The Memory Library Programmer's Manual

THOMAS RICHTER

Contents

1 Introduction to the MemoryLib

The very purpose of the memory library is to offer the power of “virtual memory” to the Amiga Operating System. Now, what is virtual memory then? It’s the illusion of memory that is “not really there”. Hence, programs using virtual memory access so called *logical* memory addresses that are not necessarily represented by physical hardware. The contents of these addresses is then currently not available in system RAM, but represented on some external hardware, typically a hard disk. If this happens, one says that the corresponding logical address is *swapped out.* Whenever a program tries to access a logical address that is currently not represented by the hardware, the memory library interacts, allocates real physical memory from the system, and loads the corresponding data from the swap device — as said typically a hard disk — into the physical memory. All this happens under the feet of the application program, as a completely transparent operation. An application using the memory library will not be aware that parts of the addresses it is currently using are not in main memory, just because the memory library will load them into memory whenever the program tries to access them. Therefore the word “illusion”. This illusion is quite useful, though: Programs that require more memory than available main memory could use virtual memory instead. The memory library keeps care about which parts of the required memory are really loaded in the main memory, and which parts may reside on disk. Another application of virtual memory is to mirror a huge file in memory, as if it has been loaded as a single gigantic block. As an application accesses this file, the memory library loads and disposes parts of this file automatically for you — hence, it seeks in the file as the application accesses different portions of its image.

Obviously, virtual memory requires some support by the hardware. The responsible hardware circuit is the MMU, short for *memory management unit.* The memory library does not attempt to program the MMU itself; rather, it leaves this dirty work for another library, the mmu library. This library comes with its own documentation you should have a look into as well.

On the other hand, virtual memory requires some support of the operating system as well. Unfortunately, the initial design of Amiga OS did not keep care about virtual memory, mainly because at the time of its writing MMUs were not an option for a low-priced customer system. Some preparations have been made, at least, but little application programs keep care. The conclusion of this constructional problem is that *there is no transparent integration of virtual memory into Amiga OS.* This does not mean that there cannot be virtual memory at all. It just means that applications that want to use virtual memory have to make special preparations to make use of it. The memory library will not, though, make old applications use virtual memory transparently. As said, there is no way to make this happen. The memory library will, though, provide the required interface functions to manage virtual memory, should an application require it.

1.1 Supported Hardware

The memory library supports all hardware the mmu library supports. In fact, as long as an mmu library is present, it is completely independent of the underlying hardware. At the time of writing, the 68020/68851 combination as well as the 68030, 68040 and 68060 processors are supported by the mmu library, and hence by the memory library.

1.2 Basic Concepts

The memory library handles two basic objects: The `AdrSpace` and the `VMPool`. An `AdrSpace` defines the stage on which virtual memory plays — if an exec `Task` *enters* an `AdrSpace`, it shares the same logical addresses with all other `Tasks` that are part of this `AdrSpace`. Hence, the `AdrSpace` links virtual addresses to a swapping mechanism, and hence defines the meaning of virtual addresses. If you know the interface of the mmu library, then this might sound quite common. A very similar concept exists on the mmu library, namely that of an `MMUContext`. In fact, an `AdrSpace` is a natural extension of the mmu library `MMUContext`, and each `MMUContext` may carry *at most one* `AdrSpace`. On the contrary, each `AdrSpace` may also only extend one `MMUContext`. Hence, there is a one-to-one relation between `address spaces` and `MMUContexts`. Similar to the `MMUContext`, `AdrSpaces` don't have any documented structure, you won't need any to make use of it.

Each exec `Task` may run as part of an `AdrSpace`, and as exec schedules `Tasks`, it also schedules `AdrSpaces`. Therefore, `AdrSpaces` are similar to "processes" of other operating systems, e.g. Unix, and exec `Tasks` are more like "threads" because any two `Tasks` belonging to the same address space may see and modify the data of each other. Virtual memory allocated within one `AdrSpace` is, however, invisible to `Tasks` of other `AdrSpaces`.

Unlike mmu library `MMUContexts`, *no initial* `AdrSpace` *exists* if the system boots up. Hence, all `Tasks` are "only" able to see the system wide shared memory. You need to build an `MMUContext` and an `AdrSpace` on top if you want to use virtual memory, and `Tasks` have to enter this `AdrSpace` explicitly to make use of it.

Virtual memory does have its restrictions since other parts of the operating system are not prepared to handle virtual memory. Virtual memory may not be accessed from within supervisor mode, e.g. within interrupts. The corresponding memory will simply not "appear" at its place, and the result is either reading of invalid data, a MuForce hit, or a system crash. Virtual memory *must not* be accessed while in a **Forbid()** or **Disable()** state. The reason for this restriction is quite simple: Access of virtual memory may require that its contents must be swapped in, i.e. loaded from disk, and this requires a `Task` switch by very construction. For the very same reason, virtual memory must not be accessed while the swapping device is blocked in some way, e.g. the corresponding filing system containing the swap file has been shut down. Since one cannot ensure which parts of the operating system may access memory without keeping care of these side conditions, and whether the Os passes data structures to other `Tasks` not sharing the same `AdrSpace` in first place, *placing Os structures in virtual memory is illegal.*

`VMPools` are, similar to their exec counterpart, memory pools virtual memory can be allocated from. An `AdrSpace` may provide more than one `VMPool`. Each pool may have its own swapping mechanism if required, and each pool may define special properties of the memory it administrates — as for example the exec memory flags `MEMF_CHIP`, or special MMU caching modes as `MAPP_CACHEINHIBIT`. Except that, a `VMPool` may also mirror the contents of a file. For this application, it cannot be used for memory allocation and release. Rather, its logical addresses are used to address the data within the mirror file.

