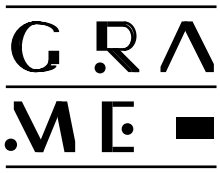

MidiShare Library

Player Library 2.0

GRAME Research Lab.
9, rue du Garet - BP 1185
69202 LYON CEDEX 01 France
Ph: +33 (0)4 72 07 37 00 Fax: +33 (0)4 72 07 37 01
http://www.grame.fr

e-mail : grame@rd.grame.fr

GRAME

Summary

Introduction

About this manual

This manual is intended for developers who want to add a sequencer component in their application using the Player library. It contains a complete description of all the library functions and data structures, as well as several examples of code. This library is available on Macintosh and PC/Windows (as a DLL).

Introduction

The *Player* library is provided to allow the development of a complete multi-tracks MidiShare sequencer. Each Player is a MidiShare application, has 256 tracks, four different synchronization mode, events chase and loop markers. A Player use MidiShare sequences. Functions are provided to read on write MIDIfiles and convert them to the MidiShare sequence format.

Overview of a Player

This section gives an overview of the Player library functions.

Opening and Closing a Player

A new empty Player is opened using the `OpenPlayer` function which gives as result the reference number of the corresponding MidiShare application. This reference number will be used for all the Player functions. An empty Player use by default a tempo of 120 bpm, a 4/4 time signature and a tick_per_quarter value of 500 so that 1 tick (the internal time unit) correspond exactly to 1 ms. MidiShare sequences will be loaded into a Player using the `SetAllTrackPlayer` function. A MidiShare sequence contains usually a tempo-map track which will be used instead of the default tempo and time signature values.

After it's creation, the Player should be connected to the MIDI input/output (that is to the MidiShare application), but because the Player is manipulated using it's MidiShare reference number, it's possible to use some MidiShare functions to manipulate it : a Player can be connected to other players or MidiShare applications using `MidiConnect`, the name of a Player can be accessed an changed using `MidiGetName/MidiSetName` functions...

The Player will be closed using the `ClosePlayer` function. `ClosePlayer` frees the Player internal score and close the MidiShare application.

Transport control : playback

A Player has the usual transport control:

-`StartPlayer` starts from the beginning of the score.

-`PausePlayer` stops the Player without sending the pending events (key-off, sustain off ..)

-`ContPlayer` plays from the current position.

-`StopPlayer` stops the Player and send the chased events (key-off...)

-`ForwardStepPlayer` plays the following chord and take care of solo/muted tracks state.

-`BackwardStepPlayer` plays the previous chord and take care of solo/muted track state.

The Player's running state can be accessed at any time using the `GetStatePlayer` function. The state can be one of the following: `kIdle, kPause, kRecording, kPlaying or kWaiting.` The Player is in `kWaiting` mode when waiting for synchronization.

Transport control : recording

To begin a new recording, one must first select a track in record mode using the `RecordPlayer` function and then start the Player (from the beginning or from the current position). During the recording, the incoming events are stored in the selected track. A recording can be done in two mode :

-`kMixMode` : recorded events are mixed in the recording track.

-`kEraseMode` : the track is first deleted and recorded events are then inserted in the track.

The recording state will be controlled with the `SetRecordModePlayer` function. A record filter can be installed for the Player using the `SetRecordFilterPlayer` function.

Synchronization management

A Player can be in four different synchronization modes which can be changed using the `SetSynchroInPlayer` function:

-k`InternalSync` mode: The Player uses it's internal tempo-map which is on the track 0.

-k`ClockSync` mode: The Player recognizes synchronization events at it's input and can be driven by **Start**, **Stop**, **Continue**, **Clock** and **SongPos** events.

-k`SMPTESync` mode: This is a global synchronization mode for all MidiShare applications. A SMPTE offset can be set, and the Player will automatically starts when the SMPTE offset is reached. The Player will stop when incoming MTC disappears.

-k`ExternalSync` mode: The Player does not use it's internal Tempo Map anymore, recognizes incoming **Tempo** events and the internal tempo can be changed with the `SetTempo` function.

By default, a Player is in k`InternalSync` mode.

A Player can be used as a master to synchronize other Players or others synchronizable MidiShare applications using the `SetSynchroOutPlayer function.` The Player will send **Start**, **Stop**, **Continue**, **Song Position** and **Clock** events. By default, a Player is in `kNoSyncOut` mode.

