The MMU Library Programmer's Manual

THOMAS RICHTER

Contents

1 Introduction to the MuLib

All “modern” Amiga computers come with a special hardware component called the “MMU”. This abbreviation stands for *Memory Management Unit.* The MMU is a very powerful piece of hardware that can be seen as a translator between the CPU that carries out the actual calculation, and the surrounding hardware: memory and IO devices. Each access of the CPU to fetch or write data from the hardware or memory is filtered by the MMU, checked whether the memory region is available, write protected, can be hold in the CPU internal cache and more. The MMU can be told to translate the addresses as seen from the CPU to different addresses, hence it can be used to “re-map”, i.e. mirror parts of the memory without actually touching the memory itself.

A series of programs has been and is available that make use of the MMU: First of all, it’s needed by the operating system to tell the CPU not to hold “chip memory”, as used by the Amiga custom chips, in its cache; second, several tools re-map the Kickstart ROM to faster 32Bit RAM by using the MMU to translate the ROM addresses — as seen from the CPU — to the RAM addresses where the image of the ROM is kept. Third, a number of debugging tools make use of it to detect accesses to physically unavailable memory regions, and hence to find bugs in programs; amongst them is the “Enforcer” by Michael Sinz. Fourth, the MMU can be used to create the illusion of “almost infinite memory”, with so-called “virtual memory systems”. Last but not least, a number of miscellaneous applications have been found for the MMU as well, for example for display drivers of emulators.

Unfortunately, the Amiga Os did not provide *any* interface to the MMU so far, everything boils down to hardware hacking and every program hacks the MMU tables as it wishes. Needless to say this prevents program A from working nicely together with program B, Enforcer with FastROM or VMM, and other combinations have been impossible up to now.

This has to change! There has to be a documented interface to the MMU that makes accesses transparent, easy and compatible. This is the goal of the “mmu.library”. In one word, *compatibility.*

1.1 Supported Hardware

The MuLib is able to program all MMUs of the Motorola MC68K processor family: The 68851, which is the external MMU of the 68020 in the form of a coprocessor, and the build-in MMUs of the 68030, the 68040 and the 68060. Motorola offered an external MMU for the 68010 quite a while ago, the 68451, but up to my knowledge it has never been used in any Amiga model or third-party expansion. It is unsupported by the library, mainly because it is conceptually very different from the other four MMU types which, in fact, have been used in the Amiga here and there.

1.2 Basic Concepts

The basic object the MMU library handles is the *MMUContext* or short *Context.* It keeps the complete configuration of the MMU. If you’re familiar with other Amiga hardware components, one might say that *the MMUContext is what the ViewPort is for the graphics engine.* Unlike the ViewPort, however, the Context does not have a documented structure, there are only functions to operate on Contexts. Since the Tasks handled by Exec really share all their memory, one should better think of them as the *threads* of other operating systems, like Unix: They run all in the same environment and share all the data and all the addresses amongst them. A Context, however, *defines* a unique address space, and different Contexts run “independent” of each other. There are means to protect data from one context to be seen from another, or to write-protect them at least. Shared memory concepts are possible, too. Each Exec Task “belongs to” or “runs as a part of” a Context, and as Exec schedules Tasks, it automatically schedules Contexts, too. If a Task belonging to a different Context gains the CPU, a “Context swap” is initiated by the MuLib in addition. Therefore, a “Context” should be seen as the AmigaOs equivalent of a Unix process.

As soon as the MuLib is loaded, *two* Contexts will be build. For first, the so called “public Context”. This Context describes the global address space all Amiga applications are part of, unless they decide to detach themselves from the public Context. You usually need not to enter the public Context explicitly; as soon as a new Task is created, it will belong to this Context anyhow; you furthermore need not to create this Context, it will be build by the library on startup. The second Context is the "public supervisor Context". This is the Context "supervisor code", mostly system maintenance code, runs in. Note that this is different from the old "Exec world" where user code and supervisor code shared a common environment. *The MuLib enforces a distinct user/supervisor model.* Especially, this means that you may have data available in supervisor mode which is *not* available in user mode. As for the "public Context", you need not to create the "public supervisor Context" as it will be build by the library anyways on startup. To conclude:

- Each Exec Task belongs to a MMUContext. Without further calls to the MuLib, it belongs to the public Context.
- The MuLib distinguishes between user and supervisor mode accesses.
- Contexts are schedules as the Tasks belonging to them are scheduled.
- User mode code runs in the MMUContext of the currently active task. Supervisor code runs in the supervisor Context of the currently active user Context. The supervisor Context of the public Context is the public supervisor Context.
- Contexts come in pairs. Each user mode Context comes with a corresponding supervisor mode Context. Several user mode Contexts may share one supervisor mode Context, but not the other way round.
- On startup, the MuLib will build two Contexts, the public Context and the public supervisor Context.

From the user's point of view, the Context is "just a handle" to the administration data keeping all the information required to swap the MMU setup. It does not have a documented structure, even though it is referred as a `struct MMUContext *` for all the library functions.

2 MuLib Contexts and Exec Tasks

2.1 Looking for Contexts

The MuLib provides several calls to get handles to Contexts. The following library vectors are available:

```
struct MMUContext *ctx;
struct Task       *task;

        ctx = CurrentContext(task);
```

This returns the user mode Context the provided Task belongs to. You may pass in **NULL** to get the Context the current Task is part of. So to say, **CurrentContext(NULL)** is pretty much the MuLib equivalent of **FindTask(NULL)**. The following

```
struct MMUContext *public;

        public = DefaultContext();
```

however, returns the public user Context. It need not to be identical to the result code of **CurrentContext(NULL)** because the current task might have been "detached" from the public context and might run in its own environment. Unlike **CurrentContext()**, **DefaultContext()** does not take any arguments.

```
struct MMUContext *ctx,*sctx;

        sctx = SuperContext(ctx);
```

The **SuperContext()** call returns the corresponding supervisor mode Context for the user mode Context passed in. Hence,

```
SuperContext(CurrentContext(NULL));
```

returns the current supervisor Context, and

```
SuperContext(DefaultContext());
```

returns the public supervisor Context. None of these calls can fail, the result code is always valid.

2.2 Attaching Tasks to Contexts

Even though each Exec Task is already part of a Context — the public Context, namely — you may want to detach a task from the public Context and may run it as part of a different Context. Especially, since Exec always creates Tasks in the public Context, this is the *only* way to run a Task in its private environment: Namely, create a new Context, create a new Task, and attach the new Task to the Context. The following call will do this for you:

```
struct MMUContext *new,*old;
struct Task *task;

        old = EnterMMUContext(new,task);
```

The function **EnterMMUContext** returns the handle to the Context the task belonged to before **EnterMMUContext()** has been called. **EnterMMUContext()** does not only attach tasks to private Contexts, it is also able to remove a task from one private Context and to attach it to a different private Context. If the **new** argument of **EnterMMUContext()** is set to **NULL**, the task will leave its current Context and will be attached to the public Context.

> *NULL Means Failure.* **EnterMMUContext()** returns **NULL** in case it failed. It does *not* return **NULL** in case the task was attached to the public Context before; in this case, it returns the handle to the public Context. Always remember to check for result codes.

The MuLib provides a short-hand for removing a task from a private Context and to re-attach it to the public Context. This is

```
struct MMUContext *old;
struct Task *task;

        old = LeaveMMUContext(task);
```

The call **LeaveMMUContext()** is fully equivalent to

```
EnterMMUContext(NULL,..);
```

It might seem senseless to call **EnterMMUContext()** with the context argument set to the public Context because the **NULL** argument seems to do right the same with less trouble. However, *this is not equivalent.* For some of the advanced features of the MuLib, you *have to* enter a Context *explicitly*, regardless of whether you want to detach from the public context or not. **EnterMMUContext()** allocates some internal data structures for the task passed in which are required for these features, and **LeaveMMUContext()** will release these structures again. The task will use the same MMU setup in both cases, but the MuLib will keep some additional structures with the task if the argument is non-**NULL**. Nevertheless, no matter why you entered a Context, you have to call **LeaveContext()** before your Task shuts down.

2.3 Advanced Information about Contexts and Tasks

To be able to schedule the Contexts, the MuLib makes use of the **tc_Launch()** and **tc_Switch()** function pointers in the Exec Task structure as soon as you enter a Context — i.e. call **EnterMMUContext()** with a non-**NULL** argument. This means that these two Exec hooks will be no longer available for you if you have to make use of the advanced features of the MuLib. They remain available as long as you never enter a Context, though, and hence the MuLib remains backwards compatible to those applications that have to play with the two Exec hooks.

The MuLib provides more flexible replacement hooks under the name of "Switch and Launch Exception Hooks" if you need this feature. Please study the **MuLib Exceptions** chapter for more details.

2.4 Function Reference

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

Table 1: Context and Task Control Functions

MuLib function	Description
CurrentContext()	Get a handle to the Context of a given Task
DefaultContext()	Get a handle to the public Context
SuperContext()	Get the handle to a supervisor Context
EnterMMUContext()	Attach a task to a Context
LeaveMMUContext()	Run a task as part of the public Context

3 Working on Contexts

The MuLib provides a series of functions to modify existing Ccontexts, hence to modify the MMU setup in a easy, straightforward and hardware independent way. All the functions fall in two categories: *Low level* and *high level* calls. The high level calls are very convenient to use, allow rather abstract modifications on the MMU setup, and provide an easy to use interface. However, the high-level functions are slow and not interrupt-callable; they must be called from within a task, and they might break a **Forbid()** state. High-level calls might fail, due to out-of-memory conditions, but the high-level functions provide means to "lock" the MMU setup and hence to avoid interactions in critical situations; error handling functions are available as well. Furthermore, calling high-level functions does not directly cause a modification of the MMU setup. Instead, all modifications are recorded, but not written out to the hardware yet. By one call, the changes are translated to the lower level.

The low level functions, however, write more or less directly to the hardware, causing an instant change of the MMU setup. They give no control about this modification whatsoever, and they require a special preparation step by some high-level calls. Furthermore, the low-level functions are cumbersome to use and not as handy as the high-level routines, and even slower if large modifications have to be made. However, the low level functions are faster for smaller modifications, and fully interrupt-callable.

If the high-level routines operate on the low-level, a special hook is provided for the low-level users to get informed if their setup is about to be overwritten, check the "Page Access Exception Hook" in the "MMU Exception Handling" section below.

3.1 Concepts

A "MMU Page" is the smallest block of memory individually handled by the MMU. Typically, pages are 1K or 4K large, but the size depends on the Context and on what the hardware is able to offer. The 68851 and the 68030 provide page sizes of 256 bytes up to 32K, in powers of two, whereas the 68040 and the 68060 can only handle 4K and 8K pages. Pages start and end always at multiples of their sizes, i.e. 4K pages start at 4K boundaries. Hence, one should think of the full address space of 2^{32} bits divided into pages of equal size, adjacent to each other whose boundaries are aligned to multiples of powers of two. The MuLib function

```
struct MMUContext *ctx;
ULONG pagesize;

        pagesize = GetPageSize(ctx);
```

will return the page size for the Context passed in.