- An `AdrSpace` is the extension of an `MMUContext`. They are in a one-to-one relationship.
- An `AdrSpace` assigns a common meaning to logical addresses for all `Tasks` that entered it. `Tasks` belonging to different `AdrSpaces` may use the very same logical address for quite different purposes, and may find different data under the very same address.
- Exec `Tasks` must *enter* `AdrSpaces` explicitly to make use of the virtual memory within. If they don't, they are only able to access the shared public memory.
- As exec schedules its `Tasks`, it also schedules `AdrSpaces`.
- Virtual memory must not be accessed while in **Forbid()** or **Disable()** state, or from within supervisor mode.
- Os structures must never be allocated within virtual memory, neither may you pass any virtual memory address to the Os.
- `VMPools` are build within `AdrSpaces`. They are the entity you allocate virtual memory from.
- All memory chunks allocated from a `VMPool` share the same memory and caching flags.

2 Address Spaces

The first step to build a new `AdrSpace` is to build a new `MMUContext`. This job is out of scope of the memory library, but requires the mmu library, namely its **CreateMMUContext()** function. The mmu library provides a huge set of tags for this function, but in the most frequent case, the following should be enough:

```
struct MMUContext *ctx;

ctx = CreateMMUContext(MCXTAG_SHARE,DefaultContext(),TAG_DONE);
```

This will build a new `MMUContext` that shares all definitions from the public global context, i.e. modifications of the global context will be forwarded automatically, except for memory regions declared otherwise.

A new address space is then created on top of this context by the following call:

```
struct MMUContext *ctx;
struct AdrSpace *adr;
char *swapfile;

adr = NewAdrSpace(MEMTAG_CONTEXT,ctx,
                  MEMTAG_VMSWAPTYPE,MEMFLAG_SWAPFILE,
                  MEMTAG_SWAPFILENAME,swapfile, /* swap to this file */
                  MEMTAG_VMSIZE,64*1024*1024,   /* 64 MB size */
                  TAG_DONE);
```

The **NewAdrSpace()** function is tag-based, it provides much more options that are discussed below in full detail in section 4. For the moment, the above example should be enough — it builds an address space on top of the passed-in context that swaps into a file of the indicated name, and is 64MB large.

Address spaces and contexts are disposed in reverse order by the following two functions:

```
struct AdrSpace *adr;

DeleteAdrSpace(adr);
```

and, after disposing the address space,

```
struct MMUContext *ctx;

DeleteMMUContext(ctx);
```

which is again an mmu library call.

2.1 Address Spaces plus Tasks and MMUContexts

A `Task` need not to be part of an `AdrSpace` at all — note that this is an important difference to the mmu library concept where a task is at least part of the public global context. If a task hasn't entered an `AdrSpace`, it may of course still address objects, but only those that are in the global shared memory Amiga Os provides since ever. Virtual memory requires, though, that a `Task` *enters* an `AdrSpace` explicitly, which then defines how the virtual addresses are interpreted.

Entering an `AdrSpace` is done by the following call:

```
struct AdrSpace *adr;
struct Task *task;
BOOL ok;

ok = EnterAddressSpace(adr,task);
```

This call returns `TRUE` for success or `FALSE` on failure. Hence, this call is the "memory" analogue of the mmu library call **EnterMMUContext()**. Similar to the mmu library, it requires the `tc_Switch` and `tc_Launch` function pointers of the task to be patched over.

> *BOOL is not a Context!* Note that **EnterAddressSpace()** may fail if the system runs out of memory. Entering an address space requires the allocation of some private administration structures. Unlike **EnterMMUContext()**, which returns on success the old context the task was part of, this function returns a boolean success/failure indicator.

Similarly, a task may leave an address space on demand:

```
BOOL ok;
struct Task *task;

ok = LeaveAddressSpace(task);
```

Afterwards, the task will have access to system-wide shared memory only. This call cannot fail unless you pass in incorrect parameters, hence you don't need to check the return value typically.

Since `AdrSpaces` and `MMUContexts` are in one-to-one relation, you may want to find out the address space a context is attached to. The following call will either return this address space, or `NULL` in case the context doesn't carry an address space:

```
struct AdrSpace *adr;
struct MMUContext *ctx;

adr = AdrSpaceOf(ctx);
```

Similarly, the underlying context of an address space is returned by the following:

```
struct AdrSpace *adr;
struct MMUContext *ctx;

ctx = MMUContextOf(adr);
```

As an address space is always build on top of a context, this call cannot fail.

By the following call, you find the address space a task is part of:

```
struct Task *task;
struct AdrSpace *adr;

adr = CurrentAddressSpace(task);
```

This returns `NULL` for tasks that may only access global shared memory, or its address space. It will return the address space of the calling task if you set `task` to `NULL`.

2.2 Function Reference

This chart provides a brief reference to the functions mentioned in this chapter:

Table 1: Basic Address Space Functions

MemLib function	Description
NewAdrSpace()	Create a new address space
DeleteAdrSpace()	Delete an address space
EnterAddressSpace()	Attach a task to an address space
LeaveAddressSpace()	Remove a task from an address space
CurrentAddressSpace()	Return the address space of a task
AdrSpaceOf()	Return the address space of a context
MMUContextOf()	Return the context of an address space

3 Memory Pools

The memory library handles virtual memory in memory pools; these pools are responsible for administration of virtual memory, and virtual memory is allocated from and disposed into these pools. Similar to exec memory pools, all memory allocated from a pool shares the same properties concerning the memory type and, additionally, the same caching mode; e.g. a pool may administrate chip memory only, or may hold only cache-inhibited memory. Further, you may build more than one `VMPool` within an `AdrSpace`. The following call is required to construct a new pool:

```
struct AdrSpace *adr;
struct VMPool *pool;

pool = NewVMPool( MEMTAG_ADRSPACE, adr, TAG_DONE);
```

This will construct a new pool, or will return `NULL` on failure. More options can be given to fine-tune the memory pool, they will be presented in section 5.

> *Not an Exec Pool.* Memory pools created by `NewVMPool` are very different from exec library pools. You *must not* allocate from memory library pools with exec functions, and vice versa.

A pool can be disposed again; this will also release all the memory handled by the pool:

```
struct VMPool *pool;

DeleteVMPool(pool);
```

> *Enter, then Build!* The calling task of all pool and virtual memory related functions, including the construction and deletion of pools, *must* have entered the address space it builds pools for, or it allocates memory within.

