MidiShare™

Developer Documentation

version 1.91

GRAME - Computer Music Research Lab.
9, rue du Garet BP 1185
FR - 69202 LYON CEDEX 01
Ph: +33 (0)4 720 737 00
Fax: +33 (0)4 720 737 01

http://midishare.sourceforge.net/

Summary

Introduction

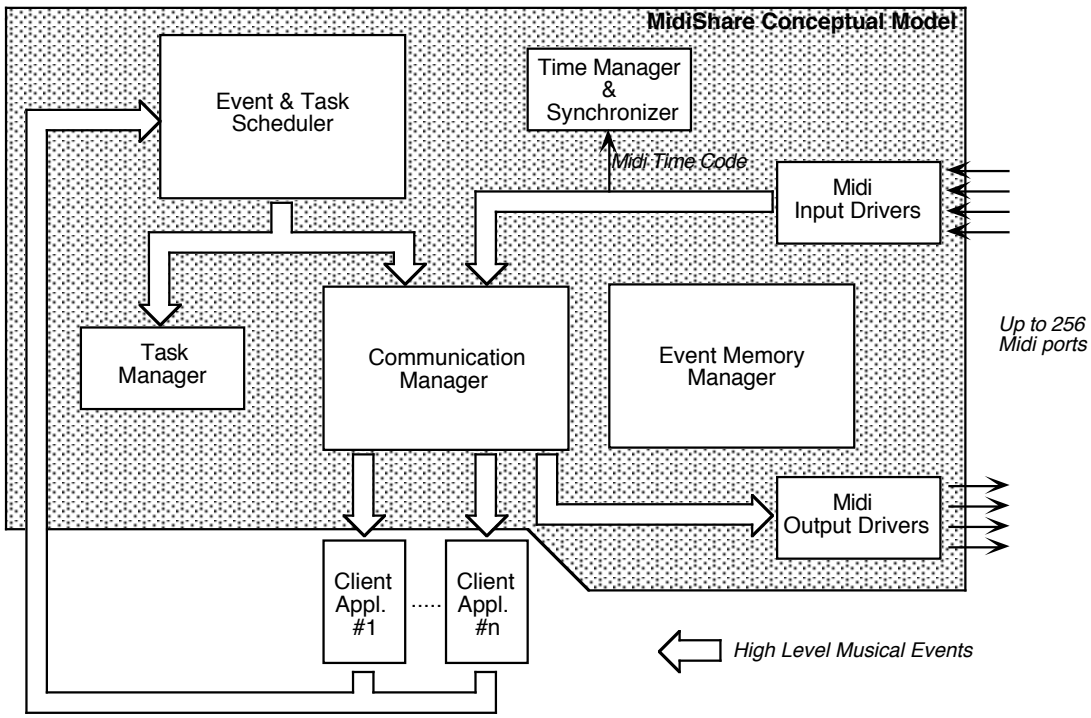
About this manual

This manual is intended for developers who wish to write MIDI applications using MidiShare. It contains a complete description of all the MidiShare functions and data structures, as well as several examples of code. This manual describes MidiShare 1.68 both for Apple Macintosh and Atari computers.

MidiShare is a real-time multi-tasking MIDI operating system specially devised for the development of musical applications. Its innovative features and careful design (the result of 6 years of research and development), provide developers with a powerful and efficient toolbox for developing MIDI applications.

MidiShare is based on a client/server model. It is composed of six main components : an *event memory manager*, a *time manager and synchronizer*, a *task manager*, a *communication manager*, an *event and task scheduler* and *Midi drivers*.

Figure 1 The conceptual Model of MidiShare



The figure 1 shows the conceptual model of MidiShare :

- The *Event and Task Scheduler* is in charge of delivering scheduled events and tasks at the right date. It allows events to be sent in the future as well as functions to be called in the future. This ability to schedule function calls is a very powerful mechanism which is particularly useful in real-time applications where multiple tasks need to run in parallel with precise timings. The scheduling algorithm used ensures a very low and constant time overhead per event, even when the scheduler is heavily loaded.
- The *Time Manager and Synchronizer* maintains the current date of the system. It offers 1ms resolution and supports accurate transparent SMPTE synchronization.
- The *Communication Manager* routes events received from scheduler and the input drivers to the client applications and output drivers according to the connections set between applications.
- The *Task Manager* is in charge of calling the tasks delivered by the scheduler.

- The *Event Memory Manager* is a dynamic memory manager, specially designed for real-time operations at interrupt level. It provides applications with a convenient and efficient way for storing, copying and deleting MidiShare events without using the host memory manager.
- The *Midi Drivers* are in charge of the physical Midi communications with up to 256 Midi ports.

MidiShare avoids many of the complexities and limitations of other MIDI Operating Systems and offers the advantages of code efficiency, portability and simplicity of application development.

Communication is based on high level events instead of packets of Midi bytes. These events are easier and faster to process than packets of Midi bytes. For example, large system exclusive messages never need to be split into multiple packets. They are sent, received and processed as a whole, like any other Midi events. Events are not limited to strict MIDI messages, MidiShare offers full support for Midi File 1.0 events. Lyrics, tempo changes and cue points for example can be sent and received by client applications like Midi events. Future versions of MidiShare will provide additional events for multimedia.

MidiShare allows multi-port configurations to be handled using upto 256 ports. All MidiShare events are stamped with a full Midi device address defined by a physical port number and a Midi channel. Client applications just need one input and one output connection to communicate with all MIDI devices (up to 4096 devices).

All of the above and the fact that the full device address of an event is never lost during inter-application communication makes application code considerably simpler than with other MIDI operating systems.

It should also be noted that MidiShares internal buffers and queues are dynamically sized avoiding the overflow problems encountered with other systems.

Several mechanisms have been implemented to control the real time behavior of MidiShare applications. *Receive Alarms* can be installed by client applications to deal with incoming events when they occur (in real time at interrupt level). A *Context alarm* can also be installed to inform applications of changes in the MidiShare configuration. Function calls can be scheduled in two ways : functions scheduled by *MidiTask* are done in true real-time, while those scheduled by *MidiDTask* benefit from the scheduler, but are only pseudo real-time as they are functions that can't be called at interrupt level (i.e. those that use the host Memory Manager).

Overview of a MidiShare application

This section gives an overview of MidiShare functions as used by a typical application.

Opening and Closing a MidiShare session

First of all, an application must make sure that MidiShare is installed in memory, this check can be completed by the `MidiShare` function.

If MidiShare is installed then the function `MidiOpen` should be called to start a MidiShare session. This allows MidiShare to record information relating to the application context (i.e. its name, the value of A5 register, etc.) and to create a reception FIFO and to attribute a unique reference number to the application.

Before closing, an application must call the counterpart `MidiClose` function, giving its application reference number as an argument. MidiShare can thus be aware of the precise number of active MidiShare applications. In theory, there is no objection to an application performing several `MidiOpen`'s, under the condition that it performs as many corresponding `MidiClose`'s. In total, there must not be more than 63 simultaneously open MidiShare applications .

As long as no `MidiOpen`'s are performed, MidiShare is dormant and has no effect on the operation of the computer. Following the first `MidiOpen`, MidiShare becomes active, it then creates a task which will be called by interrupt every millisecond and initiates ACIA interruption vectors and registers corresponding to the physical MIDI ports. MidiShare returns to its dormant state after the last `MidiClose` is performed.

Communications and Connections

For an application to be able to transmit and receive events, it must first be connected to one or more source and destination.

MidiShare is built around an internal communication mechanism which allows the exchange of events in real-time between client applications. An application can be thought of as a black box, receiving a flow of events at input and producing a flow of events at output. These 'black boxes' can be freely connected to others, thus forming an arbitrary complex network. This is one of the major advantages of MidiShare, that it allows transparent, powerful collaboration between applications that are otherwise totally independent.

MIDI hardware input and output is performed by a pseudo-application, which is always refereed to as application number 0 and named 'MidiShare'. To communicate with the 'outside world', your applications input and/or output should be connected to this application.

The implementation of these connections is very simple. The `MidiConnect` function allows the switching on or off connections between a source and destination applications and the `MidiIsConnected` function gives the state (on or off) of a connection. There are no restrictions in establishing connections, an application can be source or destination of as many applications as you wish and of course looping is possible.

In some special cases, it is important that an application can obtain information regarding the other active MidiShare applications. The `MidiCountAppls` function

returns the number of open MidiShare applications. The `MidiGetIndAppl` function returns the reference number of an application by giving an order number (between 1 and the result of `MidiCountAppls`). It is also possible to find the reference number of an application by name using the `MidiGetNamedAppl` function. In the same way, knowing an application reference number, it is possible to find its name using the `MidiGetName` function. And last, the `MidiSetName` function allows change of an applications name.

When writing 'meta-applications' for the management of connections requiring information on context modifications in MidiShare (opening of new applications, changing connections, etc.) , all that is required is the definition of a context alarm using the `MidiSetApplAlarm` and `MidiGetApplAlarm` functions. This alarm function will be automatically called by MidiShare to inform the application of all the occurred context changes.

Sending and receiving

Once connections have been established, an application can send and receive MIDI events. Each application owns a reception FIFO in which MidiShare puts a copy of received events. These events may come from other applications or from the different MIDI ports in active use. MidiShare can in theory handle up to 256 ports. The implementation of MIDI ports is controlled by the `MidiSetPortState` and `MidiGetPortState` routines, these functions must be used with caution since they affect all applications.

The `MidiCountEvs` function allows an application at any time to know the number of events waiting in its reception FIFO. This number of events is only limited by the amount of memory available to MidiShare. The available events are picked up by repeated calls to the `MidiGetEv` function. The `MidiAvailEv` function is similar, except it allows the reading of a received event while leaving it in the FIFO. The `MidiFlushEvs` function eliminates all the events on wait in the reception FIFO.

Events received by applications are duplicates, the application can therefore freely dispose of them without any repercussion on other applications. However, an application must not forget to free them when it no longer needs them.

Each application can select the events it receives by using a filter. The filtering process is local to the application and has no influence on the events received by the other applications. The implementation of these filters is achieved by two routines : `MidiSetFilter` and `MidiGetFilter`.

MidiShare drives an internal absolute clock of 32 bits which is automatically switched on with the first `MidiOpen` and keeps running until the last `MidiClose`. This clock is used to date (in milliseconds) all the received events, as well as to specify the sending dates of events to be transmitted. Moreover, it provides all the applications with an absolute time reference. Its value can be read by the `MidiGetTime` function.

Three functions facilitate the transmission of events. The `MidiSendIm` function allows the immediate transmission of an event and the `MidiSend` and `MidiSendAt` functions allow time delayed transmission, MidiShare automatically managing the scheduling of the transmission time.

(thanks to this mechanism, applications can plan transmissions at millisecond accuracy up to many days in advance.)

Once an event is transmitted (by the means of `MidiSend`, `MidiSendAt` or `MidiSendIm`), it is no longer accessible by the application. This event must no longer be refereed to, as to do so would cause irreparable damage to MidiShare's event organization.

Event management

The memory management of a standard application is generally performed by the computers 'Memory Manager' (MM). The MM deals with dynamic allocation, freeing memory blocks of arbitrary length, and memory compacting when necessary (in the case of excessive fragmentation of memory). A traditional MM is unsuitable for use in a real time context for the following reasons:

- Only large memory blocks can be allocated efficiently. For example, the Macintosh MM has an overhead of several bytes per allocated blocks, which is prohibitive for the very small groups of bytes associated with MIDI events.
- The allocation time of a block is not constant, but depends on several factors, one of which being the fragmentation state of the memory. Allocation times can be very long if memory needs to be compacted and therefore a traditional MM cannot guarantee a response time.
- A traditional MM is not re-entrant, therefore no routine under interruption can use it either directly or indirectly, without disorganizing the memory space.

To overcome these short-comings, MidiShare possess its own memory manager, which is adapted to the Midi event management and is available under interruption.

MidiShare drives a group of events common to all the applications. Each event has compulsory fields (date, channel, port, type, etc.) and variable fields that depend of its type.

Allocation is performed by the `MidiNewEv` function which returns an event of a suitable type. The counterpart de-allocation is done by the `MidiFreeEv` function. Another way of allocating an event is to duplicate an existing event by the `MidiCopyEv` function.

It is possible at any time to discover the available remaining event space, by the `MidiFreeSpace` function.

Access to the common event fields can be done directly, but access to the variable fields is achieved through the `MidiSetField` and `MidiGetField` functions.

Some categories of events do not have a fixed number of fields, for example System Exclusive messages, in this case the `MidiCountFields` function returns the number of fields in the variable length event and the `MidiAddField` function allows the addition of a field at the tail of the variable length event.

For some special applications, it may be necessary to access the basic functions of the memory manager. All the events managed by MidiShare are implemented in fixed-size cells (16 bytes). Most events need just one cell, others like the System Exclusive use a variable number of linked cells. Most applications normally do not have to worry about storage 'details', nevertheless, two functions are provided for low level memory management. The first one, `MidiNewCell`, allows to allocate a simple cell. The second one, `MidiFreeCell`, operates in reverse and de-allocates a cell.

Sequence management

MidiShare provides basic functionalities for the managing of sequences of time ordered events. The `MidiNewSeq` function allocates a new sequence, empty at the start and the `MidiAddSeq` function inserts an event into the sequence, maintaining the time order.

The `MidiApplySeq` function is an iterating function which allows the processing of a sequences events, by a user defined function, the address of which is passed as a parameter.

The `MidiClearSeq` function flushes the contents of a sequence and the `MidiFreeSeq` function frees the sequence events.

Real time tasks

The MidiShare scheduling mechanism is based around the concept of alarms. An alarm is a function whose address is sent to MidiShare by an application, MidiShare will then call this function in real time to indicate the occurrence of an event, even if the application is in interrupt.

Each application can define two categories of alarms, the first is defined by the `MidiSetApplAlarm` function, this warns of any change in the global context of MidiShare (see paragraph "Communications and connections"). The second category is defined by the `MidiSetRcvAlarm` function which informs of the presence of new events in the reception FIFO. This alarm is always called under interruption, and therefore, it must not use the Macintosh Memory Manager either directly or indirectly. However, it can have a free access to all the MidiShare functions (apart from `MidiOpen` or `MidiClose`) and it may also access the global variables of the application, as before the call, MidiShare restores its context register.

Macintosh desk accessories cannot have global variables, so to make up for this drawback the `MidiSetInfo` routine allows each application to define a data area. The area pointer remains accessible by the `MidiGetInfo` function, even during alarms, and it can also be used for global data areas for desk accessories and other application.

Once the RcvAlarm is set, the application can organise its real-time tasks utilizing its private FIFO. As opposed to traditional MIDI events, private events are messages that an application sends to itself, an application generally makes use of these to remember that a task has to be done at a precise date.

When the date of a private event falls, MidiShare puts the event into the applications reception FIFO, where it waits to be picked up and handled in the same way as MIDI events.

MidiShare implements a second mechanism to manage tasks. This is a time-delayed function call using `MidiTask` (or `MidiCall`) and the `MidiDTask` functions. To achieve this call, MidiShare collects the call arguments, as well as the functions address to be called and triggers a special event (`typeProcess` or `typeDProcess`). When a `typeProcess` event falls MidiShare restores the application context and proceeds to the call the function. However, when a `typeDProcess` event falls, the function is not processed immediately, but placed in a waiting list belonging to the application.

The `MidiCountDTasks` allows an application to find the number of tasks currently waiting to be executed, they can then be executed when required using `MidiExec1DTask` which executes the next task on wait. (Note that actual execution must be initiated by the application)

As the MidiTasks are processed under interruption, they must not call the operating system either directly or indirectly. The MidiDTasks allow a by-pass of this obstacle since the application triggers their processing (generally in the main loop).

Under certain circumstances, 'forgetting' an already launched but not yet processed MidiTask or MidiDTask, can be useful. The `MidiForgetTask` function is used for this purpose. Also an applications MidiDTask waiting list can be deleted by `MidiFlushDTasks` function.

In order to simplify communication between application tasks and to the manage sharing of variables, two non-interruptable, pointer-handling routines are

provided. The `MidiReadSync` function reads and sets to NIL the value at a memory address, and the `MidiWriteSync` function updates the value of an address only if its current value is NIL.

Midi Time Code Synchronisation

MidiShare can be synchronised to an external Midi Time Code (MTC) using the `MidiSetSyncMode` function. `MidiSetSyncMode` takes a parameter describing the chosen synchronisation mode (internal or external) and the synchronisation input port to be used. The synchronisation mode is global and it affects all MidiShare applications. The function `MidiGetSyncInfo` provides information regarding the synchronisation process.

When the synchronisation mode is set to internal (the default mode), MidiShare is driven by an internal interrupt every millisecond. (the "size" of a MidiShare time unit is one millisecond) The function `MidiGetTime` gives MidiShare's internal time, which is the time elapsed since the very first `MidiOpen`, expressed in milliseconds.

When the synchronisation mode is set to external, MidiShare looks for an incoming MTC. When enough MTC's are detected, MidiShare becomes locked to the signal. It warns all the MidiShare applications, by calling their ApplAlarm, if any, with the code using `MidiSyncStart`. A typical sequencer might use this information to start playing a sequence according to the position of the tape. The function `MidiGetExtTime` returns the position of the tape in milliseconds.

When an incoming MTC is no longer detected, MidiShare becomes unlocked, it automatically adjusts its time unit to one millisecond and again informs the MidiShare applications via their ApplAlarm with the code `MidiSyncStop`. A typical sequencer application may for example decide to stop playing its sequences in this situation.

While MidiShare is locked, it maintains a constant offset between its internal time and the external time (the time of the tape), by automatically adjusting the size of the time unit to follow the speed variations of the incoming MTC. The size of the MidiShare time unit will be exactly one millisecond when the MTC runs at its nominal speed, it will increase when the MTC slows down and decrease when the MTC's speed increases. For example with an MTC format of 25 frames/second, one frame represents 40 milliseconds (1000/25). In this case MidiShare will adjust the size of its time unit in order to always have 40 time units per frame whatever the actual speed of the incoming MTC. Consequently, from the point of view of a MidiShare application, the duration of one frame at 25 frames/seconds will always be 40 milliseconds.