Tracks management

Each Player contains a unique internal score where all tracks are mixed. A Player has 256 tracks which can be individually accessed, changed and recorded. Tracks are internally distinguish by the reference number of events which are on them. This means that all events of track 0 have a reference number of 0, all events on track 1 have a reference number of 1 and so on. When loading a multi-tracks MIDIFile (format 1 or 2) the different tracks of the MIDIfile are automatically dispatched on tracks of the MidiShare sequence. If one wants to set a new multi-tracks sequence in a Player, the different tracks must have different reference numbers. A new MidiShare sequence is loaded into the Player using the functions `SetAllTrackPlayer` or `SetTrackPlayer`. Internal tracks can be extracted (for example to be saved as MIDIfiles) using `GetAllTrackPlayer` or `GetTrackPlayer` functions.

By using the function `SetParamTrackPlayer`, each track can be muted or played solo. The track 0 is use for the tempo map and should be managed carefully.

Position management

The position in a score in managed internally using a tick unit. The position will be manipulated either with musical time (bar, beat, unit) or real-time (hours, min, sec, millisec). The Player's current position can be changed using the `SetPosMsPlayer` to access the real-time or `SetPosBBUPlayer` to access the musical time in bar, beat, unit.

Events Chase

Chase Events makes sure that a Player is playing the correct patch regardless of the position in which the track was started. To do that, the Player keeps internally the whole state of program change, volume, pan or other controller information which are inserted in the tracks. This state in automatically updated when the Player moves in both directions. Chase Events can then re-send MIDI events that occurred previously in the tracks.

The following types of events are chased: program-change, controller (volumes, sustain..), pitch-bend, channel pressure, key pressure and tune.

Loop management

Loop markers can be set using `SetLoopStartMsPlayer`, `SetLoopEndMsPlayer`, `SetLoopStartBBUPlayer` and `SetLoopEndBBUPlayer` functions (even during playback) and loop mode will be switch on/off using `SetLoopPlayer`.

State management

For displaying purpose, the `GetStatePlayer` function allows to read the internal state of the Player (position in millisecond and musical time, tempo, time signature ....) at any time. An application typically call this function at regular interval (using a MidiShare task or in it's event loop) to display the current state. The function `GetEndScore` allows to read various information about the score last position.

Midifile management

Two functions, `MidifileLoad` and `MidifileSave` allow to read and write MIDIfiles and convert them to MidiShare sequence format. When loading a multi-tracks MIDIfile, the different tracks of the MIDIfile will correspond to different tracks of the MidiShare sequence.

Some examples

We give here some simple examples of the use of the Player library. 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 some Players functions are in Pascal format (starting with `\p` like in Ò`\pExample1`Ó). If you run these examples on another computer, you need to remove the `\p`.

Example 1 : Opening and closing a Player

Program 1 opens a new empty Player using the `OpenPlayer` function, connect it to the Midi Output, then closes the Player using the `ClosePlayer` function, that's all.

Listing 1

```
#include <Player.h>

main()
{
            short   myRefNum;

            myRefNum = OpenPlayer("\pExample1");

            MidiConnect(myRefNum,0,1);

            ClosePlayer(myRefNum);
}
```

Example 2 : Loading and Playing a MIDIfile

Now an example using a MIDIfile, assuming that we have a MIDIfile whose filename is "HD500:test1", we open a Player with the test1 file, connect it to the Midi Output, then play the MIDIfile.

Listing 2

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

main()
{
            short   myRefNum;
            MidiseqPtr mySeq;
            MidiFileInfos myInfo;

            myRefNum = OpenPlayer("\pExample2");
            MidiConnect(myRefNum,0,1);
            mySeq = MidiNewSeq();
            MidiFileLoad( "HD500:midifile1", mySeq, &myInfo);
            SetAllTrackPlayer (myRefnum, mySeq, info.clicks);

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

            StartPlayer(myRefNum );

            printf("(type <ENTER> to stop playing )\n");
            getc(stdin);

            StopPlayer(myRefNum );

            ClosePlayer(myRefNum);
}
```

Example 3 : Recording a score and save it as a MIDIfile

In this example we will open a new Player, connect it to the Midi Input and the Midi Output, record some events and then save the result as a MIDIfile.

Listing 3

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


main()
{
            short  myRefNum;
            MidiseqPtr mySeq, myRecSeq;
            MidiFileInfos myInfo;

            myRefNum = OpenPlayer("\pExample3");
            MidiConnect(myRefNum,0,1);
            MidiConnect(0,myRefNum,1);

            RecordPlayer (myRefnum, 1); // record on track one

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

            ... play some events...

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

            myRecSeq= GetAllTrackPlayer(myRefnum);

            myInfo.format = midifile1;
            myInfo.timedef = TicksPerQuarterNote;
            myInfo.clicks = 500;

            MidiFileSave( "HD500:test2", myRecSeq, &myInfo);

            MidiFreeSeq (myRecSeq);

            ClosePlayer(myRefNum);
}
```