The following function pair is used to allocate memory from a pool and to release it again. They are the memory library analogues of the well-known exec **AllocMem()** and **FreeMem()** functions:

```
struct VMPool *pool;
ULONG size;
APTR mem;

mem = AllocVMemory(pool,size);
```

The returned pointer points to the allocated virtual memory; it is `NULL` on failure. Note that virtual memory allocation may fail similar to ordinary memory allocation, even though it is less likely to fail. You should therefore check for out of memory situations as well.

Virtual memory is released again by

```
struct VMPool *pool;
APTR mem;
ULONG size;

FreeVMemory(pool,mem,size);
```

Allocation size and released size *must* coincide; it is illegal to break up a large memory block into several small blocks and to release these small blocks separately.

> *Forbid is Forbidden.* You *must not* call the above functions within **Forbid()** or **Disable()** state. **AllocVMemory()** will simply fail if you try to, other functions may even crash. Virtual memory administration requires the task switching enabled since memory must be swapped in or out.

3.1 Function Reference

The following chart gives the function reference for the last chapter:

Table 2: Basic Memory Pool Functions

MemLib function	Description
NewVMPool()	Create a new virtual memory pool
DeleteVMPool()	Dispose a pool and its contents
AllocVMemory()	Allocate virtual memory from a pool
FreeVMemory()	Release virtual memory

3.2 Memory Control Functions

Virtual memory must not be accessed while the task switching is disabled. Further, you must not place Os structures into virtual memory. Clearly, this restricts the usefulness of the virtual memory provided by this library. To weaken the above restrictions somewhat, the memory library can be told not to swap out certain memory regions, or even to place them in a physical memory buffer that is large enough to hold the selected chunk as a continuous memory block similar to ordinary shared memory.

The following call will "lock" virtual memory blocks in memory, hence will prohibit that they are swapped out. Once you've locked a block, you are free to use the memory within this block even with multitasking disabled:

```
struct AdrSpace *adr;
APTR mem;
ULONG size;
BOOL ok;

ok = LockMemory(adr,mem,size);
```

The call returns a boolean success/failure indicator — note that this call is likely to fail if you want to lock a very large memory block in main memory, especially if the system is low on memory anyhow.

The next call releases a lock again:

```
struct AdrSpace *adr;
ULONG size;
APTR mem;

UnlockMemory(adr,mem,size,FALSE);
```

Afterwards, this memory may be swapped out again immediately. The fourth parameter of this call should be set to `FALSE` for typical applications.

LockMemory() and **UnlockMemory()** nest, i.e. each call to **LockMemory()** must be matched by one and only one call to **UnlockMemory()** unlocking the same memory block that has been locked by **LockMemory()**. Unlocking a memory block in different chunks than locking the same block is *illegal.*

If you set the fourth parameter of **UnlockMemory()** to `TRUE`, all locks on this memory region are released immediately, overriding any locks, and overriding the nesting. This option should be used with great care.

Even though **LockMemory()** guarantees that the locked memory is represented in physical memory, it does not guarantee that it is represented in a continuous memory block, neither does it tell you anything about the physical addresses of these blocks. It may therefore happen that a huge memory block of, for example, 64K is represented by tiny discontinuous blocks of 4K that are splattered all around the physical space. The CPU won't care about this fragmentation as the MMU generates the illusion of a continuous memory block, but memory in this form is unsuitable for DMA devices, as hard-disks. Hence, even the virtual memory became "less virtual" by **LockMemory()** is not yet "real enough" for DMA devices and hence unsuitable to be passed to any kind of IO function, e.g. **Read()** or **DoIO()**. The following function pair will ensure that the memory block is really in one piece:

```
struct AdrSpace *adr;
ULONG size;
APTR mem,physical;
```

```
physical = HoldMemory(adr,mem,size);
```

This call returns the true physical address where the selected virtual memory block is represented, and *this* value is finally suitable to be passed to any IO function.

Needless to say, this operation is costly because it requires a continuous chunk of system memory, and may therefore fail — it returns `NULL` on failure.

The next call will release the hold-lock again:

```
struct AdrSpace *adr;
ULONG size;
APTR mem;

UnholdMemory(adr,mem,size,FALSE);
```

You need to pass in here the logical address, i.e. the same arguments you used to "held" a memory block. The meaning of the last `FALSE` argument is similar to that of the **UnlockMemory()** call; it defines whether **UnholdMemory()** nests or releases all locks at once, ignoring the nest-count. You should use the nesting variant for almost all applications and leave the argument as `FALSE`.

> *Don't Lock Yourself Out.* Locking or holding memory is a relatively expensive operation as it requires that the selected memory regions are really represented in main memory. This contradicts of course the virtual memory concept that allows memory blocks larger than the available main memory. Therefore, try to avoid locking huge blocks at once, lock only small blocks, one after another. Don't hold locks too long or the system may run out of memory.

3.3 Function Reference

The following chart gives again the function reference for the last chapter:

Table 3: Memory Control Functions

MemLib function	Description
LockMemory()	Disallow swapping of a block
UnlockMemory()	Re-allow swapping
HoldMemory()	Lock memory into a continuous block
UnholdMemory()	Release the hold-lock

4 Building Address Spaces

We saw in the last chapter how to build simple address spaces using the **NewAdrSpace()** function. We will now present some advanced options, in the form of additional tags.

4.1 Tags Selecting the Target Context

The following tags can be specified to select the MMU context an `AdrSpace` is build on:

MEMTAG_CONTEXT Defines the mmu.library `MMUContext` directly the address space shall be build on. At most one address space per context; do *not* attach an address space to the public context or to a supervisor context, this won't work.

MEMTAG_TASK If `MEMTAG_CONTEXT` is missing, this tag can be used alternatively to specify the target context. The `MMUContext` the specified task is part of will then be used as target for the new `AdrSpace`. The default for this tag is the current task.

