MidiShare

Kernel Development Guide

v.1.0

GRAME Research Lab.
9, rue du Garet BP 1185
FR - 69202 LYON CEDEX 01
Tél +33 (0)4 720 737 00 Fax +33 (0)4 720 737 01
grame@rd.grame.fr
http://www.grame.fr/Research/

GRAME

Summary

1. Introduction

MidiShare is a real-time multi-tasks MIDI operating system specially devised for the development of musical applications. MidiShare provides high level services to the field of computer music and MIDI applications. It is freely available to users and developers. MidiShare exists since 1989 and several releases have been developped for various platforms : Macintosh, Atari, Windows 3.1, Windows 95, Windows 32 bits thunk release. Embedded versions have also been developped.

The MidiShare source code is now publicly available under the GNU Library General Public License. You should have received a copy of this license along with the MidiShare source code; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

This manual is intended for developers who wish to contribute to the MidiShare kernel development. It contains a complete description of the kernel architecture and of the related source code. A special attention is given to the portability issues.

For more information concerning the MidiShare functions and the MidiShare applications development issues, you should refer to the MidiShare documentation, available at the MidiShare ftp sites :

```
ftp://ftp.grame.fr/pub/MidiShare
```

or

```
ftp://putney.music.uiowa.edu/pub/MidiShare
```

Special MidiShare mailing lists exist for both MidiShare applications and kernel development:

- the “midishare-dev” list : for support and discussion about MidiShare compatible applications development.
  To subscribe, send the following :
  `subscribe midishare-dev <your-email-adress>`
  in the body of a message to majordomo@rd.grame.fr

- the “midishare-kernel” list : for support and discussion about MidiShare kernel development, including porting and extension of the kernel.
  To subscribe, send the following :
  `subscribe midishare-kernel<your-email-adress>`
  in the body of a message to majordomo@rd.grame.fr

If you plan to contribute to the kernel development, you should subscribe to the “midishare-kernel” mailing list where are discussed all the changes and the development issues. This document assumes that you are familiar with the MidiShare Development documentation, where are explained all the services provided by the kernel to its client applications.

2. Kernel Description

2.1. General architecture

MidiShare is based on a client/server model. It is composed of six main components : a Memory Manager, a Time Manager, a Task Manager, a Communication Manager, a Scheduler and a Ports Manager.

The figure 1 shows the conceptual model of MidiShare :

- The *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* maintains the current date of the system. It offers 1ms resolution and supports accurate transparent SMPTE synchronization.
- The *Communication Manager* routes events received from the scheduler to the client applications and ports manager according to the connections set between applications.
- The *Task Manager* is in charge of calling the tasks delivered by the scheduler.
- The *Memory Manager* is 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 *Ports Manager* is in charge of the communications with the drivers. It supports up to 256 ports. It provides the mechanisms to plug the drivers in the kernel at wake up time.