The function `MidiGetExtTime` returns the external time (the time of the tape expressed in milliseconds).

While MidiShare is locked :

```
MidiGetTime() - MidiGetExtTime() == constant offset
```

The difference between MidiShare's internal time and the tape time expressed in millisecond is a constant. Two functions are provided to convert between external and internal time `MidiInt2ExtTime` and `MidiExt2IntTime` :

```
MidiInt2ExtTime( MidiGetTime() ) == MidiGetExtTime()
MidiExt2IntTime( MidiGetExtTime() ) == MidiGetTime()
```

Two additional functions, `MidiTime2Smpte` and `MidiSmpte2Time`, are provided to make conversions between time expressed in millisecond and SMPTE time locations. For example :

```
MidiTime2Smpt ( MidiGetExtTime(), 3, &loc )
```

This will set loc with the current SMPTE location of the tape using SMPTE format 3 (30 frames / seconds).

These functions can be used to convert SMPTE locations from one format to another. For example suppose we want to derive a SMPTE location from a current 30 drop frame format, we can write :

```
MidiTime2Smpte( MidiSmpte2Time (&loc), 2, &loc);
```

where 2 means 30 drop frame.

Some examples

We give here some very simple examples of MidiShare programs. In order to keep the listings short they have no 'user-interface', just a command line like in a traditional UNIX environment. They where written for the Macintosh but they can be easily adapted for other computers. The specific differences with the Macintosh is that the string arguments to MidiShare functions are in Pascal format (starting with `\p` like in `"\pExample1"`) and the user defined functions (like tasks and alarms) that are passed to MidiShare functions are prefixed with the `PASCAL` keyword. If you run these examples on another computer, you need to remove both the `\p` and the `PASCAL` keyword.

Example 1 : the shortest MidiShare program

Program example are often too long to type so here is the shortest MidiShare program one can write. It starts a MidiShare session using the `MidiOpen` function and then closes the session using the `MidiClose` function, that's all.

Listing 1

```
#include <MidiShare.h>

main()
{
  short  myRefNum;

  myRefNum = MidiOpen("\pExample1");
  MidiClose(myRefNum);
}
```

Example 2 : still short but safer

The previous example was not very safe. Usually you need first to test if MidiShare is available, then check its version number and finally test if you have succeeded in opening a MidiShare session.

Listing 2

```
#include <stdio.h>
#include <stdlib.h>
#include <MidiShare.h>

main()
{
  short  myRefNum;

  if (! MidiShare() ) {
    printf("error : MidiShare not available\n");
    exit(0);
  }

  if ( MidiGetVersion() < 168 ) {
    printf("error : MidiShare version 1.68 or greater required\n");
    exit(0);
  }
```

```
  myRefNum = MidiOpen("\pExample2");

  if (myRefNum < 0) {
    printf("Unable to open a MidiShare session (code %d)\n",
      myRefNum);
    exit(0);
  }

  MidiClose(myRefNum);
}
```

Example 3 : waiting

In this example we spend 3 seconds printing dots. The checking of the previous example has been removed for sake of simplicity.

Listing 3

```
#include <stdio.h>
#include <stdlib.h>
#include <MidiShare.h>


main()
{
  short  myRefNum;
  long stopdate;

  myRefNum = MidiOpen("\pExample3");
  stopdate = MidiGetTime() + 3000;

  printf("waiting");
  while (MidiGetTime() < stopdate) {
    printf(".");
  }
  printf("\n");

  MidiClose(myRefNum);
}
```

Example 4 : multitasking

The previous example used a very trivial method of time handling. In this example we use a method in which several tasks are scheduled in the future. The `PrintTask` function is used as a repetitive task to print characters. The `StopTask` is used to inform the program stop.

Listing 4

```
#include <stdio.h>
#include <stdlib.h>
#include <MidiShare.h>

long gStopflag;

pascal
void PrintTask (long dt, short ref, long c, long delay, long a3);
```

```
pascal
void StopTask (long dt, short ref, long a1, long a2, long a3);

main()
{

  short  myRefNum;
  short  i;
  long dt;

  myRefNum = MidiOpen("\pExample4");
  dt = MidiGetTime();

  /* schedule the stop task                    */
  gStopflag= 0;
  MidiTask(StopTask, dt+6150, myRefNum, 0, 0, 0);

  /* schedule the print task with different */
  /* delays and characters to print           */
  MidiDTask(PrintTask, dt+100, myRefNum, ' ', 100, 0);
  MidiDTask(PrintTask, dt+201, myRefNum, 'H', 200, 0);
  MidiDTask(PrintTask, dt+302, myRefNum, 'E', 300, 0);
  MidiDTask(PrintTask, dt+403, myRefNum, 'L', 400, 0);
  MidiDTask(PrintTask, dt+604, myRefNum, 'L', 600, 0);
  MidiDTask(PrintTask, dt+1005, myRefNum, 'O', 1000, 0);

  printf("Running :\n");

  while (gStopflag == 0) {
     for (i = MidiCountDTasks(myRefNum); i; i--) {
       MidiExec1DTask(myRefNum);
     }
  }

  printf("\nStopped\n");

  MidiClose(myRefNum);
}


pascal
void PrintTask (long dt, short ref, long c, long delay, long a3)
{
  fputc(c, stdout);
  fflush(stdout);
  MidiDTask(PrintTask, dt+delay, ref, c, delay, 0);
}

pascal
void StopTask (long dt, short ref, long a1, long a2, long a3)
{
  gStopflag= 1;
}
```

In the above example you may have noticed that two different functions, MidiTask and MidiDTask, are used to schedule function calls.

Function calls scheduled with MidiTask are automatically executed in real time at interrupt level by MidiShare. These functions must be very fast (< 1ms) and must not call any slow or non-reentrant Operating System functions.

Function calls scheduled with MidiDTask behaves differently. They are not executed automatically but stored in a special list of pending tasks. The application can periodically (for example in its main event loop) execute pending tasks by calling MidiExec1DTask as in this example. In this case slow or non-reentrant functions can be safely called within the scheduled function.

In both cases the scheduled functions can use global variables, as the A5 register of the application is automatically restored by MidiShare before calling the scheduled function.

Example 5 : real-time event processing

In this example we see how to install a receive alarm to process incoming events in real time. The processing is very simple, received events are delayed accordingly to their Midi channel (delay = channel number * 100ms).

Listing 5

```
#include <stdio.h>
#include <stdlib.h>
#include <MidiShare.h>


pascal void DelayRcvAlarm (short ref);

void main()
{

  short  myRefNum;
  short  i;
  long dt;

  /* Open the MidiShare session                                    */
  myRefNum = MidiOpen("\pExample5");

  /* Install the receive alarm                                     */
  MidiSetRcvAlarm( myRefNum, DelayRcvAlarm );

  /* Connect the application to MidiShare physical I/Os            */
  /* the 3 arguments are the reference number of the source        */
  /* the reference number of the destination
  MidiConnect (myRefNum, 0, 1);
  MidiConnect (0, myRefNum, 1);

  printf("Now Midi events are delayed\n");
  printf(" <type the ENTER key to stop the program>\n");

  getc(stdin);

  printf("\nStopped\n");

  /* close the MidiShare session                                   */
  MidiClose(myRefNum);
}


pascal void DelayRcvAlarm (short ref)
{
  MidiEvPtr e;

  while ( e = MidiGetEv(ref) ) {
    Date(e) += Chan(e)*100;
    MidiSend(ref, e);
  }
}
```

The receive alarm is called at interrupt level every time new events are received by the application. The argument passed to the receive alarm is the reference number of the application.

Example 6 : a small sequencer

This example implements a small sequencer able to record and play back MIDI events.

Listing 6

```
#include <stdio.h>
#include <stdlib.h>
#include <MidiShare.h>

long     gStopFlag;
MidiSeqPtr  gSequence;

pascal
void record (short aRefNum);

pascal
void play (long time, short refnum, long nextEv, long unused1, long unused2);

void main ()
{
  short     myRefNum;

  /* OPEN A MIDISHARE SESSION */
  myRefNum = MidiOpen("\pExample6");


  printf("type <ENTER> to start recording\n");
  getc(stdin);

  /* START RECORDING */
  gSequence = MidiNewSeq();       /* sequence for recording     */
  MidiSetRcvAlarm(myRefNum, record);    /* set rcv alarm for rec      */
  MidiConnect(0, myRefNum, true); /* connect input              */

  printf("\n\n Now recording.... \n");
  printf("(type <ENTER> to stop recording and play back)\n");
  getc(stdin);

  /* PLAY BACK */
  MidiConnect(0, myRefNum, false);/* disconnect input           */
  MidiConnect(myRefNum, 0, true); /* connect output       */
  MidiSetRcvAlarm(myRefNum, 0);         /* remove the rcv alarm */
  play(MidiGetTime(), myRefNum, (long) FirstEv(gSequence), 0, 0);

  printf("\n\nNow playing back....\n");
  printf("(type <ENTER> to stop and exit program)\n");
  getc(stdin);

  /* STOP PLAY BACK AND EXIT */
  gStopFlag = 1;            /* set to stop playing  */
  MidiFreeSeq(gSequence);               /* free the sequence    */
  MidiClose(myRefNum);            /* close the session    */
}

/* THE RECEIVE ALARM TO RECORD EVENTS */
pascal void record (short ref)
{
  MidiEvPtr e;

  while (e = MidiGetEv(ref)) {          /* get received events  */
    MidiAddSeq(gSequence,e);            /* store into the seq   */
  }
}
```

```
/* THE TASK TO PLAY BACK EVENTS */
pascal void play (     long time, short refnum, long nextEv,
                  long unused1, long unused2)
{
  long        date;
  MidiEvPtr   e;

  /* If we have not been stopped  */
  /* and still have events to play*/
  if (!gStopFlag && nextEv) {
    e = (MidiEvPtr) nextEv;
    date = Date(e);

    /* for all the events at the same date    */
    while (e && Date(e) == date)  {
      MidiSendIm(refnum, MidiCopyEv(e));     /* Send a copy    */
      e = e->link;                           /* Go to next one */
    }

    /* If we still have events to play in future?  */
    if (e) {
      /* schedule the play task again  */
      MidiTask(play, Date(e)-date+time, refnum, (long)e, 0, 0);
    }
  }
}
```

Reference

MidiShare Events

The MidiShare communication is based on *events*. An event is a time-stamped data object that can be sent or received by client applications.

Typology

The table 1 below represents the different types of events handled by MidiShare. This typology contains all of the standard Midi messages, plus specific messages such as the typeNote corresponding to a note with its duration, typeStream which corresponds to a series of arbitrary bytes (possibly including data and status codes), and typePrivate which is used for an applications private messages.

All these codes may be used by the MidiNewEv function to allocate an event of the desirable type and are accessed by the evType field of an event.

Table 1

Name	Code	Comment
typeNote	0	pitch, velocity and duration (16bit)
typeKeyOn	1	pitch and velocity
typeKeyOff	2	pitch and velocity
typeKeyPress	3	pitch and after touch pressure
typeCtrlChange	4	control and value
typeProgChange	5	program change
typeChanPress	6	channel after touch pressure
typePitchWheel	7	Lsb and Msb
typeSongPos	8	Lsb and Msb
typeSongSel	9	song selection
typeClock	10	-
typeStart	11	-
typeContinue	12	-
typeStop	13	-
typeTune	14	-
typeActiveSens	15	-
typeReset	16	-
typeSysEx	17	data1..dataN
typeStream	18	byte1..byteN
typePrivate	19..127	arg1, arg2, arg3, arg4
typeProcess	128	arg1, arg2, arg3, arg4
typeDProcess	129	arg1, arg2, arg3, arg4
typeQFrame	130	msg type (0..7) and value
TypeCtrl14b	131	
TypeNonRegParam	132	
TypeRegParam	133	
TypeSeqNum	134	extended types from MidiFile 1.0
TypeText	135	
TypeCopyright	136	
TypeSeqName	137	
TypeInstrName	138	
TypeLyric	139	
TypeMarker	140	
TypeCuePoint	141	
TypeChanPrefix	142	
TypeEndTrack	143	
TypeTempo	144	
TypeSMPTEOffset	145	
TypeTimeSign	146	
TypeKeySign	147	
TypeSpecific	148	
TypeReserved	149..254	
TypeDead	255	-

Events Internal structure

The MidiShare memory management is organised around fixed-sized cells (16 bytes). All the events are composed of a header cell that may be followed by one or more extension cells. Figure 1 describes the different fields forming the common cell :

The `Link` field is used internally for linking cells.

The Date field contains the falling date of the event (from 0 to 2^{31} - 1).

The `refNum` field contains the application reference of the event sender.

The `evType` field contains the type of the event.

The `Port` field contains the destination MIDI port of the event.

The `Chan` field contains the MIDI channel of the event.

These six fields are always present and always have the same meaning, whatever the type of the event, and they can be accessed directly. The following Info part of an event contains special fields who's purpose depends on the event type. In some cases, the Info part contains a pointer to one or several extension cells. Direct access to these special fields is possible provided one takes into account the different memory structures. Otherwise the special functions `MidiGetField` and `MidiSetField` can be used, these hide the internal event structure and allow direct access to the special fields by specifying an index between 0 and `MidiCountFields() - 1`.

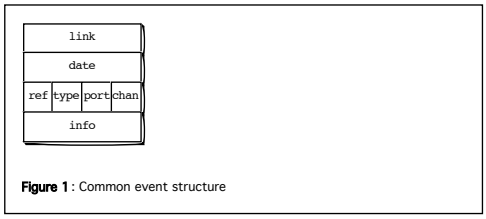


Figure 1 : Common event structure

Midi messages with 0, 1 or 2 data bytes, use only one cell, as shown in figure 2. These two supplementary fields are accessible by the `MidiGetField` and `MidiSetField` functions with index 0 and 1.

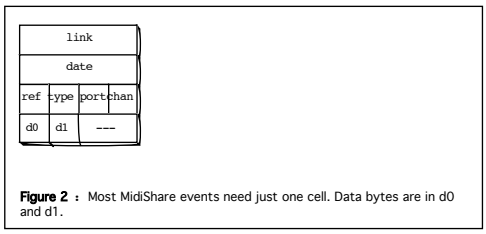


Figure 2 : Most MidiShare events need just one cell. Data bytes are in d0 and d1.

Notes (figure 3) have three more fields at their disposal : 0, 1 and 2 for pitch, velocity and duration. The access functions `MidiGetField` and `MidiSetField` automatically selects the 8, 16 or 32 bit fields.

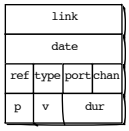


Figure 3 : MidiShare notes are events with pitch, velocity and a 16-bits duration

System Exclusive type messages or Stream type messages include variable number of fields. They use the structure described on figure 4, built with elementary cells linked one to another. The `MidiGetField` and `MidiSetField` functions are able to follow the links giving access to data. The `MidiAddField` function allows the addition of fields at the tail of the message.

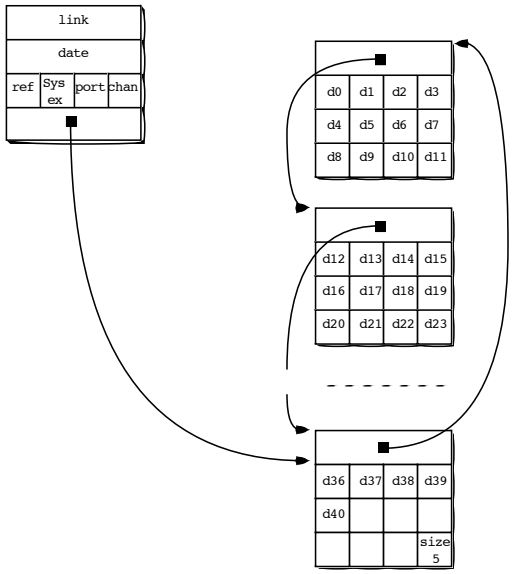


Figure 4 : Events of variable length (here a system exclusive) use multiple cells linked together.

Private or internal type events need the use of one extension cell. They are composed of four 32 bit fields (from 0 to 3) being able to contain any information left to the choice of the application.

Midi Error Codes

Table 2 List of the error codes returned by some MidiShare functions.

Name	Code	Comment
`MIDIerrSpace`	-1	No more space available
`MIDIerrRefNum`	-2	Bad reference number
`MIDIerrBadType`	-3	Bad type of event
`MIDIerrIndex`	-4	Wrong field index of access to an event

Table 3 List of the global system error codes

Name	Code	Comment
`MIDInoErr`	0	No error
`MIDIerrDriverLoad`	1	Failure in loading a driver
`MIDIerrTime`	2	Failure in opening time interrupts

Midi Change Codes

When an application needs to know about context modifications, for example the opening and closing of applications, opening and closing of midi ports and changes in connections between applications, it can install an ApplAlarm (see MidiSetApplAlarm). This ApplAlarm function is then called by MidiShare every time a context modification occurs and it is passed a 32-bits code describing the modification. The hi 16-bits part of this code is the refNum of the application involved in the context modification, the low 16-bits part describe the type of change as listed below.

Table 4 List of the change codes sent by MidiShare to ApplAlarm

Name	Code Mac	Code Atari	Comment
`MIDIOpenAppl`	1	-	A new application is opened
`MIDICloseAppl`	2	-	An application is closed
`MIDIChgName`	3	-	An application name is changed
`MIDIChgConnect`	4	-	A connection is changed
`MIDIOpenModem`	5	-	The Modem port is opened
`MIDICloseModem`	6	-	The Modem Port is closed
`MIDIOpenPrinter`	7	-	The Printer port is opened
`MIDIClosePrinter`	8	-	The Printer Port is closed
`MIDISyncStart`	9	550	Start of synchronization
`MIDISyncStop`	10	551	End of synchronization
`MIDIChangeSync`	11	552	The synchronization mode is changed

MidiAddField

DESCRIPTION

Adds a field at the tail of an event of variable length (for example a System Exclusive or a Stream) and assigns to it the value transmitted as a parameter.

