() or MIDIPacketListAdd() for this
                   packet list.
      time       = The new event's time (when to play the MIDI event
                   when this is output data).
      nData      = The length of the new event, in bytes.
      data       = The new event.  May be a single MIDI event, or a
                   partial sys-ex event.  Running status is *not*
                   permitted.
   Returns null if there was not room in the packet for the event; otherwise,
   returns a packet point which should be passed as CurPacket in a
```

```
   subsequent call to this function.  The maximum size of a packet list is
   65536 bytes.  Large sysex messages must be sent in smaller packet
   lists.



/// MIDI SEND FUNCTIONS ////////////////////////////////////////////////

OSStatus MIDISend(MIDIPortRef port, MIDIEndpointRef dest,
                                            const MIDIPacketList *packetList);
      port       = The output port through which the MIDI is to be sent.
      dest       = The destination to receive the events.
      packetList = The MIDI events to be sent.
   Events with future timestamps are scheduled for future delivery.
   CoreMIDI performs and needed MIDI merging.



/// OTHER USEFUL FUNCTIONS /////////////////////////////////////////////

OSStatus MIDIRestart(void);
   This function forces CoreMIDI to ask its drivers to rescan for hardware.
   (OSX10.1 and later).

OSStatus MIDISendSysex(MIDISysexSendRequest* request);
      request = contains the destination, and a pointer to the MIDI data
                to be sent.
   request->data must point to a single complete or partial MIDI
   system-exclusive message.


/// LINKS AND MISC. NOTES //////////////////////////////////////////////

MIDIServices.h Documentation:
   http://developer.apple.com/DOCUMENTATION/MusicAudio/Reference/CACoreMIDIRef/MIDIServ
ices/CompositePage.html

CoreMIDI Documentation:
   https://developer.apple.com/documentation/MusicAudio/Reference/CACoreMIDIRef/MIDISer
vices/index.html

OSStatus Information:
   https://developer.apple.com/documentation/Carbon/Reference/ErrorHandler/Reference/re
ference.html#//apple_ref/doc/uid/TP40000867-CH201-DontLinkElementID_2

// Example MIDI I/O program: http://bl0rg.net/~manuel/midi-merge.c
// MIDIOutputPortCreate: http://xmidi.com/docs/coremidi34.html
// MIDI echo example: http://www.allegro.cc/forums/thread/598206
// MidiPacket *MIDIPacketNext(MidiPacket *pkt);
// MidiPacket *MIDIPacketListInit(MidiPacketList *pktlist);
// MIDIPacket *MIDIPacketListAdd(MIDIPacketList *pktlist, ByteCount
//    listSize, MidiPacket *curPacket, MIDITimeStamp time, ByteCount nData,
//    Byte *data);
// MIDISend(MIDIPortRef port, MIDIEndpointRef dest, const MIDIPacketList*pktlist);
// OSStatus string: http://lists.apple.com/archives/QuickTime-API/2006/Oct/msg00092.htm
l


*/
```

```
//
// Programmer:  Craig Stuart Sapp
// Date:        Mon Jun  8 14:54:42 PDT 2009
// Filename:    testout.c
// Syntax:      C; Apple OSX CoreMIDI
// $Smake:      gcc -o %b %f -framework CoreMIDI -framework CoreServices
//              note: CoreServices needed for GetMacOSSStatusErrorString().
//
// Description: This program plays two MIDI notes (middle C, and C#) on all
//              MIDI output ports which the program can find. Similar to
//              testout.c, but writes all MIDI messages to CoreMIDI at
//              the same time and relies on time stamps for the timing
//              of the notes rather than sleep().
//
// Derived from "Audio and MIDI on Mac OS X" Preliminary Documentation,
// May 2001 Apple Computer, Inc. found in PDF form on the developer.apple.com
// website, as well as using links at the bottom of the file.
//

#include <CoreMIDI/CoreMIDI.h>    /* interface to MIDI in Macintosh OS X */
#include <unistd.h>               /* for sleep() function                */
#include <mach/mach_time.h>       /* for mach_absolute_time()            */
#define MESSAGESIZE 3             /* byte count for MIDI note messages   */

void playPacketListOnAllDevices   (MIDIPortRef     midiout,
                                   const MIDIPacketList* pktlist);

