

# *C M I D I*

CMIDI.π		
Name	obj size	
♦ CMIDIClient.c	448	↑
♦ CMIDIDataPort.c	382	□
♦ CMIDIInputPort.c	518	▒
♦ CMIDIOutputPort.c	640	↓
♦ CMIDIPort.c	934	↕
♦ CMIDITimePort.c	1472	□

## MIDI Manager Class Library for THINK C™

*Version 2.1*

### Programmer's Reference Manual

Paul D. Ferguson

## **Copyright Notices**

CMIDI is copyright © 1991, 1992, 1993 Paul D. Ferguson. All rights reserved.

Apple, PatchBay, and Macintosh are registered trademarks of Apple Computer, Inc.

THINK C is a trademark of Symantec Corporation.

The style of this manual is unabashedly borrowed from the THINK C documentation. Not only is imitation the sincerest form of flattery<sup>1\*</sup>, so too can it be justified as being familiar to THINK C programmers.

## **Contacting the Author**

Please send any comments, suggestions, or bug reports to me at:

CompuServe: 70441,3055.

SnailMail: Paul Ferguson  
258 Rodonovan Drive  
Santa Clara, CA 95051

VoiceMail: (408) 249-7189

I look forward to hearing from you.

—*Fergy*

---

<sup>1\*</sup> Or as Ben Franklin said, “There is much difference between imitating a good man and counterfeiting him.”

# Contents

1	Introduction.....	1
	Audience.....	1
	Big, Fat Disclaimer.....	2
	Caveat Programmer.....	2
	Software Requirements.....	2
	What's New in CMIDI 2.1.....	3
2	CMIDI Programming Basics.....	4
	Supported MIDI Manager Calls.....	5
	CMIDI Classes.....	6
	Creating CMIDI Objects.....	6
	Reading MIDI Data.....	8
	Interrupt Level Read Hooks.....	8
	Polled Read Hooks.....	8
	A Better Approach.....	9
	Time Bases.....	10
	Quitting Time.....	11
	MIDI Packet Objects.....	11
3	CMIDIClient.....	12
4	CMIDIDataPort.....	14
5	CMIDIInputPort.....	15
6	CMIDIOutputPort.....	17
7	CMIDIPort.....	19
8	CMIDITimePort.....	22
9	Global Variables.....	25
10	Creating a MIDI Manager Library.....	26
	Using THINK C 4.0 or MIDI Manager 1.2.....	27

Apple's MIDI Manager is the future of MIDI programming on the Macintosh. The MIDI Manager allows applications to exchange MIDI messages with external devices (keyboards, synthesizer modules, drum machines) as well as with other MIDI Manager compatible applications. Using PatchBay, MIDI musicians can connect multiple MIDI Manager applications under MultiFinder.

The **CMIDI class library** provides an object oriented programming (OOP) interface to the MIDI Manager. A wide range of MIDI applications, from simple editors and librarians to sophisticated real-time musical tools,

lend themselves to object oriented design.

Built on the OOP extensions in Symantec Corporation's THINK C 5.0 compiler, CMIDI defines a set of classes integrated into the THINK Class Library (TCL). CMIDI classes define MIDI Manager objects including input, output, and time ports.

## **Audience**

I assume that you are an experienced Macintosh programmer and are already familiar with THINK C, TCL, and object oriented programming.

I also assume that you are familiar with the MIDI Manager and have the developer's documentation and software "*MIDI Management Tools*" Version 2.0 or later. Presently, the

only place to obtain this package is  
through APDA<sup>21</sup> (part number M0240LL/D or later). CMIDI is useless  
without these files.

These classes make it easier to deal with the MIDI Manager, but they don't completely insulate the programmer from it. For example, it is up to you to understand what an input port readHook routine does, and how to initialize and manipulate MIDI ports. In the documentation which follows, I often refer you to the "*MIDI Management Tools*" documentation for specific information about the MIDI Manager.

---

<sup>21</sup> The Apple Programmer's and Developer's Association. For more information, contact APDA on CompuServe, AppleLink, or elsewhere.

### Big, Fat Disclaimer

This source code and documentation is made available as freeware from Paul Ferguson. The source code and documentation are copyrighted in their entirety by the author. All rights reserved.

**Commercial distribution of the source code or documentation is expressly prohibited without written permission from the author.**

You may freely use these routines in applications which you develop, provided that you acknowledge my copyright in your application and documentation.

(I wouldn't mind if you sent me a copy of your program, either.)

### Caveat Programmer

This code is thoroughly untested.

Let me repeat that: *This code is thoroughly untested.*

Do I have to say it again?