PROTOTYPE

```
C Atari        void          MidiAddField (e, v);
C Mac ANSI     pascal void   MidiAddField (MidiEvPtr e, long v);
Pascal Mac     procedure     MidiAddField (e:MidiEvPtr; v:longint);
```

ARGUMENTS

e : a MidiEvPtr, it is a pointer to the event to be modified.

v : a 32-bit integer, it is the value of the field to be added. This value is always a long for a purpose of uniformity, but it is internally translate to the right size (a byte in this case). The value of v is actually between 0 and 127 for a System Exclusive and between 0 and 255 for a Stream.

EXAMPLE 1 (ANSI C)

Creates the System Exclusive message "F0 67 18 05 F7"

```
MidiEvPtr e;

e = MidiNewEv (typeSysEx);
MidiAddField (e, 0x67L);
MidiAddField (e, 0x18L);
MidiAddField (e, 0x05L);
```

***Note** : the leading F0 byte and the tailing F7 byte are automatically added by MidiShare when the message is transmitted. They must not be added by the user.*

EXAMPLE 2 (ANSI C)

Creates the Stream message "F8 F0 67 F8 18 05 F7" that mixes two MidiClock messages (F8) into a System Exclusive.

```
MidiEvPtr e;
long   i;

e = MidiNewEv(typeStream);
MidiAddField (e, 0xF8L);
MidiAddField (e, 0xF0L);
MidiAddField (e, 0x67L);
MidiAddField (e, 0xF8L);
MidiAddField (e, 0x18L);
MidiAddField (e, 0x05L);
MidiAddField (e, 0xF7L);
```

***Note** : Streams are sent without any transformation (no running status, no check of coherence). They can be used for example to send a long system exclusive split into several chunks with a little delay between. They can also be used as in the example to mix real time messages in a long system exclusive for maintaining synchronization.*

EXAMPLE 3 (ANSI C)

Create a system exclusive message from an array of values:

```
char          tab[3] = {10, 20, 30};
MidiEvPtr   aSysEx;

MidiEvPtr Array2SysEx( short len, char* vect, short chan, short port )
{
  MidiEvPtr e;

  e = MidiNewEv( typeSysEx );                 /* a new, empty sysex   */
  Chan(e) = chan; Port(e) = port;       /* set destination info */
  while (len—) MidiAddField(e, *vect++);    /* append fields        */
  return e;
}

aSysEx = Array2SysEx(3, tab, 0, 0);
```

MidiAddSeq

DESCRIPTION

Inserts an event in to a sequence while maintaining the dates in time order.

PROTOTYPE

C Atari	`void MidiAddSeq (s, e);`
C Mac ANSI	`pascal void MidiAddSeq (MidiSeqPtr s, MidiEvPtr e);`
Pascal Mac	`procedure MidiAddSeq (s:MidiSeqPtr; e:MidiEvPtr);`

ARGUMENTS

s : a MidiSeqPtr, it is a pointer to the sequence to be modified.

e : a MidiEvPtr, it is a pointer to the event to be added.

EXAMPLE (ANSI C)

Creates a sequence of 10 midi clock every 250 ms.

```
MidiSeqPtr  s;
MidiEvPtr e;
long d;

s = MidiNewSeq();
for (d=0; d< 2500; d+=250)
{
  e = MidiNewEv (typeClock);
  Date(e) = d;
  MidiAddSeq (s, e);
}
```

***Note** : if you are concerned about application speed, you must realise that sequences are single linked lists of time ordered events, so it takes more time for MidiAddSeq to insert an event in the middle of a sequence than at either at the beginning or the end.*

DESCRIPTION

This function is an iteration. It allows an application apply a function to all the events of a sequence.

PROTOTYPE OF MIDIAPPLYSEQ

```
C Atari        void          MidiApplySeq (s, MyProc);
C Mac ANSI     pascal void   MidiApplySeq (MidiSeqPtr s, ApplyProcPtr MyProc);
Pascal Mac     procedure     MidiApplySeq (s:MidiSeqPtr; Myproc:ApplyProcPtr );
```

ARGUMENTS OF MIDIAPPLYSEQ

`s` : a `MidiSeqPtr`, is a pointer to the sequence to be browsed;

`MyProc` : a `ApplyProcPtr` is the address of the function to apply to each event of the sequence.

PROTOTYPE OF MYPROC

```
C Atari        void          MyProc (e);
C Mac ANSI     pascal void   MyProc (MidiEvPtr e);
Pascal Mac     procedure     MyProc (e:MidiEvPtr);
```

ARGUMENT OF MYPROC

e : a MidiEvPtr, is a pointer to the current event in the sequence.

EXAMPLE (ANSI C)

Transpose a sequence by one octave.

```
MidiSeqPtr  s;

void TransposeOctave (MidiEvPtr e)
{
  if ( EvType(e) == typeNote ||
       EvType(e) == typeKeyOn ||
       EvType(e) == typeKeyOff ||
       EvType(e) == typeKeyPress )
     {
       Pitch(e) += 12; /* normally one must check boundaries */
     }
}
....

MidiApplySeq(s, TransposeOctave);
```

Note for Mac users : *MidiShare was originally developed in Pascal on the Macintosh. Therefore, in C, all functions passed as arguments of a MidiShare function must be declared as Pascal. In the previous example, TransposeOctave should be declared as :*

```
pascal void TransposeOctave (MidiEvPtr e)
```

MidiAvailEv

DESCRIPTION

Gives a pointer to the first event at the head of the reception FIFO, without extracting it. MidiAvailEv can be used if an application wants to test the first event in its reception FIFO, without processing it.

PROTOTYPE

C Atari
```
MidiEvPtr            MidiAvailEv (refnum) ;
```
C Mac ANSI
```
pascal MidiEvPtr  MidiAvailEv (short refnum) ;
```
Pascal Mac
```
Function          MidiAvailEv (refnum: integer): MidiEvPtr;
```

ARGUMENTS

`refNum` : a 16-bit integer, it is the reference number of the application.

RESULT

The result is a MidiEvPtr, a pointer to the first event in the reception FIFO, or NIL if the reception FIFO is empty.

EXAMPLE (ANSI C)

A function that calculates for how long events have been waiting in the reception FIFO

```
long CalculateWaitTime (short refNum)
{
  MidiEvPtr e;

  if (e = MidiAvailEv (refNum))
    return MidiGetTime() - Date(e);
  else
    return 0;
}
```

***Note** : as the event is still in the reception FIFO it must not be destroyed or transmitted. It can just be tested or duplicated.*

DESCRIPTION

Initiates a time delayed function call. When the calling date falls, the call is automatically realized by MidiShare under interrupt. MidiCall is presented here for historical reasons, and MidiTask is a better choice of code for completing this task.

PROTOTYPE OF MIDICALL

C Atari
```
void          MidiCall (MyProc, date, refNum, a1, a2, a3);
```
C Mac ANSI
```
pascal void   MidiCall (TaskPtr MyProc, long date, short refNum,
                        long a1, long a2, long a3);
```
Pascal Mac
```
Procedure     MidiCall (MyProc:TaskPtr; date:longint;
                        refNum:integer; a1,a2,a3: longint);
```

ARGUMENTS OF MIDICALL

`MyProc` : a TaskPtr, is the address of the function to be called.

`date` : a 32-bit integer, is the date at which this call is scheduled.

`refNum` : a 16-bit integer, it is the reference number of the application.

`a1,a2,a3` : are 32-bit integers left at the user's disposal, as arguments of MyProc

PROTOTYPE OF MYPROC

C Atari
```
void          MyProc (date, refNum, a1, a2, a3);
```
C Mac ANSI
```
pascal void   MyProc (long date, short refNum, long a1, long a2,
                                                        long a3);
```
Pascal Mac
```
procedure     MyProc (date:longint; refNum:integer; a1,a2,a3:
                                                        longint);
```

ARGUMENT OF MYPROC

`date` : a 32-bit integer, is the date of the call .

`refNum` : a 16-bit integer, is the reference number of the application.

`a1,a2,a3` : are 32-bit integers that can be used by the application.

EXAMPLE (ANSI C)

Send periodically (every 10 ms), a MidiClock message for 30 seconds.

```
void MyClock (long date, short refNum, long delay, long limit, long a3)
{
  if (date < limit)
  {
     MidiSendIm (refNum, MidiNewEv(typeClock));
     MidiCall (MyClock, date+delay, refNum, delay, limit, a3);
  }
}
...........
long d;
...........
d = MidiGetTime();
MyClock (d, myRefNum, 10L, d+30000L, 0L); /* Start now the clock for 30s */
```

***Note** : As this call occurs under interruptions, a few precautions should be taken when using it, for example not invoking non-reentrant routines of the Operating System (such as the Memory Manager on the Macintosh for example). However, most of the MidiShare functions are reentrant, they can be used safely under interruption.*

***Note for Mac users :** MidiShare was originally developed for Pascal on the Macintosh. Therefore, in C, all functions passed as arguments for a MidiShare function should be declared as Pascal. In the previous example, MyClock should be declared as :*

```
pascal void MyClock(long date, short refNum, long delay,
                    long limit, long a3);
```

MidiClearSeq

DESCRIPTION

Frees the content of a sequence. MidiClearSeq de-allocates all the events of the given sequence, consequently this sequence becomes empty.

PROTOTYPE

C Atari
```
void          MidiClearSeq (s);
```
C Mac ANSI
```
pascal void   MidiClearSeq (MidiSeqPtr s);
```
Pascal Mac
```
procedure     MidiClearSeq (s:MidiSeqPtr);
```

ARGUMENTS

s : a MidiSeqPtr,is a pointer on a sequence whose events are to be freed.

EXAMPLE (ANSI C)

Suppress all but the first event of a sequence.

```
void ClearAllButFirst (MidiSeqPtr s)
{
  MidiEvPtr e;

  if (s && First(s))              /* Check a non empty sequence */
  {
    e = MidiCopyEv(First(s));     /* make a copy of the first event   */
    MidiClearSeq(s);              /* clear the content of the sequence
  */
    MidiAddSeq(s, e);       /* add the event to the empty sequence    */
  }
}
```

***Note** : a sequence consist of a header of 4 pointers. The first one points to the first event of the sequence. The second one points to the last event. The other two pointers are reserved for future extensions and must be NIL. In an empty sequence, the pointers to the first and last events are NIL.*

MidiClose

DESCRIPTION

This is used for closing of a MidiShare application. Every opening of MidiShare with MidiOpen must be matched by a call to MidiClose, so that MidiShare can keep track of active applications and release the corresponding internal data structures. All the MidiShare applications owning a "context alarm" will be informed of this closing.

PROTOTYPE

C Atari	`void MidiClose (refNum);`
C Mac ANSI	`pascal void MidiClose (short refNum);`
Pascal Mac	`procedure MidiClose (refNum:integer);`

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application, given by the corresponding MidiOpen.

EXAMPLE (ANSI C)

A do-nothing MidiShare application.

```
#include MidiShare.h
#include <stdio.h>

short  myRefNum;

main()
{
  if ( ! MidiShare() ) exit(1);         /* Check MidiShare loaded     */
  myRefNum = MidiOpen("Sample");        /* Ask for a reference number*/
  if ( myRefNum < 1 ) exit(1);    /* Check MidiOpen success     */
                                  /* Print the reference number */
  printf( "refNum : %i \n", myRefNum);
  MidiClose(myRefNum);                  /* And close      */
}
```

***Note** : MidiClose takes care of deleting all the connections of the concerned application. Therefore if an application sends some Midi events and without delay, does a MidiClose, these sent events will probably not be actually transmitted. They just go back to the MidiShare Memory Manager.*

***Note for Mac users :** MidiShare was originally developed for Pascal on the Macintosh. Therefore, in C, all strings passed as arguments of a MidiShare function must be Pascal strings. In the previous example, one must write :*

```
myRefNum = MidiOpen("\pSample");
```

DESCRIPTION

Connects or disconnects two applications. The MidiConnect function allows the switching on or off of a connection between a source application and a destination application. There is no restrictions in the establishing of these connections, an application can be the source or destination of as many other applications as you wish. Loops are permitted.

PROTOTYPE

```
C Atari       void          MidiConnect (src,dest,state);
C Mac ANSI    pascal void   MidiConnect (short src,short dest,boolean state);
Pascal Mac    Procedure     MidiConnect (src, dest:integer; state:boolean);
```

ARGUMENTS

src : a 16-bit integer, is the reference number of the source application.

dest : a 16-bit integer, is the reference number of the destination application.

state : a boolean, indicates if a connection must be switched on (True) or off (False).

EXAMPLE (ANSI C)

Open a MidiShare application and connect it to the physical Midi inputs and outputs.

```
#include MidiShare.h
#define PHYSMIDI_IO 0       /* The MidiShare physical Midi I/O ports*/

Main()
{
  short  myRefNum;

  myRefNum = MidiOpen("MidiSample");
  MidiConnect (PHYSMIDI_IO, myRefNum, TRUE);/* to receive events    */
  MidiConnect (myRefNum, PHYSMIDI_IO, TRUE);/* to transmit events   */

  /* ....... */

  MidiClose(myRefNum);
}
```

***Note** : the physical Midi inputs and outputs are represented by the pseudo application called "MidiShare" with a reference number of 0 (zero). This pseudo application is automatically created when MidiShare wakes up at the very first MidiOpen.*

MidiCopyEv

DESCRIPTION

Duplicates an event, taking into account the structure of the event. It can be used to copy any type of events, from simple notes to large system exclusives.

PROTOTYPE

C Atari
```
MidiEvPtr          MidiCopyEv (e);
```
C Mac ANSI
```
pascal MidiEvPtrMidiCopyEv (MidiEvPtr e);
```
Pascal Mac
```
Function           MidiCopyEv (e: MidiEvPtr): MidiEvPtr;
```

ARGUMENTS

e : a MidiEvPtr, is a pointer to the event to be copied.

RESULT

The result is a MidiEvPtr, a pointer to the copy if the operation was successful. The result is NIL if MidiShare was not able to allocate enough memory space for the copy.

EXAMPLE (ANSI C)

Send from now, 10 times an identical note of pitch 60 every 250 ms.

```
MidiEvPtr e;
short     myRefNum;
long   d;
short     i;
........

e = MidiNewEv (typeNote);    /* create template note */
Pitch(e)= 60;          /* fill up its parameters     */
Vel(e) = 80;
Dur(e) = 250;
Chan(e)   = 0;
Port(e) = 0;

for (d=MidiGetTime (), i=0; i<10; i++, d+=250)
                /* send the 10 copies of the template */
  MidiSendAt (myRefNum, MidiCopyEv(e), d);
MidiFreeEv(e);   /* and free the template      */
```

***Note** : it is very important that, once an event is sent, it must <u>never</u> be used again by the application. Therefore, if an application needs to send the same event several times duplicate copies must be used.*

DESCRIPTION

Gives the number of Midi applications currently active.

PROTOTYPE

C Atari
```
short         MidiCountAppls ();
```
C Mac ANSI
```
pascal short  MidiCountAppls ();
```
Pascal Mac
```
Function      MidiCountAppls : integer;
```

RESULT

The result is a 16-bit integer, the number of currently opened Midi applications.

EXAMPLE (ANSI C)

Print the name of all the actives MidiShare applications

```
void PrintApplNames(void)
{
  short  ref;
  short  i;

  printf( "List of MidiShare applications :\n" );
  for( i = 1; i <= MidiCountAppls(); ++i )
  {
     ref = MidiGetIndAppl(i);
     printf("%i : %s \n", ref, MidiGetName( ref ) );
  }
}
```

Note for Mac users : *MidiShare was originally developed for Pascal on the Macintosh. Therefore, in C, the result of MidiGetName is a Pascal string that must be converted to a C string before being printed.*

MidiCountDTasks

DESCRIPTION

Returns the number of time delayed tasks waiting in the list of an application. Delayed tasks are function calls that where scheduled with MidiDTask and that are now ready to be executed in the DTasksFifo of the application.

PROTOTYPE

```
C Atari        long          MidiCountDTasks (refNum);
C Mac ANSI     pascal long   MidiCountDTasks (short refNum);
Pascal Mac     Function      MidiCountDTasks (refNum: integer):longint;
```

ARGUMENTS

refNum : a 16-bit integer, the reference number of the application.

RESULT

The result is a 32-bit integer, the number of waiting DTasks.

EXAMPLE (ANSI C)

Execute the waiting DTasks of a MidiShare application.

```
void ExecuteAllDTasks(short refNum)
{
  long n;

  for (n=MidiCountDTasks(refNum), n>0; n—)
  {
    MidiExec1DTask(refNum);
  }
}
```

***Note** : A typical application might execute the queued tasks from with in its 'main' loop, and since it is not under interruption, operating system calls may be performed.*
However this does mean that the execution time of these functions cannot be accurately predicted.

DESCRIPTION

Gives the number of events on wait into the reception FIFO of the application.

PROTOTYPE

C Atari	`long MidiCountEvs (refnum);`
C Mac ANSI	`pascal long MidiCountEvs (short refnum);`
Pascal Mac	`Function MidiCountEvs (refnum: integer) : longint;`

ARGUMENTS

refNum : a 16-bit integer, the reference number of the application.

RESULT

The result is a 32-bit integer, the number of waiting events in the reception FIFO.

EXAMPLE (ANSI C)

A receive alarm that processes all the received events by adding to their date a one second delay.

```
void OneSecDelay (short refNum)
{
  MidiEvPtr e;
  long n;

  for ( n = MidiCountEvs(refNum); n > 0; —n )
  {
    e = MidiGetEv (refNum);       /* Get an event from the FIFO */
    Date(e) += 1000;              /* Add 1000 ms to its date    */
    MidiSend(refNum,e);           /* Then send the event        */
  }
}
......
/* Activate the receive alarm */
MidiSetRcvAlarm(myRefNum,OneSecDelay);
```

***Note** : such a function can be called repeatedly in the main event loop of the application, but for really accurate time control, it must be installed as a receive alarm with MidiSetRcvAlarm.*

***Note for Mac users :** MidiShare was originally developed for Pascal on the Macintosh. Therefore, in C, all procedures passed as arguments of a MidiShare function must be declared as Pascal. In the previous example, OneSecDelay must be declared as :*

```
pascal void OneSecDelay (short refNum)
```

MidiCountFields

DESCRIPTION

Gives the number of fields of an event.