////////////////////////////////////////////////////////////////////////

int main(void) {

   // Prepare MIDI Interface Client/Port for writing MIDI data:
   MIDIClientRef midiclient  = NULL;
   MIDIPortRef   midiout     = NULL;
   OSStatus status;
   if (status = MIDIClientCreate(CFSTR("TeStInG"), NULL, NULL, &midiclient)) {
       printf("Error trying to create MIDI Client structure: %d\n", status);
       printf("%s\n", GetMacOSStatusErrorString(status));
       exit(status);
   }
   if (status = MIDIOutputPortCreate(midiclient, CFSTR("OuTpUt"), &midiout)) {
       printf("Error trying to create MIDI output port: %d\n", status);
       printf("%s\n", GetMacOSStatusErrorString(status));
       exit(status);
   }

   // Prepare a MIDI Note-On message to send
   MIDITimeStamp timestamp = mach_absolute_time();
   Byte buffer[1024];            // storage space for MIDI Packets (max 65536)
   MIDIPacketList *packetlist = (MIDIPacketList*)buffer;
   MIDIPacket *currentpacket = MIDIPacketListInit(packetlist);
   Byte notemessage[MESSAGESIZE] = {0x90, 60, 90};
   currentpacket = MIDIPacketListAdd(packetlist, sizeof(buffer),
         currentpacket, timestamp, MESSAGESIZE, notemessage);

   // setup another note to play one second later with same loudness
   notemessage[1] = 61;            // pitch = C#4
   timestamp += 1000000000;        // one billion nanoseconds later
   currentpacket = MIDIPacketListAdd(packetlist, sizeof(buffer),
          currentpacket, timestamp, MESSAGESIZE, notemessage);

   // turn off the second note played one second later
   notemessage[1] = 61;            // pitch = C#4
   notemessage[2] = 0;             // turn off the note
```

```
   timestamp += 1000000000;        // one billion nanoseconds later
   currentpacket = MIDIPacketListAdd(packetlist, sizeof(buffer),
          currentpacket, timestamp, MESSAGESIZE, notemessage);

   // turn off the first note played one second later
   notemessage[1] = 60;            // pitch = C4
   notemessage[2] = 0;             // turn off the note
   timestamp += 1000000000;        // one billion nanoseconds later
   currentpacket = MIDIPacketListAdd(packetlist, sizeof(buffer),
          currentpacket, timestamp, MESSAGESIZE, notemessage);

   // send the MIDI data and don't wait around for a little while
   // to see what happens.
   playPacketListOnAllDevices(midiout, packetlist);
   sleep(5);

   if (status = MIDIPortDispose(midiout)) {
      printf("Error trying to close MIDI output port: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   midiout = NULL;

   if (status = MIDIClientDispose(midiclient)) {
      printf("Error trying to close MIDI client: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   midiclient = NULL;

   return 0;
}


////////////////////////////////////////////////////////////////////////

//////////////////////////////
//
// playPacketOnAllDevices -- play the list of MIDI packets
//    on all MIDI output devices which the computer knows about.
//    (Send the MIDI message(s) to all MIDI out ports).
//

void playPacketListOnAllDevices(MIDIPortRef midiout,
      const MIDIPacketList* pktlist) {
   // send MIDI message to all MIDI output devices connected to computer:
   ItemCount nDests = MIDIGetNumberOfDestinations();
   ItemCount iDest;
   OSStatus status;
   MIDIEndpointRef dest;
   for(iDest=0; iDest<nDests; iDest++) {
      dest = MIDIGetDestination(iDest);
      if (status = MIDISend(midiout, dest, pktlist)) {
          printf("Problem sendint MIDI data on port %d\n", iDest);
          printf("%s\n", GetMacOSStatusErrorString(status));
          exit(status);
      }
   }
}
```

```
//
// Programmer:  Craig Stuart Sapp
// Date:        Tue Jun  9 16:00:00 PDT 2009
// Filename:    testin.c
// Syntax:      C; Apple OSX CoreMIDI
// $Smake:      gcc -o %b %f -framework CoreMIDI -framework CoreServices
//              note: CoreServices needed for GetMacOSSStatusErrorString().
//
// Description: This program reads in some MIDI data and the time stamps
//              which are attached to the data.
//
// Derived from "Audio and MIDI on Mac OS X" Preliminary Documentation,
// May 2001 Apple Computer, Inc. found in PDF form on the developer.apple.com
// website, as well as using links at the bottom of the file.
//

#include <CoreMIDI/CoreMIDI.h>  /* interface to MIDI in Macintosh OS X */
#include <stdio.h>

void   printPacketInfo          (const MIDIPacket* packet);
void   myReadProc               (const MIDIPacketList *packetList,
                                 void* readProcRefCon, void* srcConnRefCon);