***THIS CODE IS THOROUGHLY UNTESTED!***

I created these classes primarily for my own use in developing MIDI Manager applications (mostly the shareware program "*Chroma*"). I have used some, but not all, of the features in these classes in this development. The rest, well...

Don't be fooled by this great documentation into thinking that just because I said a class method will do a certain thing, it actually will. I'm a dreamer, not a tester (pardon me, I meant "quality assurance engineer").

### Software Requirements

You must have the following to use CMIDI:

Product	Preferred	Supported
Macintosh System Software	7.0/7.1	6.0.4
THINK C	5.0	4.0
MIDI Manager Developer's Kit	2.0	1.2

**Figure 1-1** Software requirements

CMIDI is designed for use with THINK C 5.0 (as of this writing, the latest version is 5.0.4). If you are still using THINK C 4.0, upgrade.

If you absolutely can't, you can still use this source code with some minor modifications.

This version of CMIDI is designed for version 2.0 of the MIDI Manager. Applications developed using CMIDI are compatible with version 1.2, and the CMIDI routines compensate for differences between 1.2 and 2.0. If you only have access to version 1.2 for your development system, you will need to make modifications to the source files. Methods which are valid only under version 2.0 are noted.

Refer to the last section of this manual for more details about using MIDI Manager 1.2 or THINK C 4.0.

### **What's New in CMIDI 2.1**

If you have used CMIDI version 2.0, the differences between it and this release are minimal.

The most significant difference is that this version uses new static function pointers for MIDI Manager functions `MIDIWrite`, `MIDIDiscard`, `MIDIFlush`, and `MIDIPoll` when running with MIDI Manager 2.0 or later. The advantage to this approach, as described in the *MIDI Management Tools 2.0 Addendum*, is better performance by bypassing the Macintosh trap dispatcher.

The documentation and source code in version 2.1 have been cleaned up as well.



## **Assumptions**

I assume you are familiar with programming in the THINK C environment and that you understand the concepts of THINK C's object oriented programming and class library.

I also assume you have an understanding of the MIDI Manager programming interfaces. As with TCL, you must be familiar with its concepts, philosophy and theory of operations. For example, you will need to create your application's read hook functions, which requires an intimate understanding of the MIDI Manager.

Before you begin, you will need to convert the MIDI Manager MPW MIDIGlue.o object file and MIDI.h header file to THINK C format. Refer to the last chapter of this manual “Creating a MIDI Manager Library” for details on how to do this.

This chapter discusses the implementation of CMIDI classes, the object hierarchy, and provides some sample code indicating usage. For detailed information about specific classes, refer to the appropriate section of this manual.

## **Supported MIDI Manager Calls**

The CMIDI class methods support the following MIDI Manager functions.

CMIDI	CMIDI	CMIDI	CMIDI
Client	Time	Input	
Output			
Function	Port	Port	Port
SndDispVersion	•		
MIDIGet/SetConnectProc*	•		
MIDIGetPorts	•		
MIDISignIn/Out	•		
MIDIWorldChanged	•		
MIDIAddPort	•	•	•
MIDIGetPortInfo	•	•	•
MIDIGet/SetPortName	•	•	•
MIDIGet/SetRefCon	•	•	•
MIDIGet/SetCurTime	•		
MIDIGet/SetOffsetTime	•		
MIDIGet/SetSync	•		
MIDIStart/StopTime	•		
MIDIWakeUp	•		
MIDIDiscardPacket*		•	
MIDIFlush		•	
MIDIGet/SetReadHook		•	
MIDIGet/SetTCFormat		•	•
MIDIWritePacket			•
* MIDI Manager Version 2.0 only			

Figure 2-1 Supported MIDI Manager functions

The following MIDI Manager calls are not supported in CMIDI.

MIDIRemovePort	MIDIGetClients
MIDIGet/SetClientName	MIDIConnect/UnConnectData
MIDIConnect/UnConnectTime	MIDIGet/SetClRefCon
MIDIConvertTime	MIDIGetClientIcon

MIDISetRunRate	All MDVR calls
----------------	----------------

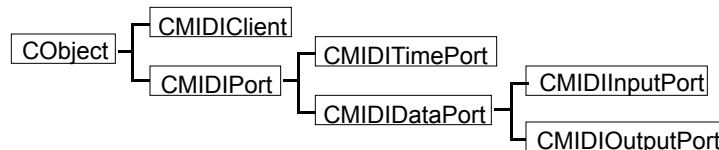
**Figure 2–2** Unsupported MIDI Manager functions

These functions are mostly used by patchers or other clients. Refer to the “*MIDI Management Tools*” manual for details about these

calls. If you need the functionality of one of these, you can define additional methods for CMIDIClient or one of the port objects.