PROTOTYPE

C Atari	`long MidiCountFields (e);`
C Mac ANSI	`pascal long MidiCountFields (MidiEvPtr e);`
Pascal Mac	`Function MidiCountFields (e: MidiEvPtr): longint;`

ARGUMENTS

e : a MidiEvPtr, a pointer to the concerned event.

RESULT

The result is a 32-bit integer, the number of fields of the event.

EXAMPLE (ANSI C)

An universal method for printing of a MidiShare event.

```
void PrintEv(MidiEvPtr e)
{
  long i, n;
  n = MidiCountFields(e);
  printf( "Event %x content :\n", e );
  printf( " link : %x\n", Link(e) );
  printf( " date : %i\n", Date(e) );
  printf( " type : %i\n", EvType(e) );
  printf( "  ref : %i\n", RefNum(e) );
  printf( " port : %i\n", Port(e) );
  printf( " chan : %i\n", Chan(e) );
  printf( " %i fields : ( ", n );
  for(i=0; i<n; ++i) printf("%i ",MidiGetField(e,i) );
  printf( ")\n" );
}
```

***Note** : MidiShare events carry two kinds of information, common information, like date, type, channel, port etc. and specific information that depend of the type of event. Fields allow a uniform method of access to this specific information. Some events have fixed number of fields (for example notes have three fields : pitch (8-bit), velocity (8-bit) and duration (16-bit)). Some others, like system exclusive have a variable number of fields.*

DESCRIPTION

As with MidiTask, MidiDTask allows an application to initiate a time delayed function call, but unlike MidiTask, the call is not achieved under interruption as soon as falling time is due. The address of the routine to be executed and the corresponding arguments are stored in a special buffer. The application can then process these waiting tasks, one by one, using to MidiExec1DTask.

PROTOTYPE OF MIDIDTASK

C Atari
```
MidiEvPtr          MidiDTask (MyProc, date, refNum, a1, a2, a3);
```
C Mac ANSI
```
pascal MidiEvPtrMidiDTask (ProcPtr MyProc, long date,
                           short refNum,long a1,long a2,long a3);
```
Pascal Mac
```
Function           MidiDTask (MyProc:ProcPtr; date:longint;
                           refNum:integer;a1,a2,a3:longint):MidiEvPtr;
```

ARGUMENTS OF MIDIDTASK

MyProc : is the address of the function to be called.

date : a 32-bit integer, it is the date at which this call is scheduled.

refNum : a 16-bit integer, it is the reference number of the application.

a1,a2,a3 : are 32-bit integers left at the user's disposal, as arguments to MyProc

RESULT OF MIDIDTASK

The result, a MidiEvPtr, is a pointer to a typeDProcess MidiShare event. The result is NIL if MidiShare runs out of memory.

PROTOTYPE OF MYPROC

C Atari
```
void          MyProc (date, refNum, a1, a2, a3);
```
C Mac ANSI
```
pascal void   MyProc (long date,short refNum,long a1,long a2,
                                                 long a3);
```
Pascal Mac
```
procedure     MyProc (date:longint; refNum:integer;
                                                 a1,a2,a3: longint);
```

ARGUMENT OF MYPROC

date : a 32-bit integer, it is the date of the call .

refNum : a 16-bit integer, it is the reference number of the application.

a1,a2,a3 : are 32-bit integers that can be freely used.

EXAMPLE (ANSI C)

```
Schedule Action() procedure call 1000 ms ahead.

MidiEvPtr myDTask;

MyDTask = MidiDTask( Action, MidiGetTime()+1000, myRefNum, a1, a2, a3);
```

***Note** : The result, in myDTask, can be used to test the success of MidiDTask. It can also be used by MidiForgetTask to try to "forget" a scheduled task before it happens.*

DESCRIPTION

Processes the first time delayed task on wait in the applications queue. The time delayed tasks scheduled by MidiDTask are not processed at a given time, but instead must be called using MidiExec1DTask which executes the first task in its queue.

PROTOTYPE

C Atari
```
void          MidiExec1DTask (refnum);
```
C Mac ANSI
```
pascal void   MidiExec1DTask (short refnum);
```
Pascal Mac
```
procedure     MidiExec1DTask (refnum: integer);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

EXAMPLE (ANSI C)

Execute the waiting DTasks of a MidiShare application.

```
void ExecuteAllDTasks(short refNum)
{
  long n;

  for (n=MidiCountDTasks(refNum), n>0; n—)
  {
    MidiExec1DTask(refNum);
  }
}
```

***Note** : Generally this function is called from within an applications 'main' function, and as this is not under interruption it is possible to perform operating system calls.*

MidiExt2IntTime

DESCRIPTION

Converts an external time in millisecond to the value of an internal time. The convertion is made by subtracting the current offset between internal and external time.

PROTOTYPES

C Atari
```
long          MidiExt2IntTime (time);
```
C Mac ANSI
```
pascal long   MidiExt2IntTime (long time)
```
Pascal Mac
```
Function      MidiExt2IntTime (time : longint): longint;
```

ARGUMENTS

time : a 32-bits time in milliseconds

RESULT

the corresponding internal time, a 32-bits value in milliseconds.

Note: *When MidiShare is locked we have the following equivalence :*

```
MidiExt2IntTime( MidiGetExtTime() ) == MidiGetTime ()
```

We have also :

```
TSyncInfo   myInfo;
MidiGetSyncInfo(&myInfo);
MidiExt2IntTime(x) == x - myInfo.syncOffset
```

DESCRIPTION

Flushes all the waiting DTasks in the application DTask list.

PROTOTYPE

C Atari
```
void          MidiFlushDTasks (refnum);
```
C Mac ANSI
```
pascal void   MidiFlushDTasks (short refnum);
```
Pascal Mac
```
procedure     MidiFlushDTasks (refnum: integer);
```

ARGUMENTS

refNum : a 16-bit integer, is the reference number of the application.

EXAMPLE (ANSI C)

Flushes all the waiting DTasks in the application DTask list.

```
short  myRefNum;

.....

MidiFlushDTasks (myRefNum);
```

MidiFlushEvs

DESCRIPTION

Flushes all the waiting events in the reception FIFO of the application.

PROTOTYPE

C Atari
```
void          MidiFlushEvs (refNum);
```
C Mac ANSI
```
pascal void   MidiFlushEvs (short refNum);
```
Pascal Mac
```
procedure     MidiFlushEvs (refNum : integer);
```

ARGUMENTS

refNum : a 16-bit integer, is the reference number of the application.

EXAMPLE (ANSI C)

Flushes all the waiting events in the application reception FIFO.

```
short  myRefNum;

.....

MidiFlushEvs (myRefNum);
```

DESCRIPTION

Tries to "forget" a previously scheduled Task or DTasks. This is a very powerful, but also dangerous function. An application must be sure that the task has not yet executed before calling MidiForgetTask.

PROTOTYPE

C Atari	`void MidiForgetTask (v);`
C Mac ANSI	`pascal void MidiForgetTask (MidiEvPtr *v);`
Pascal Mac	`procedure MidiForgetTask (var v: MidiEvPtr);`

ARGUMENTS

v : is the address of a variable pointing to a previously scheduled Task or DTask but not yet executed. The variable may also contain NIL. In this case MidiForgetTask does nothing.

SIDE EFFECT

The variable, which address is given in parameter, is set to NIL by MidiForgetTask.

EXAMPLE 1 (ANSI C)

Create an infinite periodic clock (every 250ms) and stop it with MidiForgetTask.

```
MidiEvPtr theClock;

void InfClock (long date,short refNum,long delay,long a2,long a3)
{
  MidiSendIm (refNum, MidiNewEv(typeClock));
  theClock = MidiTask (InfClock, date+delay,refNum,delay,a2,a3);
}
/* Start the clock    */
InfClock(MidiGetTime (), myRefNum, 250L, 0L, 0L);
.........                    /* Wait some time */
MidiForgetTask(&theClock);/* And forget it       */
```

EXAMPLE 2 (ANSI C)

In the previous example theClock always point to a valid task because InfClock never stop by itself. If the task may decide to stop itself, it must set the pointer to NIL in order to avoid to forget an invalid task.

```
MidiEvPtr theClock;

void CountClock (long date, short refNum, long delay,long count, long a3)
{
  if (count > 0)
  {
     MidiSendIm (refNum, MidiNewEv(typeClock));
     theClock = MidiTask (CountClock, date+delay, refNum, delay,
                          count-1, a3);
```

```
  } else {
     theClock = NIL;          /* here the task decide to stop itself */
                        /* so set the pointer to NIL  */
  }
}

/* Start 100  clocks  */
CountClock(MidiGetTime (), myRefNum, 250L, 100L, 0L);
.........                    /* Wait some time */
MidiForgetTask(&theClock);      /* And forget it        */
```

If MidiForgetTask happens before the end of the 100 clocks, theClock points to a valid task and MidiForgetTask(&theClock) is safe. If MidiForgetTask happens after the end of the 100 clocks, theClock contains NIL and MidiForgetTask(&theClock) is safe and will do nothing.

DESCRIPTION

Frees a cell allocated by MidiNewCell function. This is the lowest level command for accessing the MidiShare Memory Manager. One must be sure to use MidiFreeCell on an individual cell allocated with MidiNewCell and not on complete MidiShare events. Not doing so may result in the lose of cells.

PROTOTYPE

C Atari
```
void          MidiFreeCell (c);
```
C Mac ANSI
```
pascal void   MidiFreeCell (MidiEvPtr c);
```
Pascal Mac
```
procedure     MidiFreeCell (c: MidiEvPtr);
```

ARGUMENTS

c : a MidiEvPtr, a pointer to a basic cell of 16 bytes.

EXAMPLE (ANSI C)

Free a cell previously allocated.

```
MidiEvPtr aCell;

aCell = MidiNewCell();

....

MidiFreeCell( aCell );
```

***Note** : Cells allocated with MidiNewCell must be freed with MidiFreeCell and not with MidiFreeEv.*

MidiFreeEv

DESCRIPTION

Frees a MidiShare event allocated with MidiNewEv. MidiFreeEv takes into account the event structure by checking the events type. For this reason, MidiFreeEv must not be used on cell allocated with MidiNewCell.

PROTOTYPE

C Atari	`void MidiFreeEv (e);`
C Mac ANSI	`pascal void MidiFreeEv (MidiEvPtr e);`
Pascal Mac	`procedure MidiFreeEv (e: MidiEvPtr);`

ARGUMENTS

e : a MidiEvPtr, it is a pointer to a MidiShare event.

EXAMPLE (ANSI C)

A receive alarm that delete all the received events.

```
short  myRefNum;
.....

void DeleteAll( short refNum )
{
  MidiEvPtr e;
  long n;

  for ( n = MidiCountEvs(refNum); n > 0; –n )
  {
    e = MidiGetEv ( refNum );     /* Get an event from the FIFO */
    MidiFreeEv( e );      /* Then free it                       */
  }
}
......
/* Activate the receive alarm     */
MidiSetRcvAlarm( myRefNum, DeleteAll );
```

***Note** : For this example it would be simpler and faster to use MidiFlushEvs to achieve the same result.*

***Note for Mac users :** MidiShare was originally developed for Pascal on the Macintosh. Therefore, in C, all function passed as arguments of a MidiShare function must be declared as Pascal. In the previous example, DeleteAll must be declared as :*

```
pascal void DeleteAll( short refNum )
```

MidiFreeSeq

DESCRIPTION

Frees a sequence and its content. MidiFreeSeq first de-allocates all the events of the given sequence and then the sequence header itself.

PROTOTYPE

C Atari
```
void          MidiFreeSeq (s);
```
C Mac ANSI
```
pascal void   MidiFreeSeq (MidiSeqPtr s);
```
Pascal Mac
```
procedure     MidiFreeSeq (s:MidiSeqPtr);
```

ARGUMENTS

s : a MidiSeqPtr, is a pointer on a sequence to be freed.

EXAMPLE (ANSI C)

Frees a previously allocated sequence s.

```
MidiSeqPtr  s;

s = MidiNewSeq();
....
MidiFreeSeq(s);
```

***Note** : Once freed, s is no longer a valid pointer.*

MidiFreeSpace

DESCRIPTION

Returns the available free MidiShare event space. MidiFreeSpace allows to know at any time the number of cells remaining available from the MidiShare memory manager.

PROTOTYPE

C Atari	`long`	**`MidiFreeSpace`**`();`
C Mac ANSI	`pascal long`	**`MidiFreeSpace`**`(void);`
Pascal Mac	`Function`	**`MidiFreeSpace`** `: longint;`

ARGUMENTS

none

RESULT

The result is a 32-bit integer, the number of available free cells in the MidiShare memory manager.

EXAMPLE (ANSI C)

Print informations about MidiShare memory space.

```
void PrintMemInfo(void)
{
  printf("MidiShare memory :\n");
  printf(" free space  : %i cells\n", MidiFreeSpace());
  printf(" used space  : %i cells\n", MidiTotalSpace() - MidiFreeSpace());
  printf(" total space : %i cells\n", MidiTotalSpace());
}
```

***Note** : MidiFreeSpace inhibits all interrupts during its execution. If the remaining space is very large MidiFreeSpace can take a long time to execute and may cause overrun errors with fast incoming Midi data.*

MidiGetApplAlarm

DESCRIPTION

Returns the context alarm of an application. MidiGetAlarm allows to know the address of the context alarm function associated to the application. This alarm is automatically called by MidiShare to inform the application of all the changes that happen to the active Midi applications (name or connection changes, closing, opening, etc.)

PROTOTYPE OF MIDIGETAPPLALARM

```
C Atari        ApplAlarmPtr         MidiGetApplAlarm (refNum);
C Mac ANSI     pascal ApplAlarmPtr  MidiGetApplAlarm (short refNum);
Pascal Mac     Function             MidiGetApplAlarm (refNum: integer)
                                                     :ApplAlarmPtr;
```

ARGUMENTS OF MIDIGETAPPLALARM

refNum : a 16-bit integer, it is the reference number of the application.

RESULT

The result, a ApplAlarmPtr, is the address of the alarm routine or NIL if no such routine was installed.

PROTOTYPE OF AN APPLALARM ROUTINE

```
C Atari        void          MyApplAlarm (refNum,code);
C Mac ANSI     pascal void   MyApplAlarm (short refNum, long code);
Pascal Mac     procedure     MyApplAlarm (refNum: integer; code:longint);
```

ARGUMENTS OF AN APPLALARM ROUTINE

refNum : a 16-bit integer, it is the reference number of the application.

code : a 32-bit integer, the context modification code.

EXAMPLE (ANSI C)

Temporarily disables the applications context alarm.

```
ApplAlarmPtr p;
.....
p = MidiGetApplAlarm( myRefNum );
MidiSetApplAlarm( NIL);      /* Disable application context alarm*/
.....
MidiSetApplAlarm( p);        /* Restore application context alarm*/
```

MidiGetError

DESCRIPTION

A kernel operation may fail outside the call of the MidiShare API. These errors are global to the system and concern all the clients. MidiGetError gives the list of the system errors under the form of a bit field.

PROTOTYPE

C

```
long                MidiGetError () ;
```

RESULT

a 32 bit value. Possible errors are any combination of:

- MIDInoErr : no error
- MIDIerrDriverLoad : indicates a failure to load a driver
- MIDIerrTime : indicates a failure to activate time interrupts

***Note** : MidiGetError() is not available before MidiShare version 1.91.*

DESCRIPTION

Extracts the first event on in the reception FIFO. The received events, stored automatically by MidiShare in the application reception FIFO, can be picked up by successive calls to MidiGetEv function.

PROTOTYPE

C Atari
```
MidiEvPtr          MidiGetEv (refNum) ;
```
C Mac ANSI
```
pascal MidiEvPtrMidiGetEv (short refNum);
```
Pascal Mac
```
Function           MidiGetEv (refNum: integer) : MidiEvPtr;
```

ARGUMENTS

refNum : a 16-bit integer, is the reference number of the application.

RESULT

A MidiEvPtr, is a pointer to the first event in the reception FIFO, or NIL if the FIFO is empty. The event is extracted from the reception FIFO.

EXAMPLE (ANSI C)

A receive alarm that processes all the received events by adding to their date a one second delay.

```
void OneSecDelay (short refNum)
{
  MidiEvPtr e;
  long n;
  for ( n = MidiCountEvs(refNum); n > 0; –n )
  {
    e = MidiGetEv (refNum);     /* Get an event from the FIFO */
    Date(e) += 1000;            /* Add 1000 ms to its date          */
    MidiSend(refNum,e);         /* Then send the event              */
  }
}
.....
/* Activate the receive alarm     */
MidiSetRcvAlarm(myRefNum,OneSecDelay);
```

***Note** : such a function can be called repeatedly in the main event loop of the application, but for really accurate time control, it must be installed as a receive alarm with MidiSetRcvAlarm.*

***Note for Mac users :** MidiShare was originally developed for Pascal on the Macintosh. Therefore, in C, all functions passed as arguments of a MidiShare function must be declared as Pascal. In the previous example, OneSecDelay must be declared as:*

```
pascal void OneSecDelay (short refNum)
```

MidiGetExtTime

DESCRIPTION

Gives the current external time i.e. the position of the tape converted in milliseconds.

PROTOTYPES

C Atari	`long MidiGetExtTime ();`
C Mac ANSI	`pascal long MidiGetExtTime (void);`
Pascal Mac	`function MidiGetExtTime : longint;`

ARGUMENTS

none

EXAMPLE (ANSI C)

Gives the SMPTE current location of the tape.

```
TSyncInfo       myInfo;
TSmpteLocationmyLoc;


MidiGetSyncInfo(&myInfo);
MidiTime2Smpte( MidiGetExtTime(), myInfo.syncFormat, &myLoc);
```

Note

When the tape is stopped, MidiGetExtTime returns the stop position of the tape converted in milliseconds.

MidiGetField

DESCRIPTION

Gives the index field value of an event. Field index start from 0. Depending of the event type and field nature, the field format can be 8, 16 or 32-bit. MidiGetField deals with all the format conversion and the result is always a 32-bit integer.

PROTOTYPE

C Atari
```
long          MidiGetField (e, f);
```
C Mac ANSI
```
pascal long   MidiGetField (MidiEvPtr e, long f);
```
Pascal Mac
```
Function      MidiGetField (e: MidiEvPtr; f: longint): longint;
```

ARGUMENTS

e : a MidiEvPtr, it is a pointer to the event to be accessed.

f : a 32-bit integer, it is the field number to be read (numbered from 0).

RESULT

The result is a 32-bit integer, the value of the field. Fields are considered as unsigned.

EXAMPLE (ANSI C)

An universal method for printing of a MidiShare event.

