6 The Midi Kit

Midi Kit Inheritance Hierarchy

6   The Midi Kit

The Musical Instrument Digital Interface (MIDI) is a standard for representing and communicating musical data. Its fundamental notion is that instantaneous musical events generated by a digital musical device can be encapsulated as "messages" of a known length and format. These messages can then be transmitted to other computer devices where they're acted on in some manner. The MIDI standard allows digital keyboards to be de-coupled from synthesizer boxes, lets computers record and playback performances on digital instruments, and so on.

The Midi Kit understands the MIDI software format (including Standard MIDI Files). With the Kit, you can create a network of objects that generate and broadcast MIDI messages. Applications built with the Midi Kit can read MIDI data that's brought into the computer through a MIDI port, process the data, write it to a file, and send it back out through the same port. The Kit contains four classes:

- The BMidi class is the centerpiece of the Kit. It defines the tenets to which all MIDI-processing objects adhere, and provides much of the machinery that realizes these ideas. BMidi is abstract—you never create direct instances of the class. Instead, you construct and connect instances of the other Kit classes, all of which derive from BMidi. You can also create your own classes that derive from BMidi.

- BMidiPort knows how to read MIDI data from and write it to a MIDI hardware port.

- BMidiStore provides a means for storing MIDI data, and for reading, writing, and performing Standard MIDI Files.

- BMidiText is a debugging aid that translates MIDI messages into text and prints them to standard output. You should only need this class while you're designing and fine-tuning your application.

To use the Midi Kit, you should have a working knowledge of the MIDI specification; no attempt is made here to describe the MIDI software format.

The BeBox comes equipped with four MIDI hardware ports. These are standard MIDI ports that accept standard MIDI cables—you don't need a MIDI interface box. The ports are aligned vertically at the back of the computer. Top-to-bottom they are MIDI-In A, MIDI-Out A, MIDI-In B, and MIDI-Out B. Currently, the Midi Kit only talks to the top set of ports (MIDI-In A and MIDI-Out A).

BMidi

Derived from: public BObject

Declared in: <midi/Midi.h>

Overview

BMidi is the centerpiece of the Midi Kit. It provides base class implementations of the functions that create a MIDI performance. BMidi is abstract; all other Kit classes—and any class that you want to design to take part in a performance—derive from BMidi. When you create a BMidi-derived class, you do so mainly to re-implement the hook functions that BMidi provides. The hook functions allow instances of your class to behave in a fashion that the other objects will understand.

The functions that BMidi defines fall into four categories:

- *Connection functions.* The connection functions let you connect the output of one BMidi object to the input of another BMidi object.

- *Message-generation functions.* Some BMidi objects generate (or otherwise procure) MIDI data. To do this, a derived class must implement the **Run()** hook function. **Run()** is the brains of a MIDI performance; other performance functions, such as **Start()** and **Stop()** control the performance.

- *"Spray" functions.* If a BMidi object wants to send a MIDI message to other BMidi objects, it does so by calling one of the output, or "spray," functions. There's a spray function for each type of MIDI message; for example, **SprayNoteOn()** corresponds to MIDI's Note On message. When a message is sprayed, it's sent to each of the objects that are connected to the output of the sprayer.

- *Input functions.* When a message is sprayed, the receivers of the message are notified by the automatic invocation of particular "input" functions. For example, when a BMidi object calls **SprayNoteOn()**, each of the objects that it's connected to becomes the target of the **NoteOn()** function. How the receiving object responds depends on the object's class: The input functions are virtual; the BMidi class implementations are empty.

Forming Connections

A fundamental concept of the Midi Kit is that MIDI data should "stream" through your application, passing from one BMidi-derived object to another. Each object does

whatever it's designed to do: Sends the data to a MIDI port, writes it to a file, modifies it and passes it on, and so on.

You form the chain of BMidi objects that propagate MIDI data by connecting them to each other. This is done through BMidi's **Connect()** function. The function takes a single argument: The object you want the caller to connect to. By calling **Connect()**, you connect the output of the calling object to the input of the argument object.

For example, let's say you want to connect a MIDI keyboard to your computer, play it, and have the performance recorded in a file. To set this up, you connect a BMidiPort object, which reads data from the MIDI port, to a BMidiStore object, which stores the data that's sent to it and can write it to a file:

```
/* Connect the output of a BMidiPort to the input of a
 * BMidiStore.
 */
BMidiPort *m_port = new BMidiPort();
BMidiStore *m_store = new BMidiStore();

m_port->Connect(m_store);
```

Simply connecting the objects isn't enough, however; you have to tell the BMidiPort to start listening to the MIDI port, by calling its **Start()** function. This is explained in a later section.

Once you've made the recording, you could play it back by re-connecting the objects in the opposite direction:

```
/* We'll disconnect first, although this isn't strictly
 * necessary.
 */
m_port->Disconnect(m_store);
m_store->Connect(m_port);
```

In this configuration, a **Start()** call to **m_store** would cause its MIDI data to flow into the BMidiPort (and thence to a synthesizer, for example, for realization).