### CMIDI Classes

CMIDI defines six classes derived from CObject. Figure 2–3 illustrates their relationship.



**Figure 2–3** CMIDI class hierarchy

Two abstract classes are defined: CMIDIPort and CMIDIDataPort. You normally don't create instances of these, rather you create CMIDITimePort, CMIDIInputPort, and CMIDIOutputPort objects.

The CMIDIClient class is responsible for initializing the MIDI Manager interface, and registering the application with the MIDI Manager. A global CMIDIClient object, `gMIDIClient`, must be created and initialized prior to creating any port objects.

MIDI port objects manage sending and receiving MIDI data. An application may have one or more of each type of port object. Refer to the “*MIDI Management Tools*” manual for more information about MIDI Manager ports.

### Creating CMIDI Objects

At runtime, CMIDI objects are created. The hierarchy of these objects is simple: an application can have zero, one, or more input, output, or time ports. Any of these port objects may be subclassed to specialize or expand their behavior. Every application must have exactly one CMIDIClient object, referenced by the global variable `gMIDIClient`.

Let's look at the basic flow of CMIDI programming. The code fragment in figure 4 shows the creation of one of each type of object.

By design, you can create any of the CMIDI objects and send them messages even if the MIDI Manager drivers aren't loaded. Obviously no MIDI packets can be read or written in this situation, but otherwise the objects should behave normally. Methods which return an error code will return `ErrNoMIDI` (see `CMIDIClient.h`).

Note that this example does not include any error checking on creation of the objects. Of course, you would never program this way, would you?

```

#include "CMIDIClient.h"
#include "CMIDIInputPort.h"
#include "CMIDIOutputPort.h"
#include "CMIDITimePort.h"

extern CMIDIClient * gMIDIClient;
extern pascal myConnProc(short refNum ...);

void CMyApp::IMyApp(...)
{
    ...
    // Initialize gMIDIClient and our port objects.
    // And remember, error checking is for wimps!

    gMIDIClient = new CMIDIClient;
    err = gMIDIClient->IMIDIClient(MIDIRes);

    itsMIDITime = new CMIDITimePort;
    err = itsMIDITime->IMIDITimePort(
        "\pMy Time",    // Port name
        'ATim',         // Port ID
        TRUE,           // Visible?
        midiFormatMSec); // Time format
    itsMIDITime->LoadPatches('ATim', 128);
    itsMIDITime->StartTime();
    itsMIDITime->SetConnection(myConnProc);

    itsMIDIOut = new CMIDIOutputPort;
    err = itsMIDIOut->IMIDIOutputPort(
        "\pMy Out",    // Port name
        'Out ',        // Port ID
        TRUE,           // Visible?
        itsMIDITime,    // Time base object
        0L);            // Time offset
    itsMIDIOut->LoadPatches('Out ', 128);

    itsMIDIIn = new CMIDIInputPort;
    err = itsMIDIIn->IMIDIInputPort(
        "\pMy In",    // Port name
        'In ',        // Port ID
        TRUE,           // Visible?
        itsMIDITime,    // Time base object
        midiGetNothing, // Time offset
        INBUFSIZE,     // Input buffer size
        midiReader);    // Read hook
    itsMIDIIn->LoadPatches('In ', 128);
}

```

**Example 2-4** Initializing CMIDI objects

### **Reading MIDI Data**

The MIDI Manager notifies an application of incoming MIDI data via a readHook procedure. There are two times when your readHook can be called: at interrupt or non-interrupt time. For time-critical applications you typically must have a readHook routine which is called at interrupt time.

For other types of applications like patch librarians or editors it may be simpler to have a non-interrupt readHook. You can receive MIDI data by polling the MIDI Manager from your application, document, or other object.

### **Interrupt Level Read Hooks**

Interrupt level routines in the Macintosh are subject to significant restrictions. Most QuickDraw routines cannot be called, nor can any Toolbox calls which might move memory.

Since standard TCL applications use handles for objects (known as indirect objects), most objects are subject to being moved by the memory manager. If you reference any indirect objects, they should be locked using `CObject::Lock`.

Consequently, using CMIDI presents some issues to address if you wish to process MIDI data during an interrupt. If your readHook routine references any object handles, those handles should be locked down before any MIDI packets are received by the readHook.

If your application uses an interrupt-level readHook you may wish to optimize its speed by storing a copy of your output port object's reference number (using `GetRefNum`) in a global variable that the readHook can access after restoring register A5. You may then call MIDI Manager functions such as `MIDIWritePacket` directly, bypassing the overhead of method calls within your readHook.