Reference

Constants

Player state

```
enum playerstatus {kIdle = 0, kPause, kRecording, kPlaying, kWaiting };
```

Track control and state

```
#define kMaxTrack       256
#define kMuteOn         1
#define kMuteOff        0
#define kSoloOn         1
#define kSoloOff        0
enum trackparameter     { kMute = 0,  kSolo };
```

Recording state

```
#define kNoTrack        -1
#define kEraseMode      1
#define kMixMode        0
```

Loop control

```
enum loop {kLoopOn = 0, kLoopOff};
```

Step playing control

```
#define kStepPlay       1
#define kStepMute       0
```

Synchronisation management

```
enum rcvsynchro { kInternalSync = 0, kClockSync, kSMPTESync,
kExternalSync };
enum sendsynchro { kNoSyncOut = 0,  kClockSyncOut };
```

MIDIFile management

```
enum midifiletypes     { midifile0 = 0, midifile1, midifile2};

#define TicksPerQuarterNote        0
#define Smpte24                    24
#define Smpte25                    25
#define Smpte29                    29
#define Smpte30                    30
```

Data Types

The PlayerState data structure allows to read the internal state of the Player for displaying purposes.

```
typedef struct PlayerState* PlayerStatePtr;
typedef struct PlayerState{
  long date;   /* date in millisecond */
  long tempo;  /* tempo in microsec/per/quarter-note */
  short tsnum; /* time signature */
  short tsdenom;
  short tsclick;
  short tsquarter;
  short bar;      /* position in musical time */
  short beat;
  short unit;
  short state;    /* player state */
  short syncin;   /* synchronisation IN state */
  short syncout;  /* synchronisation OUT state */

}PlayerState;
```

`MidiFileInfos` data structure must be used when reading and writing MIDIfile.

```
typedef struct MidiFileInfos* MidiFileInfosPtr;
typedef struct MidiFileInfos
{
  long format;        /* file format  */
  long timedef;       /* time definition  */
  long clicks;        /* tick/per/quarter-note */
  long tracks;        /* number of tracks */

}MidiFileInfos;
```

TPos data structure allows to provide position in musical time (bar, beat, unit) to the Player.

```
typedef struct Pos* PosPtr;
typedef struct Pos{
  short bar;      /* position in musical time */
  short beat;
  short unit;
}Pos;
```

Error Codes

Table 1 List of the error codes returned by some Player functions.

Name	Code	Comment
PLAYERnoErr	-1	/* no error */
PLAYERerrAppl	-2	/* Unable to open MidiShare application */
PLAYERerrEvent	-3	/* No more MidiShare Memory */
PLAYERerrMemory	-4	/* No more Memory */
PLAYERerrSequencer	-5	/* Player error */

Table 2 List of the error codes returned by MIDIfile functions.

Name	Code	Comment
NoErr	0	/* no error */
ErrOpen	1	/* file Open error */
ErrRead	2	/* file read error */
ErrWrite	3	/* file write error */
ErrVol	4	/* Volume info volume */
ErrGetInfo	5	/* GetFInfo error */
ErrSetInfo	6	/* SetFInfo error */
ErrMidiFileFormat	7	/* bad MidiFile format */

BackwardStepPlayer

DESCRIPTION

This function allows to implement step playing. `BackwardStepPlayer` plays the previous group of notes (notes at the same date) and allows to move in the score by group of notes. `BackwardStepPlayer` takes care of the tracks state (mute or solo mode) and works only when the Player is idle.

PROTOTYPE

C ANSI `void BackwardStepPlayer (short refnum, short state);`

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

state : k`StepPlay` , kStepMute : event will be played or not.

(Warning : in version 2.0, this argument is no longer used, events will always be played)

EXAMPLE (ANSI C)

`BackwardStepPlayer` will be used to implement fast/backward playing. The function BackwardButton calls `BackwardStepPlayer` in a loop while mouse down.

```
void BackwardButton (refnum)
{

            while (gMouseDown){
               BackwardStepPlayer (refNum , kStepPlay);
               ... wait some time...
            }

}
```

See also ForwardStepPlayer.