```
void PrintEv(MidiEvPtr e)
{
  long i, n;

  n = MidiCountFields(e);
  printf( "Event %x content :\n", e );
  printf( " link : %x\n", Link(e) );
  printf( " date : %i\n", Date(e) );
  printf( " type : %i\n", EvType(e) );
  printf( "  ref : %i\n", RefNum(e) );
  printf( " port : %i\n", Port(e) );
  printf( " chan : %i\n", Chan(e) );
  printf( " %i fields : ( ", n );
  for(i=0; i<n; ++i) printf("%i ",MidiGetField(e,i) );
  printf( ")\n" );
}
```

***Note** : MidiShare events carry two kind of information : common information, like date, type, channel, port ..., and specific information that depend of the type of event. Fields allow a uniform method of access to these specific data's. Some events have fixed number of fields (for example notes have three fields : pitch (8-bit), velocity (8-bit) and duration (16-bit)). Some others, like system exclusives have a variable number of fields.*

MidiGetFilter

DESCRIPTION

Gives the associated filter of an application. Each application can select the events to be received by using a filter. The filtering process is local to the application and has no influence on the events received by other applications.

PROTOTYPE

C Atari
```
FilterPtr          MidiGetFilter (refNum);
```
C Mac ANSI
```
pascal FilterPtrMidiGetFilter (short refNum);
```
Pascal Mac
```
Function         MidiGetFilter (refNum: integer): FilterPtr;
```

ARGUMENTS

refNum : a 16-bit integer, the reference number of the application

RESULT

the result is a FilterPtr, a pointer to the filter associated to the application, or NIL if there is no such filter (in this case the application accepts any events)

EXAMPLE (ANSI C)

<< to be supplied >>

MidiGetIndAppl

DESCRIPTION

Gives the reference of number of an application from is order number. The MidiGetIndAppl function allows to know the reference number of any application by giving its order number (a number between 1 and MidiCountAppls()).

PROTOTYPE

C Atari
```
short         MidiGetIndAppl (index);
```
C Mac ANSI
```
pascal short  MidiGetIndAppl (short index);
```
Pascal Mac
```
Function      MidiGetIndAppl (index: integer) : integer;
```

ARGUMENTS

index : a 16-bit integer, is the index number of an application between 1 and MidiCountAppls().

RESULT

The result is an application reference number or MIDIerrIndex if the index is out of range.

EXAMPLE (ANSI C)

Print the name of all the actives MidiShare applications

```
void PrintApplNames(void)
{
  short  ref;
  short  i;

  printf( "List of MidiShare applications :\n" );
  for( i = 1; i <= MidiCountAppls(); ++i )
  {
     ref = MidiGetIndAppl(i);
     printf("%i : %s \n", ref, MidiGetName( ref ) );
  }
}
```

Note for Mac users : *MidiShare was originally developed for Pascal on the Macintosh. Consequently, in C, the result of MidiGetName is a Pascal string that must be converted to a C string before being printed.*

MidiGetInfo

DESCRIPTION

Gives the content of a 32-bit field an application can use for any purpose. This field remains accessible by MidiGetInfo during alarms and interrupts. It can be used as a global context if necessary (for example for desk accessories on the Macintosh).

PROTOTYPE

C Atari
```
Ptr           MidiGetInfo (short refNum);
```
C Mac ANSI
```
pascal void*  MidiGetInfo (short refNum);
```
Pascal Mac
```
Function      MidiGetInfo (refNum: integer) : Ptr;
```

ARGUMENTS

refNum : a 16-bit integer, the reference number of the application

RESULT

The result is a 32-bit integer, the last value set by MidiSetInfo

EXAMPLE (ANSI C)

<< to be supplied >>

MidiGetName

DESCRIPTION

Gives the name of an application. Knowing an application reference number, it is possible to find its name using the MidiGetName function. It is also possible to find the reference number of an application via its name using the MidiGetNamedAppl function.

PROTOTYPE

C Atari
```
MidiName         MidiGetName (refNum);
```
C Mac ANSI
```
pascal MidiName MidiGetName (short refNum);
```
Pascal Mac
```
Function         MidiGetName (refNum: integer) : MidiName;
```

ARGUMENTS

refNum : a 16-bit integer, the reference number of the application

RESULT

The result is pointer on a character string representing the application name.

EXAMPLE (ANSI C)

Print the name of all the active MidiShare applications

```
void PrintApplNames(void)
{
  short  ref;
  short  i;

  printf( "List of MidiShare applications :\n" );
  for( i = 1; i <= MidiCountAppls(); ++i )
  {
     ref = MidiGetIndAppl(i);
     printf("%i : %s \n", ref, MidiGetName( ref ) );
  }
}
```

Note for Mac users : *MidiShare was originally developed for Pascal on the Macintosh. Consequently, in C, the result of MidiGetName is a Pascal string that must be converted to a C string before being printed.*

MidiGetNamedAppl

DESCRIPTION

Returns the reference number of an application. Knowing an application name, it is possible to find its reference number using the MidiGetNamedAppl function. It is also possible to find the name of an application via its reference number using the MidiGetName function.

PROTOTYPE

C Atari
```
short         MidiGetNamedAppl (MidiName name);
```
C Mac ANSI
```
pascal short  MidiGetNamedAppl (MidiName name);
```
Pascal Mac
```
Function      MidiGetNamedAppl (name: MidiName) : integer;
```

ARGUMENTS

name : the application name.

RESULT

The result is the reference number of the application.

EXAMPLE (ANSI C)

Find the reference number of the "MidiShare" pseudo-application.

```
short  r;

/* MidiShare reference is always 0 */
r = MidiGetNamedAppl("MidiShare");
```

Note for Mac users : *MidiShare was originally developed for Pascal on the Macintosh. Consequently, in C, all strings passed as arguments of a MidiShare function must be Pascal strings. In the previous example, one must write :*

```
MidiGetNamedAppl("\pMidiShare")
```

DESCRIPTION

Gives the Midi port state. The switching on or off of Midi ports is controlled by the MidiSetPortState and MidiGetPortState routines. These must be used with care since they affect all the applications.

PROTOTYPE

C Atari
```
Boolean          MidiGetPortState (port);
```
C Mac ANSI
```
pascal Boolean   MidiGetPortState (short port);
```
Pascal Mac
```
Function         MidiGetPortState (port: integer): boolean;
```

ARGUMENTS

port : a port number from 0 to 255.

RESULT

The result is true if the port is open or false if the port is closed.

EXAMPLE (ANSI C)

Print the state of all the Midi ports.

```
void PrintPortsState(void)
{
  short i;

  printf( "Midi ports state :\n");
  for( i = 0; i < 256; ++i )
  {
    if ( MidiGetPortState( i ) )
      printf(" %i is open \n", i );
    else
      printf(" %i is closed \n", i );
  }
}
```

Note : *On the Atari, there is just one Midi port (port 0), and on the Macintosh there are just two ports (port modem: 0, port printer: 1). But the future LAN version of MidiShare will allow up to 256 ports to be used. Therefore, applications must consider that 256 ports are available.*

MidiGetRcvAlarm

DESCRIPTION

Gives the address of a reception alarm of an application. The reception alarm informs of the presence of these new events in the reception FIFO. This alarm is always called under interruption. Therefore, it must not make use, either directly or indirectly, the Macintosh Memory Manager. However it can have a free access to all the MidiShare functions (except MidiOpen and MidiClose). It can also use the global variables of the application, because, before the call, MidiShare restores the global context register of the application.

PROTOTYPE

C Atari
```
RcvAlarmPtr          MidiGetRcvAlarm (refNum);
```
C Mac ANSI
```
pascal RcvAlarmPtr   MidiGetRcvAlarm (short refNum);
```
Pascal Mac
```
function             MidiGetRcvAlarm (refNum:integer)
                                              :RcvAlarmPtr;
```

ARGUMENTS

refNum : a 16-bit integer, the reference number of the application

RESULT

The result, a RcvAlarmPtr, it is the address of the receive alarm routine or NIL if no such routine where installed.

PROTOTYPE OF A RCVALARM ROUTINE

C Atari
```
void        MyRcvAlarm (refNum);
```
C Mac ANSI
```
pascal void MyRcvAlarm (short refNum);
```
Pascal Mac
```
procedure   MyRcvAlarm (refNum:integer);
```

ARGUMENT OF A RCVALARM ROUTINE

refNum : a 16-bit integer, it is the reference number of the application.

EXAMPLE (ANSI C)

Temporarily disable the application receive alarm.

```
RcvAlarmPtr p;
.....
p = MidiGetRcvAlarm( myRefNum );
MidiSetRcvAlarm( NIL );      /* Disable application receive alarm     */
.....
MidiSetRcvAlarm( p );        /* Restore application receive alarm     */
```

DESCRIPTION

Fills a TSyncInfo record with information about the current state of the MTC synchronisation.

PROTOTYPES

```
C Atari        void          MidiGetSyncInfo (p);
C Mac ANSI     pascal void   MidiGetSyncInfo (SyncInfoPtr p);
Pascal Mac     procedure     MidiGetSyncInfo (p: SyncInfoPtr);
```

ARGUMENTS

p: a SyncInfoPtr, a pointer to a TSyncInfo record

DESCRIPTION OF A TSYNCINFO RECORD

```
typedef struct TSyncInfo
{
long   time;     // the current MidiShare date (in milliseconds)
long   reenter;  // the current reentrancy count of the interrupt
                        handler
unsigned short syncMode;// the current synchronisation mode
                        as defined by MidiSetSyncMode
Byte syncLocked;        // the current synchronisation state
                        (0 : unlocked 1 : locked)
Byte syncPort; // the current synchronisation port
long syncStart;  // the date MidiShare started beeing locked
                        to external sync (in ms)
long syncStop; // the date MidiShare stopped being locked
                        to external sync (in ms)
long syncOffset; // the current offset (MidiGetExtTime() -
                  MidiGetTime (), in ms)
long syncSpeed;  // the current value for the timer
              (implementation dependent)
long syncBreaks; // the current count of breaks
                        (transition from state locked to unlocked)
short  syncFormat;      // the current synchronisation format
                        (0: 24 f/s, 1: 25 f/s, 2: 30DF f/s, 3: 30 f/s)
} TSyncInfo;
```

Note 1

syncMode is an unsigned 16-bits word of structure : xa000000ppppppp.

x (bit 15) is used to choose between internal synchronisation (x=0) and external synchronisation (x=1)

a (bit 14) is used to choose between synchronisation on port p (a=0) and synchronisation on any port (a=1)

bit 13:8 are reserved for future use and must be set to 0.

p (bit 0:7) is the synchronisation port to be used when x=1 and a=0. When a=1 the port number is ignored, the first port with incoming MTC is used.

Note 2

```
While MidiShare is locked (syncLocked == 1) syncOffset is constant and we
have the following relationships :
MidiGetExtTime() == MidiGetTime () + syncOffset
MidiInt2ExtTime(x) == x + syncOffset
```

```
MidiExt2IntTime(x) == x - syncOffset
```

EXAMPLE (ANSI C)

Gives the SMPTE start location of the tape.

```
TSyncInfo       myInfo;
TSmpteLocationmyLoc;

MidiGetSyncInfo(&myInfo);
MidiTime2Smpte( MidiInt2ExtTime(myInfo.syncStart), myInfo.syncFormat,
                                                              &myLoc);
```

DESCRIPTION

Gives in milliseconds the time elapsed since the starting up of MidiShare.

PROTOTYPE

C Atari
```
long          MidiGetTime ();
```
C Mac ANSI
```
pascal long   MidiGetTime ();
```
Pascal Mac
```
Function      MidiGetTime : longint;
```

ARGUMENTS

none

RESULT

The result is a 32-bit integer, being the elapsed time in milliseconds since the starting up of MidiShare.

EXAMPLE (ANSI C)

A wait function :

```
void wait(long delay)
{
  long d;

  d = MidiGetTime () + delay;
  while (MidiGetTime () < d);
}
```

MidiGetVersion

DESCRIPTION

Gives the version number of MidiShare

PROTOTYPE

C Atari	`short MidiGetVersion ();`
C Mac ANSI	`pascal short MidiGetVersion (void);`
Pascal Mac	`Function MidiGetVersion : integer;`

ARGUMENTS

none

RESULT

The result is a 16-bit integer, the MidiShare version number. A result of 161 means <version 1.61>.

EXAMPLE (ANSI C)

Print the MidiShare version number

```
void PrintVersion(void)
{
  printf( "MidiShare version : %4.2f\n",MidiGetVersion()/100.0);
}
```

DESCRIPTION

Tries to increase the memory space of MidiShare.

PROTOTYPE

C Atari
```
long          MidiGrowSpace (n);
```
C Mac ANSI
```
pascal long   MidiGrowSpace (long n);
```
Pascal Mac
```
Function      MidiGrowSpace (n : longint): longint;
```

ARGUMENTS

n : the number of cells to increase the MidiShare memory space.

RESULT

The result is a 32-bit integer, the number of new cells actually allocated.

EXAMPLE (ANSI C)

Add 1000 cells to MidiShare memory space.

```
void TryGrowSpace(void)
{
  printf( "Try to allocate 1000 cells : %ld\n", MidiGrowSpace(1000));
}
```

***Note** : On the Atari, MidiGrowSpace can only be used from a desk accessory, and not from a normal application.*

MidiInt2ExtTime

DESCRIPTION

Convert an internal time in millisecond to an external time. The convertion is made by adding the current offset between internal and external time.

PROTOTYPES

C Atari
```
long          MidiInt2ExtTime (time);
```
C Mac ANSI
```
pascal long   MidiInt2ExtTime (long time)
```
Pascal Mac
```
function      MidiGetExtTime (time : longint) : longint;
```

ARGUMENTS

time : a 32-bits time in milliseconds

RESULT

the corresponding external time, a 32-bits value in milliseconds.

Note

When MidiShare is locked we have the following equivalence :

```
MidiInt2ExtTime( MidiGetTime () ) == MidiGetExtTime()
```

Also :

```
TSyncInfo   myInfo;

MidiGetSyncInfo(&myInfo);
MidiInt2ExtTime(x) == x + myInfo.syncOffset
```

MidiIsConnected

DESCRIPTION

Gives the state of a connection between two MidiShare applications. Connections allow real-time communications of midi events between applications.

PROTOTYPE

```
C Atari       Boolean          MidiIsConnected (src, dest);
C Mac ANSI    pascal Boolean   MidiIsConnected (short src, short dest);
Pascal Mac    Function         MidiIsConnected (src, dest: integer) : boolean;
```

ARGUMENTS

src : is the reference number of a source application

dest : is the reference number of a destination application

RESULT

The result is true when a connection exist between the source and the destination, and false otherwise.

EXAMPLE (ANSI C)

Print all the sources of an application

```
void PrintSources(short refNum)
{
  short  src;
  short  i;

  printf( "Sources of : %s\n", MidiGetName( refNum) );
  for( i = 1; i <= MidiCountAppls(); ++i )
  {
     src = MidiGetIndAppl(i);
     if ( MidiIsConnected(src, refNum) )
       printf(" %i : %s \n", src, MidiGetName( src ) );
  }
}
```

Note for Mac users : *MidiShare was originally developed for Pascal on the Macintosh. Consequently, the result of MidiGetName is a Pascal string that must be converted to a C string to be printed.*

MidiNewCell

DESCRIPTION

Allocates a simple memory cell from the MidiShare memory manager. For some special application, it may be useful to have access to the basic functions of the memory manager. All the events managed by MidiShare are implemented from fixed-sized cells (16 bytes).

PROTOTYPE

C Atari
```
MidiEvPtr          MidiNewCell ();
```
C Mac ANSI
```
pascal MidiEvPtrMidiNewCell (void);
```
Pascal Mac
```
Function         MidiNewCell : MidiEvPtr;
```

ARGUMENTS

none

RESULT

The result a MidiEvPtr, a pointer to a memory cell, or NIL when memory space is exhausted.

EXAMPLE (ANSI C)

Allocate a new cell.

```
MidiEvPtr c;

c = MidiNewCell();

.....

MidiFreeCell(c);
```

***Note** : Cells allocated with MidiNewCell must be freed with MidiFreeCell and not with MidiFreeEv.*

DESCRIPTION

Allocates a new event of desirable type.

PROTOTYPE

C Atari
```
MidiEvPtr       MidiNewEv (short typeNum);
```
C Mac ANSI
```
pascal MidiEvPtrMidiNewEv (short typeNum);
```
Pascal Mac
```
Function        MidiNewEv (typeNum: integer): MidiEvPtr;
```

ARGUMENTS

typeNum : the type of event to be allocated

RESULT

The result a MidiEvPtr, a pointer to a MidiShare event of the desired type, or NIL if the MidiShare memory space is exhausted.

EXAMPLE (ANSI C)

A function for creating note events.

```
MidiEvPtr Note(long date, short pitch, short vel, long dur,
                                    short chan, short port)
{
  MidiEvPtr e;

  if ( e= MidiNewEv(typeNote) )
  {
    Date(e) = date;
    Pitch(e) = pitch;
    Vel(e) = vel;
    Dur(e) = dur;
    Chan(e) = chan;
    Port(e) = port;
  }
  return e;
}
```

MidiNewSeq

DESCRIPTION

Allocation of a new empty sequence.

PROTOTYPE

C Atari
```
MidiSeqPtr         MidiNewSeq ();
```
C Mac ANSI
```
pascal MidiSeqPtr  MidiNewSeq ();
```
Pascal Mac
```
Function           MidiNewSeq : MidiSeqPtr;
```

ARGUMENTS

none

RESULT

The result is a MidiSeqPtr, a pointer to an empty sequence.

EXAMPLE (ANSI C)

Create a sequence of 10 Midi clocks.