You can connect any number of BMidi objects to the output of another BMidi object, as depicted below:
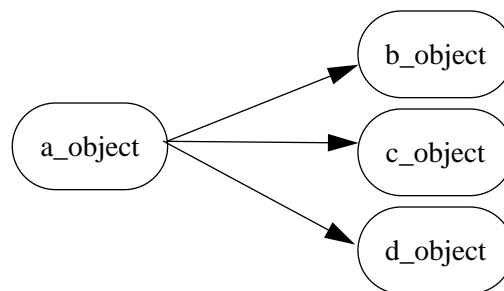
The configuration in the illustration is created thus:

```
a_object->Connect(b_object);
a_object->Connect(c_object);
a_object->Connect(d_object);
```

Every BMidi object knows which objects its output is connected to; you can get a BList of these objects through the **Connections()** function. For example, **a_object**, above, would list **b_object**, **c_object**, and **d_object** as its connections.

Similarly, the same BMidi object can be the argument in any number of **Connect()** calls, as shown below and depicted in the following illustration:

```
b_object->Connect(a_object);
c_object->Connect(a_object);
d_object->Connect(a_object);
```
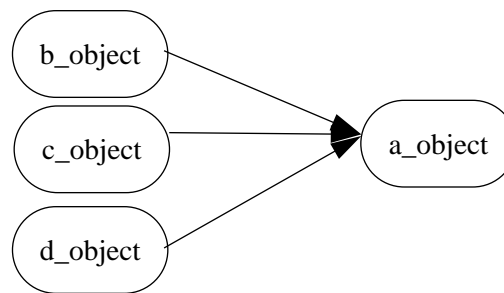
When you use a BMidi object as the argument to a **Connect()** method, the argument object *isn't* informed. In the illustration, **a_object** *doesn't* know about the objects that are connected to its input.

Generating MIDI Messages

To generate MIDI messages, you implement the **Run()** function in a BMidi-derived class. An implementation of **Run()** should include a **while()** loop that produces (typically) a single MIDI message on each pass, and then sprays the message to the connected objects. To predicate the loop you test the value of the **KeepRunning()** boolean function.

The outline of a **Run()** implementation looks like this:

```
void MyMidi::Run()
{
    while (KeepRunning()) {
        /* Generate a message and spray it. */
    }
}
```

Although your derived class can generate more than one MIDI message each time through the loop, it's recommended that you try to stick to just one.

To tell an object to perform its **Run()** function, you call the object's **Start()** function—you never call **Run()** directly. **Start()** causes the object to spawn a thread (its "run" thread) and

execute **Run()** within it. When you're tired of the object's performance, you call its **Stop()** function.

The **Run()** function is needed in classes that want to introduce new MIDI data into a performance. For example, in its implementation of **Run()**, BMidiStore sprays messages that correspond to the MIDI data that it stores. In its **Run()**, a BMidiPort reads data from the MIDI port and produces messages accordingly. If you're generating MIDI data algorithmically, or reading your own file format (BMidiStore can read standard MIDI files), then you'll need to implement **Run()**. If, on the other hand, you're creating an object that "filters" data—that accepts data at its input, modifies it, then sprays it—you won't need **Run()**.

Another point to keep in mind is that the **Run()** function can run ahead of real time. It doesn't have to generate and spray data precisely at the moment that the data needs to be realized. This is further explained in the section "Time" on page 10.

Important: The BMidi-derived classes that you create *must* implement **Run()**, even if they don't generate MIDI data; "do-nothing" implementations are acceptable, in this case. For example, if you're creating a filter (as described in a later section), your **Run()** function could be, simply

```
void MidiFilter::Run()
{}
```

Spray Functions

The spray functions are used (primarily) within a **Run()** loop to send data to the running object's connections (the objects that are connected to the running object's output). There's a separate spray function for each of the MIDI message types: **SprayNoteOn()**, **SprayNoteOff()**, **SprayPitchBend()**, and so on. The arguments that these functions take are the data items that comprise the specific messages. The spray functions also take an additional argument that gives the message a time-stamp, as explained later (again, in the "Time" section).

Input Functions

The input functions take the names of the MIDI messages to which they respond: **NoteOn()** responds to a Note On message; **NoteOff()** responds to a Note Off; **KeyPressure()** to a Key Pressure change, and so on. These are all virtual functions. BMidi doesn't provide a default implementation for any of them; it's up to each BMidi-derived class to decide how to respond to MIDI messages.

Input functions are never invoked directly; they're called automatically when a running object sprays MIDI data.

Every BMidi object automatically spawns an "input" thread when it's constructed. It's in this thread that the input functions are executed. The input thread is always running—the

Start() and Stop() functions don't affect it. As soon as you construct an object, it's ready to receive data.

For example, let's say, once again, that you have a BMidiPort connected to a BMidiStore:

```
m_port->Connect(m_store);
```

Now you open the port (a BMidiPort detail that doesn't extend to other BMidi-derived classes) and tell the BMidiPort to start running:

```
m_port->Open("midi1");
m_port->Start();
```

As the BMidiPort is running, it sends data to its output. Since the BMidiStore is connected to the BMidiPort's output, it receives this data automatically in the form of input function invocations. In other words, when m_port calls its SprayNoteOn() function (which it does in its Run() loop), m_store's NoteOn() function is automatically called. As an instance of BMidiStore, the m_store object caches the data that it receives through the input functions.

You can derive your own BMidi classes that implement the input functions in other ways. For example the following implementation of NoteOn(), in a proposed class called NoteCounter, simply keeps track of the number of times each key (in the MIDI sense) is played:

```
void NoteCounter::NoteOn(uchar channel, uchar keyNumber,
                         uchar velocity, ulong time)
{
    /* We'll assume the class has allocated an array that
     * holds the key counters.
     */
    keyCounter[keyNumber]++;
}
```

Note that the NoteOn() function in the example includes a *time* argument (the other arguments should be familiar if you understand the MIDI specification). This argument is explained in the "Time" section.