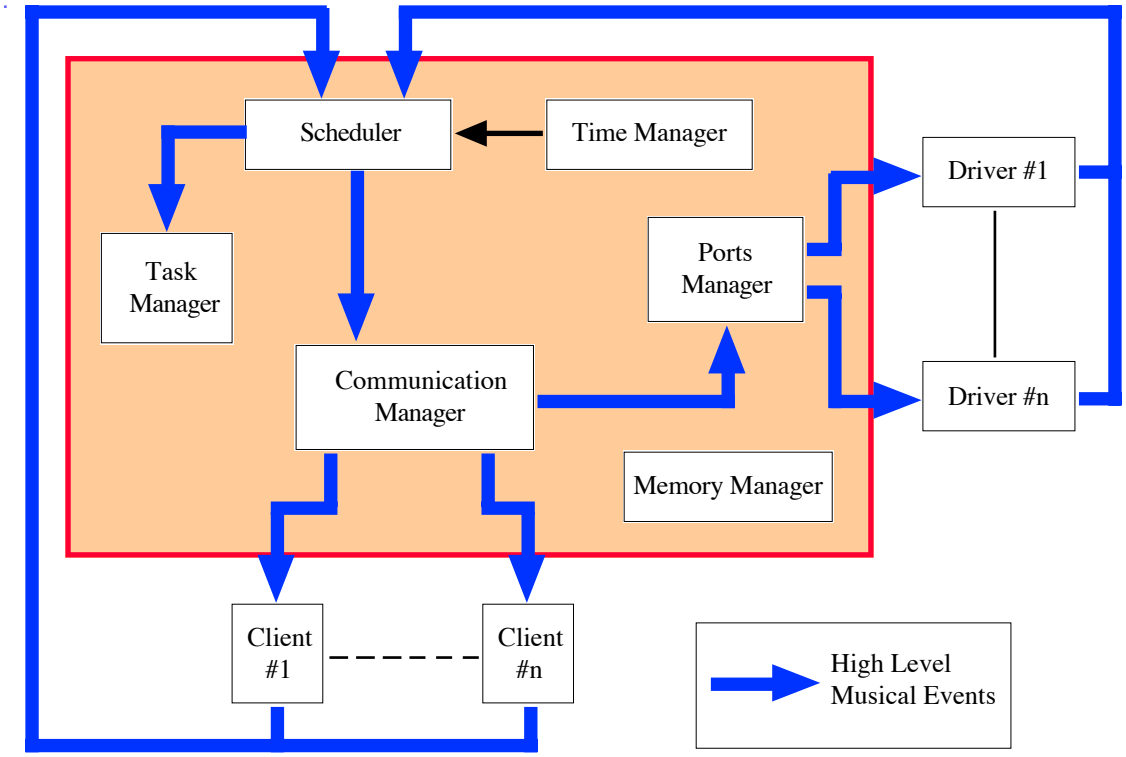

Figure 1 : the conceptual model of MidiShare

2.2. Changes

The reorganization of the source code, mainly intended to facilitate porting, was the opportunity to introduce some changes in the kernel architecture. Non public releases of MidiShare included input and output MIDI drivers. The following releases are concerned:

- MidiShare for the Macintosh up to version 1.68
- MidiShare for the Atari up to version 1.68
- MidiShare 16 bits for Windows 3.1 up to version 1.57
- MidiShare 32 bits for Windows 95 up to version 1.04

In order to facilitate drivers development and to take a better account of the different platforms particularities, the drivers are now separate components and a new manager, the Ports Manager, is part of the kernel architecture. It is in charge of loading and plugging these drivers into the kernel.

The Port Manager design is in progress. Therefore, it is not present in this distribution of the source code. It is part of the first tasks to do to complete the new implementation of the kernel. Its design will be discussed on the midishare-kernel mailing list.

2.3. General operation

2.3.1. Waking up MidiShare

As long as no client application is running, MidiShare is dormant and has no effect on the operation of the computer. Following the first MidiOpen, MidiShare wakes up and becomes active. It should then perform the tasks necessary to come into service. These tasks represent:

- initializing the kernel data structures.
- allocating and initializing the MidiShare memory space; both private memory and events memory are concerned.
- initializing the Ports Manager.
- initializing a task which will be called by interrupt at the MidiShare time resolution rate (should be every millisecond). We'll refer later to this task as the *Time Task*.

2.3.2. Putting MidiShare to sleep

MidiShare returns to its dormant state after the last MidiClose is performed. It should then execute the corresponding reverse tasks:

- stopping and releasing the Time Task.
- putting the ports Manager to sleep.
- releasing the memory space.

After returned to its dormant state, MidiShare should have again no effect on the operation of the computer.

2.3.3. The Time Task

Most of the MidiShare services are performed by the Time Task. Called at the MidiShare time resolution rate, the Time Task is in charge of the following:

- maintaining the date of the system.
- calling the Scheduler to perform a one step sorting and to get the ready events.
- dispatch the ready events according to their destinations. These destinations are determined by the current connections set. When the destination is a client application, events are stored in its fifo. When the destination is MidiShare itself, events are passed to the Ports Manager.
- activating the real-time tasks scheduled by client applications.
- activating the real-time receive alarms for the concerned applications ie applications which received a new event in their fifo.

Real-time tasks, receive alarms, inter-applications communication, input / output (using MIDI protocol or any other implemented by the drivers) are mainly dependent of the Time Task operation. As this task should be called every millisecond, a great attention must be given to its implementation and especially to its efficiency.