4.2 Tags Selecting the Size of the Virtual Memory Region

The following tags select various sizes for the virtual memory administrated by the `AdrSpace`:

MEMTAG_VMSIZE Total size of virtual memory area to be handled by this address space, not including the usual system shared memory. Default is as large as the system, the swapping device and `MEMTAG_VMMAXSIZE` (see below) allows.

MEMTAG_VMMAXSIZE This tag limits the size of the virtual memory region to be allocated to the indicated size. The resulting address space will hold at most the indicated size, but might hold less due to limitations of the swap device.

MEMTAG_MAXSYSMEM Maximal amount of physical system memory this address space is allowed to allocate. If a new page must be swapped in, the total amount of physical page memory held by this address space is compared against this threshold. If it is larger, pages are swapped out. However, if you hold locks on memory blocks by means of **LockMemory()** or **HoldMemory()**, the library may ignore this threshold temporarily as long as the locks remain; i.e., locking will not fail due to this threshold as long as enough system memory is available. Default is half the system memory minus a safety margin.

MEMTAG_CACHESIZE Defines the size in bytes of an additional write cache that is used to minimize the seeking within a swap file or swap partition/device. This cache keeps swapped out pages and is written out "in one go" as soon as necessary. Defaults to sixteen pages, or no write cache if `MEMTAG_READONLY` has been set to `TRUE`. The memory type of the cache is controlled by **MEMTAG_BUFMEMTYPE**, and defaults to `MEMF_PUBLIC`.

4.3 Tags Defining the Swap Mechanism

The following list of tags define how memory is swapped out or in, and what kind of swapping device is used. In principle, all swapping activity goes thru hooks that can be completely user defined. However, to avoid unnecessary complexity, the library comes with its own set of built-in hooks that can be selected here as well. Custom swap-hooks are described in a separate chapter.

MEMTAG_VMHOOK A custom defined swap hook as `struct Hook *` that will be called for swapping activities. More on these hooks later. Either this or the next tag must be given to define a swapping mechanism.

MEMTAG_VMSWAPTYPE Defines a library pre-defined swap hook as swapping mechanism; this is the preferred option. The following pre-defined swap hooks are available:

- **MEMFLAG_SWAPFILE** Swap to a `dos.library` file. Note that this is slow for the FFS, but is the least critical way to provide a swap target.
- **MEMFLAG_SWAPPART** Swap to a `dos.library` "Device" in the sense of a partition on a hard disk. This is the best compromise between speed and safety. As sectors within this partition are accessed directly, the partition contents will be lost, though.
- **MEMFLAG_DEVICE** Swap to an `exec.library` "device" given by a device name and a unit.

The following tags are used only for `MEMFLAG_SWAPFILE` and define further parameters for this hook:

MEMTAG_SWAPFILENAME A `char *` to the name of the swap file. Mandatory for file swapping.

MEMTAG_SWAPFILELOCK A lock onto a directory the above file name is relative to. Defaults to the current directory of the calling process.

MEMTAG_KEEPFILE If set to `TRUE`, the file is not deleted when the `AdrSpace` is shut down. Defaults to `FALSE`, i.e. the swap file will be deleted when done.

MEMTAG_PREDEFINED If set to `TRUE`, the swap file is assumed to be existent already and the contents of this file is mirrored into the virtual memory. Defaults to `FALSE`. Most useful in combination with `MEMTAG_KEEPFILE`. Examples for the application of this tag will be given in a separate chapter.

MEMTAG_READONLY Requires `MEMTAG_PREDEFINED` set to `TRUE` and allows reading from the swap file only to browse it in memory. Any changes made to the memory are discarded. Should be combined with the MMU property `MAPP_ROM` or `MAPP_READONLY` for the pools of this address space for optimal performance. Defaults to `FALSE`.

The following are only for `MEMFLAG_SWAPPART`, hence if the library swaps to a partition:

MEMTAG_SWAPDEVICENODE A pointer to a `dos.library` `struct DeviceNode` that defines the swap partition. Either this or the next tag must be given to specify a swap partition.

MEMTAG_SWAPPARTNAME The `dos.library` device name of the swap partition as `char *` with or without trailing colon, e.g. `SWAP:`.

The last set of tags is only used for `MEMFLAG_SWAPDEVICE`:

MEMTAG_DEVICENAME The name of an `exec.library` "device" to which memory shall be swapped. This must be a "trackdisk"-like device that allows random access. This tag is mandatory.

MEMTAG_DEVICEUNIT Unit number of the above device to swap to. Defaults to zero.

MEMTAG_DEVICEFLAGS Flags for "OpenDevice()" for this device. Defaults to zero.

MEMTAG_BUFMEMTYPE Intermediate buffer memory type for the device in case the device is unable to perform I/O on all system memory addresses. Default is `MEMF_24BITDMA`. Note that this is a very conservative setting that should work with almost all devices; you might want to specify `MEMF_ANY` here if you know that your selected device is able to perform this. Note further that the `memory.library` never passes logical addresses to the swapping device, but always already translated physical addresses.

MEMTAG_MASK Defines a memory mask to check the memory against. If the output buffer "and-ed" with the complement of this mask is non-zero, the device swap hook will use single-block I/O to perform the operation. Note that, similar to the above, you *need not* to specify a mask to exclude virtual memory. The library will always pass physical addresses to the device. Default is `0x00fffffc`, which is a very conservative setting and allows only device access within the 24 bit address space. A well-written device driver will accept 0xffffffff here.

MEMTAG_MAXTRANSFER The size of the largest memory block this device will be able to access at a time. Defaults to 0x001fe000.

MEMTAG_DEVICESIZE Size of the area to be reserved for the swapping activities. Limited by 2GB. Note that due to the broken design of many devices, i.e. lack of support for `TD_GETGEOMETRY`, this size cannot be obtained safely from the device itself.