### **Polled Read Hooks**

If your application does not depend on receiving MIDI data in real time, you can poll the MIDI Manager in your application or document object's `Dawdle` method. Make sure to specify `midiGetNothing` to the `IMIDIInputPort` method to prevent your read hook routine from being called at interrupt time. This will avoid the restrictions inherent in interrupt processing.



```
void CMyApp::Dawdle(long * maxSleep)
{
    itsMIDIIn->Poll(midiGetCurrent); // Get MIDI data?
    *maxSleep = 5; // Wait 5 somethings
}
```

### **Example 2–5 Polling MIDI Manager at idle time.**

Depending on how quickly you need to retrieve MIDI data, you may wish to call `CMIDIInputPort::Poll` more than once in your `Dawdle` method.

In general, frequently polling the MIDI Manager is not a good idea because of the CPU cycles involved in repeatedly calling the MIDI Manager drivers to see whether there is any MIDI data waiting. Since the MIDI Manager is most likely used in MultiFinder environments, this is a real concern.

### **A Better Approach**

You may wish to consider a “hybrid” of the two methods: Design an interrupt level read hook, which sets a global flag whenever it requires the attention of your application. Your application’s `Dawdle` methods can then poll the global flag (which is more efficient than executing the Toolbox trap for `MIDIPoll`) to know when to act. Your application can then process the MIDI data in non-interrupt mode.

The code segment in figure 6 shows an interrupt level read hook which reads note on/off messages and builds a ring buffer for processing by the application’s `Dawdle` routine.

Other types of hybrid approaches are possible, depending on your application.

```

//--- midiReader -----
// This places MIDI note on/off data into a ring
// buffer.
//-----
long MIDINotes[1000]; // Assume initialized to 0's
long * currNote = MIDINotes;
long * nextNote = MIDINotes;

pascal short midiReader(MIDIpacket * thePacket,
    long TheRefCon)
{
    long SysA5 = SetA5(TheRefCon);
    long * notePtr;

    if ((thePacket->flags == 0) &&
        (thePacket->data[0] < 0xA0)) // Note on/off
    {
        notePtr = (long *) &thePacket->data[0];
        *currNote = *notePtr;
        *currNote >>= 8;
        if (++currNote == &MIDINote[1000])
            currNote = MIDINote;
    }
    SetA5(SysA5);
    return midiMorePacket;
}
...
// Dawdle routine extracts notes.
void CMyApp::Dawdle(long * maxSleep)
{
    while (*nextNote)
    {
        this->ProcessNote(*nextNote);
        *nextNote = 0L;
        if (++nextNote == &MIDINote[1000])
            nextNote = MIDINote;
    }
}

```

**Example 2-6** A hybrid polling method.

### Time Bases

When you create a time port object, it is initially set to internal time synchronization unless a virtual connection is resolved which indicates external synchronization.

While your application is running, if an external time port is connected to your time port (via PatchBay), you should check to see whether your time port's synchronization should be changed to external. Likewise, if a connection to your time port is broken, you should

change back to internal synchronization.

In version 1.x of the MIDI Manager, the only way to accomplish this was to call `MIDIWorldChanged` in your event loop, and if it returns `TRUE`, check the time port's connections to see whether the synchronization state should be changed. Version 2.0 introduced the concept of a connection procedure to handle timing synchronization changes. Refer to the "*MIDI Management Tools*" documentation for information about writing a connection procedure.

The `CMIDITimePort` objects can automatically handle timing connections from other MIDI Manager applications. After creating a time port object, send it an `AssignIdleChore` message or a `SetConnectionProc` message.

### **Quitting Time**

When quitting, your application should send each `CMIDI` object a `SavePatches` message if you wish to save the current port connections (this is strongly recommended in "*MIDI Management Tools*"). After that, send each port object a `Dispose` message, then finally dispose of `gMIDIClient`.

### **MIDI Packet Objects**

`CMIDI` does not include a class definition to encapsulate the behavior of a MIDI Packet. MIDI packets are the central data structures that the MIDI Manager itself deals with, so it would seem natural to create a corresponding class to wrap around a MIDI packet.

In my applications, however, I haven't found the need for an explicit packet class. Incoming MIDI data is immediately converted to specific messages for objects, usually in the visual hierarchy. These messages may contain MIDI information, such as velocity or note value, which are interpreted by the target object.

For this reason, I have not attempted to create a `CMIDIPacket` class. This class should be fairly easy to define, and is left as an exercise.

## **Introduction**

CMIDIClient implements a class for registering an application with the MIDI Manager.