> *Seek the Size.* Never assume a fixed page size, and never hard-code the page size. The page size *will* be different on different Amiga models, and it even may vary from machine to machine, dependent on the requirements of the MuLib and on the configuration made by the user.

The MMU is told what to do with each page by a so-called "descriptor". It's the MuLib which creates and modifies these descriptors. However, the way how these descriptors look like depend on the type of the MMU installed, therefore the MuLib provides an abstraction of the data in the descriptor, the "Property Flags". By modifying these flags, you tell the MuLib how it should setup the MMU descriptors, and hence finally what the MMU will do; the job of the MMU is, for example, to tell the CPU which addresses are allowed to be kept in a cache for faster access. Another job of the MMU is to "translate" addresses: The addresses a program uses to access its data are called "logical addresses" because this is what the program logic sees. The MMU translates the logical addresses on a page by page basis to physical addresses — "physical" for the simple reason because these are the true hardware signals that leave the MMU and run as electrical signals to the RAM and ROM chip and other hardware. Hence, what appears outside of the MMU is different from the addresses seen by the program inside. Since two applications might run under two different Contexts, the very same logical address could be translated to two different physical addresses simply because the MMU setup is different. This is quite common on Unix machines where the program space for *each* application starts at address zero; hence Unix depends heavily on the use of a MMU. A very simple application for this feature of the MMU is to "re-map" the Kickstart ROM into RAM, which is accessible faster: What happens here is that the logical addresses of the ROM get translated to the copy of the ROM in RAM, at a different physical location.

> *Highly Logical.* All the addresses used by the MuLib and its function calls are, unless noted otherwise, *logical addresses.* Physical addresses appear only at times where the distinction has to be made, and only for re-mapping a page of logical addresses to a *different* physical address.

3.2 High-level MMU Setup

The following high-level call modifies some of the property flags of one or more pages:

```
struct MMUContext *ctx;
ULONG flags,mask,lower,size;
BOOL result;

        result = SetProperties(ctx,flags,mask,lower,size,TAG_DONE);
```

This is a tag-based call, some of the property flags require some additional tags passed in, check the list below for further details. As for all tag-based calls, the tag list must be terminated by **TAG_DONE**. There is also a non-stack based call for assembly language, named **SetPropertiesA()**. Please check the autodocs for details.

The parameters for this call are as follows: **ctx** is a handle to the Context, **lower** and **size** specify the *logical* address range to be setup. Both arguments *must* be multiples of the page size or the call will fail since the MuLib checks this requirement explicitly. The **mask** argument defines which property flags are to be changed. Each bit set to one transfers the corresponding bit from the flags parameter to the MuLib high level abstraction of a MMU descriptor. Finally, **flags** is the bit mask of the flags to be set or cleared. The following bits are defined in `mmu/context.h`:

MAPP_WRITEPROTECTED This defines the specified region to be write-protected. Especially, if a program attempts to write into the memory area, an access exception will be generated and the MuLib will call the segmentation fault exception hooks. This is the "agressive" write protection as it may generate exceptions. The "defensive" version of this Property Flag is **MAPP_ROM**.

MAPP_USED Mark the memory as "used". This is the abstraction of the MMU descriptor "U" bit which is set by the MMU each time an access to the corresponding page in memory is made. The MuLib will build the MMU descriptors with the "U" bit set if the **MAPP_USED** bit on the high-level is set. Note that reading this bit from the abstraction level by **GetProperties()** does *not* return the actual hardware MMU flag but only the pre-defined value of the "U" bit. There's usually little reason to mess with this bit with the high-level functions, just leave it alone. The only advantage of setting the "U" bit in first place is that this avoids an additional memory access of the MMU if the memory is accessed for the first time, but this is hardly noticeable. If you want to check whether a page has been used or not, you *must* use the low-level function **GetPageProperties()** instead. Similary, **SetPageProperties()** must be used to set or clear this bit.

MAPP_MODIFIED Mark the memory as "modified". This is the abstraction of the MMU "M" bit which is set on each write access to the corresponding page. Again, this bit only defines whether the MuLib high-level functions should build the descriptors with the "M" bit pre-set, it will not reflect the actual state of the true hardware descriptor. For further details, check the description of **MAPP_USED** above.

MAPP_CACHEINHIBIT Instruct the MMU to tell the CPU not to keep the corresponding memory page in cache. This is important if the page contains memory-mapped I/O registers or memory which is accessed by other hardware in parallel to the CPU, e.g. chip memory or video RAM. Unlike ordinary memory, these addresses may alter the state without interaction of the CPU, and a copy of the hardware register in cache might therefore not reflect the true state.

MAPP_SUPERVISORONLY Each access to the specified pages from user code will generate an access fault and will run the segmentation violation exception hooks. This is currently implemented by checking whether the Context is a user or a supervisor context, and marking the pages as invalid for user Contexts. Even though the 68040 and the 68060 MMUs offer a separate "supervisor only" bit, it is currently not used by the library for consistency to the 68030 and 68851 which do not offer this feature.

MAPP_USERPAGE0 Set the user page attribute 0. The user page attribute of a page appears as hardware signal at an output line of the CPU and can therefore be used for special hardware purposes. However, there is currently no Amiga hardware which uses this feature, hence just leave this bit alone. This bit is ignored by the 68030 and the 68851 anyways.

MAPP_USERPAGE1 Set user page attribute 1. As for the **MAPP_USERPAGE0** bit, this is a special hardware feature only available for the 68040 and 68060, and which is currently not made use of. Just leave this bit alone.

MAPP_GLOBAL The memory region is shared between different contexts. The MuLib makes currently no use of this flag, and it is available for the 68040 and the 68060 only. Setting this bit on a 68030 or 68851 driven system is ignored.

MAPP_BLANK The specified address range is not mapped by the hardware at all, it does not contain memory nor I/O registers. If this bit is set, read and write accesses to this area are quietly tolerated and ignored, mainly to work around faulty software and to avoid exceptions. This works currently by re-mapping the specified range to a blank "dummy" page which is elsewhere in memory. Since **MAPP_BLANK** cannot generate exceptions, this is the "defensive form" of **MAPP_INVALID** access control.

MAPP_SINGLEPAGE Tells the MuLib that it must build one hardware descriptor for each page in the specified region. Especially, this will turn off certain optimizations the MuLib would have taken to preserve memory, as for example sharing of descriptors. This bit is a *must* if you need proper access information in the form of the "used" and "modified" bits, and it is a *must* if you want to operate on the MMU descriptors by means of the low-level functions, i.e. **GetPageProperties()** and **SetPageProperties**.

> *Be Prepared for Low-Level.* Accessing MMU descriptors by means of the low-level functions *requires* a preparation step, namely, setting the **MAPP_SINGLEPAGE** bit using the high-level function. This will not only inform the MuLib that it should allow access to the descriptors, it will also ensure that each page gets its own descriptor, hence makes the low-level functions *meaningful* in first place. Needless to say, **MAPP_SINGLEPAGE** pages require more memory in general. *Do not outsmart yourself!* Experts might wonder whether this step is really required for

the 68040 and 68060 MMU which do not implement "early termination descriptors". Please feel ensured *that it really is.*

MAPP_COPYBACK If the specified pages are cache-able, i.e. **MAPP_CACHEINHIBIT** is not set, turn on the copy-back cache. This means that writes of the program will not be written back to memory immediately, but will be buffered until the cache entry is required otherwise. This will cause a quite noticeable speedup. This bit will be ignored by the 68030 and the 68851 which do not implement a copy-back cache.

MAPP_INVALID Mark the specified memory range as invalid. Accessing it by either a read or a write will cause a segmentation violation exception. If the **MAPP_REPAIRABLE** bit is not set, you may ask the MuLib to keep an additional **ULONG** with the page which will be passed to the exception hooks to identify the origin of the exception. This long word is specified by the **MAPTAG_USERDATA** tag, defined in `mmu/mmutags.h` file. This property flag is the "agressive form" of **MAPP_BLANK** as it may generate exceptions.

> *Zero is a Special Number.* You are free to mark the first — or so to say, "zeroth" — page in memory as invalid. The MuLib provides a special kludge to allow accesses to the global system constant `AbsExecBase` even with the zero page invalidated, and it will also emulate accesses to valid chip memory in this range. Needless to say that the emulation is always slower than the real thing. This kludge can be disabled for special purposes, and the low memory limit is adjustable. Study the "Building and Adjusting Contexts" section for details. For the experts: You guessed right, this is how "MuForce" works.

MAPP_REMAPPED Tell the MuLib that the physical address is different from the logical address and that the accesses to this page should be redirected to "elsewhere". The **lower** argument to **SetProperties** specifies the logical address to be re-mapped, the physical destination has to be specified by the **MAPTAG_DESTINATION** tag item, see `mmu/mmutags.h`.

MAPP_SWAPPED The specified memory area is currently "swapped out" on an external medium like a HD. In case a read or write access to this page is made, the MuLib will generate a page fault exception and call the "swapper" exception hooks to load the page back into memory again. If the **MAPP_REPAIRABLE** bit is not set, you may specify an additional **ULONG** which is passed to the exception handlers and which could be used to locate the block on the external medium. This long word is set by the **MAPTAG_BLOCKID** tag item, see `mmu/mmutags.h`.

MAPP_ROM This is the "defensive" form of **MAPP_WRITEPROTECTED** bit. The specified memory region is "simulated" read-only memory, write accesses are silently tolerated but will not alter the memory. Ideal for Kickstart re-mapping to provide a silent write protection for the ROM image.

MAPP_SHARED Shares the corresponding definition with the public Context. This bit is currently not yet implemented and should not yet be set. Unlike **MAPP_GLOBAL** which corresponds to a hardware bit for the 68040 and 68060, this is a software driven bit only which will be used for administrational purposes.

MAPP_TRANSLATED The specified memory region is — probably partially — under control of the transparent translation registers. Reprogramming the MMU for this memory area is therefore ignored by the MMU. Even though this sounds complicated, there's currently no need to care about this bit at all because the MuLib tries to get rid of the transparent translation registers very early at startup by simulating them by a proper MMU setup instead, and clearing them afterwards. Hence, you will never find this bit set anyhow, and you should never set this bit manually yourself. Just leave it alone for now.

MAPP_REPAIRABLE By setting this bit you tell the MuLib that you want to be able to repair an access to an invalid or write-protected page. The MuLib will then try to obtain the data that was written to the invalid page and will forward this data to the exception handler, or it will allow the exception handler to provide the data that should be read-in by the CPU when accessing the invalid page. Hence, by setting this bit, you could emulate some hardware registers in the specified range by means of a clever exception handler that absorbs or provides the data for the hardware registers. If this bit is not set, the MuLib will not always be able to provide the written-out data or to push back data into the CPU pipeline. Instead, the exception handler must either abort the access, or must swap in a page to allow the CPU to retry the access.