////////////////////////////////////////////////////////////////////////

int main(void) {

   // Prepare MIDI Interface Client/Port for writing MIDI data:
   MIDIClientRef midiclient  = NULL;
   MIDIPortRef   midiin     = NULL;
   OSStatus status;
   if (status = MIDIClientCreate(CFSTR("TeStInG"), NULL, NULL, &midiclient)) {
      printf("Error trying to create MIDI Client structure: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   if (status = MIDIInputPortCreate(midiclient, CFSTR("InPuT"), myReadProc,
         NULL, &midiin)) {
      printf("Error trying to create MIDI output port: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }

   ItemCount nSrcs = MIDIGetNumberOfSources();
   ItemCount iSrc;
   for (iSrc=0; iSrc<nSrcs; iSrc++) {
      MIDIEndpointRef src = MIDIGetSource(iSrc);
      MIDIPortConnectSource(midiin, src, NULL);
   }

   CFRunLoopRef runLoop;
   runLoop = CFRunLoopGetCurrent();
   CFRunLoopRun();

   return 0;
}


////////////////////////////////////////////////////////////////////////

//////////////////////////////
//
// myReadProc -- What to do when MIDI input packets are received.
//      used as an input parameter to MIDIInputPortCreate() so that
```

```
//      MIDI input knows what to do with the MIDI messages after it
//      receives them.

void myReadProc(const MIDIPacketList *packetList, void* readProcRefCon,
      void* srcConnRefCon) {
   MIDIPacket *packet = (MIDIPacket*)packetList->packet;
   int i, j;
   int count = packetList->numPackets;
   for (j=0; j<count; j++) {
      printPacketInfo(packet);
      packet = MIDIPacketNext(packet);
   }
}

//////////////////////////////
//
// printPacketInfo --
//

void printPacketInfo(const MIDIPacket* packet) {
   double timeinsec = packet->timeStamp / (double)1e9;
   printf("%9.3lf\t", timeinsec);
   int i;
   for (i=0; i<packet->length; i++) {
      if (packet->data[i] < 0x7f) {
         printf("%d ", packet->data[i]);
      } else {
         printf("0x%x ", packet->data[i]);
      }
   }
   printf("\n");
}

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

struct MIDIPacket { MIDITimeStamp timeStamp; UInt16 length; Byte data[256]; };
      timeStamp = The time at which the events occurred (if receiving MIDI),
                  or the time at which the events are to be played (if sending
                  MIDI).  Zero means "now" when sending MIDI data.  The time
                  stamp applies to the first MIDI byte in the packet.
      length    = The number of valid MIDI bytes which follow in data[].
                  It may be larger than 256 bytes if the packet is dynamically
                  allocated.
      data      = A variable-length stream of MIDI messages. Running status
                  is not allowed.  In the case of system-exclusive messages,
                  a packet may only contain a single message, or portion
                  of one, with no other MIDI events.  The MIDI messages in
                  the packet must always be complete, except for
                  system-exclusive messages.  data[] is declared to be 256
                  bytes in length so clients don't have to create custom data
                  structures in simple situations.

OSStatus MIDIInputPortCreate(MIDIClientRef client, CFStringRef portName,
      MIDIReadProc readProc, void* refCon, MIDIPortRef* outPort);
      client   = The client to own the newly-created port.
      portName = The name of the port.
      readProc = The MIDIReadProc which will be called with incoming MIDI.
      refCon   = The refCon passed to readHook.
      outPort  = On successful return, points to the newly-create MIDIPort.
   After creating a port, use MIDIPortConnectSource to establish an input
   connection from any number of sources to your port.
```

```
MIDIPortConnectSource(MIDIPortRef port, MIDIEndpointRef source,
      void* connRefCon);
      port       = The port to which the create the connection.  This port's
                   readProc is called with incomming MIDI from the source.
      source     = The source from which to create the connection.
      connRefCon = This refCon is passed to the MIDIReadProc, as a way to
                   identify the source.
   Establishes a connection from a source to a client's input port.


typedef void (*MIDIReadProc)(const MIDIPacketList *pktlist,
      void *readProcRefCon, void *srcConnRefCon);
    pktlist         = The incoming MIDI message(s).
    readProcRefCon  = The refCon you passed to MIDIInputPortCreate or
                      MIDIDestinationCreate
    srcConnRefCon   = A refCon you passed to MIDIPortConnectSource,
                      which identifies the source of the data.
   This is a callback function through which a client receives
   incoming MIDI messages.  A MIDIReadProc function pointer is
   passed to the MIDIInputPortCreate and MIDIDestinationCreate
   functions. The CoreMIDI framework will create a high-priority
   receive thread on your client's behalf, and from that thread,
   your MIDIReadProc will be called when incoming MIDI messages
   arrive. Because this function is called from a separate thread,
   be aware of the synchronization issues when accessing data in
   this callback.