MEMTAG_DEVICEORIGIN Byte offset from the beginning of the device to the first block of the device where swapped data will be held. This is a pointer to a "`QUAD word`", i.e. a two-`ULONG` array containing the high and low word defining the size as 64 bit integer. If any part of the swap area exceeds the 64 bit limit, the device must be either TD64 or NSD compliant with the further restriction that `NSDPatch` must be running to activate NSD support.

MEMTAG_SECTORSIZE Size of a sector (or "block") of the device in bytes. This value is only used for single-block transfers and defaults to 512 bytes.

4.4 Miscellaneous Tags

The following tags define various miscellaneous options for the `AdrSpace`:

MEMTAG_SWAPFAILHOOK A pointer to a `struct Hook` that will be called whenever swapping memory in or out fails. If this happens, the library tried already to bring up a Retry/Cancel requester, and tries several times to submit the I/O operation, but without success. Your swap hook may either terminate the program, or release memory you do not require immediately. A separate chapter discusses the use of this failure hook.

MEMTAG_WINDOWPTRPTR Defines a pointer to a window pointer — note the double indirection — where error requesters concerning the virtual memory system will appear. If the pointer the argument is set to `(struct Window *)(~0)`, no requesters will be generated and the library will react as if all requesters have been canceled. If set to `NULL`, requesters will appear on the Workbench. Default is `NULL`, or a pointer to the `pr_WindowPtr` component of the `MEMTAG_TASK`, should this tag be available.

4.5 Example Code

The following code segment shows how to build your own address space from a shared copy of the global context, and how to attach your task to this new address space:

```
struct MMUContext *ctx;
struct AdrSpace   *adr;
/*
 * create a new MMU Context to operate in
 */
ctx = CreateMMUContext(MCXTAG_SHARE,CurrentContext(NULL),TAG_DONE);

if (ctx == NULL) {
        Printf("Failed to create a new MMU context.\n");
        adr = NULL;
} else {
        adr = NewAdrSpace(MEMTAG_MAXSYSMEM,64<<12, /* at most 256 K */
                          MEMTAG_VMSWAPTYPE,MEMFLAG_SWAPPART,
                          MEMTAG_SWAPPARTNAME,"SWAP:",
                          MEMTAG_VMMAXSIZE,64*1024*1024, /* at most 64MB */
                          MEMTAG_PROVIDESMEM,TRUE,       /* true memory  */
                          MEMTAG_CONTEXT,ctx,
                          TAG_DONE);
}

if (adr) {
        if (EnterAddressSpace(adr,NULL)) {
                Printf("Entered the address space successfully.\n");

                /* more code goes here: Build memory pools, etc... */

                ...

                /* detach the current task from the address space */
                LeaveAddressSpace(NULL);
        }
        DeleteAdrSpace(adr);    /* dispose the address space */
        DeleteMMUContext(ctx);  /* and the context below */
}
```

5 Building VMPools

Similar to address spaces, VMPools offer a lot of additional options on creation. The following chapter presents all additional tags.

The group of the next three tags defines in one way or another the AdrSpace the VMPool shall become part of.

MEMTAG_ADRSPACE A pointer to the struct AdrSpace within which this pool shall be created.

MEMTAG_CONTEXT A pointer to a MMUContext that is attached to an AdrSpace; the pool will then be build within the AdrSpace that is attached to the specified context.

MEMTAG_TASK A pointer to a struct Task that is attached to an AdrSpace; the pool will be build within this AdrSpace. Defaults to the current task.

Due to the default of the above tags — building the pool within the address space of the current task — you typically need not to specify any of the above at all.

5.1 Private Swapping Mechanisms

Memory within VMPools is swapped in or out using the swap hook of the address space they are part of. However, you may also specify a local swap hook exclusively for your pools. This is useful, for example, if you require virtual memory at the one hand, but also require the mirror of an additional file in memory at the other hand; you would define an address space with an ordinary file or partition swap hook, a pool within this address space providing the virtual memory, and a second pool with a custom swap hook that maps the contents of the input file to memory.

You enforce private swapping mechanisms for pools by either of the two following tags:

MEMTAG_VMHOOK A custom defined swap hook as `struct Hook *` that will be called for swapping activities.

MEMTAG_VMSWAPTYPE Defines a library pre-defined swap hook as swapping mechanism.

The second tag requires additional specifications about the location of the swap file; these are identically to the tags of **NewAdrSpace()** described in section 4.3.

Pools that use private swapping mechanisms must be build with the following tags as well

```
MEMTAG_PREALLOC,TRUE,        /* build the puddle immediately */
MEMTAG_FIXEDSIZE,TRUE,       /* do not enlarge pool */
```

or the construction of the `VMPool` will fail.

5.2 Pool Management Tags

Another set of tags defines the type of the `VMPool`. Pools are not limited to providing the memory for **AllocVMemory()** and **FreeVMemory()**; they can be used as well to mirror the image of a file into memory. This requires, however, that you tell the pool that you don't want to activate its memory administration functions as otherwise these functions would overwrite the mirror of the file in memory.

MEMTAG_PREALLOC If set to `TRUE`, the pool memory will be allocated already at full size from the available virtual memory space of the AddressSpace on setup. Full size means here "as large as the swap hook permits". For a file type swap hook this would be the size of the swap file. If set to `FALSE`, which is the default, memory puddles are build as soon as required. This option is useful for mapping files into memory — use a file type swap hook, and set this option to `TRUE`.

MEMTAG_FIXEDSIZE If set to `TRUE`, this tag forbids the creation of new puddles if the pool runs out of memory. This is only useful if `MEMTAG_PREALLOC` is `TRUE` as well or you will never be able to do anything useful at all with this pool. The default is `FALSE`.

MEMTAG_PROVIDESMEM Is `TRUE` by default, and should remain `TRUE` to be able to allocate virtual memory from this pool by means of **AllocVMemory()**. If set to `FALSE`, the memory allocation and release functions will fail on this pool, but **PoolVBase()** and **PoolVSize()** will be enabled. This indicates that you either want to manage the memory provided by this pool yourself, or that this pool is used for providing virtual addresses for mirroring a file to memory.