Creating a MIDI Filter

Some BMidi classes may want to create objects that act as filters: They receive data, modify it, and then pass it on. To do this, you call the appropriate spray functions from within the implementations of the input functions. Below is the implementation of the NoteOn() function for a proposed class called Transposer. It takes each Note On, transposes it up a half step, and then sprays it:

```
void Transposer::NoteOn(uchar channel, uchar keyNumber,
                        uchar velocity, ulong time)
{
    uchar new_key = max(keyNumber + 1, 127);
    SprayNoteOn(channel, new_key, velocity, time);
}
```

There's a subtle but important distinction between a filter class and a "performance" class (where the latter is a class that's designed to actually realize the MIDI data it receives). The distinction has to do with time, and is explained in the next section. An implication of the distinction that affects the current discussion is that it may not be a great idea to invest, in a single object, the ability to filter *and* perform MIDI data. By way of calibration, both BMidiStore and BMidiPort are performance classes—objects of these classes realize the data they receive, the former by caching it, the latter by sending it out the MIDI port. In neither of these classes do the input functions spray data.

Time

Every spray and input function takes a final *time* argument. This argument declares when the message that the function represents should be performed. The argument is given as an absolute measurement in *ticks*, or milliseconds. Tick 0 occurs when you boot your computer; the tick counter automatically starts running at that point. To get the current tick measurement, you call the global, Kernel Kit-defined **system_time()** function and divide by 1000.0 (**system_time()** returns microseconds).

A convention of the Midi Kit holds that time arguments are applied at an object's input. In other words, the implementation of a BMidi-derived input function would look at the time argument, wait until the designated time, and then do whatever it does that it does do. However, this only applies to BMidi-derived classes that are designed to perform MIDI data, as the term was defined in the previous section. Objects that filter data *shouldn't* apply the time argument.

To apply the *time* argument, you call the **SnoozeUntil()** function, passing the value of *time*. For example, a "performance" **NoteOn()** function would look like this:

```
void MyPerformer::NoteOn(uchar channel, uchar keyNumber,
                         uchar velocity, ulong time)
{
    SnoozeUntil(time);
    /* Perform the data here. */
}
```

If *time* designates a tick that has already tocked, **SnoozeUntil()** returns immediately; otherwise it tells the input thread to snooze until the designated tick is at hand.

An extremely important point, with regard to The SnoozeUntil() function, as used here, may cause spraying objects (objects that are spraying

Spraying Time

If you're implementing the Run() function, then you have to generate a time value yourself which you pass as the final argument to each spray functionthat you call. The value you generate depends on whether you class runs in real time, or ahead of time.

Running in Real Time

If your class conjures MIDI data that needs to be performed immediately, you should use the **B_NOW** macro as the value of the *time* arguments that you pass to your spray functions. **B_NOW** is simply a cover for (**system_time()**/1000.0) (converted to an integer). By using **B_NOW** as the *time* argument you're declaring that the data should be performed in the same tick in which it was generated. This probably won't happen; by the time the input functions are called and the data realized, a few ticks will have elapsed. In this case, the expected **SnoozeUntil()** calls (within the input function implementations) will see that the time value has passed, and so will return immediately, allowing the data to be realized as quickly as possible.

The lag between the time that you generate the data and the time it's realized depends on a number of factors, such as how loaded down your machine is and how much processing your BMidi objects perform. But the Midi Kit machinery itself shouldn't impose a latency that's beyond the tolerability of a sensible musical performance.

Running Ahead of Time

If you're generating data ahead of its performance time, you need to compute the time value so that it pinpoints the correct time in the future. For example, if you want to create a class that generates a note every 100 milliseconds, you need to do something like this:

```
void MyTicker::Run()
{
    ulong when = B_NOW;
    uchar key_num;

    while (KeepRunning()) {

        /* Make a new note. */
        SprayNoteOn(1, 60, 64, when);

        /* Turn the note off 99 ticks later. */
        when += 99;
        SprayNoteOff(1, 60, 0, when);

        /* Bump the when variable so the next Note On
         * will be 100 ticks after this one.
         */
        when += 1;
    }
}
```

When a MyTicker object is told to start running, it generates a sequence of Note On/Note Off pairs, and sprays them to its connected objects. Somewhere down the line, a performance object will apply the time value by calling SnoozeUntil().

Tethering MyTicker

But what, you may wonder, keeps MyTicker from running wild and generating thousands or millions of notes—which aren't scheduled to be played for hours—as fast as possible?

The answer is in the mechansim that connects a spray function to an input function: The BMidi class creates a port (in the Kernel Kit sense) for every object. When you invoke a spray function, the data is encoded in a message and written to each of the connected objects' ports. The input functions (invoked on the connected objects) then read from their respective ports. The secret here is that these ports are declared to be 1 (one) message deep. So, as long as one of the input function calls SnoozeUntil(), the spraying object will never be more than one message ahead.

A useful feature of this mechanism is that if you connect a series of BMidi object that *don't* invoke SnoozeUntil(), you can process MIDI data faster than real-time. For example, let's say you want to spray data from one BMidiStore object, pass the data through a filter, and then store it in another BMidiStore. The BMidiStore input functions don't call SnoozeUntil(); thus, data will flow out of the first object, through the filter, and into its destination as quickly as possible, allowing you to process hours of real-time data in just a few seconds. Of course, if you add a performance object into this mix (so you can hear the data while it's being processed), the data flow will be tethered, as described above.