2.3.4. Events Memory Management

The MidiShare memory management is based on fixed-sized cells (16 bytes) organized into events. The services provided by the kernel are widely based on these events: they are used for inter application communication, connections set, tasks storage, MIDI and MIDIFile data storage... All the events are composed of a header cell that may be followed by one or more extension cells. Figure 2 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 have always 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.

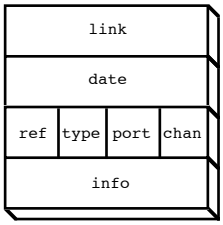

Figure 2 : common event structure

MIDI messages with 0, 1 or 2 data bytes, use only one cell, as shown in figure 2. Their datas are stored in the Info field.

Some events (private events, process or dprocess events) need more data storage. An extension cell is allocated and the Info field of the main cell points to this extension like shown in figure 3.

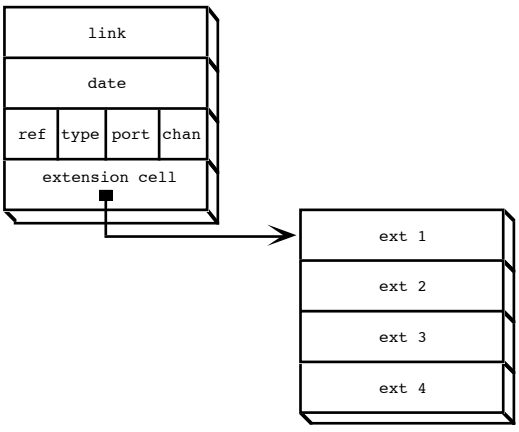

Figure 3 : 2 cells event structure

Variable length messages like System Exclusive, Stream or some MidiFile type messages uses several linked cells to store their variable number of fields. Their structure is described on figure 4. The main cell points to the last extension cell, which points to the first extension cell. This structure is intended to provide the most efficient way to add datas to a variable length message. In particular, it will minimize the operations required to process a MIDI System Exclusive stream from input.