## **Heritage**

Superclass CObject

Subclasses None

## **Using CMIDIClient**

Each Macintosh application that wishes to use the MIDI Manager must register itself with the MIDI Manager. This process allows other MIDI Manager applications to recognize it, and establish patches (connections) to

your application. The CMIDIClient class is responsible for this.

You must have one and only one CMIDIClient object in your application. The global variable

gMIDIClient, which is declared in CMIDIClient.c, must be created and initialized prior to any port objects.

Refer to the CMIDI Programming Basics chapter for examples of how to create and initialize gMIDIClient.

#### Variables

##### Variable

midiMgrVerNum

##### Type Description

unsigned long Returned by SndDispVersion indicating what version of the MIDI Manager is present.

If IMIDIClient was unsuccessful at signing into the MIDI Manager, midiMgrVerNum will be zero.

#### Methods

##### IMIDIClient

```
void IMIDIClient(short theIconID);
```

This method initializes a CMIDIClient object. It signs into the MIDI Manager using the creator type of the application (found in gSignature), icon number theIconID, and the application's file name.

NOTE: This method uses an ICN# resource rather than an ICON resource so that you may simply specify your application's bundle ICN# resource.

### **Dispose**

```
void Dispose(void);
```

Dispose of this object. Calls MIDISignOut.

### **GetPorts**

```
MIDIIDListHdl GetPorts(void);
```

Return a list of ports. See the “*MIDI Management Tools*” documentation of MIDIGetPorts for more information about the fields of the MIDIIDList structure this handle points to.

### **WorldChanged**

```
Boolean WorldChanged(void);
```

Calls MIDIWorldChanged. If the MIDI Manager is not present, returns FALSE.

### **GetVerNum**

```
unsigned long GetVerNum(void);
```

Return midiMgrVerNum. This is the full 32-bit version number from the BNDL resource of the MIDI Manager. If the MIDI Manager is not present, midiMgrVerNum will be zero.

### **GetShortVerNum**

```
unsigned short GetShortVerNum(void);
```

Return the upper word of midiMgrVerNum which contains the MIDI Manager's version number, for example 0x0120 = version 1.2, 0x0200 = version 2.0. You can use this in your application to check for specific functionality or to alert the user of an incompatibility.

### **Introduction**

CMIDIDataPort is an abstract class for implementing input and output ports.

### **Heritage**

Superclass CMIDIPort

Subclasses CMIDIInputPort  
CMIDIOutputPort

### **Using CMIDIDataPort**

CMIDIDataPort contains methods common to both input and output ports. You should not create objects of this type, but rather create CMIDIInputPort and



# CMIDIOutputPort objects.

## Variables

None.

## Methods

### LoadPatches

```
OSErr LoadPatches (ResType theResType, short theResID);
```

If itsResult is zero, check for a resource of type theResType and ID equal theResID. If present, call MIDIConnectData for each connection. If a resource error occurs or the specified resource does not appear to be a valid patch list for this port, an error is returned.

### GetTCFormat

```
short GetTCFormat (void);
```

Return the port's current time code format. Valid return values (midiFormatMSec, etc.) are defined in MIDI.h. If the MIDI Manager is not present, this method returns a -1.

### SetTCFormat

```
void SetTCFormat (short theFormat);
```

Set the port's time code format. Valid format types are defined in MIDI.h

### **Introduction**

CMIDIInputPort implements a MIDI Manager input port.

### **Heritage**

Superclass CMIDIDataPort

Subclasses None

### **Using CMIDIInputPort**

An application may have one or more input ports for reading incoming MIDI data.

Incoming MIDI Manager packets are read by a read hook procedure, which can be set using `SetReadHook`. See the CMIDI Programming

Basics chapter and the “*MIDI Management Tools*” manual for more information about

creating CMIDIInputPorts and read hooks.

### Variables

Variable	Type	Description
midiDiscardProc		
midiFlushProc		
midiPollProc	static ProcPtr	Function pointers directly to the entry points for these MIDI Manager functions.

### Methods

IMIDIInputPort

```
OSErr IMIDIInputPort(StringPtr theName, OSType thePortID,  
Boolean theVisibleFlag, CMIDITimePort * theTimePort, long  
theOffset, short theBufSize, ProcPtr theReadHook);
```

This method initializes the input port object and registers the port with the MIDI Manager.

Specify the port name, four character port ID, time offset, buffer size, and read hook as per “*MIDI Management Tools*”. Set `theVisibleFlag` to TRUE if the port should be visible in PatchBay (only MIDI Manager 2.x or later supports invisible input ports). Pass in a `CMIDITimePort` object in `theTimePort`, or NULL if no time base is needed. Following common convention, this method stores the current A5 register value in the port’s `refCon`. You may change this by calling `SetRefCon`.