> *Repair Service is Expensive.* Even though the **MAPP_REPAIRABLE** bit is a very powerful feature, it has its price. First of all, access to the CPU pipeline has to be emulated for most CPUs, which means that this is slow. Furthermore, the MuLib does not offer any additional page data for **MAPP_REPAIRABLE** pages, hence **MAPP_BLOCKID** or **MAPP_USERDATA** are not available. The "MuForce" debugging tool uses this feature to present the data that was written out on an access fault, and to push back "dummy" data into the faulty program.

MAPP_IMPRECISE Only meaningful if **MAPP_CACHEINHIBIT** is set, too, this tells the 68060 MMU to be a bit "sloppy" on true physical bus errors. Therefore, this bit should be set only for memory or I/O areas that cannot generate bus errors, but which cannot tolerate caching. This bit is safely ignored by all other MMUs. Typically, this bit is set for video RAM, like the native "chip memory" of the Amiga motherboard or the RAM on graphics cards. This memory is always valid to access, but it cannot be cached because additional circuits like the blitter operate on the memory, bypassing the CPU.

MAPP_INDIRECT The corresponding page in memory is handled by a descriptor you constructed and your code has full control over. The MuLib will just generate a reference to your hardware descriptor, but will otherwise not care about it. Hardware descriptors should be build by the **BuildIndirect()** and defined by the **SetIndirect()** call in a hardware-independent way, and should be read by **GetIndirect()** only. A hardware descriptor is always four bytes long, and must be placed at a long word boundary or even at a cache line boundary — which is 16 bytes — in case you want to read it back with **GetIndirect()** later. Its physical address is specified by the **MAPTAG_DESCRIPTOR** tag item, defined in `mmu/mmutags.h`. On access faults, the MuLib will never report your descriptor as the descriptor that caused the exception, but instead its own "indirect descriptor" that points to your descriptor.

> *Too Indirect for Beginners.* Hardware descriptors are truly powerful because they are extremely fast. On the other hand, they are very cumbersome to handle, and definitely an advanced feature. Don't try to mess with them unless you know what you're doing. Study the "Low Level MMU Setup" chapter for details.

MAPP_BUNDLED The specified memory range is bundled to one single page in memory, repeated over and over again, filling up the full range. Hence, in a **MAPP_BUNDLED** memory region, the same physical memory page is mirrored over and over again to a continuous range of logical addresses.

MAPP_USER0 This bit is strictly for your purposes. The MuLib will completely ignore this bit, and will keep it for you. It does not correspond to any hardware function of any MMU at all — don't mix this with the "user page attribute 0". These user attributes, along with all other high-level attributes, are *also* visible for the low-level functions **GetPageProperties()** and related.

MAPP_USER1 Reserved for public use, similar to **MAPP_USER0**.

MAPP_USER2 Again kept free for you.

MAPP_USER3 And another one for you.

MAPP_NONSERIALIZED Only significant if **MAPP_CACHEINHIBIT** is set, too, and only read by the 68040 MMU, safely ignored by all others. This bit tells the 68040 that it may re-order accesses to the specified memory range in order to speed up the bus throughput. Hence, accesses on the physical bus may appear in a different order than the accesses made by software. This bit should not be set for true hardware mapped I/O, but a typical application would be video RAM like the native "chip memory" or the RAM on graphics boards. It can't be cached because custom hardware like the blitter accesses it by means of DMA, but the order of accesses does not matter.

MAPP_IO The corresponding memory range are memory mapped I/O registers. This bit has no influence on the MMU setup at all, but it is read by tools like "MuForce" or the "disassembler.library" to avoid accesses to this "memory" for hex dumps or disassembling. Custom, non-auto-configuring hardware should have this bit set to inform these tools that they should keep their hands off.

The counterpart of **SetProperties()** is the **GetProperties()** function: It returns the property flags for a given logical address:

```
struct MMUContext *ctx;
ULONG flags,address;

        flags = GetProperties(ctx,address,TAG_DONE);
```

Unlike **SetProperties()**, the address need not to be aligned to a multiple of the page size. However, the returned properties will only depend on the page the address belongs to. Additionally, the following tags can be passed in, defined in `mmu/mmutags.h`:

MAPTAG_DESTINATION Requires a pointer to **void *** as argument. If the passed in logical address is re-mapped to a different physical address, this pointer will be filled in and will point to the corresponding physical address. This tag is only made use of if the **MAPP_REMAPPED** property is found enabled.

MAPTAG_BLOCKID Requires a pointer to a **ULONG** as argument. This **ULONG** is filled in if the page is swapped out, returning an identifier which was selected by **SetProperties()**. Only available for **MAPP_SWAPPED** pages.

MAPTAG_USERDATA Requires a pointer to a **ULONG** as argument which will be filled with the "cookie" of **MAPP_INVALID** pages, if they have been set by **SetProperties()** in first place.

MAPTAG_DESCRIPTOR Takes a pointer to a **ULONG *** as argument which is filled for **MAPP_INDIRECT** pages with the pointer to the true physical hardware descriptor used to handle this page.

The returned flags value reflects the MMU properties in the high-level of the MMU setup; especially, the **MAPP_USED** and **MAPP_MODIFIED** bits do *not* correspond to the state of the MMU hardware "U" and "M" bits, but just for the pre-selected value of these bits in case the MuLib has to rebuild parts of the hardware level.

3.3 Context Locking

Since more than one Task could try to operate on the same Context at once, you are highly recommended to "lock" the context before you proceed and modify its setup by **SetProperties()**. Even though the MuLib itself keeps care that its data structures remain valid even in this situation, it might be desirable to group Context operations and to protect them by granting exclusive access to the Context. This is done by

```
struct MMUContext *ctx;

        LockMMUContext(ctx);
```

After having modified the high-level of the Context by calling **SetProperties()**, the changes must be loaded to the hardware. This step is similar to the **MakeVPort()** call of the graphics.library: It translates the abstraction layer to the true hardware data. One single call is enough to proceed:

```
struct MMUContext *ctx;
BOOL result;

        result = RebuildTree(ctx);
```

Like **SetProperties()**, this call may fail due to out of memory conditions. In this case, the hardware layer of the MMU setup remains unchanged, but all the modifications in the software layer remain active, and remain marked as "changed". Hence, if **RebuildTree()** is called again later, and if more memory is available, your changes will become active. Once you're done, you should unlock the Context again to allow other tasks to modify it:

```
struct MMUContext *ctx;

        UnlockMMUContext(ctx);
```

Since **SetProperties()** and even **RebuildTree()** may fail, you are in trouble in case the system is low on memory since it would leave you alone with a half-correct MMU setup which can't be completed, and, in worst case, can't even restored to the original setup since a second **SetProperties()** call used to restore the original settings could fail as well. Luckily, the MuLib provides functions to help you in this situation. The idea is to first make a backup of the current MMU setup, using

```
struct MMUContext *ctx;
struct MinList *ctxl;

        ctxl=GetMapping(ctx);
```

then to run all modifications, finally to call **RebuildTree()**. If something goes wrong, one single call is enough to restore the orignal MMU setup, namely

```
struct MMUContext *ctx;
struct MinList *ctxl;

        SetPropertyList(ctx,ctxl);
```

Unlike **SetProperties()**, the **SetPropertyList()** call *cannot* fail. It uses the backup made before to restore the MMU setup. It *cannot* be used, though, to restore a MMU setup which has been translated into a true hardware table already and which is loaded into the MMU, even if you call **RebuildTree()** explicitly after **SetPropertyList()**. The reason is that the high-level functions keep so called "dirty bits" of the MMU table. These bits are never visible from the outside, but are set for each memory region you modified with **SetProperties()**, and cleared on a **RebuildTree()**. This allows the rather slow **RebuildTree()** call only to re-compute the parts of the MMU table which have been modified; re-computing the full MMU setup would take too much time and would drastically reduce performance. However, **SetPropertyList()** does not only restore the high-level MMU setup, it also restores the dirty bits. To be more specific, calling **RebuildTree()** after **SetPropertyList()** would not notice that a complete restauration of the MMU setup has taken place, and would leave the low-level MMU setup alone.

Finally, regardless of whether you made use of the backup or not, it must be released by

```
struct Context *ctx;
struct MinList *ctxl;

        ReleaseMapping(ctx,ctxl);
```

You can use a backup only *once* for *one single* **SetPropertyList()** call. Afterwards, the backup will be "empty" and cannot be used anymore. Nevertheless — regardless of whether **SetPropertyList()** was called or not, you *have to* tell the MuLib that you do not require it anymore. This is done by **ReleaseMapping()**.

To conclude, a proper and safe MMU modification, including proper error handling, would look like this:

```
struct MMUContext *ctx;
struct MinList *ctxl;
BOOL fine = TRUE;

        /* Lock the context */
        LockMMUContext(ctx);

        /* make a backup */
        ctxl=GetMapping(ctx);

        /* got a backup? */
        if (ctxl) {

         /* Now run all the modifications */
         fine = SetProperties(ctx,...);
         if (fine) {
          /* etc, etc... */
          fine = SetProperties(ctx,...);
         }

         /* and finally, build the hardware table */
         if (fine) {
          fine = RebuildTree(ctx);
         }

         /* Uhoh, something went wrong! */
         if (!fine) {
                /* Restore the previous setup */
                SetPropertyList(ctx,ctxl);
         }
        } else fine = FALSE;
```

```
        UnlockMMUContext(ctx);
        ReleaseMapping(ctx,ctxl);
```

3.4 Modifying More Than One Context at Once

Please recall that the MuLib keeps user and supervisor accesses separate, and that each user Context comes with a corresponding supervisor Context. This means specifically that you sometimes want to modify two or more Contexts at once, typically the user Context and its supervisor Context. If handled the naive way, several race conditions could result: For example, consider that your program locks the user Context first, and then locks the supervisor Context. Assume further that another program attempts to modify the two Contexts simultaneously, but locking the supervisor Context first and the user Context later. This could yield to the classical "deadlock" situation where your program keeps the user Context locked but can't run on because the supervisor Context is obtained by the second Task, and the second Task can't continue because it tries to obtain the user Context which is already locked by your task. Therefore, *if you want to lock more than one Context at once*, you *absolutely must* lock the complete Context list before you lock individual Contexts. This is done by the following call:

```
LockContextList();
```

Nevertheless, you need to lock the individual Contexts afterwards. The Context list lock is released by

```
UnlockContextList();
```

when you're done. Both calls do not take any arguments.