*/
```

```
//
// Programmer:  Craig Stuart Sapp
// Date:        Wed Jun 10 12:40:22 PDT 2009
// Filename:    testin2.c
// Syntax:      C; Apple OSX CoreMIDI
// $Smake:      gcc -o %b %f -framework CoreMIDI -framework CoreServices
//              note: CoreServices needed for GetMacOSSStatusErrorString().
//
// Description: This program reads in some MIDI data and the time stamps
//              which are attached to the data. Then it compares the
//              timestamp to the time value extracted from mach_absolute_time()
//
//              This program demonstrates the the MIDI time stamps are about
//              20 to 70 microseconds before the callback function receives
//              them.  This is well below the 300 microseconds per byte
//              data rate of standard MIDI.  This means that the timestamp
//              given with the data is the time at when the last byte
//              in the message was received, and not the time when the
//              first byte in the message arrived, or the time when the
//              first byte in the message started to arrive (which might
//              be a more preferable number).
//
//

#include <CoreMIDI/CoreMIDI.h>  /* interface to MIDI in Macintosh OS X */
#include <stdio.h>
#include <mach/mach_time.h>    /* for Apple OS X timing functions */

void   printPacketInfo          (const MIDIPacket* packet);
void   myReadProc               (const MIDIPacketList *packetList,
                                 void* readProcRefCon, void* srcConnRefCon);

////////////////////////////////////////////////////////////////////////

int main(void) {

   // Prepare MIDI Interface Client/Port for writing MIDI data:
   MIDIClientRef midiclient  = NULL;
   MIDIPortRef   midiin     = NULL;
   OSStatus status;
   if (status = MIDIClientCreate(CFSTR("TeStInG"), NULL, NULL, &midiclient)) {
      printf("Error trying to create MIDI Client structure: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   if (status = MIDIInputPortCreate(midiclient, CFSTR("InPuT"), myReadProc,
         NULL, &midiin)) {
      printf("Error trying to create MIDI output port: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }

   ItemCount nSrcs = MIDIGetNumberOfSources();
   ItemCount iSrc;
   for (iSrc=0; iSrc<nSrcs; iSrc++) {
      MIDIEndpointRef src = MIDIGetSource(iSrc);
      MIDIPortConnectSource(midiin, src, NULL);
   }

   CFRunLoopRef runLoop;
   runLoop = CFRunLoopGetCurrent();
   CFRunLoopRun();

   return 0;
```

```
}


////////////////////////////////////////////////////////////////////////


//////////////////////////////
//
// myReadProc -- What to do when MIDI input packets are received.
//      used as an input parameter to MIDIInputPortCreate() so that
//      MIDI input knows what to do with the MIDI messages after it
//      receives them.

void myReadProc(const MIDIPacketList *packetList, void* readProcRefCon,
      void* srcConnRefCon) {
   MIDIPacket *packet = (MIDIPacket*)packetList->packet;
   int i, j;
   int count = packetList->numPackets;
   for (j=0; j<count; j++) {
      printPacketInfo(packet);
      packet = MIDIPacketNext(packet);
   }
}



//////////////////////////////
//
// printPacketInfo --
//

void printPacketInfo(const MIDIPacket* packet) {
   double timeinsec = 0.0;
   mach_timebase_info_data_t info;
   if (mach_timebase_info(&info) == 0) {
      timeinsec = mach_absolute_time()*1e-9*(double)info.numer/info.denom;
   }

   double stampinsec = packet->timeStamp / (double)1e9;
   printf("%9.3lfs\t-\t", timeinsec);
   printf("%9.3lfs\t", stampinsec);
   // display different in times in microseconds
   printf("%9.0lfus\t", (timeinsec-stampinsec)*1000000);
   int i;
   for (i=0; i<packet->length; i++) {
      if (packet->data[i] < 0x7f) {
         printf("%d ", packet->data[i]);
      } else {
         printf("0x%x ", packet->data[i]);
      }
   }
   printf("\n");
}
```

```
//
// Programmer:  Craig Stuart Sapp
// Date:        Tue Jun  9 16:00:00 PDT 2009
// Filename:    testin.c
// Syntax:      C; Apple OSX CoreMIDI
// $Smake:      gcc -o %b %f -framework CoreMIDI -framework CoreServices
//              note: CoreServices needed for GetMacOSSStatusErrorString().
//
// Description: This program reads in MIDI data and echos notes back
//              out based on the inversion of the MIDI key value.
//

#include <CoreMIDI/CoreMIDI.h>  /* interface to MIDI in Macintosh OS X */
#include <stdio.h>

void playPacketListOnAllDevices (MIDIPortRef midiout,
                                 const MIDIPacketList* pktlist);
void myReadProc                 (const MIDIPacketList *packetList,
                                 void* readProcRefCon,
                                 void* srcConnRefCon);

MIDIPortRef gMidiout = NULL;  // MIDI output has to be global
                              // because initialization function for
                              // MIDI output and the MIDI input callback
                              // function cannot talk to each other
                              // directly.  So the MIDI input callback
                              // should check to see if the output
                              // is valid (not NULL) before it starts
                              // trying to use gMidiout.