#### **GetReadHook**

```
ProcPtr GetReadHook(void);
```

Return pointer to the current read hook. Refer to “*MIDI Management Tools*” for information about read hook procedures.

#### **SetReadHook**

```
void SetReadHook(ProcPtr theReadHook);
```

Set the port’s read hook procedure.

#### **Flush**

```
void Flush(void);
```

Call `MIDIFlush` to flush all packets currently waiting in the port’s input buffer.

#### **Poll**

```
void Poll(long offsetTime);
```

Call `MIDIPoll`. If a MIDI packet is waiting, the MIDI manager will call the port’s read hook procedure.

#### **DiscardPacket**

```
void DiscardPacket(PacketPtr thePacket);
```

If the MIDI Manager version is 2.0 or greater, call `MIDIIDiscardPacket`. Otherwise do nothing.

### **Introduction**

CMIDIOutputPort implements a MIDI output port.

### **Heritage**

Superclass CMIDIDataPort

Subclasses None

### **Using CMIDIOutputPort**

An application may have one or more output port objects for writing MIDI data.

You can send MIDI messages to the MIDI Manager in one of two ways. If the message you wish to send is

already in a valid MIDI Manager packet, you can call

`WritePacket` to send it. This is useful in read hooks to echo packets back to the MIDI Manager.

If your message is not in a MIDI Manager packet, then you can call `Write` and `WriteTS`. These methods will copy a valid MIDI message into a `MIDIPacket` structure, initialize the other fields, and call the MIDI Manager. If the length of the data exceeds 249 bytes, it is automatically broken up into multiple MIDI Manager packets. This is especially useful for sending long system exclusive messages.

#### Variables

Variable	Type	Description
<code>midiWriteProc</code>	<code>static ProcPtr</code>	Function pointer to the <code>MIDIWrite</code> function.

#### Methods

#### **IMIDIOutputPort**

```
OSErr IMIDIOutputPort(StringPtr theName, OSType thePortID,  
Boolean theVisibleFlag, CMIDITimePort * theTimePort, long  
theOffset);
```

This method initializes the output port object and registers the port with the MIDI Manager.

Specify the port name, four character port ID, and time offset as per “*MIDI Management Tools*”. Set `theVisibleFlag` equal `TRUE` if the port should be visible in `PatchBay` (only MIDI Manager 2.x or later supports invisible output ports). Pass in a `CMIDITimePort`

object in theTimePort, or NULL if no time base is needed. Following common convention, this method stores the current A5 register value in the port's refCon. You may change this by calling SetRefCon.

#### **WritePacket**

```
OSErr WritePacket(MIDIPacketPtr theMIDIPacket);
```

Call MIDIPacketWrite to send theMIDIPacket to the MIDI Manager.

#### **Write**

```
OSErr Write(char * theData, short theDataLen);
```

Copy theData to the MIDI Manager with a timestamp of zero and MIDI packet flags value of midiTimeStampCurrent. Breaks long messages into multiple MIDI Manager calls.

#### **WriteTS**

```
OSErr WriteTS(char * theData, short theDataLen, long  
theTimeStamp);
```

Copy theData to the MIDI Manager with the specified timestamp and a MIDI packet flags value of midiTimeStampValid. Breaks long messages into multiple MIDI Manager calls.

#### **DoMIDIWrite**

```
OSErr DoMIDIWrite(char * theData, short theDataLen,  
unsigned char theFlags, long theTimeStamp);
```

This method is used by Write and WriteTS to send MIDI data. It is private to CMIDIOutputPort, and should not be called directly by the application.

### **Introduction**

CMIDIPort is an abstract class for MIDI ports.

### **Heritage**

Superclass CObject

Subclasses CMIDIDataPort

CMIDITimePort

### **Using CMIDIPort**

CMIDIPort contains methods and instance variables which are common to all port classes.

### **Variables**

Variable	Type	Description
----------	------	-------------



<code>itsRefNum</code>	<code>short</code>	Reference number returned by <code>MIDIAddPort</code> .
<code>itsPortID</code>	<code>OSType</code>	Four byte port identifier.
<code>itsResult</code>	<code>OSErr</code>	Result code from <code>MIDIAddPort</code> .
<code>itsVersion</code>	<code>unsigned short</code>	Stores the short MIDI Manager version number.

`ItsVersion` flags whether the `gMIDIClient` object has been created, and whether it appears that it was successful opening the MIDI driver. All port methods check this variable before issuing any MIDI Manager trap calls. Some methods use it to check for the presence of MIDI Manager version 2.0. `ItsResult` is used by `LoadPatches` and `SavePatches` to determine whether any virtual connections were resolved when the port was created.