MEMTAG_PREDEFINED The pool contents is pre-defined by the contents of the swap device if this is `TRUE`. Otherwise, allocated memory from this pool has undefined contents. This is only useful with `MEMTAG_PROVIDESMEM` set to `FALSE` as otherwise the memory management functions of the memory library will overwrite the predefined memory contents by their own administration structures. This tag defaults to `FALSE`.

MEMTAG_READONLY If this is set to `TRUE`, changes made to the pool are not written back to swap space. Requires `MEMTAG_PROVIDESMEM` set to `FALSE` or the pool will become corrupt as soon as it must be swapped because internal administration information of the memory management is not updated properly. The default is `FALSE`.

MEMTAG_PUDDLESIZE Size of a memory puddle in bytes that will be created if a memory block runs out of data. Puddles are enforced to be at least one MMU page large. Defaults to eight pages.

MEMTAG_PUDDLETHRES Threshold defining which allocations go into separate puddles and which allocations are taken from the common puddles. The default for this tag is half the puddle size. Note that this threshold must be smaller than the puddle size.

MEMTAG_POOLPRI Priority of this pool relative to all other pools within the same address space. Higher priority pools will be swapped out later and are hence likely to keep memory resident for a longer time. Defaults to zero.

Note that the puddle administration of `VMPools` looks very similar to that of the memory pools provided by exec, but the internal workings are very different. Hence, never attempt to mix exec and memory library type of pool allocation.

5.3 Memory and Caching modes

All memory allocated from a pool share the same the same exec memory type, should the memory be resident; further, the caching flags are the same as well. This means that you can tell the memory library that all resident memory of a pool shall be allocated in chip memory such that the custom chips may find it as soon as you held it with **HoldMemory()**. You can also tell the memory library to make the memory the pool provided cache-inhibited, or write-protected. The following tags define these properties:

MEMTAG_MEMFLAGS An `exec/memory.h` type memory properties describing the physical memory to be allocated for this pool. Defaults to `MEMF_ANY`. Note that the memory type is only relevant as long as you really lock memory pages in memory, as they may otherwise get swapped out.

MEMTAG_CACHEFLAGS An `mmu/context.h` type "MMU properties" of the memory to be addressed by the pool. The original properties of the memory are filtered thru the inverse of the mask given by the next tag, then combined with this flag collection to form the properties of the allocated pages. As the default for the following mask tag is zero, these flags will be ignored unless you specify a mask. Then, this tag has the default of zero.

Note that the memory library will not allow all combinations of flags and mask; you are for example not allowed to mark `MEMF_CHIP` memory as cache-able.

MEMTAG_CACHEMASK A property mask as defined in `mmu/context.h` describing which caching flags of the above tag are relevant. A zero-bit in this mask ignores your selection and uses the corresponding property bit of the allocated physical memory, a one-bit makes your selection relevant.

Not all combinations of cache flags and mask are useful. Let's list some important combinations:

MAPP_COPYBACK Make the memory copyback-cacheable; this is the fastest caching mode. There is no need to ask for this caching type explicitly as the memory library tries to find the optimal caching mode anyhow. You may however have the copyback bit set to zero in the flags, and enabled in the mask to request writethrough caching. Since the 68020 and 68030 do not offer copyback caching, this flag is ignored on the older members of the Motorola processor family.

MAPP_CACHEINHIBIT If this is specified for flags and mask, caching is disabled completely.

MAPP_NONSERIALIZED This is ignored unless you disabled caching; it is a special non-caching mode the 68040 offers which allows re-ordering of memory accesses to speed up the access. This flag is ignored for all other processors.

MAPP_IMPRECISE This flag is also ignored unless you disabled caching. It allows the 68060 to react a bit sloppy on physical bus errors and is ignored by all other processors. Since Amiga memory should never generate bus errors, it doesn't hurt to enable this flag for cache-inhibited memory.

MAPP_ROM Enables defensive write protection; writes into memory pages will fail silently. Note that the memory library requires to write into the allocated pages; therefore, this flag *must* be combined with `MEMTAG_PROVIDESMEM` set to `FALSE`.

MAPP_WRITEPROTECTED Enables the aggressive write protection. Writes into the pool memory pages will cause either a software failure requester or a MuForce hit. Similar to the above, this rules memory allocation from this pool out.

5.4 Enhanced Virtual Memory Pool Functions

Memory pools need not just to hold memory to be allocated by **AllocVMemory()**. Memory pools may also hold a "mirror image" of the swap medium. As a typical application, a memory pool may hold the contents of a file by defining its swap hook as type `MEMFLAG_SWAPFILE`. The program can then easily read the contents of the file by reading the memory within the pool, and could even modify the file contents by writing into the pool memory. For this purpose, `VMPools` should be build with the following tag combination:

```
MEMTAG_PREALLOC,TRUE,        /* build the puddle immediately */
MEMTAG_PROVIDESMEM,FALSE,    /* no memory */
MEMTAG_FIXEDSIZE,TRUE,       /* do not enlarge pool */
```

This enables support for the following two extended memory library administration functions:

```
struct VMPool *pool;
APTR mem;

mem = PoolVBase(pool);
```

This places the start address of the pool into `mem`. Further, you may also ask the pool about its byte size:

```
struct VMPool *pool;
ULONG size;

size = PoolVSize(pool);
```

This returns the size of the pool in bytes. Note, however, that this need not to be exactly the size of the swap file in bytes; the memory library has to round up pool sizes to multiples of the MMU page size, which makes the size returned by **PoolVSize()** larger. You need to ask the dos library directly for the file size to be on the safe side.

> *Special Offer!* **PoolVBase()** and **PoolVSize()** will not work at all if you don't create the pool with the special tags shown above. They will fail immediately. On the other hand, **AllocVMemory()** and **FreeVMemory()** will fail on these pools.