A second problem is caused by the **RebuildTree()** call: If you want to compute the low-level MMU setup for two Contexts, it might happen that the first **RebuildTree()** succeeds, but the second call fails due to lack of memory; an attempt to restore the first tree could fail as well, making it impossible to restore the former setup. To help you in this situation, the MuLib provides a function that rebuilds several MMU setups at once such that either all of them are rebuild successfully, or none of them has been touched. This call comes in two forms, one parameter based form **RebuildTreesA()** which takes a **NULL**-terminated array of Context pointers, and a stack based call **RebuildTrees()** whose last argument is set to **NULL**, similar to a tag list. The last form is conveniently used from high-level languages like C.

```
struct MMUContext *ctx,*sctx;
BOOL fine;

        fine = RebuildTrees(ctx,sctx,NULL);
```

would, for example, rebuild two MMU setups at once. The following example code shows how to modify safely the property flags for the public Context and its supervisor Context at once:

```
/*
 * SetCacheMode:        Modify the cache mode of the address range
 *                      "from" to "from+size-1" in both the
 *                      default context and the default supervisor
 *                      context.
 *                      "flags" defines the new properties,
 *                      "mask" which bits are to be altered.
 *
 *                      Returns a dos-type error code.
```

```
 *
 * Taken from the "MuSetCacheMode" sources, (c) Thomas Richter.
 *
 */

#include <exec/types.h>
#include <exec/lists.h>
#include <dos/dos.h>
#include <mmu/context.h>
#include <utility/tagitem.h>

#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/mmu.h>

int SetCacheMode(ULONG from,ULONG size,ULONG flags,ULONG mask)
{
struct MMUContext *ctx,*sctx;   /* default context, supervisorcontext */
struct MinList *ctxl,*sctxl;    /* backups                            */
ULONG psize;                    /* the page size                      */
int err;

        ctx=DefaultContext();   /* get the default context */
        sctx=SuperContext(ctx); /* get the supervisor context for this one */

        psize=GetPageSize(ctx); /* get the page size */

        /* Now check for proper alignment of the data passed in */
        if (size & (psize-1)) {
                Printf("The given size 0x%lx is not divisible "
                       "by the page size 0x%lx.\n",size,psize);
                return ERROR_BAD_NUMBER;
        }
        if (from & (psize-1)) {
                Printf("The given address 0x%lx is not divisible "
                       "by the page size 0x%lx.\n",from,psize);
                return ERROR_BAD_NUMBER;
        }

        /*
        ** Page sizes of the user and the supervisor context are always
        ** identical.
        */


        /* Lock first the context list, then the two contexts */
        LockContextList();
        LockMMUContext(ctx);
        LockMMUContext(sctx);

        err=ERROR_NO_FREE_STORE;
```

```
/* Make backups of the MMU setup */
if (ctxl=GetMapping(ctx)) {
 if (sctxl=GetMapping(sctx)) {

   err=0;

   /* Set the flags in the user context */
   if (!SetProperties(ctx,flags,mask,from,size,TAG_DONE)) {
        err=ERROR_NO_FREE_STORE;
   }

   /* and just the same in the supervisor context */
   if (!SetProperties(sctx,flags,mask,from,size,TAG_DONE)) {
        err=ERROR_NO_FREE_STORE;
   }

   if (err==0) {
        /*
        ** If everything is fine so far, rebuild the trees
        ** to write this setup directly into the hardware
        */
        if (!RebuildTrees(ctx,sctx,NULL)) {
                   err=ERROR_NO_FREE_STORE;
        }
   }

   /*
   ** Uhoh, something went wrong!
   ** We better restore what we found before!
   */

   if (err) {
           SetPropertyList(ctx,ctxl);
           SetPropertyList(sctx,sctxl);
   }

   /*
   ** Now release the backups. Even if we used them,
   ** this step *IS* required.
   */
   ReleaseMapping(sctx,sctxl);
 }
 ReleaseMapping(ctx,ctxl);
}

/* Unlock the contexts and the list */
UnlockMMUContext(sctx);
UnlockMMUContext(ctx);
UnlockContextList();
```

```
        /* and say goodbye! */
        return err;
}
```

3.5 Function Reference

Here's again a quick function reference for all the calls introduced in the last section:

Table 2: High Level MMU Tree Control Functions

MuLib function	Description
GetPageSize()	Return the size of a MMU page in bytes
SetProperties()	Define property flags for one or more pages
GetProperties()	Return the property flags for one address
LockMMUContext()	Lock a context from modification
UnlockMMUContext()	Release a context lock
AttemptLockMMUContext()	Attempt to lock a context
LockContextList()	Lock the list of contexts
UnlockContextList()	Release the list lock
AttemptLockContextList()	Attempt to lock the context list
RebuildTree()	Build the low-level from the high-level data
RebuildTrees()	Rebuild more than one tree at once
GetMapping()	Make a backup of the MMU setup
ReleaseMapping()	Release a MMU setup
SetPropertyList()	Replace the high-level setup by a backup

4 Low-level MMU Setup

High-level MuLib functions have the disadvantage that they cannot be called from interrupt or supervisor code. Low level functions can, but *because* they are interrupt-callable, they provide no locking mechanism. If some other Task decides to overwrite the low-level MMU table — let it be by calling the high-level function **RebuildTree()** or by modifying the low-level directly — then your changes are lost. However, there is at least a way to handle the first situation by means of the so-called "page access exception" which will be described below. Another restriction is that the low-level functions require a special preparation step, namely the **MAPP_SINGLEPAGE** property flag *must* be set by means of the high-level functions, and the MMU tree must be rebuild afterwards. Furthermore, the low level functions operate only at one page a time.

4.1 Defining Properties on the Low Level

The function

```
struct MMUContext *ctx;
ULONG flags,mask,page;
BOOL result;

        result = SetPageProperties(ctx,flags,mask,page,TAG_DONE);
```

is the low-level equivalent of the **SetProperties()** call. All parameters and flags are identical, except that no address range can be specified, but only a single page at a time will be modified. Its logical address must be passed in as the **page** parameter. As for **SetProperties()**, it *must* be aligned to a multiple of the page size or the call will fail. Accordingly,

```
struct MMUContext *ctx;
ULONG address,flags;

        flags = GetPageProperties(ctx,address,TAG_DONE);
```

will read the flags from the low-level MMU descriptor. As a special case, the **MAPP_USED** and **MAPP_MODIFIED** properties reflect the state of the "U" and "M" bits of the true hardware descriptor and tell you whether this page has been accessed, or has been written to since the last time you cleared this bit.

> *Low, but not Ground Level.* The **SetPageProperties()** and **GetPageProperties()** function both directly modify the MMU hardware descriptors, but especially the latter does not really read the hardware descriptor except for the **MAPP_USED** and **MAPP_MODIFIED** flags. This is because most of the property flags do not correspond to features the MMU offers directly, but have to be emulated by software on some or all of the members of the MC68K MMU series. This makes little difference as long as you keep in mind that you *must not* hack on the MMU directly.

Both functions are interrupt-callable, and neither break a **Forbid()** nor a **Disable()** state. Hence, it is safe to call these from critical code if you have to. Remember, however, that both calls require the enabling of **MAPP_SINGLEPAGE** on the high level.

4.2 Reading and Writing Indirect Descriptors

The functions **SetPageProperties()** and **GetPageProperties()** are much faster than a **RebuildTree()** call of the high-level functions, but are still not the optimum of speed. As long as you have to handle small chunks of memory, indirect descriptors will work better, but offer less control and are much more cumbersome to handle.

Powerful, but Dangerous. One of the important drawbacks of indirect descriptors to keep in mind is that they *do not* support DMA operation. Especially, *never ever* read or write memory which is mapped by indirect descriptors by means of Os I/O functions like **Read()**, **Write()** or **DoIO()**. The MuLib will not be able to handle some cache related race-conditions for them. If you want to access them, make a copy of the page contents first and run the I/O operations on the copy.

An indirect descriptor works as follows: Typically, the MuLib designs all the hardware descriptors for the MMU itself, but for indirect descriptors, it will just place a reference in the MMU which points to a descriptor you have to provide. By modifying your descriptor, you get direct control over the MMU without much overhead. The descriptor is just a long word aligned long word or a long word in a cache line aligned = 16 byte aligned array which is a multiple of cache lines long. The latter, more restrictive alignment restriction holds in case you need to read back the descriptor later using **GetIndirect()**. In principle, you could setup this descriptor yourself, but *how* this must be done depends of course on the Amiga the code runs on. The MuLib helps you here by offering functions to pre-calculate the required descriptors to abstract from the hardware. Second, you could also place the descriptors in memory yourself, but due to some firmware features of several members of the 68K series, you'd better not try this yourself. The MuLib knows very well the race conditions that show up here, and knows how to handle them. Last but not least, you should also avoid reading the descriptors yourself, just for the same reason: A simple read access to a hardware MMU descriptor has some side-effects the MuLib has to keep in mind.

The first step in building an indirect descriptor is to allocate four bytes of memory, with proper alignment. The Os function **AllocMem()** is fine as long as you only want to write descriptors because it already offers long word alignment. For the more restrictive alignment requirement of **GetIndirect()**, you need to call the MuLib function

```
void *array;
ULONG size;

        array = AllocAligned(size,MEMF_PUBLIC,16);
```

where **size** is divisible by 16 as well. More about this call is in the "Miscellaneous Functions" chapter.

The second step is to obtain the *physical address* of the logical address you got from **AllocMem()**. In most cases, both will be identically, but they need not to be. The function

```
struct MMUContext *ctx;
ULONG oldflags;
void *logical;

        oldflags = PhysicalLocation(ctx,&logical,sizeof(ULONG));
        /* "logical" contains now the physical address */
```

will do this for you. The third step is to pre-calculate all the descriptors you plan to make use of. Since indirect descriptors are used for time-critical applications, this step avoids the overhead in the later usage. The following call will do this:

```
struct MMUContext *ctx;
ULONG mask,oldflags;
ULONG address,flags,descriptor;

        descriptor = BuildIndirect(ctx,address,
                                   (flags & mask) | (oldflags & (~mask)));
```

As always, this function requires the Context in the **ctx** argument. However, unlike **SetPageProperties()** or **SetProperties()**, only a subset of the property bits are provided. Especially, there is *no* **MAPP_REMAPPED** bit. This is handled differently. Instead of specifying a logical address and — possibly — a different physical address, you need to specify the physical address itself. If it is identical to the logical address, no re-mapping will occur, and if its not, the access will be re-directed to the specified page. Again, this address must be a multiple of the page size. The physical address should be obtained from the logical address by calling the **PhysicalLocation()** function, as before, *even in case you do not plan to re-map it.* It is good to write software in a defensive way, and it might happen that the memory you allocated in first place for the page has been re-mapped already. Since **BuildIndirect()** does not offer a **mask** parameter, the above example call shows how to mask in the desired flags yourself, and how to carry over parts of the old flags.