Hook Functions

Run()	Contains a loop that generates and broadcasts MIDI messages.
Start()	Starts the object's run loop. Can be overridden to provide pre-running adjustments.
Stop()	Stops the object's run loop. Can be overridden to perform post-running clean-up.

The input functions (NoteOn(), NoteOff(), and so on) are also hook functions. These are listed in the section "Input and Spray Functions" on page 17.

Constructor and Destructor

BMidi()

BMidi(void)

Creates and returns a new BMidi object. The object's input thread is spawned and started in this function—in other words, BMidi objects are born with the ability to accept incoming messages. The run thread, on the other hand, isn't spawned until **Start()** is called.

~BMidi()

virtual ~**BMidi**(void)

Kills the input and run threads after they've gotten to suitable stopping points (as defined below), deletes the list that holds the connections (but doesn't delete the objects contained in the list), then destroys the BMidi object.

The input thread is stopped after all currently-waiting input messages have been read. No more messages are accepted while the input queue is being drained. The run thread is allowed to complete its current pass through the run loop and then told to stop (in the manner of the **Stop()** function).

While the destructor severs the connections that this BMidi object has formed, it doesn't sever the connections from other objects to this one. For example, consider the following (improper) sequence of calls:

```
/* DON'T DO THIS... */
a_midi->Connect(b_midi);
b_midi->Connect(c_midi);
...
delete b_midi;
```

The **delete** call severs the connection from **b_midi** to **c_midi**, but it doesn't disconnect **a_midi** and **b_midi**. You have to disconnect the object's "back-connections" explicitly:

```
/* ...DO THIS INSTEAD */
a_midi->Connect(b_midi);
b_midi->Connect(c_midi);
...
a_midi->Disconnect(b_midi);
delete b_midi;
```

See also: **Stop()**

Member Functions

Connect()

> void **Connect**(BMidi *toObject*)

Connects the BMidi object's output to *toObject*'s input. The BMidi object can connect its output to any number of other objects. Each of these connected objects receives an input function call as the BMidi sprays messages. For example, consider the following setup:

```
my_midi->Connect(your_midi);
my_midi->Connect(his_midi);
my_midi->Connect(her_midi);
```

The output of **my_midi** is connected to the inputs of **your_midi**, **his_midi**, and **her_midi**. When **my_midi** calls a spray function—**SprayNoteOn()**, for example—each of the other objects receives an input function call—in this case, **NoteOn()**.

Any object that's been the argument in a **Connect()** call should ultimately be disconnected through a call to **Disconnect()**. In particular, care should be taken to disconnect objects when deleting a BMidi object, as described in the destructor.

See also: **~BMidi()**, **Connections()**, **IsConnected()**

Connections()

> inline BList ***Connections**(void)

Returns a BList that contains the objects that this object has connected to itself. In other words, the objects that were arguments in previous calls to **Connect()**. When a BMidi object sprays, each of the objects in its connection list becomes the target of an input function invocation, as explained in the class description.

See also: **Connect()**, **Disconnect()**, **IsConnected()**

Disconnect()

> void **Disconnect**(BMidi *toObject*)

Severs the BMidi's connection to the argument. The connection must have previously been formed through a call to **Connect()** with a like disposition of receiver and argument.

See also: **Connect()**

IsConnected()

> inline bool **IsConnected**(BMidi **toObject*)

Returns TRUE if the argument is present in the receiver's list of connected objects.

See also: **Connect()**, **Connections()**

IsRunning()

> bool **IsRunning**(void)

Returns TRUE if the object's **Run()** loop is looping; in other words, if the object has received a **Start()** function call, but hasn't been told to **Stop()** (or otherwise hasn't fallen out of the loop).

See also: **Start()**, **Stop()**

KeepRunning()

protected:

> bool **KeepRunning**(void)

Used by the **Run()** function to predicate its **while** loop, as explained in the class description. This function should *only* be called from within **Run()**.

See also: **Run()**, **Start()**, **Stop()**

Run()

private:

> void **Run**(void)

A BMidi-derived class places its data-generating machinery in the **Run()** function, as described in the section "Generating MIDI Messages" on page 7.

See also: **Start()**, **Stop()**, **KeepRunning()**

SnoozeUntil()

> void **SnoozeUntil**(ulong *tick*)

Puts the calling thread to sleep until *tick* milliseconds have elapsed since the computer was booted. This function is meant to be used in the implementation of the input functions, as explained in the section "Time" on page 10.

Start()

virtual void **Start**(void)

Tells the object to begin its run loop and execute the **Run()** function. You can override this function in a BMidi-derived class to provide your own pre-running initialization. Make sure, however, that you call the inherited version of this function within your implementation.

See also: **Stop()**, **Run()**

Stop()

virtual void **Stop**(void)

Tells the object to halt its run loop. Calling **Stop()** tells the **KeepRunning()** function to return **FALSE**, thus causing the run loop (in the **Run()** function) to terminate.   You can override this function in a BMidi-derived class to predicate the stop, or to perform post-performance clean-up (as two examples). Make sure, however, that you invoke the inherited version of this function within your implementation.

See also: **Start()**, **Run()**

Input and Spray Functions

The protocols for the input and spray functions are given below, grouped by the MIDI message to which they correspond (the input function for each group is shown first, the spray function is second).

See the class overview for more information on these functions.