```
MidiSeqPtr ClockSeq()
{
  MidiSeqPtr  s;
  MidiEvPtr e;
  long d;

  s = MidiNewSeq();
  for (d=0; d< 2500; d+=250)
  {
    e = MidiNewEv (typeClock);
    Date(e) = d;
    MidiAddSeq (s, e);
  }
  return s;
}
```

DESCRIPTION

Opening of MidiShare. MidiOpen allows the recording of some information relative to the application context (its name, the value of the global data register, etc.), to allocate a reception FIFO and to attribute a unique reference number to the application. In counterpart to any MidiOpen call, the application must call the MidiClose function before leaving, by giving its reference number as an argument. MidiShare can thus be aware of the precise number of active Midi applications.

PROTOTYPE

C Atari
```
short         MidiOpen (applName);
```
C Mac ANSI
```
pascal short  MidiOpen (MidiName applName);
```
Pascal Mac
```
Function      MidiOpen (applName: midiName): integer;
```

ARGUMENTS

applName : the name of the application.

RESULT

The result is a unique reference number identifying the application.

EXAMPLE (ANSI C)

A do-nothing MidiShare application.

```
#include MidiShare.h
#include <stdio.h>

short  myRefNum;

main()
{
  if ( ! MidiShare() ) exit(1);         /* Check MidiShare loaded     */
  myRefNum = MidiOpen("Sample"); /* Ask for a reference number*/
  if ( myRefNum < 1 ) exit(1);    /* Check MidiOpen success     */
  printf( "refNum : %i \n", myRefNum);  /* Print the reference number*/
  MidiClose(myRefNum);            /* And close                  */
}
```

Note for Mac users : *MidiShare was originally developed for Pascal on the Macintosh. Consequently, in C, all strings passed as arguments of a MidiShare function must be Pascal strings. In the previous example, one must write :*

```
myRefNum = MidiOpen("\pSample");
```

MidiReadSync

DESCRIPTION

The MidiReadSync function reads and sets to NIL a memory address. This function is **none-interruptable** in order to make easier communication between the application tasks that run at interrupt level. It can be used to implement some sort of "mail boxes" in conjunction of MidiWriteSync.

PROTOTYPE

C Atari
```
Ptr           MidiReadSync (adrMem);
```
C Mac ANSI
```
pascal void*  MidiReadSync (void* adrMem) ;
```
Pascal Mac
```
Function      MidiReadSync (adrMem: univ ptr): ptr;
```

ARGUMENTS

adrMem : the address of a variable containing a 32-bit data.

RESULT

The result is the content of the variable.

SIDE EFFECT

Once read, the content of the variable is set to NIL.

EXAMPLE (ANSI C)

<< to be supplied >>

DESCRIPTION

Sends an event. A copy of the event is sent to all the application destinations. The date field of the event is used to specify when the destinations will actually receive the event.

PROTOTYPE

```
C Atari       void          MidiSend (refNum, e);
C Mac ANSI    pascal void   MidiSend (short refNum, MidiEvPtr e);
Pascal Mac    procedure     MidiSend (refNum: integer; e : MidiEvPtr);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

e : a MidiEvePtr, it is a pointer to the event to send.

EXAMPLE (ANSI C)

A receive alarm that processes all the received events by adding a one second delay to their date.

```
void OneSecDelay (short refNum)
{
  MidiEvPtr e;
  long n;

  for ( n = MidiCountEvs(refNum); n > 0; —n )
  {
    e = MidiGetEv (refNum); /* Get an event from the FIFO */
    Date(e) += 1000;              /* Add 1000 ms to its date          */
    MidiSend(refNum,e);           /* Then send the event              */
  }
}
......
/* Activate the receive alarm*/
MidiSetRcvAlarm(myRefNum,OneSecDelay);
```

***Note** : such a function can be called repeatedly in the main event loop of the application, but for really accurate time control, it must be installed as a receive alarm with MidiSetRcvAlarm.*

***Note for Mac users :** MidiShare was originally developed for Pascal on the Macintosh. Consequently, in C, all functions passed as arguments of a MidiShare function must be declared as Pascal. In the previous example, OneSecDelay must be declared as :*

```
pascal void OneSecDelay (short refNum)
```

MidiSendAt

DESCRIPTION

Sends an event. A copy of the event is sent to all the application destinations. The date argument is used to specify when destinations will actually receive the event.

PROTOTYPE

C Atari
```
void          MidiSendAt (refNum, e);
```
C Mac ANSI
```
pascal void   MidiSendAt (short refNum, MidiEvPtr e, long d);
```
Pascal Mac
```
procedure     MidiSendAt (refNum:integer;e:MidiEvPtr;d:longint);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

e : a MidiEvePtr, it is a pointer to the event to send.

d : a 32-bit integer, the date when destinations will receive the event.

EXAMPLE (ANSI C)

Equivalence between MidiSend, MidiSendAt and MidiSendIm :

```
MidiSendAt(myRefNum,e,MidiGetTime ());
```

is equivalent to :

```
MidiSendIm(myRefNum,e);
```

is equivalent to :

```
Date(e) = MidiGetTime ();
MidiSend( myRefNum, e );
```

DESCRIPTION

Immediately sends an event. A copy of the event is sent to all the application destinations.

PROTOTYPE

C Atari
```
void          MidiSendIm (refNum, e);
```
C Mac ANSI
```
pascal void   MidiSendIm (short refNum, MidiEvPtr e);
```
Pascal Mac
```
procedure     MidiSendIm (refNum:integer;e:MidiEvPtr);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

e : a MidiEvePtr, it is a pointer to the event to send.

EXAMPLE (ANSI C)

equivalence between MidiSend, MidiSendAt and MidiSendIm :

```
MidiSendIm(myRefNum,e);
```

is equivalent to :

```
MidiSendAt(myRefNum,e,MidiGetTime ());
```

is equivalent to :

```
Date(e) = MidiGetTime ();
MidiSend( myRefNum, e );
```

MidiSetApplAlarm

DESCRIPTION

Defines the context alarm of an application. This alarm will be called by MidiShare on every application of global context modifications (opening and closing of applications, opening and closing of midi ports, changes in connections between applications, SMPTE synchronisation).

PROTOTYPE

```
C Atari        void          MidiSetApplAlarm(short refNum,ApplAlarmPtr alarm);
C Mac ANSI     pascal void   MidiSetApplAlarm(short refNum,ApplAlarmPtr alarm);
Pascal Mac     Procedure     MidiSetApplAlarm(refNum:integer;
                                              alarm:ApplAlarmPtr);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

alarm : a ApplAlarmPtr, a pointer to the application context alarm routine.

PROTOTYPE OF A APPLALARM ROUTINE

```
C Atari        void          MyApplAlarm (refNum, code);
C Mac ANSI     pascal void   MyApplAlarm (short refNum, long code);
Pascal Mac     procedure     MyApplAlarm (refNum:integer; code:longint);
```

ARGUMENT OF A APPLALARM ROUTINE

refNum : a 16-bit integer, it is the reference number of the application.

code : a 32-bit integer, the context modification code : 0xRRRRMMMM where RRRR is the Reference number of the involved application and MMMM the type of change (see Midi Change Codes).

EXAMPLE (ANSI C)

<< to be supplied >>

MidiSetField

DESCRIPTION

Attributes a value to a field of an event. The access to the compulsory fields of the event is done directly. But the access to the variables fields is achieved through the MidiSetField and MidiGetField functions.

The function deals with the conversion of this value into the concerned field format (8, 16 or 32-bit).

PROTOTYPE

C Atari
```
void          MidiSetField (MidiEvPtr e, long f, long v);
```
C Mac ANSI
```
pascal void   MidiSetField (MidiEvPtr e, long f, long v);
```
Pascal Mac
```
procedure     MidiSetField (e:MidiEvPtr; f:longint; v:longint);
```

ARGUMENTS

e : a MidiEvPtr, a pointer to the event to be modified

f : a 32-bit integer, the index number of the field to modify (from 0 to MidiCountFields(e)-1)

v : a 32-bit value to put in the field. This value will be converted to the right size (8, 16 or 32-bit)

EXAMPLE (ANSI C)

<< to be supplied >>

MidiSetFilter

DESCRIPTION

Associates a filter to an application. Each application can select the events to be received by using a filter. The filtering process is local to the application and has no influence on the events received by the other applications. The implementation of these filters is achieved by two routines : MidiSetFilter and MidiGetFilter.

PROTOTYPE

C Atari
```
void          MidiSetFilter (refNum, filter);
```
C Mac ANSI
```
pascal void   MidiSetFilter (short refNum, FilterPtr filter);
```
Pascal Mac
```
procedure     MidiSetFilter (refNum: integer;
                             filter: FilterPtr);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

filter : a FilterPtr, a pointer to the application filter.

EXAMPLE (ANSI C)

<< to be supplied >>

DESCRIPTION

Defines the global information area of an application. The Macintosh desk accessories cannot have global variables. To make up for this drawback, the MidiSetInfo routine allows each application to define a data area. This area remains accessible by MidiGetInfo function, even during the alarm, and also serves as a global context to desk accessories.

PROTOTYPE

C Atari
```
void          MidiSetInfo (refNum, infoZone);
```
C Mac ANSI
```
pascal void   MidiSetInfo (short refNum, void* infoZone);
```
Pascal Mac
```
procedure     MidiSetInfo (refNum: integer; infoZone: Ptr);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

infoZone : an arbitrary 32-bit value, generally a pointer or a handle.

EXAMPLE (ANSI C)

<< to be supplied >>

MidiSetName

DESCRIPTION

Changes the name of an application.

PROTOTYPE

C Atari
```
void          MidiSetName (refNum, name);
```
C Mac ANSI
```
pascal void   MidiSetName (short refNum, MidiName name);
```
Pascal Mac
```
procedure     MidiSetName (refNum: integer; name: midiName);
```

ARGUMENTS

refNum : a 16-bit integer, it is the reference number of the application.

name : a MidiName, the new application name.

EXAMPLE (ANSI C)

<< to be supplied >>

MidiSetPortState

DESCRIPTION

For opening and closing of a Midi port. The implementation of Midi ports is controlled by the MidiSetPortState and MidiGetPortState routines. These must be used with care since they affect all the applications.

A closed port is available for other uses (printing, AppleTalk, etc.).

The Midi applications holding a "context alarm" will be informed of this change in the ports state.

PROTOTYPE

C Atari
```
void          MidiSetPortState (port, state);
```
C Mac ANSI
```
pascal void   MidiSetPortState (short port, Boolean state);
```
Pascal Mac
```
procedure     MidiSetPortState (port: integer; state: boolean);
```

ARGUMENTS

port : a 16-bit integer, the port number to control.

state : a Boolean, True : to open a port, False : to close a port.

EXAMPLE (ANSI C)

<< to be supplied >>

MidiSetRcvAlarm

DESCRIPTION

Defines the event reception alarm of an application. The alarm will be automatically called by MidiShare to inform the application of the presence of new events in its reception FIFO. This alarm is always called under interruption. It must not use, directly or indirectly, the Macintosh Memory Manager, however it can freely access all the others MidiShare functions, particularly the event management (but not MidiOpen and MidiClose). It can also use applications global variables, since MidiShare restores its global context register, before the call.

PROTOTYPE OF MIDISETRCVALARM

C Atari	`void`	**`MidiSetRcvAlarm`**`(refNum, alarm);`
C Mac ANSI	`pascal void`	**`MidiSetRcvAlarm`**`(short refNum, RcvAlarmPtr alarm);`
Pascal Mac	`Procedure`	**`MidiSetRcvAlarm`**`(refNum:integer;alarm:RcvAlarmPtr);`

ARGUMENTS OF MIDISETRCVALARM

refNum : a 16-bit integer, the reference number of the application

alarm : a RcvAlarmPtr, a pointer to a receive alarm routine or NIL to disable receive alarms.

PROTOTYPE OF A RCVALARM ROUTINE

C Atari	`void`	**`MyRcvAlarm`** `(refNum);`
C Mac ANSI	`pascal void`	**`MyRcvAlarm`** `(short refNum);`
Pascal Mac	`procedure`	**`MyRcvAlarm`** `(refNum:integer);`

ARGUMENT OF A RCVALARM ROUTINE

refNum : a 16-bit integer, it is the reference number of the application.

EXAMPLE (ANSI C)

A receive alarm that processes all the received events by adding to their date a one second delay.

```
void OneSecDelay (short refNum)
{
  MidiEvPtr e;
  long n;

  for ( n = MidiCountEvs(refNum); n > 0; —n )
  {
    e = MidiGetEv (refNum); /* Get an event from the FIFO */
    Date(e) += 1000;              /* Add 1000 ms to its date          */
    MidiSend(refNum,e);           /* Then send the event              */
  }
}

......
/* Install the receive alarm */
MidiSetRcvAlarm(myRefNum,OneSecDelay);
```

***Note** : Such a function could be called repeatedly in the main event loop of the application, but for really accurate time control, it must be installed as a receive alarm with MidiSetRcvAlarm.*

***Note for Mac users :** MidiShare was originally developed for Pascal on the Macintosh. Consequently, in C, all functions passed as arguments of a MidiShare function must be declared as Pascal. In the previous example, OneSecDelay must be declared as :*

```
pascal void OneSecDelay (short refNum)
```

MidiSetSyncMode

DESCRIPTION

Set the synchronisation mode of MidiShare.

PROTOTYPES

C Atari	`void MidiSetSyncMode (mode);`
C Mac ANSI	`pascal void MidiSetSyncMode (unsigned short mode);`
Pascal Mac	`procedure MidiSetSyncMode (mode: integer);`

ARGUMENTS

mode : an unsigned 16-bits word of structure : xa000000ppppppppp.

x (bit 15) is used to choose between internal synchronisation (x=0) and external synchronisation (x=1)

a (bit 14) is used to choose between synchronisation on port p (a=0) and synchronisation on any port (a=1)

bit 13:8 are reserved for future use and must be set to 0.

p (bit 0:7) is the synchronisation port to be used when x=1 and a=0. When a=1 the port number is ignored, the first port with incoming MTC is used.

EXAMPLE 1 (ANSI C)

Set the synchronisation to external, on any port.

```
MidiSetSyncMode(MIDISyncExternal | MIDISyncAnyPort);
```

EXAMPLE 2 (ANSI C)

Set the synchronisation to external, on port 18.

```
MidiSetSyncMode(MIDISyncExternal | 18);
```

EXAMPLE 3 (ANSI C)

Set the synchronisation to internal.

```
MidiSetSyncMode(MIDISyncInternal);
```

DESCRIPTION

Tests if MidiShare is resident in memory by looking for a specific pattern of code. This is the first MidiShare function that an application should call.

PROTOTYPE

C Atari
```
Boolean          MidiShare ();
```
C Mac ANSI
```
pascal Boolean   MidiShare (void);
```
Pascal Mac
```
Function         MidiShare : boolean;
```

ARGUMENTS

none

RESULT

The result is true when MidiShare is loaded, false otherwise.

EXAMPLE (ANSI C)

A do-nothing MidiShare application.

```
#include MidiShare.h
#include <stdio.h>

short  myRefNum;

main()
{
  if ( ! MidiShare() ) exit(1);       /* Check MidiShare loaded
  */
  myRefNum = MidiOpen("Sample");      /* Ask for a reference number */
  if ( myRefNum < 1 ) exit(1);    /* Check MidiOpen success          */
  printf( "refNum : %i \n", myRefNum);  /* Print the reference number */
  MidiClose(myRefNum);            /* And close                       */
}
```

Note for Mac users : *MidiShare was originally developed for Pascal on the Macintosh. Consequently, in C, all strings passed as arguments of a MidiShare function must be Pascal strings. In the previous example, one must write :*

```
myRefNum = MidiOpen("\pSample");
```

MidiSmpte2Time

DESCRIPTION

Convert an SMPTE location to a time in millisecond.

PROTOTYPES

C Atari
```
long          MidiSmpte2Time (loc);
```
C Mac ANSI
```
pascal long   MidiSmpte2Time (SmpteLocPtr loc);
```
Pascal Mac
```
function      MidiSmpte2Time (loc: SmpteLocPtr): longint;
```

ARGUMENTS

loc : a pointer to a TSmpteLocation record to be converted in milliseconds.

RESULT

a 32-bits time in milliseconds

DESCRIPTION OF A TSMPTELOCATION

```
typedef struct TSmpteLocation *SmpteLocPtr;
typedef struct TSmpteLocation
{
  short  format;// (0: 24 f/s, 1: 25 f/s, 2: 30DF f/s, 3: 30 f/s)
  short  hours; // 0..23
  short  minutes;     // 0..59
  short  seconds;     // 0..59
  short  frames;// 0..30 (according to format)
  short  fracs; // 0..99 (1/100 of frames)
} TSmpteLocation;
```

EXAMPLE (ANSI C)

Gives the SMPTE location from its current format to 30 drop frame (format 2).

```
TSmpteLocation myLoc;
...
// we suppose here myLoc filled with an SMPTE location
MidiTime2Smpte( MidiSmpte2Time(&myLoc), 2, &myLoc);
// now myLoc is filled with the same SMPTE location but in 30 drop frame format.
```

DESCRIPTION

As with MidiDTask, MidiTask allows an application to initiate a time delayed function call, however with MidiTask, the call is achieved under interruption as soon as falling time is due.

PROTOTYPE OF MIDITASK

C Atari
```
MidiEvPtr          MidiTask (MyProc, date, refNum, a1, a2, a3);
```
C Mac ANSI
```
pascal MidiEvPtrMidiTask (TaskPtr MyProc, long date,
                          short refNum, long a1, long a2,long a3);
```
Pascal Mac
```
Function           MidiTask (MyProc:TaskPtr; date:longint;
                     refNum:integer; a1,a2,a3:longint):MidiEvPtr;
```

ARGUMENTS OF MIDITASK

MyProc : a TaskPtr, it is the address of the routine to be called.

date : a 32-bit integer, it is the date at which this call is scheduled.

refNum : a 16-bit integer, it is the reference number of the application.

a1,a2,a3 : are 32-bit integers left at the user's disposal, as arguments to MyProc

RESULT OF MIDITASK

The result, a MidiEvPtr, is a pointer to a typeProcess MidiShare event. The result is NIL if MidiShare run out of memory.

PROTOTYPE OF MYPROC

C Atari
```
void          MyProc (date, refNum, a1, a2, a3);
```
C Mac ANSI :
```
pascal void   MyProc (long date, short refNum,
                      long a1, long a2, long a3);
```
Pascal Mac
```
procedure     MyProc (date:longint; refNum:integer;
                      a1,a2,a3: longint);
```

ARGUMENT OF MYPROC

date : a 32-bit integer, it is the date of the call .

refNum : a 16-bit integer, it is the reference number of the application.

a1,a2,a3 : are 32-bit integers that can be freely used.

EXAMPLE (ANSI C)

Schedule a function Action() call 1000 ms ahead.