Depending on the hardware, only a subset of the following property bits is supported:

MAPP_WRITEPROTECTED The page will be write protected. Writes to this area will cause a segmentation fault.

MAPP_USED The "U" bit of the descriptor will be set. The MMU will set this bit if the page gets accessed in any way, too.

MAPP_MODIFIED The "M" bit of the descriptor will be set. The MMU sets this bit, too, on any write access that goes into this page. Due to a hardware feature of some of the 68K MMUs, *never ever* set this bit together with **MAPP_WRITEPROTECTED** and without **MAPP_USED** or the MMU might hang.

MAPP_INVALID The page will be marked as invalid. Accessing it will cause a segmentation violation exception. However, note that the bit **MAPP_REPAIRABLE** is *not* available as property bit for indirect descriptors itself. You may, however, still ask the MuLib for the repair service by setting the **MAPP_REPAIRABLE** bit in the corresponding **MAPP_INDIRECT** descriptor pointing to your descriptor. Even though indirect descriptors support the user data field to some extend, using **MAPTAD_USERDATA** is discouraged. This is because the descriptor will not be able to hold *all* 32 bits of your data, some of the lower order bits will be required for the purposes of the MMU and are therefore lost.

MAPP_CACHEINHIBIT The corresponding memory page will not be kept in the CPU cache.

MAPP_IMPRECISE Only available if **MAPP_CACHEINHIBIT** is set as well, this tells the 68060 MMU to react a bit sloppy on real bus errors. Ignored and read as zero by all other MMUs.

MAPP_NONSERIALIZED Again, this is only valid for **MAPP_CACHEINHIBIT** pages, and ignored and read as zero by all except the 68040 MMU. It tells the 68040 that it may re-order accesses to the page to improve performance.

MAPP_COPYBACK Enable the copy-back cache for cache-able pages. This bit is ignored and read as zero by the 68030 and 68851 MMU.

MAPP_USERPAGE0 Set the "user page attribute 0" CPU output line on accesses of this page. This is only available for the 68040 and 68060 and ignored and read as zero for the 68030 and 68851. There's currently no Amiga hardware I know of which keeps care about this hardware line anyhow.

MAPP_USERPAGE1 Sets the "user page attribute 1" CPU control line.

MAPP_GLOBAL Sets the "global" bit of the descriptor, which is only available for the 68040 and the 68060. It is ignored and read as zero by the 68030 and the 68851. Setting this bit means that certain specialized instructions will not flush this descriptor from the MMU cache. The MuLib writes only descriptors with this bit cleared and does not use these instructions. It will always flush descriptors independent of the G bit. There is currently little use of this bit, so please leave it alone.

Passing in unsupported bits causes the MuLib to ignore these bits. Especially, if you read back the descriptor later, you might find different properties than intended because of missing hardware support. For example, if you set the **MAPP_COPYBACK** bit on a 68030 based machine, and you check the descriptor later, you'll find this bit disabled. Especially, note that the following properties are *not* supported:

MAPP_REMAPPED is unsupported because you have to pass in the physical destination anyhow.

MAPP_REPAIRABLE is unsupported. However, you still get the same service by setting this bit "one level up" in the **MAPP_INDIRECT** descriptor pointing to your new descriptor.

MAPP_SUPERVISORONLY is unsupported. To emulate it, build separate descriptors for the user and the supervisor Context and set the user descriptor to **MAPP_INVALID**.

MAPP_ROM is unsupported. However, you are still able to get this feature if you set the descriptor to **MAPP_WRITEPROTECTED** and, additionally, set this bit "one level up" in the **MAPP_INDIRECT** descriptor.

The result code of **BuildIndirect()** is either a valid descriptor value, or the special result code `BAD_DESCRIPTOR` defined in `mmu/descriptor.h`. Especially, **NULL** does *not* indicate an error.

> *Indirection's Unmasked.* You should have noticed that **BuildIndirect()** does not come with a mask-type argument. Hence, it is not able to read and alter the current property flags of the page you want to address. Instead, you've to read the property flags yourself, for example by **GetProperties()** or **PhysicalLocation()**, and have to mask-in the desired flags yourself. This step is important because it is not clear whether the memory your page will be kept is is, for example, cache-able or not. Hence, you have to carry the cache flags over, as in the example above.

The next step is to set your descriptor to one of the pre-calculated values. This is done by

```
ULONG *descriptor,address,value;

        SetIndirect(descriptor,address,value);
```

which writes the pre-calculated value into your descriptor. This is also the function which should be called to exchange descriptors rapidly. The **descriptor** argument is the *physical* location of the hardware descriptor you allocated in the first step and whose physical address has been calculated in the second step. The **value** argument is the descriptor value calculated by **BuildIndirect()** before. Finally, **address** is the *logical* address which is covered by this descriptor. In case you want to re-use the descriptor for more than one logical address, pass in **-1L** instead as a special case.

> *Keep Care about the Cache!* Unlike the **SetPageProperties()** call, the **SetIndirect()** function does *not* touch the CPU cache for the page you've modified, mainly for speed reasons. Therefore, it is *absolutely necessary* to push back the cache of the page(s) whose MMU setup is altered by **SetIndirect()**. The Exec functions **CacheClearE()** and **CacheClearU()** will help you here. If you do not follow this rule, you might observe

strange effects up to complete CPU lockups. The one and only exception to this rule is that you do not need to push caches if you change the physical destination of the logical page(s) addressed by the indirect descriptor you installed. This works even for the 68851 and the 68030 whose cache is addressed by logical rather than physical addresses. The MuLib knows about this special case.

Finally, as a last step, you have to link in your descriptor into the MMU setup. This requires calling either **SetProperties()** and **RebuildTree()** or **SetPageProperties()** with the property flags bit **MAPP_INDIRECT** set to one and the physical address of your descriptor as **MAPTAG_DESCRIPTOR** tag item. You may also add the **MAPP_REPAIRABLE** and **MAPP_ROM** bits as mentioned in the list above. They need to be set *here* and not in your descriptor. The MMU will now use your new descriptor, and you're able to re-define the descriptor very rapidly with the **SetIndirect()** call.

In case you want to alter more than one indirect descriptor at a time, the MuLib offers a function for re-defining a complete array of descriptors at once. This function, **SetIndirectArray()** is typically faster than calling **SetIndirect()** in a loop. Its synopsis is as follows:

```
ULONG *descriptors,*values,num;

        SetIndirectArray(descriptors,values,num);
```

The first argument, **descriptors**, points to the *physical* base address of the indirect descriptors to be filled in. Note that you must have ensured that this array is really a continuous array of physical addresses, i.e. it is not possible that this array, even though a continuous range of logical addresses, is split into several non-adjacent physical memory pages. The **PhysicalLocation()** function is able to check this, see the "DMA Support Functions" chapter for more information about this call. For fragmented memory models, you have to call **SetIndirectArray()** several times, once for each fragment.

The **values** array points to a **ULONG** array of the MMU flags that should be filled in, one **ULONG** for each descriptor. The **SetIndirectArray()** function will, "morally speaking", copy the contents of this array to the first array, but considering the caveats when modifying MMU descriptors. The last argument is the number of descriptors to be set and hence the size of both arrays. Passing zero here is allowed and is a no-op.

As for **SetIndirect()**, proper cache management is up to yourself. Hence, if you alter the cache modes of some memory pages, e.g. by changing them from **MAPP_COPYBACK** to **MAPP_CACHEINHIBIT**, it is up to you to push back the CPU caches by means of **CacheClearU()** or **CacheClearE()**.

Finally, to read an indirect descriptor, use

```
struct MMUContext *ctx;
ULONG *descriptor;
struct AbstractDescriptor adt;

        GetIndirect(ctx,&adt,descriptor);
```

The **ctx** argument is the Context, as always, and **descriptor** is the *physical* address of the descriptor to be read. The **adt** structure need not to be initialized. It is filled in by the call as follows:

```
struct AbstractDescriptor {     /* defined in mmu/descriptor.h */
        ULONG   atd_Pointer;
        ULONG   atd_Properties;
        UWORD   atd_LowerLimit;
        UWORD   atd_UpperLimit;
```

```
        UBYTE   atd_ThisType;
        UBYTE   atd_NextType;
        UWORD   atd_reserved;
};
```

The **adt_Pointer** field is either the *physical* address the accesses to the page his descriptor is installed for are redirected to, or the user data if this descriptor is of invalid type. Note that providing user data for invalid indirect descriptors is discouraged because the MuLib will not be able to preserve all 32 bits of your data. Otherwise, the **adt_Pointer** component will be the same address that was passed in as physical destination to **BuildIndirect()**.

atd_Properties is the set of property flags read from the descriptor. This *need not* to be identical to the properties setup by **BuildIndirect()**, for two reasons: First, the MMU sets the "U" and "M" bits as soon as any access or a write access happens to the page or pages handled by the descriptor. Second, not all MMUs support all properties. Unavailable properties are ignored by **BuildIndirect()**, and read as zero by this function.

Please leave all other fields alone, they are not documented and should not be read, and please do *not* try to read the descriptor yourself. First, it is hardware dependent, and second, you would need to take care about some hardware features and side-effects.

> *Beware of Oddities!* The alignment rules for indirect descriptors might seem strange indeed. As long as you do not use **GetIndirect()**, long word alignment is good enough. Since **AllocMem()** guarantees even alignment to quad words, ordinary Exec memory allocations will suffer. However, special cache related considerations when *reading* the descriptors require that they do not share cache lines with ordinary program code or data. Therefore, if you allocate memory for descriptors and you suppose to call **GetIndirect()** on them, make sure that you allocate a multiple of the cache line size, which is 16 bytes, and make sure that the memory block you allocated is aligned to a cache line boundary. Hence, the MuLib function **AllocAligned()** is required here. When allocating a complete array of descriptors, each individual descriptor in this array need not to be — and will not be — aligned, but the *array boundaries* have to. Therefore, round the array size up to the next multiple of 16 bytes, and pass 16 as alignment parameter to **AllocAligned()**. Not following this guideline might appear to work most of the time, but **GetIndirect()** may return improper data and certain "surprise moments" may show up. The **SetIndirect()** calls are not touched by this problem.