Channel Pressure

virtual void **ChannelPressure**(uchar *channel*,
                    uchar *pressure*,
                    ulong *time* = B_NOW)

protected:

void **SprayChannelPressure**(uchar *channel*,
                    uchar *pressure*,
                    ulong *time*)

Control Change

virtual void **ControlChange**(uchar *channel*,
                    uchar *controlNumber*,
                    uchar *controlValue*,
                    ulong *time* = B_NOW)

protected:

void **SprayControlChange**(uchar *channel*,
                    uchar *controlNumber*,
                    uchar *controlValue*,
                    ulong *time*)

Key Pressure

virtual void **KeyPressure**(uchar *channel*,
                    uchar *note*,
                    uchar *pressure*,
                    ulong *time* = B_NOW)

protected:

void **SprayKeyPressure**(uchar *channel*,
                    uchar *note*,
                    uchar *pressure*,
                    ulong *time*)

Note Off

virtual void **NoteOff**(uchar *channel*,
                                uchar *note*,
                                uchar *velocity*,
                                ulong *time* = B_NOW)

protected:

void **SprayNoteOff**(uchar *channel*,
                                uchar *note*,
                                uchar *velocity*,
                                ulong *time*)

Note On

virtual void **NoteOn**(uchar *channel*,
                                uchar *note*,
                                uchar *velocity*,
                                ulong *time* = B_NOW)

protected:

void **SprayNoteOn**(uchar *channel*,
                                uchar *note*,
                                uchar *velocity*,
                                ulong *time*)

Pitch Bend

virtual void **PitchBend**(uchar *channel*,
                                uchar *lsb*,
                                uchar *msb*,
                                ulong *time* = B_NOW)

protected:

void **SprayPitchBend**(uchar *channel*,
                                uchar *lsb*,
                                uchar *msb*,
                                ulong *time*)

Program Change

virtual void **ProgramChange**(uchar *channel*,
                                uchar *programNumber*,
                                ulong *time* = B_NOW)

protected:

void **SprayProgramChange**(uchar *channel*,

            uchar *programNumber*,
            ulong *time*)

System Common

        virtual void **SystemCommon**(uchar *status*,
                        uchar *data1*,
                        uchar *data2*,
                        ulong *time* = B_NOW)

protected:

        void **SpraySystemCommon**(uchar *status*,
                        uchar *data1*,
                        uchar *data2*,
                        ulong *time*)

System Exclusive

        virtual void **SystemExclusive**(void **data*,
                        long *dataLength*,
                        ulong *time* = B_NOW)

protected:

        void **SpraySystemExclusive**(void **data*,
                        long *dataLength*,
                        ulong *time*)

SystemRealTime()

        virtual void **SystemRealTime**(uchar *status*, ulong *time* = B_NOW)

protected:

        void **SpraySystemRealTime**(uchar *status*, ulong *time*)

Tempo Change()

   virtual void **TempoChange**(long *beatsPerMinute*, ulong *time* = B_NOW)

protected:

        void **SprayTempoChange**(long *beatsPerMinute*, ulong *time*)aa

BMidiPort

Derived from:	public BObject
Declared in:	<midi/MidiPort.h>

Overview

The BMidiPort class provides the mechanisms for reading MIDI data from the MIDI-In ports, and for writing MIDI data to the MIDI-Out ports. The BeBox has two pairs of MIDI-In and MIDI-Out hardware ports, stacked vertically on the back panel:

MIDI-In 1

MIDI-Out 1

MIDI-In 2

MIDI-Out 2
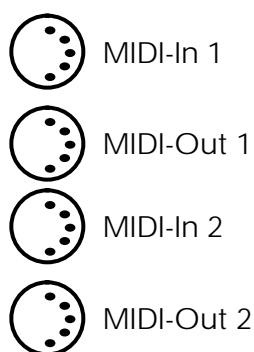
You can use a single BMidiPort object to communicate with both halves (the input side and the output side) of a single in/out pair. Thus, to talk to all four ports, you only need two BMidiPort objects. However, you can create and use any number of BMidiPort objects in your application—multiple BMidiPort objects can open and use the same hardware ports at the same time.

Opening the Ports

To obtain data from a MIDI-In port or send data to a MIDI-Out port, you must first open the ports by calling BMidiPort's **Open()** functions. The function's single argument is a string that names the identifies the in/out pair that you're opening. The two pairs of MIDI ports are named "midi1" and "midi2". For example, to open the MIDI-In 1 and MIDI-Out 1 pair, you invoke **Open()** thus:

```
BMidiPort *m_port = new BMidiPort();
m_port->Open("midi1");
```

When you're finished with the ports, you can close them through the **Close()** function. The ports are closed automatically when the BMidiPort object is destroyed.

Run() and the Input Functions

According to the BMidi rules, a BMidi-derived class implementation of **Run()** should create and spray MIDI messages. Furthermore, the implementations of the input functions should realize the messages they receive.

The BMidiPort implementation of **Run()** produces messages by reading them from the MIDI-In port and spraying them to the connected objects. The input functions send MIDI messages to the MIDI-Out port. Linguistically, this might seem backwards, but it makes sense if you think of a BMidiPort as representing not only the hardware port, but whatever is connected to the port. For example, if you're reading data that's generated by an external synthesizer, the **Run()** function can be thought of as encapsulating the synthesizer itself. From this perspective, the message-generation description of **Run()** is reasonable. Similarly, the input functions fulfill their message-realization promise when you consider them to be (for example) the synthesizer that's connected to the MIDI-Out port.