```
MidiEvPtr myTask;

myTask = MidiTask(Action,MidiGetTime ()+1000,myRefNum, a1, a2, a3);
```

***Note** : The result, in myTask, can be used to test the success of MidiTask. It can also be used by MidiForgetTask to try to "forget" a scheduled task before it happens.*

MidiTime2Smpte

DESCRIPTION

Convert a time in millisecond to an SMPTE location.

PROTOTYPES

C Atari
```
void          MidiTime2Smpte (time, format, loc);
```
C Mac ANSI
```
pascal void   MidiTime2Smpte (long time, short format,
                                                SmpteLocPtr loc);
```
Pascal Mac
```
procedure     MidiTime2Smpte (time: longint; format: integer;
                                                loc:SmpteLocPtr);
```

ARGUMENTS

time : a 32-bits time in milliseconds to convert in an SMPTE location

format : a 16-bits integer, the SMPTE format to be used : (0 : 24 f/s, 1 : 25 f/s, 2 : 30DF f/s, 3 : 30 f/s)

loc : a pointer to a TSmpteLocation record to be filled with the resulting SMPTE location.

DESCRIPTION OF A TSMPTELOCATION

```
typedef struct TSmpteLocation *SmpteLocPtr;
typedef struct TSmpteLocation
{
  short  format;// (0 : 24 f/s, 1 : 25 f/s,
                    2 : 30DF f/s, 3 : 30 f/s)
  short  hours; // 0..23
  short  minutes;     // 0..59
  short  seconds;     // 0..59
  short  frames;// 0..30 (according to format)
  short  fracs; // 0..99 (1/100 of frames)
} TSmpteLocation;
```

EXAMPLE (ANSI C)

Gives the SMPTE start location of the tape.

```
TSyncInfo       myInfo;
TSmpteLocationmyLoc;

MidiGetSyncInfo(&myInfo);
MidiTime2Smpte( MidiInt2ExtTime(myInfo.syncStart),
                            myInfo.syncFormat, &myLoc);
```

DESCRIPTION

Gives the total number of cells allocated to MidiShare. MidiTotalSpace allows an application to know at any time the total number of cells allocated by the MidiShare memory manager at startup.

PROTOTYPE

C Atari	`long MidiTotalSpace ();`
C Mac ANSI	`pascal long MidiTotalSpace (void);`
Pascal Mac	`Function MidiTotalSpace : longint;`

ARGUMENTS

none

RESULT

the result is a 32-bit integer, the total number of cells in the MidiShare memory manager.

EXAMPLE (ANSI C)

Print information about MidiShare memory space.

```
void PrintMemInfo(void)
{
  printf("MidiShare memory :\n");
  printf(" free space  : %i cells\n", MidiFreeSpace());
  printf(" used space  : %i cells\n", MidiTotalSpace() - MidiFreeSpace());
  printf(" total space : %i cells\n", MidiTotalSpace());
}
```

MidiWriteSync

DESCRIPTION

Writes a 32-bit value to a variable only if the previous variable content was NIL. This function is **non-interruptable** in order to simplify communication between application tasks that run at interrupt level. It can be used to implement "mail boxes" between tasks when used in conjunction with MidiReadSync

PROTOTYPE

C Atari
```
Ptr           MidiWriteSync (adrMem, val );
```
C Mac ANSI
```
pascal void*  MidiWriteSync (void *adrMem, void *val);
```
Pascal Mac
```
Function      MidiWriteSync (adrMem:univ ptr;val:univ ptr):ptr;
```

ARGUMENTS

adrMem : is the address of a variable to be modified.

val : is a 32-bit value to write.

RESULT

The result is the previous content of the variable.

EXAMPLE (ANSI C)

<< to be supplied >>

EVENT DESCRIPTION

A Real Time ActiveSens message.

Fields : ActiveSens events have no field.

EXAMPLE (ANSI C)

Creates a ActiveSens event. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr ActiveSens ( long date, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeActiveSens )
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;   /* These informations are common    */
     Port(e) = port;   /* to all kind of events            */
  }
  return e;
}
```

typeChanPress (code 6)

EVENT DESCRIPTION

A Channel pressure message with pressure value.

Fields : ChanPress events have 1 field numbered 0 :

0 - A channel pressure value from 0 to 127. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a ChanPress event. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr ChanPress( long date, short press, short chan, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeChanPress ) )
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;         /* These informations's are common to all */
     Chan(e) = chan;         /* kind of events                         */
     Port(e) = port;
     MidiSetField(e,0,press);      /* Field particular to ChanPress    */
  }
  return e;
}
```

EVENT DESCRIPTION

A Real Time Clock message.

Fields : Clock events have no field.

EXAMPLE (ANSI C)

Creates a Clock event. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Clock ( long date, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeClock ))/* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all   */
    Port(e) = port;         /* kind of events                    */
  }
  return e;
}
```

typeContinue (code 12)

EVENT DESCRIPTION

A Real Time Continue message.

Fields : Continue events have no field.

EXAMPLE (ANSI C)

Creates a Continue event. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Continue ( long date, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeContinue ) )
  /* Allocate a new event. Check not NIL */
  {
    Date(e) = date;         /* These informations are common to all  */
    Port(e) = port;         /* kind of events                        */
  }
  return e;
}
```

EVENT DESCRIPTION

A copyright event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeCopyrigth events have a variable number of character fields.

EXAMPLE 1 (ANSI C)

Creates a typeCopyright event from a character string. Return a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr Copyright ( long date, char *s, short chan, short port)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv(typeCopyright) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                  */
    Port(e) = port;
    for (c=0; *s; s++, c++)       /* Build the event while counting
  */
      MidiAddField(e ,*s);        /* the characters of the original string */
    if (c != MidiCountFields(e)) {      /* Check the length of the event
  */
      MidiFreeEv(e);              /* if we run out of memory : free the
  */
      return 0;       /* event and return NIL            */
    }
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Convert a typeCopyrigth event into a character string. Assume s big enough.

```
void GetText (MidiEvPtr e, char *s)
{
  short c=0, i=0;

  c = MidiCountFields(e);
  while (i<c) *s++ = MidiGetField(e, i++);
  *s = 0;
}
```

typeCtrl14b (code 131)

EVENT DESCRIPTION

A Control Change event with a controller number from 0 to 31 and a 14-bits value. When a typeCtrl14b event is sent to external Midi devices, actually two control change messages are sent, the first one for the MSB part of the value and the second one for the LSB part of the value. The message for the LSB part is sent only when the LSB part of the value is different from 0.

Fields : Ctrl14b events have 2 fields numbered from 0 to 1 :

0 - A control number from 0 to 31. (Field size : 2 byte)

1 - A control value from 0 to 16383. (Field size : 2 byte)

EXAMPLE (ANSI C)

Creates a CtrlChange event. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr CtrlChange14b( long date, short ctrl, short val, short chan, short
port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeCtrl14b ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;              /* These informations are common to all
  */
    Chan(e) = chan;              /* kind of events                   */
    Port(e) = port;
    MidiSetField(e,0,ctrl); /* Fields particular to CtrlChange  */
    MidiSetField(e,1,val);
  }
  return e;
}
```

typeCtrlChange (code 4)

EVENT DESCRIPTION

A Control Change message with controller and value.

Fields : CtrlChange events have 2 fields numbered from 0 to 1 :

0 - A control number from 0 to 127. (Field size : 1 byte)

1 - A control value from 0 to 127. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a CtrlChange event. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr CtrlChange( long date, short ctrl, short val, short chan, short
port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeCtrlChange ) )
  /* Allocate a new event.Check not NIL */
  {
    Date(e) = date;              /* These informations are common to all
  */
    Chan(e) = chan;              /* kind of events                  */
    Port(e) = port;
    MidiSetField(e,0,ctrl); /* Fields particular to CtrlChange */
    MidiSetField(e,1,val);
  }
  return e;
}
```

typeChanPrefix (code 142)

EVENT DESCRIPTION

A channel prefix event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeChanPrefix events have one field.

0 - A channel prefix number from 0 to 15. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a typeChanPrefix event. Return a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr ChanPrefix ( long date, short prefix)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv(typeChanPrefix))
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;
     MidiSetField( e, 0, prefix);
  }
  return e;
}
```

EVENT DESCRIPTION

A cue point event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeCuePoint events have a variable number of character fields.

EXAMPLE 1 (ANSI C)

Creates a typeCuePoint event from a character string. Return a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr CuePoint ( long date, char *s, short chan, short port)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv(typeCuePoint))
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;               /* These informations are common to all
  */
    Chan(e) = chan;               /* kind of events                   */
    Port(e) = port;
    for (c=0; *s; s++, c++)       /* Build the event while counting the
  */
      MidiAddField(e ,*s);        /* characters of the original string
  */
    if (c != MidiCountFields(e)) {      /* Check the length of the event
  */
      MidiFreeEv(e);              /* if we run out of memory : free the
  */
      return 0;       /* event and return NIL             */
    }
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Convert a typeCuePoint event into a character string. Assume s big enough.

```
void GetText (MidiEvPtr e, char *s)
{
  short c=0, i=0;

  c = MidiCountFields(e);
  while (i<c) *s++ = MidiGetField(e, i++);
  *s = 0;
}
```

typeDProcess (code 129)

EVENT DESCRIPTION

DProcess events are automatically created by MidiDTask. They are used to realize time delayed function calls. Once the scheduling date is due, the routine is not automatically executed, but stored in a special list. It is the applications responsability to individually execute those pending tasks using MidiExec1DTask.

Fields : DProcess events have 4 fields numbered from 0 to 3 :

0 - a TaskPtr, the address of the function to call. (Field size : 4 byte)

1 - the first argument of the function. (Field size : 4 byte)

2 - the second argument of the function. (Field size : 4 byte)

3 - the third argument of the function. (Field size : 4 byte)

EXAMPLE (ANSI C)

Creates a DProcess event in the same way than MidiDTask.

```
MidiEvPtr MakeDTask ( TaskPtr proc, long date, short refNum, long arg1,
            long arg2, long arg3)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeDProcess ) )
  /* Allocate a new event. Check not NIL*/
  {
    MidiSetField(e, 0, (long)proc);     /* Fill the 4 fields */
    MidiSetField(e, 1, arg1);
    MidiSetField(e, 2, arg2);
    MidiSetField(e, 3, arg3);
    MidiSendAt(refNum, e, date);  /* and schedule the differed task*/
  }
  return e;
}
```

typeEndTrack (code 143)

EVENT DESCRIPTION

An end of track event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeEndTrack events have no field.

EXAMPLE (ANSI C)

Creates a typeEndTrack event. Return a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr EndTrack ( long date )
{
  MidiEvPtr e;

  if ( e = MidiNewEv(typeEndTrack))
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;
  }
  return e;
}
```

typeInstrName (code 138)

EVENT DESCRIPTION

An instrument name event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeInstrName events have a variable number of character fields.

EXAMPLE 1 (ANSI C)

Creates a typeInstrName event from a character string and returns a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr InstrName ( long date, char *s, short chan, short port)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv(typeInstrName) )
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;              /* These informations are common to all
  */
     Chan(e) = chan;              /* kind of events                   */
     Port(e) = port;
     for (c=0; *s; s++, c++)      /* Build the event while counting the
  */
        MidiAddField(e ,*s);      /* characters of the original string
  */
     if (c != MidiCountFields(e)) {     /* Check the length of the event
  */
        MidiFreeEv(e);            /* if we run out of memory : free the
  */
        return 0;       /* event and return NIL            */
     }
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Converts a typeInstrName event into a character string. Assume s is big enough.

```
void GetText (MidiEvPtr e, char *s)
{
  short c=0, i=0;

  c = MidiCountFields(e);
  while (i<c) *s++ = MidiGetField(e, i++);
  *s = 0;
}
```

typeKeyOff (code 2)

EVENT DESCRIPTION

A Note Off message with pitch and velocity.

Fields : KeyOff events have 2 fields numbered from 0 to 1 :

0 - Pitch, a note number from 0 to 127. (Field size : 1 byte)

1 - Vel, a note velocity from 0 to 127. (Field size : 1 byte)

EXAMPLE 1 (ANSI C)

Creates a KeyOff event, and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using MidiSetField instead of direct structure access.

```
MidiEvPtr KeyOff( long date, short pitch, short vel, short chan, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeKeyOff ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;              /* These informations are common to all
  */
    Chan(e) = chan;              /* kind of events                   */
    Port(e) = port;
    MidiSetField(e,0,pitch);     /* These fields are particular to KeyOff
  */
    MidiSetField(e,1,vel);
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Creates a KeyOff event, and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using direct structure access instead of MidiSetField.

```
MidiEvPtr KeyKeyOff( long date, short pitch, short vel, short chan, short
port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeKeyOff ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                   */
    Port(e) = port;
    Pitch(e) = pitch; /* These fields are particular to KeyOff*/
    Vel(e) = vel;
  }
  return e;
}
```

typeKeyOn (code 1)

EVENT DESCRIPTION

A Note On message with pitch and velocity.

Fields : KeyOn events have 2 fields numbered from 0 to 1 :

0 - Pitch, a note number from 0 to 127. (Field size : 1 byte)

1 - Vel, a note velocity from 0 to 127. (Field size : 1 byte)

EXAMPLE 1 (ANSI C)

Creates a KeyOn event, and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using MidiSetField instead of direct structure access.

```
MidiEvPtr KeyOn( long date, short pitch, short vel, short chan, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeKeyOn ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;              /* These informations are common to all
  */
    Chan(e) = chan;              /* kind of events                    */
    Port(e) = port;
    MidiSetField(e,0,pitch);     /* These fields are particular to KeyOn
  */
    MidiSetField(e,1,vel);
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Creates a KeyOn event and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using direct structure access instead of MidiSetField.

```
MidiEvPtr KeyOn( long date, short pitch, short vel, short chan, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeKeyOn ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                    */
    Port(e) = port;
    Pitch(e) = pitch; /* These fields are particular to KeyOn */
    Vel(e) = vel;
  }
  return e;
}
```

EVENT DESCRIPTION

A Polyphonic Key Pressure message with pitch and pressure.

Fields : KeyPress events have 2 fields numbered from 0 to 1 :

0 - Pitch, a note number from 0 to 127. (Field size : 1 byte)

1 - Press, a key pressure from 0 to 127. (Field size : 1 byte)

EXAMPLE 1 (ANSI C)

Creates a KeyPress event and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using MidiSetField instead of direct structure access.

```
MidiEvPtr KeyPress( long date, short pitch, short press, short chan, short
port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeKeyPress ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                        */
    Port(e) = port;
    MidiSetField(e,0,pitch);/* These fields are particular to KeyPress */
    MidiSetField(e,1,press);
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Creates a KeyPress event and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using direct structure access instead of MidiSetField.

```
MidiEvPtr KeyPress( long date, short pitch, short press, short chan, short
port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeKeyPress ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                        */
    Port(e) = port;
    Pitch(e) = pitch; /* These fields are particular to KeyPress*/
    Vel(e) = press;   /* Same byte than velocity               */
  }
  return e;
}
```

typeKeySign (code 147)

EVENT DESCRIPTION

A Key Signature event (form the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeKeySign events have 2 fields :

0 - from -7 (7 flats) to 7 (7 sharps), (8-bits field)

1 - form 0 (major key) to 1 (minor key), (8-bits field)

EXAMPLE (ANSI C)

Creates a Key Signature event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr KeySign (long date, long sharpflats, long minor)
{
  MidiEvPtr e;

  if ( e = MidiNewEv(typeKeySign))
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;
    MidiSetField(e, 0, sharpflats);
    MidiSetField(e, 1, minor);
  }
  return e;
}
```

EVENT DESCRIPTION

A lyric event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeLyric events have a variable number of character fields.

EXAMPLE 1 (ANSI C)

Creates a typeLyric event from a character string and returns a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr Lyric ( long date, char *s, short chan, short port)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv(typeLyric) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                  */
    Port(e) = port;
    for (c=0; *s; s++, c++) /* Build the event while counting the    */
      MidiAddField(e ,*s);  /* characters of the original string     */
    if (c != MidiCountFields(e)) {/* Check the length of the event   */
      MidiFreeEv(e);        /* if we run out of memory : free the    */
      return 0; /* event and return NIL            */
    }
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Convert a typeLyric event into a character string. Assume s big enough.

```
void GetText (MidiEvPtr e, char *s)
{
  short c=0, i=0;

  c = MidiCountFields(e);
  while (i<c) *s++ = MidiGetField(e, i++);
  *s = 0;
}
```

typeMarker (code 140)

EVENT DESCRIPTION

A marker event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeMarker events have a variable number of character fields.

EXAMPLE 1 (ANSI C)

Creates a typeMarker event from a character string and returns a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr Marker ( long date, char *s, short chan, short port)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv(typeMarker))
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;              /* These informations are common to all
  */
     Chan(e) = chan;              /* kind of events                   */
     Port(e) = port;
     for (c=0; *s; s++, c++)      /* Build the event while counting the
  */
        MidiAddField(e ,*s);      /* characters of the original string
  */
     if (c != MidiCountFields(e)) { /* Check the length of the event
  */
        MidiFreeEv(e);            /* if we run out of memory : free the
  */
        return 0;       /* event and return NIL            */
     }
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Convert a typeMarker event into a character string. Assume s big enough.