5.5 Example Code

The first example presents how to create a standard virtual memory pool and how to allocate memory from it. This example assumes that you have already build an address space and entered it, as show in section 4.5.

```
struct VMPool *vmpool;

vmpool = NewVMPool(MEMTAG_PROVIDESMEM,TRUE,     /* build for memory allocation */
                   MEMTAG_PUDDLESIZE,5<<12,     /* 20K puddles */
                   TAG_DONE);

if (vmpool) {
        APTR mem;

        mem = AllocVMemory(vmpool,238);         /* allocate 238 bytes */
        if (mem) {
                /* similar to exec: check for failure! Virtual memory is
                 * large, but not bottomless.
                 */
                ...     /* operate on memory */
                FreeVMemory(vmpool,mem,238);
        }

        DeleteVMPool(vmpool); /* done with it. */
        /* This disposes *all* memory within the pool as well */
}
```

In the second example, we demonstrate how one could map the contents of a file into a memory pool, and uses then the **PoolVBase()** function to read from the file in memory; similar to the above, we assume that a valid address space has been build already. We use here a memory pool with a private swapping hook.

```
struct VMPool *vmpool;
BOOL readonly = FALSE;  /* allow writing to the file? */
char *file;             /* name of the file to map */

vmpool = NewVMPool(MEMTAG_PREDEFINED,TRUE,      /* mem contents is predefined */
                   MEMTAG_READONLY,readonly,    /* allow reading only? */
                   MEMTAG_PREALLOC,TRUE,        /* build the puddle immediately*/
                   MEMTAG_PROVIDESMEM,FALSE,    /* no memory */
                   MEMTAG_FIXEDSIZE,TRUE,       /* do not enlarge pool */
                   MEMTAG_VMSWAPTYPE,MEMFLAG_SWAPFILE, /* swapping mechanism */
                   MEMTAG_SWAPFILENAME,file,    /* name of the swap file */
                   MEMTAG_KEEPFILE,TRUE,        /* keep when done */
                   TAG_DONE);

if (vmpool) {
        UBYTE *mirroraddress;

        mirroraddress = (UBYTE *)PoolVBase(vmpool);

        /* mirroraddress points now to the first character of the file
```

```
         * that is, reading from mirroraddress[4711] will read the 4711th
         * byte of that file.
         */

        ... /* work on the file */

        DeleteVMPool(vmpool); /* done with it */
}
```

Note that in the above example, the file size must be obtained directly from the dos library. This is because the memory library will round up the pool size to the next full MMU page.

5.6 Function Reference

Again the function reference for the previous section:

Table 4: Memory Control Functions

MemLib function	Description
PoolVBase()	Get the virtual base address of a pool
PoolVSize()	Obtain the (rounded) size of a pool

6 Swap Fail Hooks

Under the default settings, the memory library will show failure requesters in case the swapping mechanism failed. Requesters might be generated due to two different reasons: The first reason is an input/output error within the swap hook, i.e. loading or saving swapped data to the swap device failed. This requester is automatic and will pop up even with a swap fail hook installed. It can be disabled by setting the `MEMTAG_WINDOWPTRPTR` to a pointer to `~0`. The second kind of requester will be generated if the memory library runs out of memory; it can be disabled similar to the first kind, of course, but you may also install a swap fail hook into your `AdrSpace`. This will disable the second kind of requester only, but will still make the first kind happen. Your hook will get called as soon as either the user cancels the I/O error requester, or the library is low on memory; the purpose of the hook is then to clean-up your own resources to allow continue swapping.

The swap fail hook is setup by the following tag, to be passed to **NewAdrSpace()**:

```
struct AdrSpace *adr;
struct Hook *hook;

adr = NewAdrSpace(MEMTAG_SWAPFAILHOOK, hook,
                  ...
                  TAG_DONE);
```

Your hook is called with an `Object *` set to NULL, its message, however, points to the following structure, defined in `<memory/memfailhook.h>`:

```
struct SwapFailMsg {
        APTR            sfm_AddressSpace;/* the address space that failed */
        APTR            sfm_Memory;      /* lower memory that failed */
        ULONG           sfm_Size;        /* size of the memory range */
        LONG            sfm_Error;       /* type of error */
};
```

The component `sfm_AdressSpace` points to the address space that caused the error condition. This is the — or one of – the address space(s) you installed your fail hook into. `sfm_Memory` is the lower boundary of the memory block whose swap operation failed, as given by its logical address, and `sfm_Size` the size of this block in bytes. This will be typically a multiple of the MMU page size. `sfm_Error` contains an error code describing the cause of the fault. The possible error codes include the ones defined in `<dos/dos.h>` and `<memory/memerrors.h>`. This will be either `ERROR_NO_FREE_STORE` for out of memory faults, or an I/O type of error to indicate that the user canceled the error requester of the swap hook.

The return code of the hook must be one of the three following values:

```
#define SWFRET_RETRYSWAP  0
#define SWFRET_RETRYFAULT 1
#define SWFRET_DEACTIVATE 2
```

SWFRET_RETRYSWAP indicates that your tried some countermeasures against the error, and the memory library shall retry the swap access. In case of out of memory errors, you tried successfully to release enough memory, in case of I/O errors you made the swap device available again. Do *not* generate this result code in case you couldn't do anything against the fault or the memory library will simply come back.

SWFRET_RETRYFAULT If you signal this result code, the memory.library will declare the access of the user application for handled and will restart it. Note that this differs from `SWFRET_RETRYSWAP` by giving control back to the application that requested the virtual memory in first place. If you generated this exception, you should have successfully tried to avoid the page fault by making the failing pages available directly by the mmu library as otherwise a new page fault would be generated immediately and your hook would be called again.

SWFRET_DEACTIVATE This is the worst-case handling of the fault: If this result code is received, the mmu library "Exception Hook" of the address space gets deactivated and the page fault is signaled as handled. If the failing page is not made available, the MMU will just generate another page fault which, however, will no longer reach the memory library. The fault is either captured by MuForce causing a "Hit", or falls thru to the exec access error handler. This handler will first try to call the task specific exception handler, and will otherwise cause a guru. This is obviously the least pleasant way to handle a page fault, but the last resort in case everything else failed.