Looping through a BMidiPort Object

It's possible to use the same BMidiPort object to accept data from MIDI-In and broadcast different data to MIDI-Out. You can even connect a BMidiPort object to itself to create a "MIDI through" effect: Anything that shows up at the MIDI-In port will automatically be sent out the MIDI-Out port.

Constructor and Destructor

BMidiPort()

> **BMidiPort**(void)

Connects the object to the MIDI-In and MIDI-Out ports. The MIDI-Out connection is active from the moment the object is constructed   Messages that arrive through the input functions are automatically sent to the MIDI-Out port. To begin reading from the MIDI-In port, you have to invoke the object's **Start()** function.

~BMidiPort()

> virtual **~BMidiPort**(void)

Closes the connections to the MIDI ports.

Member Functions

AllNotesOff()

bool **AllNotesOff**(bool *controlOnly*, ulong *time* = B_NOW)

Commands the BMidiPort object to issue an All Notes Off MIDI message to the MIDI-Out port. If *controlOnly* is TRUE, only the All Notes Off message is sent. If it's FALSE, a Note Off message is also sent for every key number on every channel.

Close()

void **Close**(void)

Closes the object's MIDI ports. The ports should have been previously opened through a call to **Open**().

Open()

long **Open**(const char **name*)

Opens a pair of MIDI ports, as identified by *name*, so the object can read and write MIDI data. The names that correspond to the two set of MIDI ports are "midi1" and "midi2". The object isn't given exclusive access to the ports that it has opened—other BMidiPort objects, potentially from other applications, can open the same MIDI ports. When you're finished with the ports, you should close them through a (single) call to **Close**().

The function returns **B_NO_ERROR** if the ports were successfully opened.

BMidiStore

Derived from:	public BMidi
Declared in:	<midi/MidiStore.h>

Overview

The BMidiStore class defines a MIDI recording and playback mechanism. The MIDI messages that a BMidiStore object receives (at its input) are stored as *events* in an *event list*, allowing a captured performance to be played back later. The object can also read and write—or *import* and *export*—standard MIDI files. Typically, the performance and file techniques are combined: A BMidiStore is often used to capture a performance and then export it to a file, or to import a file and then perform it.

Recording

The ability to record a MIDI performance is vested in BMidiStore's input functions (**NoteOn()**, **NoteOff()**, and so on, as declared by the BMidi class). When a BMidiStore input function is invoked, the function fabricates a discrete event based on the data it has received in its arguments, and adds the event to its event list. The event list, in a manner of speaking, *is* the recording.

Since the ability to record is provided by the input functions, you don't need to tell a BMidiStore to start recording; it can record from the moment it's constructed.

For example, to record a performance from an external MIDI keyboard, you connect a BMidiStore to a BMidiPort object and then tell the BMidiPort to start:

```
/* Record a keyboard performance. */
BMidiStore *MyStore = new BMidiStore();
BMidiPort *MyPort = new BMidiPort();

MyPort->Connect(MyStore);
MyPort->Start();
/* Start playing... */
```

At the end of the performance, you tell the BMidiPort to stop:

```
MyPort->Stop();
```

Timestamps

Events are added to a BMidiStore's event list immediately upon arrival. Each event is given a timestamp as it arrives; the value of the timestamp is the value of the *time* argument that was passed to the input function by the "upstream" object's spray function. For example, the time argument that a BMidiPort object passes through its spray functions is always **B_NOW**. Since **B_NOW** is a shorthand for "the current tick," and since time tends to move forward at a reasonably steady rate (at least so far), the events that are recorded from a BMidiPort are guaranteed to be in chronological order (as they appear in the event list).

There's no guarantee that other spraying objects will generate *time* arguments that procede in chronological order, however. And the BMidiStore object doesn't time-sort its events as they arrive; thus, after a recording has been made, events in the event list might not be in chronological order. If you want to ensure that the events are properly ordered, you should call **Sort()** after you've added events to the event list.

Note that BMidiStore's input functions don't call **SnoozeUntil()**: A BMidiStore writes to its event list as soon as it gets a new message, it doesn't wait until the time indicated by the *time* argument.

Erasing and Editing a Recording

You can't. If you make a mistake while you're recording (for example) and want to try again, you can simulate emptying the object by disconnecting the input to the BMidiStore, destroying the object, making a new one, and re-connecting. For example:

```
MyPort->Disconnect(MyStore);
delete MyStore;
MyStore = new BMidiStore();
MyPort->Connect(MyStore);
```

Editing the events in the event list is less than impossible (were such a state possible). You can't do it, and you can't simulate it, at least not with the default implementation of BMidiStore. If you want to edit MIDI data, you have to provide your own BMidi-derived class.

Playback

To "play" a BMidiStore's list of events, you call the object's **Start()** function. For example, by reversing the roles taken by the BMidiStore and BMidiPort objects, you can send the BMidiStore's recording to an external synthesizer:

```
/* First we disconnect the objects. */
MyPort->Disconnect(MyStore);

/* Now connect in the other direction...*/
MyStore->Connect(MyPort);

/* ...and start the playback. */
MyStore->Start();
```

As described in the BMidi class specification, Start() invokes Run(). In BMidiStore's implementation of Run(), the function reads events in the order that they appear in the event list, and sprays the appropriate messages to the connected objects. You can interrupt a BMidiStore playback by calling Stop(); uninterrupted, the object will stop by itself after it has sprayed the last event in the list.