### Methods

#### **IMIDIPort**

```
OSErr IMIDIPort(MIDIPortParamsPtr portParams, short
bufSize);
```

This method initializes the object and calls `MIDIAddPort` to add itself to the list of application ports. `IMIDIPort` is a protected method, you should not call it directly. This method is called by `IMIDITimePort`, `IMIDIInputPort`, and `IMIDIOutputPort`.

The global TCL variable `gSignature`, which normally contains the application's creator ID, is used as the client ID for all ports.

This method initializes the instance variable `itsVersion` to `gMIDIClient->GetShortVerNum()`, so that port methods can efficiently determine whether the MIDI Manager drivers are present. It also stores the result of `MIDIAddPort` in `itsResult` for use by `LoadPatches` and `SavePatches`.

**Dispose** `void Dispose(void);`

Note that `Dispose` does not call `MIDIRemovePort`. Doing so causes serious problems with the MIDI Manager (as I discovered after many long hours of debugging).

**GetPortInfo** `MIDIPortInfoHdl GetPortInfo(void);`

Return a data structure containing all port connections. See the *"MIDI Management Tools"* documentation of `MIDIGetPortInfo` for more information about the fields of the `MIDIPortInfoHdl` structure this handle points to.

**GetRefNum** `short GetRefNum(void);`

Return the port reference number. Directly using a port's reference number can increase performance within a read hook routine.

**GetRefCon** `long GetRefCon(void);`

Return the port's `refCon`, which is initially set to register A5.

**SetRefCon** `void SetRefCon(long theRefCon);`

Change the port's `refCon`.

**GetPortName** `void GetPortName(StringPtr theName);`

Return the port name. The name can be up to 32 characters long.

**SetPortName** `void SetPortName(StringPtr theName);`

Set the port name. The name can be up to 32 characters long.

**LoadPatches** `OSErr LoadPatches(ResType theResType, short theResID);`

This is a pure virtual function (it calls the CObject method `SubclassResponsibility`) to define the interface. The actual `LoadPatches` code resides in overridden methods in `CMIDIDataPort` and `CMIDITimePort`.

**SavePatches** `OSErr SavePatches(ResType theResType, short theResID);`

Save the current patch connections in a resource of type `theResType` with ID equal `theResID`. The name of the resource is set to the port's name.

**SetConnectionProc**

```
void SetConnectionProc(ProcPtr theConnectProc, long  
theRefCon);
```

This method is for MIDI Manager version 2.0 or later. The function `theConnectProc` will be called whenever a connection is made or broken for this port. Refer to “*MIDI Management Tools 2.0 Addendum*” for more information about connection procedures.

**GetConnectionProc**

```
void GetConnectionProc(ProcPtr * theConnectProc, long *  
theRefCon);
```

Returns the port’s connection procedure address and its associated `refCon`. Valid for MIDI Manager 2.0 or later.

### **Introduction**

CMIDITimePort implements a MIDI Manager time port.

### **Heritage**

Superclass CMIDIPort

Subclasses None

### **Using CMIDITimePort**

An application may have one or more time ports to derive timing information. You should create time port objects before any input or output ports which use the time port as a time base.

# Variables

None.

# Methods

## IMIDITimePort

```
OSErr IMIDITimePort(StringPtr theName, OSType  
thePortID, Boolean theVisibleFlag, short theFormat);
```

This method initializes a timeport object and registers the port with the MIDI Manager.

Specify the port name and four character port ID. Set theVisibleFlag to TRUE if the port should be visible in PatchBay.

Refer to MIDI.h for time format constants passed in theFormat.

### LoadPatches

```
OSErr LoadPatches(ResType theResType, short theResID);
```

If itsResult is zero, check for a resource of type theResType and ID equal theResID. If present, call MIDIConnectTime for each connection.

### GetSync

```
short GetSync(void);
```

Call MIDIGetSync. Possible return values are midiInternalSync or midiExternalSync. If the MIDI Manager is not present, returns a -1. To change port synchronization, call SetExternalSync or SetInternalSync.

**SetExternalSync**

```
void SetExternalSync(void);
```

Call `MIDISetSync` to set external synchronization.

**SetInternalSync**

```
void SetInternalSync(void);
```

Call `MIDISetSync` to set internal synchronization.

**UpdateSync**

```
short UpdateSync(void);
```

This method checks whether the time base is connected to another time base and adjusts the time port's synchronization accordingly. It is called by `IMIDITimePort` and `Perform`.