The following example program shows how to use indirect descriptors:

```
/*******************************************************
 ** IndirectTest                                      **
 **                                                   **
 ** Test indirect page descriptors of the MuLib       **
 ** Release 1.01                                      **
 **                                                   **
 ** (c) 19.03.2000 Thomas Richter                     **
 *******************************************************/

/*
 * Compile and link without startup code.
 */

/* Includes */
#include <exec/types.h>
#include <exec/memory.h>
```

```
#include <dos/dos.h>
#include <mmu/context.h>
#include <mmu/mmutags.h>
#include <mmu/descriptor.h>

#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/mmu.h>

#include <string.h>

/* Protos */
int __saveds main(void);
int RunTests(void);
void DumpData(UBYTE *src,ULONG size);

/* Statics */
char version[]="$VER: IndirectTest 1.01 (19.03.2000) (c) THOR";
struct ExecBase         *SysBase;
struct DosLibrary       *DOSBase;
struct MMUBase          *MMUBase;

/* main */
int __saveds main(void)
{
int rc=25;

        /*
        ** Since we want to link without startup code,
        ** we need to open the system libraries here...
        */
        SysBase = *((struct ExecBase **)(4L));

        /*
        ** Open DOS and MMU
        **/

        if (DOSBase = (struct DosLibrary *)OpenLibrary("dos.library",37L)) {
                if (MMUBase = (struct MMUBase *)OpenLibrary("mmu.library",42L)) {

                        rc = RunTests();

                        CloseLibrary((struct Library *)MMUBase);
                } else {
                        Printf("IndirectTest failed: This program "
                               "requires the mmu.library V42 or better.\n");
                        rc = 10;
                }

                /*
                ** Everything above 64 is a system
```

```
		** error code we print over the console.
		*/

		if (rc>64) {
			PrintFault((LONG)rc,"IndirectTest failed");
			rc = 10;
		}
		CloseLibrary((struct Library *)DOSBase);
	}

	return rc;
}

/* RunTests */
int RunTests(void)
{
struct MMUContext *ctx;
struct MinList *ctxl;
ULONG pagesize;
ULONG *descriptor,*descriptorp;
ULONG values[2];
ULONG props[2];
UBYTE *page,*pagep[2];
int rc=25;

	/*
	** Get the context we're currently using
	** and its page size
	** furthermore, allocate a page.
	*/

	ctx		= CurrentContext(NULL);
	pagesize	= GetPageSize(ctx);
	page		= AllocAligned(pagesize*2,
					MEMF_PUBLIC|MEMF_CLEAR,pagesize);

	if (page) {

	 /*
	 ** Now allocate memory for the descriptor
	 ** this must be long-word aligned, hence
	 ** an AllocMem is fine here.
	 ** However, we need to know the physical location
	 ** of the descriptor.
	 */
	 descriptor	= AllocMem(sizeof(ULONG),MEMF_PUBLIC);

	 if (descriptor) {


	  /*
```

```
** Compute physical locations
** We do not assume that PhysicalLocation()
** truncates the address. All values are
** long/page aligned longs/pages, hence never cross a
** page boundary.
*/

descriptorp    = descriptor;
PhysicalLocation(ctx,(void **)&descriptorp,&pagesize);

/* And now for the pages */

pagep[0]       = page;
props[0]       = PhysicalLocation(ctx,(void **)&pagep[0],&pagesize);
pagep[1]       = page+pagesize;
props[1]       = PhysicalLocation(ctx,(void **)&pagep[1],&pagesize);


if (pagep[0] && pagep[1] && descriptorp) {

 /*
 ** Lock the context and make a backup of it.
 **
 */
 LockMMUContext(ctx);

 if (ctxl=GetMapping(ctx)) {

      /*
      ** Pre-calculate the values for the descriptors.
      ** The first descriptor maps the page to its TRUE physical
      ** location, the second one to the ROM, write-protecting
      ** it.
      ** Note that we need to use the physical addresses here.
      **
      ** MAPP_ROM protection must be archived by setting this
      ** property bit "one level up".
      **
      ** We furthermore set USED and MODIFIED to avoid unnecessary
      ** MMU writebacks, and transfer the old properties back
      ** into the descriptor properties
      **
      ** Note that this call returns BAD_DESCRIPTOR in case
      ** of an error, not NULL.
      */

      values[0] = BuildIndirect(ctx,(ULONG)(pagep[0]),
                                props[0]|MAPP_USED|MAPP_MODIFIED);
      values[1] = BuildIndirect(ctx,(ULONG)(pagep[1]),
                                props[1]|MAPP_USED|MAPP_WRITEPROTECTED);
```

```
if ((values[0] != BAD_DESCRIPTOR) &&
    (values[1] != BAD_DESCRIPTOR)) {
 /*
 ** Install the descriptor
 ** The first parameter is the physical address
 ** of the descriptor, the second the
 ** logical address of the page
 ** and the last the descriptor to install
 */

 SetIndirect(descriptorp,(ULONG)page,values[0]);

 /*
 ** Now install this descriptor
 ** We set this to MAPP_ROM because we want emulated
 ** ROM writeprotection.
 ** This is ignored if the descriptor itself is
 ** not write protected anyhow.
 ** We need the physical location of the descriptor
 ** here.
 */

 if (SetProperties(ctx,MAPP_ROM|MAPP_INDIRECT,
                      MAPP_ROM|MAPP_INDIRECT,
                      (ULONG)page,pagesize,
                      MAPTAG_DESCRIPTOR,descriptorp,
                      TAG_DONE)) {

  if (RebuildTree(ctx)) {

   /* Everything's fine.
   ** copy some dummy data into the page
   */
   memset(page,'*',(size_t)pagesize);

   /* now print parts of it */
   DumpData(page,0x10);

   /*
   ** install the other descriptor
   */
   SetIndirect(descriptorp,(ULONG)page,values[1]);

   /*
   ** Dump it again. Should be all zero now.
   */
   DumpData(page,0x10);

   /* Try to write to it. This should
   ** fail quietly.
   */
```

```
  *page = 'A';

  /* And dump it again */
  DumpData(page,0x10);

  /*
  ** install the old descriptor
  ** again
  */
  SetIndirect(descriptorp,(ULONG)page,values[0]);

  /*
  ** Now reset the context data.
  ** Disable the MAPP_ROM and MAPP_INDIRECT
  ** features. This call shouldn't fail or
  ** we are in trouble
  */

  if (SetProperties(ctx,0,MAPP_ROM|MAPP_INDIRECT,
                    (ULONG)page,pagesize,TAG_DONE)) {

       /* Restore the former MMU tree */
       if (RebuildTree(ctx)) {

               /*
               ** everything is fine now.
               */
               rc = 0;
       }
  }

  if (rc) {
       /*
       ** We're now in trouble.
       ** The old context couldn't be restored.
       ** Therefore, we do not release the descriptors
       ** such that the accesses are at least right,
       ** and restore the high-level by SetPropertyList()
       ** below. This will cause a mild memory leak,
       ** but the system will be fine.
       */
       Printf("IndirectTest: Can't restore the context.\n");
       descriptor = NULL;
  }
 }  else Printf("IndirectTest: Building the context failed.\n");
} else Printf("IndirectTest: Can't install the "
              "new descriptor.\n");

/*
** In case of an error, we restore now the high
```

```
                ** level of the context.
                ** This is all we could do.
                ** The high-level looks then fine again,
                ** and the low level contains either an
                ** indirect descriptor which we can't get
                ** rid of, but which maps ok, or is
                ** unchanged. The system will be fine
                ** in both cases.
                */

                if (rc) {
                       SetPropertyList(ctx,ctxl);
                }

               } else Printf("Can't build the new descriptors.\n");
               /* Release the mapping */
               ReleaseMapping(ctx,ctxl);

          } else rc = ERROR_NO_FREE_STORE;

          /*
          ** Release the MMU Context lock
          */
          UnlockMMUContext(ctx);

         } else Printf("IndirectTest: Can't perform the logical "
                       "to physical translation.\n");

         /*
         ** now release the descriptor
         */
         if (descriptor) {
                 FreeMem(descriptor,sizeof(ULONG));
         }
        } else rc = ERROR_NO_FREE_STORE; /* of if descriptor */

        FreeMem(page,pagesize*2);
       } else rc = ERROR_NO_FREE_STORE; /* of if page */

       return rc;
}

/* DumpData */
void DumpData(UBYTE *src,ULONG size)
{
        /*
        ** A pretty dumb memory dump
        */

        Printf("Memory contents at 0x%08lx : ",src);
        while(size) {
```

```
                Printf("%02lx ",*src);
                src++;
                size--;
        }
        Printf("\n");
}
```

4.3 Function Reference

The following is again the function reference for this chapter. The **PhysicalLocation()** call is explained and listed in the "DMA support functions" chapter below.

Table 3: Low-Level MMU Tree Control Functions

MuLib function	Description
SetPageProperties()	Define the low-level MMU setup
GetPageProperties()	Read the low-level MMU setup
BuildIndirect()	Pre-calculate an indirect descriptor
SetIndirect()	Define an indirect descriptor
SetIndirectArray()	Define an array of indirect descriptors
GetIndirect()	Read an indirect descriptor

5 DMA Support Functions

All the addresses used by software are *logical* addresses, they are translated by the MMU before they arrive at the memory and I/O chips in the form of electrical signals. However, DMA controllers bypass the CPU and the MMU and address memory directly, of course using physical addresses as there is no MMU between the controller and the RAM. Additionally, the data in RAM might not be "up to date" because the most recent data in the cache might not yet be written out to RAM. Moreover, the CPU might read parts of the I/O buffer and hence might load obsolete data into the cache without noting that this data is about to be altered by an external bus master. Therefore, the Os provides DMA control functions: Namely, to translate logical to physical addresses and to avoid cache race conditions.

5.1 Logical to Physical Translation Functions

As for most MuLib related calls, two different functions are available to translate a logical to a physical address: A high-level call which makes use of the data set constructed and controlled by **SetProperties()** and **GetProperties()**, and a low-level call which operates on the MMU descriptors written by **SetPageProperties()** and read by **GetPageProperties()**. Which of the two calls is apropriate depends on your requirements: The high-level call

```
struct MMUContext *ctx;
ULONG props,length;
void *address;

        props = PhysicalLocation(ctx,&address,&length);
```

translates a complete address range, starting at the address and length passed it. It returns the property flags for this address range. It sets both **length** and **address** to **NULL** in case the corresponding physical page does not exist.