The events' timestamps are used as the *time* arguments in the spray functions that are called from within Run(). But with a twist: The *time* argument that's passed in the first spray call (for a given performance) is always B_NOW; subsequent *time* arguments are re-computed to maintain the correct timing in relation to the first event. In other words, when you tell a BMidiStore to start playing, the first event is performed immediately regardless of the actual value of its timestamp.

Setting the Current Event

A playback needn't begin with the first event in the event list. You can tell the BMidiStore to start somewhere in the middle of the list by calling SetCurrentEvent() before starting the playback. The function takes an integer argument that gives the index of the event that you want to begin with.

If you want to start playing from a particular time offset into the event list, you first have to figure out which event lies at that time. To do this, you ask for the event that occurs at or after the time offset (in milliseconds) through the EventAtDelta() function. The value that's returned by this function is suitable as the argument to SetCurrentEvent(). Here, we prime a playback to begin three seconds into the event list:

```
long firstEvent = MyStore->EventAtDelta(3000);
MyStore->SetCurrentEvent(firstEvent);
```

Keep in mind that EventAtDelta() returns the index of the first event at *or after* the desired offset. If you need to know the actual offset of the winning event, you can pass its index to DeltaOfEvent():

```
long firstEvent = MyStore->EventAtDelta(3000);
long actualDelta = MyStore->DeltaOfEvent(firstEvent);
```

Reading and Writing MIDI Files

You can also add events to a BMidiStore's event list by reading, or *importing*, a Standard MIDI File. To do this, you locate the file that you want to read, create a BFile to represent it, and pass the object to the **Import()** function:

```
BFile midi_file;

/* We'll assume that a_dir is a legitimate directory.  */
if (a_dir.Contains("myfile.mid"))
{
    /* Get the file...*/
    a_dir.GetFile("myfile.mid", &midi_file);

    /* ...and import it. */
    MyStore->Import(&midi_file);
}
```

Note that the BFile object isn't open (you shouldn't call BFile's **Open()** function before you call **Import()**).

You can import any number of files into the same BMidiStore object. After you import a file, the event list is automatically sorted.

One thing you shouldn't do is import a MIDI file into a BMidiStore that contains events that were previously recorded from a BMidiPort (in an attempt to mix the file and the recording). Nor does the reverse work: You can't import a file and *then* record from a BMidiPort. The file's timestamps are incompatible with those that are generated for events that are received from the BMidiPort; the result certainly won't be satisfactory.

To write the event list as a MIDI file, you call BMidiStore's **Export()** function:

```
BFile midi_file;

/* We'll assume that a_dir is a legitimate directory. The
 * file should be empty, so we delete it first if it exists.
 */
if (a_dir.Contains("myfile.mid"))
{
    a_dir.GetFile("myfile.mid", &midi_file);
    a_dir.Remove(&midi_file);
}

/* Create the file. */
a_dir.Create(&midi_file);

/* And export the BMidiStore. */
MyStore->Export(&midi_file, 1);
```

Export()'s second argument is an integer that declares the format of the file. The MIDI specification provides three formats: 0, 1, and 2. As with **Import()**, the BFile mustn't be open.

Constructor and Destructor

BMidiStore()

> **BMidiStore**(void)

Creates a new, empty BMidiStore object.

~BMidiText()

> virtual ~**BMidiStore**(void)

Frees the memory that the object allocated to store its events.

Member Functions

BeginTime()

> inline ulong **BeginTime**(void)

Returns the time, in ticks, at which the most recent performance started. This function is only valid if the object has actually performed.

CountEvents()

> inline ulong **CountEvents**(void)

Returns the number of events in the object's event list.

CurrentEvent()

> inline ulong **CurrentEvent**(void)

Returns the index of the event that will be performed next.

See also: **SetCurrentEvent()**

DeltaOfEvent()

> ulong **DeltaOfEvent**(ulong *index*)

Returns the "delta time" of the *index*'th event in the object's list of events. An event's delta time is the time span, in ticks, between the first event in the event list and itself.

See also: **EventAtDelta()**

EventAtDelta()

ulong **EventAtDelta**(ulong *delta*)

Returns the index of the event that occurs on or after *delta* ticks from the beginning of the event list.

See also: **DeltaOfEvent()**

Export()

void **Export**(BFile **aFile,* long *format*)

Writes the object's event list as a standard MIDI file in the designated format. The BFile must be allocated, must refer to an actual file, and its data portion must not be open. The events are time-sorted before they're written.

See also: **Import()**

Import()

void **Import**(BFile **aFile*)

Reads the standard MIDI file from the BFile given by the argument. The BFile must not be open.

See also: **Export()**

SetCurrentEvent()

void **SetCurrentEvent**(ulong *index*)

Sets the object's "current event"—the event that it will perform next—to the event at *index* in the event list.

See also: **CurrentEvent()**

SetTempo()

void **SetTempo**(ulong *beatsPerMinute*)

Sets the object's tempo—the speed at which it performs events—to *beatsPerMinute*. The default tempo is 60 beats-per-minute.

See also: **Tempo()**

SortEvents()

>   void **SortEvents(**bool *force* = FALSE**)**

Time-sorts the events in the BMidiStore. The object maintains a (conservative) notion of whether the events are already sorted; if *force* is **FALSE** (the default) and the object doesn't think the operation is necessary, the sorting isn't performed. If force is **TRUE**, the operation is always performed, regardless of its necessity.