ClosePlayer

DESCRIPTION

Closes a Player given it's reference number. This function automatically free the internal score and close the MidiShare application. `ClosePlayer` can not be called at interrupt level.

PROTOTYPE

C ANSI
```
short ClosePlayer (short refnum);
```

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

See also OpenPlayer.

ContPlayer

DESCRIPTION

Starts playing from the current position in the score.

PROTOTYPE

C ANSI `void` **`ContPlayer`** `(short refnum);`

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

See also StartPlayer, StopPlayer, PausePlayer.

ForwardStepPlayer

DESCRIPTION

This function allows to implement step playing. `ForwardStepPlayer` plays the next group of notes (notes at the same date) and allows to move in the score by group of notes. `ForwardStepPlayer` takes care of the tracks state (mute or solo mode) and works only when the Player is idle.

PROTOTYPE

C ANSI `void` **`ForwardStepPlayer`** `(short refnum, short state);`

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

state : k`StepPlay` , kStepMute : event will be played or not.

(Warning : in version 2.0, this argument is no longer used, events will always be played)

EXAMPLE (ANSI C)

`ForwardStepPlayer` will be used to implement fast/forward playing. The function ForwardButton calls `ForwardStepPlayer` in a loop while mouse down.

```
void ForwardButton (refnum)
{

            while (gMouseDown){
               ForwardStepPlayer (refNum , kStepPlay);
               ... wait some time...
            }

}
```

See also BackwardStepPlayer.

GetAllTrackPlayer

DESCRIPTION

Returns all the tracks contained in a Player as a MidiShare sequence. All tracks are mixed in a unique MidiShare sequence but are distinguish by the reference number of their events. The returned sequence is a COPY of the internal score. `GetAllTrackPlayer` can not be called at interrupt level.

PROTOTYPE

C ANSI `MidiSeqPtr` **`GetAllTrackPlayer`** `(short refnum);`

ARGUMENTS

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

RESULT

The result is MidiShare sequence where all tracks are mixed or 0 if the sequence can not be allocated.

EXAMPLE (ANSI C)

Get all tracks from a Player and save the sequence in a MIDIfile.

```
void SaveMidiFile (short refNum)
{
            MidiSeqPtr mySeq;
            MidiFileInfos info;

            mySeq= GetAllTrackPlayer(refNum);

            myInfo.format = midifile1;
            myInfo.timedef = TicksPerQuarterNote;
            myInfo.clicks = 500;

            MidiFileSave( "HD500:test2", mySeq, &myInfo);

            MidiFreeSeq(mySeq);


}
```

See also GetTrackPlayer, SetAllTrackPlayer, SetTrackPlayer.

GetEndScorePlayer

DESCRIPTION

Returns information about the last event in the score: position in musical time and millisecond.

PROTOTYPE

C ANSI
```
void GetEndScorePlayer(short refNum, PlayerStatePtr state);
```

ARGUMENTS

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

state : a PlayerStatePtr will be filled with information about the last event in the score : position in (Bar, Beat,Unit) and millisecond.

GetParamPlayer

DESCRIPTION

Returns the current value of the parameter **solo** or **mute** in a track.

PROTOTYPE

C ANSI
```
short GetParamPlayer(short refNum,short tracknum,short param);
```

ARGUMENTS

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

RESULT

A short : is the value of the corresponding parameter.

See also SetParamPlayer.

GetStatePlayer

DESCRIPTION

Returns the state of the Player at the current position. This function is usually used for state displaying purpose.

PROTOTYPE

C ANSI
```
void GetStatePlayer (short refNum, PlayerStatePtr state);
```

ARGUMENTS

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

state : a PlayerStatePtr will be filled with the state of the Player.

GetTrackPlayer

DESCRIPTION

Returns a track contained in a Player as a MidiShare sequence given the track number. The returned sequence is a COPY of the internal track. `GetTrackPlayer` can not be called at interrupt level.

PROTOTYPE

C ANSI `MidiSeqPtr` **`GetTrackPlayer`** `(short refnum, short tracknum);`

ARGUMENTS

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

tracknum : the track number.

RESULT

The result is a MidiShare sequence or 0 if the sequence can not be allocated.