Consider now a situation where a continuous block of logical addresses is build by remapping two non-adjacent physical memory blocks in a way that they form one continous memory block. Clearly, if a continuous logical memory block crossing the border line of the two physical blocks should be transported by a DMA access, this DMA access has to be broken up into two smaller accesses: The first DMA transfer accesses the logical region up to the end of the first physical memory block, and the second access starts at the first byte of the second physical block up to the end of the specified logical range. More complicated situations may arise if the logical memory range is split into several non-adjacent physical blocks. For that reason, **PhysicalLocation()** requires a *pointer* to the length of the memory block as well, and might return a smaller length if the full range of logical addresses is not continuously mapped. In case this happens, you have to transfer the beginning of the block as returned by **PhysicalLocation()**, then have to add the returned length to the *logical* address passed in *in the first step*, and have to call **PhysicalLocation()** again until the full memory block is transfered. The following code segment shows how this might look like:

```
int RunTransfer(BYTE *base,ULONG len)
{
UBYTE *physical;
ULONG phylen;
ULONG flags;
int error;

        while (len) {
                physical = base;
                phylen   = len;
```

```
        /* translate the address range */
                flags = PhysicalLocation(ctx,(void **)(&physical),&phylen);
                if (phylen==0) {
                        /*
                        ** Generate an error as the user
                        ** tries to read from or write to
                        ** invalid or blank memory.
                        */
                        return INVALID_ARGUMENTS;
                }
                error = TransferData(physical,phylen);
                if (error)
                        return error;

                /* and now go for the next block */
                base += phylen;
                len  -= phylen;
        }

        /* everything's fine */
        return RESULT_FINE;
}
```

However, the above code segment *does not* handle any caching problems, and does not lock the Context in any way. Moreover, the address returned by **PhysicalLocation()** reflects the state of the high-level MMU setup. This setup may differ from the true physical programming of the MMU if the high-level setup has been altered, but **RebuildTree()** hasn't been called yet, or the low-level setup has been modified by **SetPageProperties()** bypassing the high-level functions altogether. Furthermore, as a high-level function, **PhysicalLocation()** is not interrupt-callable. It is for these reasons usually not very well suited for DMA device drivers, but might find different applications. The corresponding low-level function

```
struct MMUContext *ctx;
void *logical,*physical;

        physical = PhysicalPageLocation(ctx,logical);
```

is interrupt-callable and reflects the current state of the MMU low-level setup, hence will be able to "see" modifications made by **SetPageProperties()**, but it translates only one address at a time. It does not check complete memory regions for overlapping. It doesn't keep care about proper cache handling, and neither about proper handling of page boundaries. Both is up to your code.

5.2 DMA Memory Control Functions

Instead of using **PhysicalLocation()**, you might want to use some smarter functions, namely **DMAInitiate()** and **DMATerminate()**. This function pair is interrupt-callable, and — additionally — does not care about high-level modifications that have not yet been entered the lower level by means of **RebuildTree()**. It doesn't obtain its information from the low-level either, but uses a data base which is updated from the high-level on each **RebuildTree()** call. This data base is guaranteed to be consistent all the time, Context-locking is not required. Since this data base is a copy from the internal high-level database the MuLib keeps, **DMAInitiate()** can't see modifications performed on the MMU setup by **SetPageProperties()** as well.

The following function should be run to to initiate a DMA transfer, and to translate logical to physical addresses:

```
struct MMUContext *ctx;
void *address;
ULONG length;
BOOL writetoram,ok;

        ok = DMAInitiate(ctx,&address,&length,writetoram);
```

The parameters **address** and **length** specify the logical memory block to be transfered, and as for **PhysicalLocation()**, the function may not only modify the **address** parameter to submit the correct physical page, but may also change the **length** call in case the logical address range passed in is not continuously mapped to a single memory block. The **write** argument must be set to **TRUE** for data transports from the DMA device *into* memory, which would be, for example, a *read access* for a harddrive. It must be set to **FALSE** for transfering data from memory to the device. Each call to **DMAInitiate()**, even a call with a **FALSE** result code, must be matched by one and only one call to **DMATerminate()**:

```
struct MMUContext *ctx;

        DMATerminate(ctx);
```

Note that this is different to how **CachePreDMA()** and **CachePostDMA()** handle errors or non-continuous memory blocks.

The following example code show a typical application of **DMAInitiate()** and **DMATerminate()**, like in a DMA device driver; the logical address base of the transfer, and the length as well as the data transport direction are passed in as arguments:

```
int RunDMATransfer(BYTE *base,ULONG len,BOOL writetoram)
{
UBYTE *physical;
ULONG phylen;
BOOL fine;
int error;

        while (len) {
                physical = base;
                phylen   = len; /* translate this address range */
                fine = DMAInitiate(ctx,(void **)(&physical),&phylen,writetoram);
                if (!fine) {
                        /*
                        ** Generate an error as the user
                        ** tries to read from or write to
                        ** invalid or blank memory.
                        */
                        DMATerminate(ctx);  /* <-- REQUIRED !! */
                        return INVALID_ARGUMENTS;
                }
                /* Initiate the DMA cycle */
                error = InitiateTheDMA(physical,phylen);
                /*
                ** In this very simple application, we do not
```

```
            ** multi-thread. In the ideal case, we would be
            ** free here to initiate the I/O of other
            ** devices.
            */
            WaitForDMACompletion();

            /* Terminate the DMA */
            DMATerminate(ctx);
            if (error)      return error;

            /* and now go for the next block */
            base += phylen;
            len  -= phylen;
        }
        /* everything's fine */
        return RESULT_FINE;
}
```

Similar to **PhysicalLocation**(), the **DMAInitiate**() and **DMATerminate**() functions do not touch the CPU caches. Proper cache handling is up to your code, again.

> *Comes in Pairs.* To note this again: It is *very* important that **DMATerminate()** is called correctly, *once and only once* for each single **DMAInitiate()** call, regardless of the return code. If you don't follow this guideline, all further **RebuildTree()** calls will wait forever.

Since there is currently no mechanism how a Task invoking a DMA transfer makes its Context known to the DMA device — note that most DMA transfers are initiated by a filing system, and not directly by the task requiring the data — the **ctx** parameter of these functions is currently ignored, but reserved for future applications and refinements. The **DMAInitiate**() and **DMATerminate**() functions operate always on the data set of the public Context. Therefore, for consistency with future improvements, please pass in the public Context *only*.

5.3 DMA and Cache Control functions

Even though **DMAInitiate**() handles the translation from logical to physical addresses, it does not keep care about proper CPU cache handling. Two Os functions handle all this in once, **CachePreDMA**() and **CachePostDMA**(). Both functions are not part of the mmu.library, precisely speaking, but are Exec functions instead. Nevertheless, the function code is provided by the MuLib as soon as it is loaded. Unfortunately, the logic of both functions is a bit tricky, therefore both should be discussed here again:

```
APTR logical,physical;
ULONG length,flags;

        physical = CachePreDMA(logical,&length,flags);
```

This function should be called before a DMA operation is started. It's purposes are manifold: First, it translates the logical address passed in into a physical address, its result code. Second, it checks for non-continuously mapped memory. In case the data block passed in is not one continuous block of physical memory, the length is truncated and a smaller length counter is returned. This is why you have to pass a *pointer* to the length — the function might *alter* this parameter. Finally, a flags value must be set. The following bits are currently defined in `exec/execbase.h`:

DMA_Continue This flag must be set on the second and all further calls to **CachePreDMA()** in case your code came back to complete a truncated DMA transfer.

DMA_ReadFromRAM Set this flag to indicate that the intended DMA transfer is from RAM into the external device, e.g. a harddisk *write* access.

Note that **CachePreDMA()** does not return an error in case the memory passed in is not mapped at all. The function will either provide a dummy page for the operation, or will terminate with the infamous "guru". Furthermore, no **Context** parameter is available, which means that **CachePreDMA()** performs the page translation always on the public Context. As there is currently no DMA device which is able to keep the Contexts correct — and even very few of them actually call this function anyhow — this is not really a loss.

Unlike the **DMAInitiate()** interface, **CachePostDMA()** must be called only *once*, namely after the DMA transfer is complete — or has been aborted. Here is its syntax:

```
APTR logical;
ULONG length,flags;

        CachePostDMA(logical,&length,flags);
```

The **logical** parameter has to be set to the *initial logical* address the DMA transfer has been started with. Note that you *must not* pass in a logical address which has been used in some subsequent calls to **CachePreDMA()**. The **length** parameter has to be set to the complete length of the DMA transfer, as initially intended. Do not pass in a truncated length as returned by **CachePreDMA()**. Finally, the following flags are available:

DMA_NoModify Set this flag in case the memory range hasn't been modified, hence to allow the code to avoid an unnecessary cache flush.

DMA_ReadFromRAM This flag *must* be set consistent to the **CachePreDMA()** call, namely if the data transfer direction is from RAM to the DMA controller.

Here's again an example code which shows how the two function should be called. Its arguments are as in the examples above.

```
int RunDMATransfer(BYTE *base,ULONG len,BOOL writetoram)
{
UBYTE *physical,*logical;
ULONG phylen,remaining,flags,iflags;
BOOL fine;
int error;

        logical   = base;
        remaining = len;
        error     = RESULT_FINE;
        /* Set DMA_ReadFromRAM on RAM --> device transfer */
        flags     = (writetoram)?(0):(DMA_ReadFromRAM);
        iflags    = (writetoram)?(0):(DMA_ReadFromRAM|DMA_NoModify);

        while (remaining) {
                phylen   = remaining; /* translate this address range */
                physical = CachePreDMA(logical,&phylen,flags);

                /*
```

```
		** We do not get a result code here. Maybe we
		** should check whether "physical" is zero.
		**/
		if (physical==NULL) {
			/*
			** Generate an error as the user
			** tries to read from or write to
			** invalid or blank memory.
			**
			** Note that CachePostDMA() requires
			** the initial parameters!
			*/

			error = INVALID_ARGUMENTS;
			break;
		}
		/* Initiate the DMA cycle */
		error = InitiateTheDMA(physical,phylen);
		/*
		** In this very simple application, we do not
		** multi-thread. In the ideal case, we would be
		** free here to initiate the I/O of other
		** devices.
		*/
		WaitForDMACompletion();

		/* Terminate the DMA */
		if (error)	break;

		/* and now go for the next block */
		logical   += phylen;
		remaining -= phylen;

		/* we *MUST* set this flag, though */
		flags     |= DMA_Continue;
	}

	/* And finally, only once, the following must be called... */
	CachePostDMA(base,len,iflags);
	/* but with the initial parameters */

	return error;
}
```

Complicated Memory Access. DMA is a touchy business, even more on the more advanced members of the MC68K family which implement a copyback cache. Therefore, please study the example above *carefully*. Especially consider situations where one continuous block of logical memory does not refer to one continuous block of physical memory, i.e. the memory model is fragmented. Failing to call these two functions, or **DMAInitiate()** due to "speed reasons" is a very poor excuse and might fail for more advanced applications of the "MMU Majik".

Maybe some background information about the cache problems should be provided: Even though it seems to be perfectly sufficient to flush the caches before the DMA transfer to make sure the DMA device reads proper data, *it is not.* This is due to the way how the 68040 and 68060 caches work: They do not buffer single bytes, but complete "cache lines" which are 16 bytes in a row, aligned to 16 byte boundaries. If the CPU reads from RAM, it typically fills the full cache line, hence loads 16 bytes instead of just the requested size; and if the CPU has to perform a write access, it first *reads* a full cache line from memory.