Tempo()

>   ulong **Tempo(**void**)**

Returns the object's tempo in beats-per-minute.

See also: **SetTempo()**

BMidiText

Derived from: public BMidi

Declared in: <midi/MidiText.h>

Overview

A BMidiText object displays, to standard output, a textual description of each MIDI message it receives. You use BMidiText objects to debug and monitor your application; it has no other purpose.

To use a BMidiText object, you construct it and connect it to some other BMidi object as shown below:

```
BMidiText *midiText;

midiText = new BMidiText();
otherMidiObj->Connect(midiText);

/* Start a performance here ... */
```

BMidiText's output (the text it displays) is timed: When it receives a MIDI message that's timestamped for the future, the object waits until that time has come to display its textual representation of the message. While it's waiting, the object won't process any other incoming messages. Because of this, you shouldn't connect the same BMidiText object to more than one BMidi object. To monitor two or more MIDI-producing objects, you should connect a separate BMidiText object to each.

The text that's displayed by a BMidiText follows this general format:

     timestamp: *MESSAGE TYPE*; *message data*

(Message-specific formats are given in the function descriptions, below.) Of particular note is the *timestamp* field. Its value is the number of milliseconds that have elapsed since the object received its first message. The time value is computed through the use of an internal timer; to reset this timer—a useful thing to do between performances, for example—you call the **ResetTimer()** function.

The BMidiText class doesn't generate or spray MIDI messages, so the performance and connection functions that it inherits from BMidi have no effect.

Constructor and Destructor

BMidiText()

BMidiText(void)

Creates a new BMidiText object. The object's timer is set to zero and doesn't start ticking until the first message is received. (To force the timer to start, call **ResetTimer(TRUE)**.)

~BMidiText()

virtual ~**BMidiText**(void)

Does nothing.

Member Functions

ChannelPressure()

virtual void **ChannelPressure**(char *channel*,
                    char *pressure*,
                    ulong *time* = B_NOW**)**

Responds to a Channel Pressure message by printing the following:

timestamp: CHANNEL PRESSURE; channel = *channel*, pressure = *pressure*

The *channel* and *pressure* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

ControlChange()

virtual void **ControlChange**(char *channel*,
                    char *ctrl_num*,
                    char *ctrl_value*,
                    ulong *time* = B_NOW**)**

Responds to a Control Change message by printing the following:

timestamp: CONTROL CHANGE; channel = *channel*, control = *ctrl_num*, value = *ctrl_value*

The *channel*, *ctrl_num*, and *ctrl_value* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

KeyPressure()

> virtual void **KeyPressure**(char *channel*,
>   char *note*,
>   char *pressure*,
>   ulong *time* = B_NOW)

Responds to a Key Pressure message by printing the following:

timestamp: KEY PRESSURE; channel = *channel*, note = *note*, pressure = *pressure*

The *channel*, *note*, and *pressure* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

NoteOff()

> virtual void **NoteOff**(char *channel*,
>   char *note*,
>   char *velocity*,
>   ulong *time* = B_NOW)

Responds to a Note Off message by printing the following:

timestamp: NOTE OFF; channel = *channel*, note = *note*, velocity = *velocity*

The *channel*, *note*, and *velocity* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

NoteOn()

> virtual void **NoteOn**(char *channel*,
>   char *note*,
>   char *velocity*,
>   ulong *time* = B_NOW)

Responds to a Note On message by printing the following:

timestamp: NOTE ON; channel = *channel*, note = *note*, velocity = *velocity*

The *channel*, *note*, and *velocity* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

PitchBend()

> virtual void **PitchBend**(char *channel*,
>   char *lsb*,
>   char *msb*,
>   ulong *time* = B_NOW)

Responds to a Pitch Bend message by printing the following:

timestamp: PITCH BEND; channel = *channel*, lsb = *lsb*, msb = *msb*

The *channel*, *lsb*, and *msb* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

ProgramChange()

> virtual void **ProgramChange**(char *channel*,
>   char *program_num*,
>   ulong *time* = B_NOW)

Responds to a Program Change message by printing the following:

timestamp: PROGRAM CHANGE; channel = *channel*, program = *program_num*

The *channel* and *program_num* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

ResetTimer()

> void **ResetTimer**(bool *start* = FALSE)

Sets the object's internal timer to zero. Lacking a *start* argument—or with a *start* of FALSE—the timer doesn't start ticking until the next MIDI message is received. If *start* is TRUE, the timer begins immediately.

The timer value is used to compute the timestamp that's displayed at the beginning of each message description.

SystemCommon()

> virtual void **SystemCommon**(char *status*,
>   char *data1*,
>   char *data2*,
>   ulong *time* = B_NOW)

Responds to a System Common message by printing the following:

timestamp: SYSTEM COMMON; status = *status*, data1 = *data1*, data2= *data2*

> The *channel*, *data1*, and *data2* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

SystemExclusive()

> virtual void **SystemExclusive(**void **data*,
> long *data_length*,
> ulong *time* = B_NOW**)**

Responds to a System Exclusive message by printing the following:

timestamp: SYSTEM EXCLUSIVE;

> This is followed by the data itself, starting on the next line. The data is displayed in hexadecimal, byte by byte. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

SystemRealTime()

> virtual void **SystemRealTime(**char *status*, ulong *time* = B_NOW**)**

Responds to a System Real Time message by printing the following:

timestamp: SYSTEM REAL TIME; status = *status*

> The *status* value is taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).