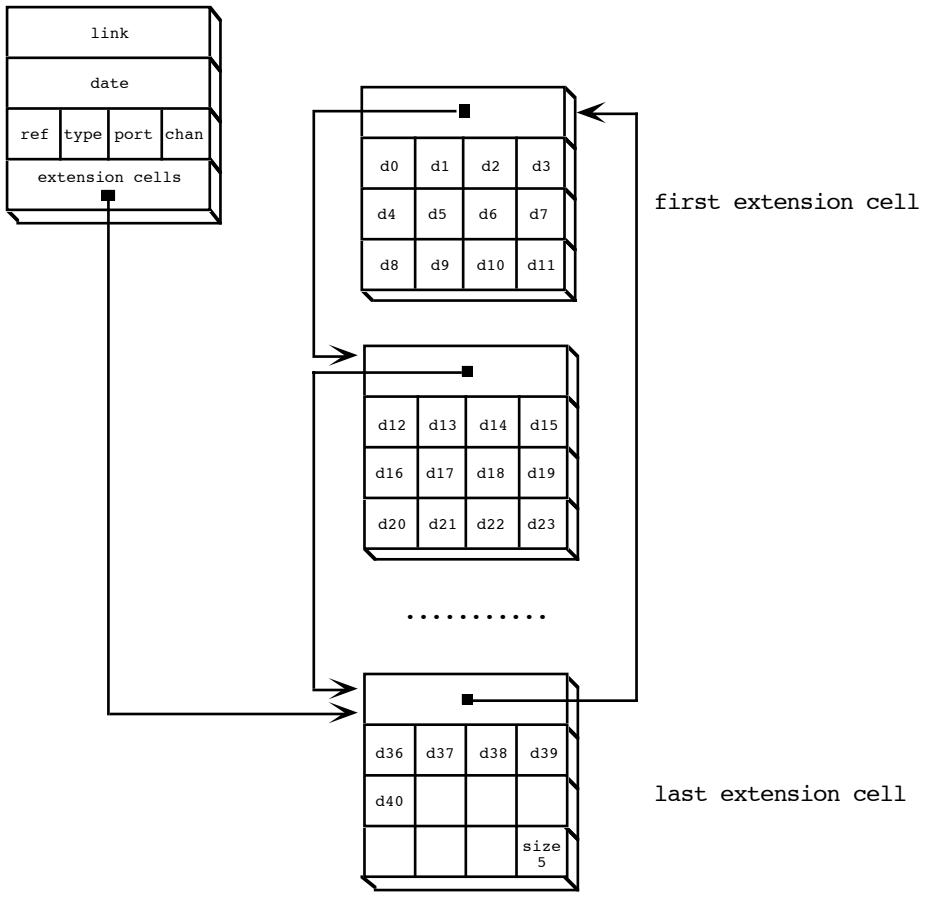

Figure 4 : variable length events structure

3. Portability issues

Several low level implementation parts of the kernel are platform dependent and generaly concern critical services provided by the host operating system. Among them are: installing an interrupt service routine, sharing the events memory, switching from the MidiShare kernel to a client process and synchronizing the client processes. Modern operating systems provides high level functions to perform all these tasks but they are generally not suitable to operate in a realtime context. In particular, acuracy and efficiency issues are critical for a correct operating of the MidiShare kernel. Therefore, the implementation problems raised for each point are detailed below.

3.1. The MidiShare process

The MidiShare process represents the entity which owns the necessary resources for the kernel operations ie memory and interrupt service routine. It must ensure of the permanence of these resources all along the kernel active period, from wake up time until sleep time. The MidiShare process do not necessary refer to a process as commonly defined by the operating systems. Its implementation is generally dictated by the host operating system and may or may not be such a process. One of the design issue is to choose the most appropriate system resource for the implementation of the MidiShare process. Efficiency and accuracy must be particularly taken into account : for example, if the MidiShare process is implemented as an operating system process and if the process context switching costs are high, the resulting kernel will certainly be inefficient as it will have to pay these costs every millisecond.

More generally, the interrupt service routine latency time should not exceed 5% of the interrupt periodicity time. In the case of MidiShare, it means that serving the interrupt should be done in less than 50 μsec.

Implementation

The following functions are to provide by any platform dependant implementation :

`void SpecialWakeUp (TMSGlobalPtr g)`
called at wakeup time. If necessary, this function should create the process which will own and manage the kernel critical resources. The argument ‘g’ points to the global MidiShare environment.

`void SpecialSleep (TMSGlobalPtr g)`
called at sleep time. If a separate process exists, in charge of the kernel resources, this function should cleanly close and release this process. The argument ‘g’ points to the global MidiShare environment.

`void OpenTimeInterrupts (TMSGlobalPtr g)`
this is the last task invoked at wakeup time. This function should activate an interrupt routine called every millisecond. The argument ‘g’ points to the global MidiShare environment.

`void CloseTimeInterrupts(TMSGlobalPtr g)`
this is the first task invoked at sleep time. This function should remove the interrupt routine and if necessary, release its associated resources. The argument ‘g’ points to the global MidiShare environment.

3.2. Memory allocation

Memory is allocated and owned by the MidiShare process. The implementation can freely use any way to allocate the kernel internal memory. It is different for events and filters memory :

- events memory must be shared by the MidiShare process and all the client applications : a pointer to a MidiShare event must refer to the same memory location whatever its context of use. It's the responsability of the MidiShare process to allocate such a memory at wake up time.
- filters memory must be shared by its application owner and the MidiShare process. It's different from the events memory because a filter is private to one application. Filter memory is allocated by the MidiShare process at application request by the way of the MidiNewFilter function.

The source code clearly separates these services using an allocation function which takes as argument the type of the desired memory.

Implementation

The following functions are to provide by any platform dependant implementation :

`FarPtr(void) AllocateMemory (MemoryType type, long size)`
this function should allocates memory according to the desired 'type' argument. The 'size' argument specifies the desired memory size.

`void DisposeMemory (FarPtr(void) memPtr)`
this function should release any memory block previously allocated using 'AllocateMemory '. the 'memPtr' argument points to the memory block.