```
void GetText (MidiEvPtr e, char *s)
{
  short c=0, i=0;

  c = MidiCountFields(e);
  while (i<c) *s++ = MidiGetField(e, i++);
  *s = 0;
}
```

typeNonRegParam (code 132)

EVENT DESCRIPTION

A Non Registered Parameter event with a 14-bit parameter number and a 14-bit parameter value. When a typeNonRegParam event is sent to external Midi devices, actually four control change messages are sent, two to select the non-registered parameter number, and two for the parameter value using the 14-bits data-entry controller.

Fields : typeNonRegParam events have 2 fields numbered from 0 to 1 :

0 - A Non Registered Parameter number from 0 to 16383. (Field size : 2 bytes)

1 - A parameter value from 0 to 16383. (Field size : 2 bytes)

EXAMPLE (ANSI C)

Creates a Non Registered Parameter event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr NonRegParam( long date, short param, short val, short chan, short
port)
{
  MidiEvPtr e;

  if (e = MidiNewEv(typeNonRegParam))
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;              /* These informations are common to all
  */
    Chan(e) = chan;              /* kind of events                   */
    Port(e) = port;
    MidiSetField(e,0,param);     /* Fields particular to NonRegParam
  */
    MidiSetField(e,1,val);
  }
  return e;
}
```

typeNote (code 0)

EVENT DESCRIPTION

A note with pitch, velocity and duration. When a Note event is sent to external Midi devices, actually a NoteOn message is first sent followed, after a delay specified by the duration, by a NoteOn with a velocity of 0 to end the note.

Fields : Note events have 3 fields numbered from 0 to 2 :

0 - Pitch, a note number from 0 to 127. (Field size : 1 byte)

1 - Vel, a note velocity from 0 to 127. (Field size : 1 byte)

2 - Dur, a note duration from 0 to 2^{15}-1. (Field size : 2 bytes)

EXAMPLE 1 (ANSI C)

Creates a Note event and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using MidiSetField instead of direct structure access.

```
MidiEvPtr Note( long date, short pitch, short vel, short duration,
                short chan, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeNote ) ) /* Allocate a new event. Check not NIL
  */
  {
    Date(e) = date;   Port(e) = port;   /* These informations are common
  */
    Chan(e) = chan;                     /* to all kind of events
  */
    MidiSetField(e,0,pitch);      /* These fields are particular to Notes
  */
    MidiSetField(e,1,vel);
    MidiSetField(e,2,dur);
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Creates a Note event and returns a pointer to the event or NIL if there is no more memory space. Fields are modified using direct structure access instead of MidiSetField.

```
MidiEvPtr Note( long date, short pitch, short vel, short duration,
                short chan, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeNote ) ) /* Allocate a new event. Check not NIL
  */
  {
    Date(e) = date;   Port(e) = port;   /* These informations are common
  */
```

```
    Chan(e) = chan;                    /* to all kind of events
  */
    Pitch(e)  = pitch;       /* These fields are particular to Notes  */
    Vel(e) = vel;     Dur(e) = dur;
  }
  return e;
}
```

typePitchWheel (code 7)

EVENT DESCRIPTION

A Pitch Bender message with a 14 bits resolution.

Fields : PitchWheel events have 2 fields numbered from 0 to 1 :

0 - LS 7-Bits of 14-bits pitch swing, from 0 to 127. (Field size : 1 byte)

1 - MS 7-Bits of 14-bits pitch swing, from 0 to 127. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a PitchWheel event with a parameter between -8192 and 8191. This returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr PitchWheel( long date, short wheel, short chan, short port)
{
  const offset = 8192;
  const min = -8192;
  const max = 8191;
  MidiEvPtr e;

  wheel = (wheel>max) ? max : (wheel<min) ? min : wheel;

  if ( e = MidiNewEv( typePitchWheel ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;              /* These informations are common to all
  */
    Chan(e) = chan;              /* kind of events                   */
    Port(e) = port;
    MidiSetField(e,0,(wheel+offset) & 0x7F);  /* LS-7bits Field */
    MidiSetField(e,1,(wheel+offset)>>7 & 0x7F);    /* MS-7bits Field */
  }
  return e;
}
```

typePrivate (code 19 to 127)

EVENT DESCRIPTION

A private event with 4 fields which can be freely used by the application.

Fields : Private events have 4 fields numbered from 0 to 3.
Fields size : 4 bytes

EXAMPLE (ANSI C)

```
<to be supplied>
```

typeProcess (code 128)

EVENT DESCRIPTION

Process events are automatically created by MidiCall and MidiTask. They are used to realize time delayed function calls. The function call is achieved under interruption as soon as the scheduling date is due.

Fields : Process events have 4 fields numbered from 0 to 3 :

0 - a TaskPtr, the address of the function to call. (Field size : 4 byte)

1 - the first argument of the function. (Field size : 4 byte)

2 - the second argument of the function. (Field size : 4 byte)

3 - the third argument of the function. (Field size : 4 byte)

EXAMPLE (ANSI C)

Creates a Process event in the same way than MidiTask.

```
MidiEvPtr MakeTask ( TaskPtr proc, long date, short refNum, long arg1,
            long arg2, long arg3)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeProcess ) )
  /* Allocate a new event. Check not NIL*/
  {
    MidiSetField(e, 0, (long)proc);     /* Fill the 4 fields          */
    MidiSetField(e, 1, arg1);
    MidiSetField(e, 2, arg2);
    MidiSetField(e, 3, arg3);
    MidiSendAt(refNum, e, date);        /* and schedule the task      */
  }
  return e;
}
```

typeProgChange (code 5)

EVENT DESCRIPTION

A Program Change message with a program number.

Fields : ProgChange events have 1 field numbered 0 :

0 - A program number from 0 to 127. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a ProgChange event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr ProgChange( long date, short prog, short chan, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeProgChange ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                        */
    Port(e) = port;
    MidiSetField(e,0,prog); /* Field particular to ProgChange  */
  }
  return e;
}
```

typeQuarterFrame (code 130)

EVENT DESCRIPTION

A Midi time code quarter frame message with message type and value. These two fields are automatically assembled by MidiShare into one byte when the message is sent.

Fields : QuarterFrame events have 2 fields numbered from 0 to 1 :

0 - A message type from 0=Frame count LSB nibble to 7=Hours count MS nibble. (Field size : 1 byte)

1 - A count nibble from 0 to 15. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a QuarterFrame event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr QuarterFrame( long date, short type, short nibble, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeQuarterFrame ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;              /* These informations are common to all
  */
    Port(e) = port;              /* kind of events                   */
    MidiSetField(e,0,type); /* Fields particular to QuarterFrame     */
    MidiSetField(e,1,nibble);
  }
  return e;
}
```

typeRegParam (code 133)

EVENT DESCRIPTION

A Registered Parameter event with a 14-bit parameter number and a 14-bit parameter value. When a typeRegParam event is sent to external Midi devices, actually four control change messages are sent, two to select the registered parameter number, and two for the parameter value using the 14-bits data-entry controller.

Fields : typeRegParam events have 2 fields numbered from 0 to 1 :

0 - A Registered Parameter number from 0 to 16383. (Field size : 2 byte)

1 - A Registered Parameter value from 0 to 16383. (Field size : 2 byte)

EXAMPLE (ANSI C)

Creates a Registered Parameter event. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr RegParam( long date, short param, short val, short chan, short
port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeRegParam ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Chan(e) = chan;         /* kind of events                  */
    Port(e) = port;
    MidiSetField(e,0,param);      /* Fields particular to RegParam   */
    MidiSetField(e,1,val);
  }
  return e;
}
```

typeReserved (code 149 to 254)

EVENT DESCRIPTION

These events are RESERVED for future use.

EVENT DESCRIPTION

A Real Time Reset message.

Fields : Reset events have no field.

EXAMPLE (ANSI C)

Creates a Reset event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Reset ( long date, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeReset ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Port(e) = port;         /* kind of events                        */
  }
  return e;
}
```

typeSeqName (code 137)

EVENT DESCRIPTION

A sequence name event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeSeqName events have a variable number of character fields.

EXAMPLE 1 (ANSI C)

Creates a typeSeqName event from a character string and returns a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr SeqName ( long date, char *s, short chan, short port)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv( typeSeqName ) )
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;              /* These informations are common to all
  */
     Chan(e) = chan;              /* kind of events                   */
     Port(e) = port;
     for (c=0; *s; s++, c++)      /* Build the event while counting the
  */
        MidiAddField(e ,*s);      /* characters of the original string
  */
     if (c != MidiCountFields(e)) {     /* Check the length of the event
  */
        MidiFreeEv(e);            /* if we run out of memory : free the
  */
        return 0;       /* event and return NIL            */
     }
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Convert a typeSeqName event into a character string. Assume s big enough.

```
void GetText (MidiEvPtr e, char *s)
{
  short c=0, i=0;

  c = MidiCountFields(e);
  while (i<c) *s++ = MidiGetField(e, i++);
  *s = 0;
}
```

typeSeqNum (code 134)

EVENT DESCRIPTION

A Sequence number event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeSeqNum events have 1 field :

0 - Sequence number form 0 to 65535 (2-bytes field)

EXAMPLE (ANSI C)

Creates a Sequence Number event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr SeqNum( long date, short num, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeSeqNum) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Port(e) = port;         /* kind of events                        */
    MidiSetField(e,0,num);  /* the sequence number field             */
  }
  return e;
}
```

typeSMPTEOffset (code 145)

EVENT DESCRIPTION

A SMPTE Offset event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeSMPTEOffset events have 2 fields :

0 - Hours, minute and second parts of the SMPTE Offset in seconds from 0 to 1048575 (20- bits field)

1 - Frames and 100ths of a frame part of the SMPTE Offset in 100ths of a frame form 0 to 4095 (12-bits field)

EXAMPLE 1 (ANSI C)

Creates a SMPTE Offset event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr SMPTEOffset(long hr, long mn, long sec, long frames, long
subframes)
{
  MidiEvPtr e;

  if (e = MidiNewEv(typeSMPTEOffset))
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = 0;
     MidiSetField(e, 0, hr*3600 + mn*60 + sec);
     MidiSetField(e, 1, (frames*100 + subframes));
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Read the different parts of an SMPTE Offset event.

```
long GetHours (MidiEvPtr e) {
  return MidiGetField(e,0) / 3600;
}

long GetMinutes (MidiEvPtr e) {
  return MidiGetField(e,0) % 3600 / 60;
}

long GetSeconds (MidiEvPtr e) {
  return MidiGetField(e,0) % 60;
}

long GetFrames (MidiEvPtr e) {
  return MidiGetField(e,1) / 100;
}

long GetSubFrames (MidiEvPtr e) {
  return MidiGetField(e,1) % 100;
}
```

typeSongPos (code 8)

EVENT DESCRIPTION

A Song Position Pointer message with a 14 bit location (unit : 6 Midi Clocks).

Fields : SongPos events have 2 fields numbered from 0 to 1 :

0 - LS 7-Bits of 14-bits location, from 0 to 127. (Field size : 1 byte)

1 - MS 7-Bits of 14-bits location, from 0 to 127. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a SongPos event with a location in Midi clocks. The location is internally divided by 6. Return a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr SongPos( long date, short pos, short port)
{
  MidiEvPtr e;

  pos = pos / 6;

  if ( e = MidiNewEv( typeSongPos) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;              /* These informations are common to all
  */
    Port(e) = port;              /* kind of events                   */
    MidiSetField(e,0,pos & 0x7F); /* LS-7bits Field */
    MidiSetField(e,1,pos>>7 & 0x7F);    /* MS-7bits Field */
  }
  return e;
}
```

typeSongSel (code 9)

EVENT DESCRIPTION

A Song Select message with a song number.

Fields : SongSel events have 1 field numbered 0 :

0 - A song number from 0 to 127. (Field size : 1 byte)

EXAMPLE (ANSI C)

Creates a SongSel event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr SongSel ( long date, short song, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeSongSel ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Port(e) = port;         /* kind of events                        */
    MidiSetField(e,0,song); /* Field particular to SongSel           */
  }
  return e;
}
```

EVENT DESCRIPTION

A sequencer specific event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeSpecific events have a variable number of 8-bits fields.

EXAMPLE (ANSI C)

Creates a typeSpecific event from an array of bytes. Returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Specific ( long date, short len, Byte *p)
{
  MidiEvPtr e;
  short  c;

  if ( e = MidiNewEv( typeStream ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;
    c = len;
    while (c--) MidiAddField(e,*p++);
    if (MidiCountFields(e) < len ) /* if event smaller than len then*/
    {
      MidiFreeEv(e);             /* we run out of memory, free it    */
      e = nil;             /* and return nil             */
    }
  }
  return e;
}
```

typeStart (code 11)

EVENT DESCRIPTION

A Real Time Start message.

Fields : Start events have no field.

EXAMPLE (ANSI C)

Creates a Start event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Start ( long date, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeStart ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Port(e) = port;         /* kind of events                        */
  }
  return e;
}
```

EVENT DESCRIPTION

A Real Time Stop message.

Fields : Stop events have no field.

EXAMPLE (ANSI C)

Creates a Stop event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Stop ( long date, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeStop ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Port(e) = port;         /* kind of events                   */
  }
  return e;
}
```

typeStream (code 18)

EVENT DESCRIPTION

Stream messages are arbitrary streams of bytes sent by the MidiShare driver without any processing.

Fields : Stream events have a variable number of fields.

EXAMPLE (ANSI C)

Creates a Stream event from an array of shorts and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Stream ( long date, short len, short *p, short port)
{
  MidiEvPtr e;
  short c;

  if ( e = MidiNewEv( typeStream ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all  */
    Port(e) = port;         /* kind of events                        */
    c = len+1;
    while (--c) MidiAddField(e,*p++);
    if (MidiCountFields(e) < len ) /* if event smaller than len then*/
    {
      MidiFreeEv(e);              /* we run out of memory, free it   */
      e = nil;              /* and return nil             */
    }
  }
  return e;
}
```

EVENT DESCRIPTION

A System Exclusive message.

Fields : SysEx events have a variable number of fields. The leading F0 and tailing F7 codes MUST NOT be included. They are automatically supplied by MidiShare. The channel field of the event is OR'ed with the first data byte after the manufacturer ID. This works for setting the channel of many system exclusive messages.

EXAMPLE (ANSI C)

Creates a SysEx event from an array of shorts and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr SysEx ( long date, short len, short *p, short chan, short port)
{
  MidiEvPtr e;
  short c;

  if ( e = MidiNewEv( typeSysEx ) )
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;         /* These informations are common to all   */
    Chan(e) = chan;         /* kind of events                    */
    Port(e) = port;
    c = len+1;
    while (--c) MidiAddField(e,*p++);
    if (MidiCountFields(e) < len ) /* if event smaller than len then*/
    {
      MidiFreeEv(e);              /* we run out of memory, free it    */
      e = nil;             /*    and return nil                */
    }
  }
  return e;
}
```

typeTempo (code 144)

EVENT DESCRIPTION

A tempo event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeTempo events have one field.

0 - A tempo value in microseconds/Midi quarter-note 0 to 127. (Field size : 4 bytes)

EXAMPLE 1 (ANSI C)

Creates a typeTempo event from a floating point tempo value in quarter-notes per minutes. Returns a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr TempoChange ( long date, float tempo)
{
  MidiEvPtr e;

  if ( e = MidiNewEv(typeTempo))
  /* Allocate a new event. Check not NIL*/
  {
    Date(e) = date;
    MidiSetField(e, 0, (long)(60000000.0 / tempo));
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Converts a tempo event in microseconds per quarter-note in to a floating point tempo value in quarter-notes per minutes.

```
float GetTempo (MidiEvPtr e)
{
  return 60000000.0 / (float) MidiGetField(e,0);
}
```

EVENT DESCRIPTION

A text event (from the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeText events have a variable number of character fields.

EXAMPLE 1 (ANSI C)

Creates a typeText event from a character string and returns a pointer to the event or NIL if there is not enough memory space.

```
MidiEvPtr Text ( long date, char *s, short chan, short port)
{
  MidiEvPtr e;
  long c=0;

  if ( e = MidiNewEv( typeText ) )
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;               /* These informations are common to all
  */
     Chan(e) = chan;               /* kind of events                   */
     Port(e) = port;
     for (c=0; *s; s++, c++)       /* Build the text event while counting
  */
        MidiAddField(e ,*s);       /* the characters of the original string
  */
     if (c != MidiCountFields(e)) {/* Check the length of the text event
  */
        MidiFreeEv(e);             /* if we run out of memory : free the
  */
        return 0;       /* text event and return NIL             */
     }
  }
  return e;
}
```

EXAMPLE 2 (ANSI C)

Convert a typeText event into a character string. Assume s big enough.

```
void GetText (MidiEvPtr e, char *s)
{
  short c=0, i=0;

  c = MidiCountFields(e);
  while (i<c) *s++ = MidiGetField(e, i++);
  *s = 0;
}
```

typeTimeSign (code 146)

EVENT DESCRIPTION

A Time Signature event (form the MidiFile 1.0 specification). This event CANNOT be sent to external Midi devices.

Fields : typeTimeSign events have 4 fields :

0 - Numerator (8-bits field)

1 - denominator in power of two (8-bits field)

2 - Midi Clocks per metronome clicks (8-bits field)

3 - notated 32th of note per quarter-note (8-bits field)

EXAMPLE (ANSI C)

Creates a Time Signature event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr TimeSign (long date, long num, long denom, long click, long
quarterDef)
{
  MidiEvPtr e;

  if ( e = MidiNewEv(typeTimeSign))
  /* Allocate a new event. Check not NIL*/
  {
     Date(e) = date;
     MidiSetField(e, 0, num);
     MidiSetField(e, 1, denom);
     MidiSetField(e, 2, click);
     MidiSetField(e, 3, quarterDef);
  }
  return e;
}
```

EVENT DESCRIPTION

A Tune message.

Fields : Tune events have no field.

EXAMPLE (ANSI C)

Creates a Tune event and returns a pointer to the event or NIL if there is no more memory space.

```
MidiEvPtr Tune ( long date, short port)
{
  MidiEvPtr e;

  if ( e = MidiNewEv( typeTune ) )/* Allocate a new event. Check not NIL
  */
  {
    Date(e) = date;               /* These informations are common to all
  */
    Port(e) = port;               /* kind of events                   */
  }
  return e;
}
```