EXAMPLE (ANSI C)

Get a track from a Player and save the sequence in a MIDIfile.

```
void SavePlayer (short refNum)
{
            MidiSeqPtr mySeq;
            MidiFileInfos info;

            mySeq= GetTrackPlayer(myRefnum,  1);

            myInfo.format = midifile0;
            myInfo.timedef = TicksPerQuarterNote;
            myInfo.clicks = 500;

            MidiFileSave( "HD500:test2", mySeq, &myInfo);

            MidiFreeSeq(mySeq);
}
```

See also GetAllTrackPlayer, SetAllTrackPlayer, SetTrackPlayer.

MidiFileLoad

DESCRIPTION

Load a MIDIfile and convert it into a MidiShare sequence. When loading a multi-tracks MIDIFile (format 1 or 2) the different tracks of the MIDIfile are distinguish by the reference number which are on them. It means that all events of track 0 have a reference number of 0, all events on track 1 have a reference number of 1 and so on. The function tries to restore (if possible) the Midi event port number using the InstrName event which has possibly been written by the MidiFileSave function (see MidiFileSave).

PROTOTYPE

C ANSI
```
long MidiFileLoad (char * filename, MidiSeqPtr s,
                                    MidiFileInfosPtr info);
```

ARGUMENTS

filename : a C string, the filename of the MIDIFile.

s : a MidiSeqPtr : the sequence to be loaded.

info : a MidiFileInfosPtr: used to record some information about the MIDIfile.

RESULT

The result is a MIDIfile error code.

EXAMPLE (ANSI C)

Load a MIDIfile.

```
void LoadMF()
{
            MidiSeqPtr mySeq = MidiNewseq();
            MidiFileInfos myInfo;

            MidiFileLoad( "HD500:midifile1", mySeq, &myInfo);

}
```

See also MidiFileSave.

DESCRIPTION

Save a MidiShare sequence in a MIDIfile. When saving a format 1 MIDIfile, the function uses the events reference number to create different tracks in the MIDIfile. The function always write tracks in ascending order starting from track 0 (which is the TempoMap). Notes events are written as a KeyOn/KeyOff pair and EndTrack events are written automatically. If the sequence does not contain a Tempo Map, the function writes a default Tempo of 120 bpm and a Time Signature of 4/4. To save the Midi event port number, a InstrName event is included at the beginning of each track. This event contains the port number coded as a string like "Port 3" for example. This information will be used by the MidiFileLoad function to restore (if possible) the Midi event port number. Note that the port number will be the same for all event on the same track.

PROTOTYPE

C ANSI
```
long MidiFileSave (char * filename, MidiSeqPtr s,
                                    MidiFileInfosPtr info);
```

ARGUMENTS

filename : a C string, the filename of the MIDIFile.

s : a MidiSeqPtr : the sequence to be saved.

info : a MidiFileInfosPtr : used to record some information about the MIDIfile.

RESULT

The result is a MIDIfile error code.

EXAMPLE (ANSI C)

Save a MIDIfile.

```
void SaveSeq(MidiSeqPtr seq)
{
            MidiFileInfos myInfo;


            myInfo.format = midifile1;
            myInfo.timedef = TicksPerQuarterNote;
            myInfo.clicks = 500;

            MidiFileSave( "HD500:test2", seq, &myInfo);
```

```
}
```

See also MidiFileLoad.

OpenPlayer

DESCRIPTION

`Opens` a new empty Player and give it's MidiShare reference number as result. The tempo is assumed to be 120 bpm and the internal resolution is set by default to 500 ticks_per_quarter so that the internal time unit (tick) correspond exactly to one ms. Sequences will be loaded in a Player using SetAllTrackPlayer function. Any opened Player should be closed using the ClosePlayer function. `OpenPlayer` can not be called at interrupt level.

PROTOTYPE

C ANSI
```
short OpenPlayer (MidiName playerName);
```

ARGUMENTS

playerName: the name of the Player that is the name of the corresponding MidiShare application.

RESULT

The result is a Player reference number.

EXAMPLE 1 (ANSI C)

Opening of a new empty Player.

```
void Example1 ()
{
            short  myRefnum;

            myRefnum = OpenPlayer ("\pExample1");
            ....
            ....

}
```

EXAMPLE 2 (ANSI C)

Opening of a Player with a MIDIfile.

```
void Example2 ()
{
            short  myRefnum;
            MidiFileInfos myInfo;

            MidiSeqPtr mySeq = MidiNewSeq();
            MidiFileLoad( "HD500:midifile1", mySeq, &myInfo);

            myRefNum = OpenPlayer("\pExample2");

            SetAllTrackPlayer (myRefNum , mySeq ,  myInfo.clicks);
            ....
            ....
}
```

See also ClosePlayer.