Memory types are defined as below:

```
typedef enum {
   kStdMemory,         /* no constraints on the allocated memory */
                       /* should only be accessed by the calling */
                       /* process                                */

   kSharedMemory,      /* memory should be shared by the kernel */
                       /* and all the client applications.      */

   kernelSharedMemory /* memory should be shared by the kernel */
                       /* and the calling process               */
} MemoryType;
```

3.3. Tasks and Alarms

The MidiShare kernel provides realtime tasks and realtime alarms to its client applications. Realtime tasks are stored in typeProcess events, then inserted in the scheduler and activated at interrupt level by the MidiShare process at falling date. Realtime alarms are activated at application request in two cases :

- when a context change occurs: it corresponds to an application alarm, setup using MidiSetApplAlarm, it can be activated at any time, at interrupt or user level.
- when news events are stored in the application fifo: it corresponds to a receive alarm, setup using MidiSetRcvAlarm, it will always be activated at interrupt level.

The way used to activate a client task is platform dependent. Some operating systems allows the MidiShare process to directly call the client code: on the MacOS for

example, the application context is limited to a register context and can be easily restored at no cost, allowing a task to be called by the kernel for an application account. Some other operating systems prevents such mechanisms by keeping separate address spaces for each application: this is for example the case of Windows 95 where realtime tasks are implemented as part of a thread owned by the client application ; activating a task consists then in waking up this thread, which raises all the problems of acuracy, priority and context switching costs.

As realtime tasks and alarms are among the key services of the kernel, their implementation must take a great care of acuracy and efficiency. Activation of a client task is defined in the implementation interface file.

Implementation

The following functions are to provide by any platform dependant implementation :

```
TApplContextPtr CreateApplContext ()
```
called when a MidiShare client opens. This function should provide the information necessary to restore its context at interrupt time.

```
void DisposeApplContext (TApplContextPtr context)
```
called when a MidiShare client quit. If necessary, this function should release the resources allocated to create the application context.

```
void CallApplAlarm (TApplContextPtr context, ApplAlarmPtr alarm,
                    short refNum, long alarmCode)

void CallRcvAlarm  (TApplContextPtr context, RcvAlarmPtr alarm,
                    short refNum)
```
these functions should provide a way to activate the corresponding alarms. They receive as arguments the previously created application context and the arguments of the alarm.

```
void CallTaskCode  (TApplContextPtr context, TTaskExtPtr task,
                    long date, short refNum)

void CallDTaskCode (TApplContextPtr context, TTaskExtPtr task,
                    long date, short refNum)
```
these functions should provide a way to activate the corresponding tasks. They receive as arguments the previously created application context and part of the arguments of the alarm. User arguments can be accessed using the 'task' pointer.

Note that 'CallDTaskCode' should not provide any particular mechanism to restore the application context: as the function is called by the client application and not at interrupt time, the application context should be already available to the caller.

3.4. Processes and synchronization

Concurrent access to critical sections of the MidiShare code raises the synchronization problems. Preemptive operating systems provides the necessary mechanisms to solve these problems but again, these mechanisms are generaly not suitable in a realtime context : for example in Windows 95, waiting for a semaphore induce priority

inversions and results in bad performances from the efficiency and acuracy point of view.

The current implementation generaly avoids the use of semaphores by the way of lock free shared linked lists. The principle consists in trying to modify a shared linked list using the synchronization mechanisms generally available with modern micro-processors instructions set : for example, the Motorola PowerPC microprocessor family allows to put reservations on a memory zone, then, conditionnal instructions operates only if the reservation is not altered, on the contrary, the program have to loop until success. Such mechanisms needs to be implemented using assembly language.

The only semaphore defined by the system is used to synchronize the kernel wakeup and sleep tasks. The semaphore is activated when a client application call MidiOpen or MidiClose.

Implementation

The following functions are to provide by any platform dependant implementation :

Functions to modify shared linked lists:

```
FarPtr(void) PopSync  (msListPtr adr)
```
should pop and return the first element of the list pointed by 'adr'

```
void PushSync (msListPtr adr, msListPtr link)
```
should push the 'link' element to the list pointed by 'adr'

```
void PushSyncList (msListPtr adr, msListPtr head, msListPtr tail)
```
should link the 'tail' element to the list pointed by 'adr' and store the 'head' element at the head of the list.