////////////////////////////////////////////////////////////////////////

int main(void) {

   // Prepare MIDI Interface Client/Port for writing MIDI data:
   MIDIClientRef midiclient  = NULL;
   MIDIPortRef   midiin      = NULL;
   OSStatus status;
   if (status = MIDIClientCreate(CFSTR("TeStInG"), NULL, NULL, &midiclient)) {
      printf("Error trying to create MIDI Client structure: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   if (status = MIDIInputPortCreate(midiclient, CFSTR("InPuT"), myReadProc,
         NULL, &midiin)) {
      printf("Error trying to create MIDI output port: %d\n", status);
      printf("%s\n", GetMacOSStatusErrorString(status));
      exit(status);
   }
   if (status = MIDIOutputPortCreate(midiclient, CFSTR("OuTpUt"), &gMidiout)) {
       printf("Error trying to create MIDI output port: %d\n", status);
       printf("%s\n", GetMacOSStatusErrorString(status));
       exit(status);
   }

   ItemCount nSrcs = MIDIGetNumberOfSources();
   ItemCount iSrc;
   for (iSrc=0; iSrc<nSrcs; iSrc++) {
      MIDIEndpointRef src = MIDIGetSource(iSrc);
      MIDIPortConnectSource(midiin, src, NULL);
   }

   CFRunLoopRef runLoop;
   runLoop = CFRunLoopGetCurrent();
```

```
   CFRunLoopRun();

   return 0;
}


//////////////////////////////////////////////////////////////////////

//////////////////////////////
//
// myReadProc -- What to do when MIDI input packets are received.
//      used as an input parameter to MIDIInputPortCreate() so that
//      MIDI input knows what to do with the MIDI messages after it
//      receives them.

void myReadProc(const MIDIPacketList *packetList, void* readProcRefCon,
      void* srcConnRefCon) {

   static Byte     packetbuffer[65000] = {0};
   MIDIPacket     *packet              = (MIDIPacket*)packetList->packet;
   MIDIPacketList *outPacketList       = (MIDIPacketList*)packetbuffer;
   MIDIPacket     *outPacket           =  MIDIPacketListInit(outPacketList);
   MIDITimeStamp   timestamp           = 0; // play output data immediatly
   Byte            databuffer[1024];       // storage for MIDI message data
   int i, j;

   int count = packetList->numPackets;
   for (i=0; i<count; i++) {
      if (packet->length < 3) {
         continue;
      }
      if (!(((packet->data[0] & 0xf0) == 0x80) ||
          ((packet->data[0] & 0xf0) == 0x90))) {
         continue;
      }
      for (j=0; j<packet->length; j++) {
         databuffer[j] = packet->data[j];
      }
      databuffer[1] = 127 - databuffer[1];

      outPacket = MIDIPacketListAdd(outPacketList, sizeof(packetbuffer),
         outPacket, timestamp, packet->length, databuffer);
      if (gMidiout != NULL) {
         playPacketListOnAllDevices(gMidiout, outPacketList);
      }
      outPacket = MIDIPacketListInit(outPacketList);
      packet = MIDIPacketNext(packet);
   }
}



//////////////////////////////
//
// playPacketListOnAllDevices -- play the list of MIDI packets
//    on all MIDI output devices which the computer knows about.
//    (Send the MIDI message(s) to all MIDI out ports).
//

void playPacketListOnAllDevices(MIDIPortRef midiout,
      const MIDIPacketList* pktlist) {
   // send MIDI message to all MIDI output devices connected to computer:
   ItemCount nDests = MIDIGetNumberOfDestinations();
   ItemCount iDest;
```

```
    OSStatus status;
    MIDIEndpointRef dest;
    for(iDest=0; iDest<nDests; iDest++) {
       dest = MIDIGetDestination(iDest);
       if (status = MIDISend(midiout, dest, pktlist)) {
           printf("Problem sendint MIDI data on port %d\n", iDest);
           printf("%s\n", GetMacOSStatusErrorString(status));
           exit(status);
       }
    }
}
```