PausePlayer

DESCRIPTION

Stop a Player without sending the chased events (key-off ...).

PROTOTYPE

C ANSI `void` **`PausePlayer`** `(short refnum);`

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

See also StartPlayer, StopPlayer, ContPlayer.

RecordPlayer

DESCRIPTION

Sets a track in record mode. A Player can record in one track at a time. RecordPlayer called with kNoTrack parameter will disable all recording. The track being recorded can be changed while recording.

PROTOTYPE

C ANSI `void **RecordPlayer** (short refnum, short tracknum);`

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

tracknum : a 16-bit integer, it is the number of the track to be set in record mode: from 0 to 255. Parameter `kNoTrack` will switch off all recording.

EXAMPLE 1 (ANSI C)

Sets a track in record mode and then start the Player.

```
void StartRecord (short refnum, short tracknum)
{

            RecordPlayer (refnum, tracknum);
            StartPlayer(refnum);

}
```

EXAMPLE 2 (ANSI C)

Stop recording .

```
void StopAllRecording (short refnum)
{

            RecordPlayer (refnum, kNoTrack);

}
```

See also SetRecordModePlayer.

SetAllTrackPlayer

DESCRIPTION

Replaces all tracks of a Player with a new MidiShare sequence. All tracks are mixed in a unique MidiShare sequence and should be distinguish by the value of the refnum field of the Midi events .

The MidiShare sequence given as parameter will be internally used. It means that **one must copy it before using** `SetAllTrackPlayer` if one wants to keep it. `SetAllTrackPlayer` can not be called at interrupt level.

PROTOTYPE

C ANSI
```
long SetAllTrackPlayer (short refnum, MidiSeqPtr s,
                                      long tpq);
```

ARGUMENTS

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

s : a MidiSeqPtr, is a pointer on a sequence to be set in the Player.

tpq : tick_per_quarter : the number of ticks per quarter note, usually read in the MIDIfile header.

EXAMPLE (ANSI C)

Sets a new sequence read from a MIDIfile into an already opened Player. The sequence is used internally by SetAllTrackPlayer.

```
short       myRefNum;
.....

void SetNewMIDIfile( short refNum, filename)
{
            MidiSeqPtr seq;
            MidiFileInfos info;

            seq = MidiNewSeq();
            MidiFileLoad(filename, seq, &info);

            SetAllTrackPlayer (refnum, seq, info.clicks);

}
```

See also GetAllTrackPlayer, GetTrackPlayer, SetTrackPlayer.

SetLoopEndBBUPlayer

DESCRIPTION

Sets the position of the loop end marker with a position given in musical time (Bar, Beat, Unit). This function can be used when the Player is moving .

PROTOTYPE

C ANSI `long` **`SetLoopEndBBUPlayer`** `(short refnum, PosPtr pos);`

ARGUMENTS

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

pos : a PosPtr is a pointer to a position in musical time (Bar, Beat, Unit).

RESULT

The result is a Player error code. This error is returned when trying to set a Loop end marker **before** a the Loop start marker or a Loop start **after** a Loop end marker.

See also SetLoopEndMsPlayer, SetLoopStartMsPlayer, SetLoopStartBBUPlayer, SetLoopPlayer.

SetLoopEndMsPlayer

DESCRIPTION

Sets the position of the loop end marker with a date given in millisecond. This function can be used when the Player is moving.

PROTOTYPE

C ANSI `long` **`SetLoopEndMsPlayer`** `(short refnum long date);`

ARGUMENTS

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

date : a date in millisecond.

RESULT

The result is a Player error code. This error is returned when trying to set a Loop end marker **before** a the Loop start marker or a Loop start **after** a Loop end marker.

See also SetLoopEndBBUPlayer, SetLoopStartMsPlayer, SetLoopStartBBUPlayer, SetLoopPlayer.

SetLoopPlayer

DESCRIPTION

Switch on/off the loop state for the Player.

PROTOTYPE

C ANSI `void **SetLoopPlayer** (short refnum, short state);`

ARGUMENTS

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

state : k`LoopOn` or k`LoopOff`.

See also SetLoopEndBBUPlayer, SetLoopEndMsPlayer, SetLoopStartMsPlayer, SetLoopStartBBUPlayer.

SetLoopStartBBUPlayer

DESCRIPTION

Sets the position of the loop start marker with a position given in musical time (Bar, Beat, Unit). This function can be used when the Player is moving.