7 Custom Swap Hooks

Even though the built-in swap hooks allow swapping to files, partitions and devices cover the most common cases, you may require your own private swap mechanism for special applications. The memory library may even help here: To signal a private swap hook, pass a pointer to your `struct Hook` when creating either an `AdrSpace` or a `VMPool` by the following tag:

```
struct Hook *hook;

New...(MEMTAG_VMHOOK,hook,
       ...
       TAG_DONE);
```

Once help of your swap hook is required, it will be called with the `Object` pointer set to `NULL` and the `Message` parameter pointing to the following structure:

```
struct VMHookPacket     {
        ULONG   vhp_Type;       /* packet type                  */
        APTR    vhp_HookInfo;   /* user defined data            */
        APTR    vhp_Range;      /* physical address             */
        ULONG   vhp_Size;       /* address size                 */
        ULONG   vhp_Offset;     /* ID for swapping in/out       */
};
```

The most important component here is `vhp_Type` defining the type of the packet, and the usage of other components:

VMPACK_INIT Send for initialization of your hook. `vhp_HookInfo` will be `NULL` here, and `vhp_Range` will point to your hook structure. The aim of this packet is to initialize your hook. The hook shall either return an error code as documented in `<dos/dos.h>` or, alternatively, `<memory/memerrors.h>`, or shall return 0 to indicate success. On success, you may have filled `vhp_HookInfo` by a pointer to a private administration structure that is from now on passed to each call of the hook in this very same component.

VMPACK_EXIT Send to shut down a swap hook. `vhp_HookInfo` points to a private structure you might have placed here on `VMPACK_INIT`, `vhp_Range` points again to the hook itself. This call shall release all resources, including memory for your administration structure.

VMPACK_OPEN Request to open and allocate the swap resource. `vhp_HookInfo` points to your private administration structure, `vhp_Size` defines the requested byte size of the swap domain. This call may open a swap file, for example, if a previous `VMPACK_SIZE` hasn't done so already. This hook returns an error code of the same type as `VMPACK_INIT`.

VMPACK_SIZE Request and inquiry the available byte size of the swap medium. This packet may be sent before `VMPACK_OPEN` takes place to ask the hook about the available range. `vhp_HookInfo` points again to your administration structure, `vhp_Size` to a user requested upper limit. This might be `~0` to indicate that the hook shall return the maximal possible size. `vhp_Offset` will be set to the MMU page size. The available swap size has to be placed into `vhp_Size` again, and the hook shall return an error code, or 0 for success. If your hook doesn't round the available size to full MMU page sizes, the memory library will round the size down to the next page. If you require other rounding, make sure to align the size to the next suitable page size.

VMPACK_CLOS Close or shutdown a swap medium. `vhp_HookInfo` points to your administration data, no other information is provided. This packet will be generated if and only if `VMPACK_OPEN` succeeded. Hence, if you opened a swap medium already in `VMPACK_SIZE`, you may need to close it in `VMPACK_EXIT` instead as the size inquiry and open may fail and `VMPACK_CLOSE` will never reach your hook.

VMPACK_READ Read data from the swap device into a memory library supplied buffer. This defines therefore the "swap in" operation. `vhp_HookInfo` points to your administration data, `vhp_Range` to the *physical* address of the buffer to fill in. `vhp_Size` is the byte siz of the buffer. Finally, `vhp_Offset` is set to the byte offset from the beginning of the swap medium to the requested data. Hence, this addresses the data to be swapped in. This call shall either return 0 or an error code, but do not yet generate an error requester here.

VMPACK_WRIT Write out data from a memory library buffer to a swap medium; hence, "swap out" data. Otherwise, identically to `VMPACK_READ`.

VMPACK_ALRT Generate an alert requester due to a failure of either swap-in `VMPACK_READ` or swap-out `VMPACK_WRIT`. `vhp_HookInfo` points to your administration data, `vhp_Range` to a `struct Window`, defining an intuition window whose IDCMP shall be blocked. Note that you get a pointer here, not a double pointer. This pointer is `NULL` to indicate that the requester shall appear on the workbench, or `(APTR)(~0)` to signal that no requester shall be generated at all. `vhp_Size` is an error code, as returned by the two packet types above. This call shall return either `TRUE` to signal that the access shall be retried, or `FALSE` to cancel the I/O operation. In this case, the error will be delivered to the swap fail hook, see section 6, should it be present; otherwise, the faulty task will run into an access error — and will hence guru.

VMPACK_TICK This packet doesn't define any particular action, it just gets generated each second. The packet can either be used to define a time base, or to enforce cache flushes, or to shut down a drive motor — whatever regular activity your hook may require. You may just ignore this packet in case you don't need it. As always, `vhp_HookInfo` points to your administration data.

References

[1] Thomas Richter: *The MMU Library Programmer's Manual.* Thomas Richter, on Aminet as `MuManual.lha` (2000,2001)

[2] Motorola MC68030UM/AD Rev. 2: *MC68030 Enhanced 32-Bit Microprocessor User's Manual, 3rd ed.* Prentice Hall, Englewood Cliffs, N.J. 07632 (1990)

[3] Motorola MC68040UM/AD Rev. 1: *MC68040 Microprocessor User's Manual, revised ed.* Motorola (1992,1993)

[4] Motorola MC68060UM/AD Rev. 1: *MC68060 Microprocessor User's Manual.* Motorola (1994)

[5] Motorola MC68000PM/AD Rev. 1: *Programmer's Reference Manual.* Motorola (1992)

[6] Yu-Cheng Liu: *The M68000 Microprocessor Family.* Prentice-Hall Intl., Inc. (1991)

[7] Dan Baker (Ed.): *Amiga ROM Kernel Reference Manual: Libraries. 3rd. ed.* Addison-Wesley Publishing Company (1992)

[8] Dan Baker (Ed.): *Amiga ROM Kernel Reference Manual: Includes and Autodocs. 3rd. ed.* Addison-Wesley Publishing Company (1991)

[9] Ralph Babel: *The Amiga Guru Book.* Ralph Babel, Taunusstein (1993)