Function to exchange a value in memory:

```
Boolean CompareAndSwap (FarPtr(void) *adr, FarPtr(void) compareTo,
                        FarPtr(void) swapWith)
```
should compare the 'compareTo' argument with the value adressed by 'adr'. If equal, the function should swap it with the 'swapWith' argument and return true, otherwise, it should do nothing and return false.

Function to forget a realtime task:

```
Boolean  ForgetTaskSync (MidiEvPtr * taskPtr, MidiEvPtr content)
```
should store a typeDead value into the type field of the 'content' argument, clear the event pointed by the 'taskPtr' argument and return true in case of success. This function should take account of that it could be interrupted by the the task to forget itself.

Function for applications fifo management:

```
MidiEvPtr ClearFifo  (TFifoPtr fifo)
```
should provide a synchronized way to clear the head and count fields of the 'fifo' argument. The fifo tail field should points to the fifo head.

```
MidiEvPtr PopFifoEv  (TFifoPtr fifo)
```
should provide a synchronized way to pop an event from the 'fifo' argument and to update the fifo count field.

```
void PushFifoEv (TFifoPtr fifo, MidiEvPtr ev)
```
should provide a synchronized way to push an event to the 'fifo' argument and to update the fifo count field.

Semaphores implementation:

`MutexResCode OpenMutex (MutexRef ref)`
should allocate a reservation on the mutex defined by the 'ref' argument. Only the kWakeUpMutex constant is currently defined. Other types should be rejected. The returned value is defined as below:

```
enum { kSuccess, kTimeOut, kUnknownMutex, kFailed };
```

`MutexResCode CloseMutex (MutexRef ref)`
should release a reservation previously allocated using 'OpenMutex'

4. Source code

4.1. Headers

Header files generally used by all the kernel components

file	content
msTypes.h	type definitions
msDefs.h	MidiShare constants definitions, basic MidiShare data types structures
msExtern.h	defines all the functions to provide for a platform dependant implementation

Header files specific to each kernel components

file	content
msKernel.h	kernel global data structure
msAlarms.h	interface for the alarm call function
msAppFun.h	interface for the application configuration functions
msAppls.h	constants definition, data types and data structures for the clients applications management
msConnx.h	interface for the connections management functions
msEvents.h	interface for the events memory management functions
msFields.h	interface for the events fields access functions
msFilter.h	interface for the filters management functions
msInit.h	interface for the initialization functions
msMail.h	interface for the mailbox functions
msMemory.h	constants definition, data types and data structures for the memory management
msSeq.h	interface for the MidiShare sequences functions
msSmpte.h	interface for the smpte functions
msSorter.h	the sorter data structures and external functions
msSync.h	some macros build on top of the synchronization functions
msTasks.h	task extension structure and task management functions
msTime.h	interface for the MidiShare time management functions
msXmtRcv.h	interface for the MidiShare input / output functions

4.2. Kernel

The kernel implementation provides the glue for all its components. It is in charge of initializing all these components and providing the time task. MidiShare time functions are implemented in this section.

file	content
msHandler.c	includes the clock handler implementation and the dispatch of the events returned by the sorter.
msInit.c	provides initialization functions (including MidiShareWakeUp and MidiShareSleep). Also in charge of the MidiShare version number.
msSmpte.c	smpte functions (not yet implemented)
msTime.c	includes time and time conversions functions.

4.3. Clients

The clients part of the implementation includes all the functions dedicated to an application configuration, connections management, tasks and alarms management, sending and receiving events.

file	content
msAlarms.c	implementation of an application alarm call
msAppls.c	provides applications configuration functions (MidiOpen, MidiClose, setting and getting alarms, setting and getting filters...)
msConnx.c	inter-applications connections management implementation
msFilter.c	filters management implementation
msMail.c	synchronized mailbox functions
msTasks.c	realtime and deferred tasks management
msXmtRcv.c	provides functions for sending and receiving MidiShare events.

4.4. Memory

The memory section of the implementation is in charge of the musical events memory management. It includes also the sequences management functions.

file	content
msEvents.c	provides the functions for the MidiShare events management.
msFields.c	utilities for event fields access
msMemory.c	global memory management : opening and closing the memory, growing the MidiShare memory space, getting free space.
msSeq.c	sequences management implementation

4.5. Sorter

file	content
msSorter.c	the sorter implementation