You may call `UpdateSync` directly from your application when you wish to have a time port's synchronization checked. If `gMIDIClient->WorldChanged()` returns `TRUE`, you should send all `CMIDITimePort` objects an `UpdateSync` message.

`UpdateSync` returns the (possibly new) current sync setting for the port.

**GetCurTime**

```
long GetCurTime(void);
```

Call `MIDIGetCurTime`. If the MIDI Manager is not present, returns zero.

**SetCurTime**

```
void SetCurTime(long theTime);
```

Call `MIDISetCurTime`.

**StartTime**

```
void StartTime(void);
```

Call `MIDIStartTime`.

**StopTime**

```
void StopTime(void);
```

Call `MIDIStopTime`.

**GetOffsetTime**

```
long GetOffsetTime(void);
```

Call `MIDIGetOffsetTime`. If the MIDI Manager is not present, returns zero.

**SetOffsetTime**

```
void SetOffsetTime(long theOffset);
```

Call `MIDISetOffsetTime`.

**WakeUp**

```
void WakeUp(long theBaseTime, long thePeriod, ProcPtr  
theTimeProc);
```

Call `MIDIWakeUp`.

**SetConnection**

```
void SetConnection(ProcPtr theConnectionProc);
```

This method performs one of two actions.

If the version of MIDI Manager running is at least 2.0, and theConnectionProc is not NULL, then it calls SetConnectionProc using the port's refCon (usually A5).

Otherwise it send gApplication an AssignIdleChore (this) message (see Perform).

## **Perform**

```
void Perform(long * maxSleep);
```

This method allows you to automatically check for connections made while your program runs. Pass this time port object to gApplication->AssignIdleChore, and the application will call its Perform method during idle time. This method checks for a change in its MIDI world, and calls the UpdateSync method when a change is detected.

If you create a connectionProc (MIDI Manager version 2.x or later) you do not need to use this method; connection procedures are more efficient.

## Introduction

CMIDI contains one global object,  
`gMIDIClient`.

### **gMIDIClient**

#### **Global Objects**

```
CMIDIClient * gMIDIClient;
```

Every application which uses CMIDI objects must initialize `gMIDIClient` before any port objects are created. Refer to the section on `CMIDIClient` (pp. 12–13) for more information.



To use Apple's MIDI Manager development software (orderable from APDA: Part No. M0240LL) with THINK C, you must first convert the MPW object file `MIDIGlue.o` and header file `MIDI.h`.

The result is a THINK C library file which should be included in any project using MIDI Manager calls and a THINK C compatible header file.

You need to do this conversion only once.

**Step 1.** Convert the MPW file `MIDIGlue.o` to a THINK C library.

- (a) Start oConv (supplied with THINK C), and check the box marked “.v” file.
- (b) Select the `MIDIGlue.o` file, and click on Convert. After it completes, click on Convert again. (It's very important to do this *twice!*). After quitting oConv, you should see two new files, `MIDIGlue.π` and `MIDIGlue.v`.
- (c) Open `MIDIGlue.π`. You should see one module titled “SndDispVersion”. (If you don't, or the name is in all uppercase letters, you have made a mistake and should start over.) Select Build Library... to create a library file. You may wish to store it in the THINK “Mac Libraries” folder with the other library files. You can then delete `MIDIGlue.π` and `MIDIGlue.v`, they are no longer needed.

**Step 2.** Convert the MPW header file `MIDI.h` to THINK format.

---

**NOTE:** The file `MIDI.h` provided with THINK C 5.0 is from MIDI Manager version 1.2. Since these libraries are for MIDI Manager 2.0, you should substitute the `MIDI.h` file found on your MIDI Management Tools diskette for the one in THINK C, and make the following changes.

---

- (a) Add the preprocessor statement:  

```
#define _H_MIDI
```

at the beginning of the file.
- (b) Remove the `extern` keyword from the declaration of `SndDispVersion`.

- (b) If you wish, using a tool like ResEdit or DiskTop, change the file creator to 'KAHL' so that it has a THINK C document icon (this step is not required).

### **Using THINK C 4.0 or MIDI Manager 1.2**

With slight modifications, you can use CMIDI with either THINK C 4.0 or MIDI Manager 1.2. You are strongly urged to upgrade, however.

If you are using THINK C 4.0, change the `class` definitions in the `.h` files to `struct` and remove the access specifiers `private:`, `protected:`, and `public:`.

Change the C++ style comments (`//...`) to standard comments (`/*...*/`) in all source files.

You must also remove all references to MIDI Manager 2.0-specific features in the source files; these function calls and constants will cause compile or link errors. Figure 2-1 indicates the 2.0 specific methods.