PROTOTYPE

C ANSI `long` **`SetLoopStartBBUPlayer`** `(short refnum,PosPtr pos);`

ARGUMENTS

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

pos : a PosPtr is a pointer to a position in musical time (Bar, Beat, Unit).

RESULT

The result is a Player error code. This error is returned when trying to set a Loop end marker **before** a the Loop start marker or a Loop start **after** a Loop end marker.

See also SetLoopEndBBUPlayer, SetLoopEndMsPlayer, SetLoopStartMsPlayer, SetLoopPlayer.

SetLoopStartMsPlayer

DESCRIPTION

Sets the position of the loop start marker with a date given in millisecond. This function can be used when the Player is moving.

PROTOTYPE

C ANSI `long` **`SetLoopStartMsPlayer`** `(short refnum, long date);`

ARGUMENTS

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

date : a date in millisecond.

RESULT

The result is a Player error code. This error is returned when trying to set a Loop end marker **before** a the Loop start marker or a Loop start **after** a Loop end marker.

See also SetLoopEndBBUPlayer, SetLoopEndMsPlayer, SetLoopStartBBUPlayer, SetLoopPlayer.

SetParamPlayer

DESCRIPTION

Sets parameters which define the behavior of a track: a track can be **muted** or played **solo**.

C ANSI `void SetparamPlayer (short refNum, short tracknum, short param, short val);`

ARGUMENTS

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

tracknum : a 16-bit integer, is the track number to be set.

param : is the parameter to be set (`kMute` or `kSolo`).

val : is the value to be set, can be `kMuteOn, kMuteOff, kSoloOn or kSoloOff.`

EXAMPLE (ANSI C)

Mute the 10 first tracks.

```
short       myRefNum;
.....

void MuteTracks( short refNum, seq)
{
            short i;

            for (i = 1 ; i<= 10; i++)
             SetParamPlayer (refnum, i, kMute, kMuteOn);

}
```

See also GetParamPlayer .

SetPosBBUPlayer

DESCRIPTION

Changes the current position in the score. The new position is given as a position in musical time (Bar, Beat, Unit). This function can be used when the Player is moving.

PROTOTYPE

C ANSI `void SetPosBBUPlayer (short refnum, PosPtr pos);`

ARGUMENTS

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

pos : a PosPtr is a pointer to a position in musical time (Bar, Beat, Unit).

EXAMPLE (ANSI C)

Move the Player to the beginning of the 10° bar.

```
void MovePos (short refNum)
{
            Pos  pos;

            pos.bar = 10;
            pos.beat = 1;
            pos.unit = 1;

            SetPosBBUPlayer (refNum, &pos);

}
```

See also SetPosMsPlayer.

SetPosMsPlayer

DESCRIPTION

Changes the current position in the score. The new position is given as a date in millisecond. This function can be used when the Player is moving. The conversion between the argument date in ms and the internal date in ticks is always done using the Tempo Map information of the Player.

PROTOTYPE

C ANSI `void SetPosMsPlayer (short refnum, long date);`

ARGUMENTS

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

date : a date in millisecond.

EXAMPLE (ANSI C)

Move the Player to the 10 s of the score.

```
void MovePlayer (short refNum)
{
            long date;

            date = 10000;

            SetPosMsPlayer (refNum, date);

}
```

See also SetPosBBUPlayer.

SetRecordFilterPlayer

DESCRIPTION

Allows to set a record filter for the Player. Midi events can be filtered by types, channels and ports. The record filter does not interfere with the receive filter of the Player.

PROTOTYPE

C ANSI
```
void SetRecordFilterPlayer (short refnum,
                            FilterPtr filter);
```

ARGUMENTS

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

filter: a MidiShare FilterPtr (defined in MidiShare.h).

EXAMPLE (ANSI C)

Set the filter to record synchronization events.

```
void SetFilter (FilterPtr filter)
{
            AcceptBit(filter->EvType,typeSongPos);
            AcceptBit(filter->EvType,typeClock);
            AcceptBit(filter->EvType,typeStart);
            AcceptBit(filter->EvType,typeContinue);
            AcceptBit(filter->EvType,typeStop);

            SetRecordFilterPlayer (refNum, filter);
}
```

SetRecordModePlayer

DESCRIPTION

Set the recording mode. A recording can be done in two mode :

- `kMixMode` : recorded events are mixed in the recorded track.

- `kEraseMode` : the track is completely deleted and recorded events are then inserted in the track. By default the recording state is set in `kMixMode`.

PROTOTYPE

C ANSI `void` **`SetRecordModePlayer`** `(short refnum, short state);`

ARGUMENTS

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

state : the recording state : can be k`EraseMode` or k`MixMode.`

See also RecordPlayer.

SetSMPTEOffsetPlayer

DESCRIPTION

For SMPTE synchronization mode, allows to set a SMPTE offset.

PROTOTYPE

C ANSI
```
void SetSMPTEOffsetPlayer (short refnum, SmpteLocPtr smptepos);
```

ARGUMENTS

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

smptepos : a MidiShare SmpteLocPtr (defined in MidiShare.h).

SetSynchroInPlayer

DESCRIPTION

Set the Player mode of synchronization, a Player can be in four synchronization mode :

-k`InternalSync` : the Player uses its internal tempo map

-k`ClockSync` : The Player recognizes MIDI Clock at it's input and can be driven by Start, Stop, Continue, Clock and SongPos Midi messages.

-k`SMPTESync` : This is a global synchronization mode for all MidiShare applications. A SMPTE offset can be set, and the Player will automatically starts when the SMPTE offset is reached. The Player will stop when incoming MTC disappears.

-k`ExternalSync` mode: The Player does not use it's internal Tempo Map anymore, recognizes incoming **Tempo** events and the internal tempo can be changed with the `SetTempo` function.

The synchronization mode can be changed only when the Player is stopped. By default, the synchronization mode in k`InternalSync`.

PROTOTYPE

C ANSI
```
void SetSynchroInPlayer (short refnum, short
                                        synchrostate);
```

ARGUMENTS

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

synchrostate : the synchronization state which can be k`InternalSync`, k`ClockSync`, k`SMPTEsync` or k`ExternalSync` .

EXAMPLE (ANSI C)

Set a Player in k`ClockSync` mode.

```
void SetClockSync (short refNum)
{
            SetSynchroInPlayer (refNum , kClockSync);
}
```

See also SetSynchroOutPlayer .

SetSynchroOutPlayer

DESCRIPTION

Set the Player mode of synchronization sending, which can be kNoSyncOut or kClockSyncOut. In this case start, stop, continue, and clock Midi messages are sent when the Player is running. By default, the synchronization mode in kNoSyncOut.

PROTOTYPE

C ANSI
```
void SetSynchroOutPlayer(short refNum, short
                                        synchrostate);
```

ARGUMENTS

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

synchrostate : the synchronization state which can be kNoSyncOut or kClockSyncOut.

See also SetSynchroInPlayer .

SetTempoPlayer

DESCRIPTION

Allows to change the current tempo when the Player is in `kExternalSync` mode. Tempo is a value in micro-sec/per/quarter-note.

PROTOTYPE

C ANSI `void` **`SetTempoPlayer`**`(short refNum, long tempo);`

ARGUMENTS

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

tempo : the new tempo value in micro-sec/per/quarter-note.

SetTrackPlayer

DESCRIPTION

Replace a track in a Player with a new MidiShare sequence. The existing track will be erased before the new track is set.

The MidiShare sequence given as parameter will be internally used. It means that **one must copy it before using** `SetTrackPlayer` if one wants to keep it. `SetTrackPlayer` can not be called at interrupt level.

PROTOTYPE

C ANSI
```
long SetTrackPlayer(short refnum, short tracknum,
                                  MidiSeqPtr s);
```

ARGUMENTS

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

tracknum : a 16-bit integer, is the track number to be set.

s : a MidiSeqPtr, is a pointer on a sequence to be set in the Player.

RESULT

The result is a Player error code.

EXAMPLE (ANSI C)

Example1 : Set a new sequence into an already opened Player on track 1. The sequence is used internally by the `SetTrackPlayer` .

```
short       myRefNum;
.....

void SetTrack( short refNum, seq)
{

            SetTrackPlayer (refnum, 1, seq);

}
```

See also GetAllTrackPlayer, GetTrackPlayer, SetAllTrackPlayer.

StartPlayer

DESCRIPTION

Starts a Player from the beginning of the score.

PROTOTYPE

C ANSI `void StartPlayer (short refnum);`

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

See also StopPlayer, ContPlayer, PausePlayer.

StopPlayer

DESCRIPTION

Stop a Player and send the chased events (key-off ...)

PROTOTYPE

C ANSI `void` **`StopPlayer`** `(short refnum);`

ARGUMENTS

refnum : a 16-bit integer, is the reference number of the Player.

See also StartPlayer, ContPlayer, PausePlayer.

Index