Consider now the following situation: The I/O buffer to be read from an external DMA device is not aligned to a 16 byte boundary. Caches are flushed, initially, but before the DMA transfer is initiated, another program writes to a memory location directly in front of the I/O buffer. The cache line for the written byte overlaps now with the I/O buffer, and as the CPU reads and writes full cache lines at once, it will be filled with data from the I/O buffer. Let's suppose the DMA operation completes successfully, hence the I/O buffer is really filled with the data read from the DMA device. However, if the DMA driver code would now run a cache flush, *the first bytes of the I/O buffer would be overwritten with obsolete data.* This is because the first bytes of the I/O buffer are located in a cache line which has been filled before the device started reading, by a memory access which did not even go into the I/O buffer. Therefore, *copyback caching has to be disabled for the pages at the boundary of the I/O buffer if the buffer is not aligned to cache lines.* This is what **CachePreDMA()** and **CachePostDMA()** perform.

A clever DMA device could avoid these problems by transfering the initial and final bytes of a non-aligned I/O buffer by means of programmed I/O, or by copying the initial and final page to a private memory buffer and perform the DMA from there.

5.4 Function Reference

We conclude with the function reference for this chapter:

Table 4: DMA Support Functions

MuLib function	Description
PhysicalLocation()	Translate a logical memory block to physical
PhysicalPageLocation()	Low-level translate a logical address
DMAInitiate()	Initiate a DMA transfer
DMATermiate()	Terminate a DMA transfer
Exec function	**Description**
CachePreDMA()	Cache handling before DMA start
CachePostDMA()	Cache handling after DMA completed

6 MMU Exception Handling

The MuLib offers various sources for exceptions: First, native MMU exceptions, generated by page faults or segmentation violations. Second, replacement functions for the **tc_Switch()** and **tc_Launch()** function pointers in the Task structure which are no longer available if the Task has been attached to a Context explicitly, because the MuLib requires these Exec vectors itself. Third, an exception handler which is called as soon as the high-level MuLib routines try to install a descriptor on the low-level. All exceptions are installed and removed in the same way, the interface is identical for all of them. The first step is to build an exception hook handle, and to attach it to a Context; this is done by the following routine:

```
struct ExceptionHook *hk;

        hk = AddContextHook(...);
```

The call is entirely tag-based, it does not take any "fixed" parameters. The following tag values are defined in `mmu/mmutags.h`:

MADTAG_TYPE Selects the type of the exception hook to be build. The types will be discussed in detail below, but to give a brief overview: **MMUEH_SWAPPED** handlers will be called in case the CPU accesses a page which has been marked as "swapped out", i.e. the **MAPP_SWAPPED** property flag is set.

The **MMUEH_SEGFAULT** handlers are invoked on an access to a **MAPP_INVALID** page, or on a write to a **MAPP_WRITEPROTECTED** page, as well as on a user mode access to a **MAPP_SUPERVISORONLY** page. They will *not* be called on a true hardware access error because it is not the purpose of the MuLib to handle these. In case you want to fetch true physical bus errors, you have to replace the default exec bus error handler using the **SetBusError()** function described below. Two types of segmentation fault handlers exist, global and Context specific handlers. The MuLib tries first to find a Context specific exception handler, and if this is not possible, it tries to run a global handler. If this fails, the default handler will be called, which will, unless replaced by **SetBusError()** — you guessed right — run into a "Guru". MuForce, for example, will install a global exception hook.

The **MMUEH_SWITCH** and **MMUEH_LAUNCH** handlers are task specific exception hooks which replace the **tc_Switch()** and **tc_Launch()** pointers of the Task structure which are no longer available for Tasks that have been explicitly attached to a Context.

Last but not least, the **MMUEH_PAGEACCESS** handlers are called by the MuLib as part of the **RebuildTree()** function when the library touches a low-level MMU descriptor with a set **MAPP_SINGLEPAGE** bit. It allows programs that operate on the low-level by **SetPageProperties()** and friends to get informed if their MMU setup is about to be overwritten. MuGuardianAngel makes use of this technique, for example.

More about the specific exception classes will be found below.

MADTAG_CONTEXT This selects the Context the exception is supposed to be attached to. A Context tag item is required for the page fault **MMUEH_SWAPPED** and the page access **MMUEH_PAGEACCESS** handlers. It can be left blank or set to **NULL** for the task-specific exceptions **MMUEH_SWITCH** and **MMUEH_LAUNCH**. One special case is the segmentation fault handler **MMUEH_SEGFAULT**: If no Context is given, a global segmentation fault handler will be installed which will be called from all Contexts provided no Context specific handler is able to handle the exception.

MADTAG_TASK This tag item selects an exclusive task for which the exception hook should be called. Since more than one task could be attached to a Context at a time, this tag item can be used to make a selection. However, adding too many task specific hooks to one Context may slow down exception handling considerably. Especially, do *not* add task specific hooks to supervisor Contexts. This tag item *must* be set for the **MMUEH_SWITCH** and **MMUEH_LAUNCH** handlers because this exception type is Task specific by its purpose; this tag is optional for all other types.

MADTAG_CODE This tag item *must* be given for all exception types because it points to the start address of the exception handler code. Assembly is, even though not required, highly recommended, to keep exception handling as fast as possible. All exception hooks are called in the same way:

Register a0 points to the **ExceptionData** structure for the **MMUEH_SWAPPED** or **MMUEH_SEGFAULT** handlers. It is loaded with a pointer to the **PageAccessData** structure for **MMUEH_PAGEACCESS** handlers. Both structures are explained below. Undefined for all other exceptions.

Register a1 loaded with the data you provided with the **MADTAG_DATA** tag item.

Register a4 for convenience for C style exceptions, this is loaded with the same data as register a1, namely the user specific data provided by **MADTAG_DATA**.

Register a5 points to your code itself and is available as scratch register otherwise.

Register a6 points to the library base of the MuLib. This is *definitely not a scratch register*.

Registers **d0**,**d1 a0**-**a1** and **a4** and **a5** are available as scratches, you may overwrite or modify them in your handler. In case your code was able to handle the exception, you *must* set the "Z" CPU condition code and clear register **d0** and exit with an **RTS**. The MuLib code will now test the "Z" bit directly for speed reasons. If your code bails out with a clear "Z" bit and register **d0** not equal zero, the MuLib assumes that your exception handler hasn't been able to handle the specific exception and will call the next handler with a lower priority of the same handler type. The lowest priority handler is always the default handler which will, in worst case, generate a guru.

MADTAG_DATA Handler specific data you may supply. This data will be loaded into the **a1** and **a4** registers before your code will be called.

MADTAG_NAME A name for the exception hook. Even though the name will be installed into the hook data, no further use is made of it, and currently can't be made of because the **ExceptionHook** structure is intentionally undocumented.

MADTAG_PRI The priority for the exception hook, defaults to 0. Higher priority handlers are called first, lower priority handlers later. Furthermore, for **MMUEH_SEGFAULT** handlers, the MuLib first tries to call Context specific handlers, and re-directs the exception to the global handlers in case no single Context handler accepted the fault.

The **AddContextHook()** function returns a handle to an *intentionally undocumented* **ExceptionHook** structure, or **NULL** in case it could not setup the hook structure, let it be due to missing data, or due to lack of memory. Keep the returned "hook" as a "magic cookie" for all further references.

If a Context hook as been build, it is not yet active. This gives you a chance to prepare and setup your program until the hook is really ready to be called. The final activation step is performed by

```
struct ExceptionHook *hk;

        ActivateException(hk);
```

Unlike **AddContextHook()**, this function and its counterpart

```
struct ExceptionHook *hk;

        DeactivateException(hk);
```

are interrupt-callable. One should furthermore note that calls to **ActivateException()** and **DeactivateException()** do *not* nest. The latter, **DeactivateException()**, *absolutely must* be called before the context hook is removed again by

```
struct ExceptionHook *hk;

        RemContextHook(hk);
```

The **RemContextHook()** call will release the hook structure and all administration information required for the hook, and will unlink the hook from the system.

6.1 Page Fault and Segmentation Fault Handlers

In the following, the **MMUEH_SEGFAULT** and **MMUEH_SWAPPED** exceptions are discussed in detail. Both exceptions are generated by the MMU due to an invalid access to a page in memory. The latter indicates that an attempt was made to access a "swapped out" page, whereas the first usually indicates some kind of software fault. Exception handlers of the first kind would typically print some debug information, whereas the second type should be used to re-load the "swapped" data from disk. The **MMUEH_SWAPPED** exception handlers are *only* called in case the faulty access hit a **MAP_SWAPPED** page, all other MMU related exceptions generate a "segmentation violation" by means of the **MMUEH_SEGFAULT** handler. The exception handlers never see faults that are handled by the MuLib transparently, namely accesses to **AbsExecBase**, read accesses to valid memory in the zero page, accesses to **MAPP_BLANK** memory or write accesses to **MAPP_ROM** pages.

> *Not the Real Thing!* An emulation can never be as fast as the real thing. Therefore, *avoid* these accesses. Make a backup of **AbsExecBase** to a static variable instead. For C authors, this is just the matter of chosing the proper prototype file, contact the manufacturer of your compiler. Reads from the zero page chip memory no longer happen if you use a recent Os release, 3.0 or better. Otherwise, use "MuMove4K" to move the chip memory out of this critical area.

Branch cache faults of the 68060 are also handled by the library and hence do not require any special care by your code. Furthermore, physical bus errors never reach the MuLib exception handling core as they are filtered out immediately. If you *must* handle them, you have to replace the exec bus error handler using the **SetBusError()** low-level function, but the MuLib will otherwise not be able to help you to handle these exceptions, you are on your own! Due to the tricky nature of these faults, especially for the 68040, and the rare use of them on the Amiga — I do not know a single one — I decided not to handle them in the library.

In case a MMU generated "access fault" has been detected the MuLib can't handle itself, it first checks the origin of this exception. This means it checks whether the fault is due to a "swapped" page, or due to a software fault. It determinates whether the access was made in user or supervisor mode and loads the Context responsible for this access. It then tries to find an exception handler:

- Try the handler of highest priority first. Try Context specific handlers first, if no Context specific handler can be found, or none of these handlers is able to handle the fault, go for the global handlers.

- Check whether this hook is activated or not, skip it if it is deactivated.

- Check whether the hook is "task specific", i.e. whether the **MADTAG_TASK** tag was given at the time the hook was build with **AddContextHook()**. If the hook is task specific, check whether the faulty task is identical to the task handled by the hook and skip the handler if this is not the case.
- As soon as a match has been found, call the routine at the address provided by the **MADTAG_CODE** tag by loading the registers as listed above. This routine should be written in assembly, mainly for speed reasons.
- If the handler returns with a set **Z** processor bit, return to the MuLib handling code and proceed as indicated in the "ExceptionData" structure described below.
- If the handler returns with the **Z** bit cleared, continue the search. If no useable handler could be found, run into the Exec default code.

In case you want make use of the exception data structure — and you usually do — please keep in mind that it is only valid as long as your exception hook runs. This means that you possibly have to make a copy of it for later use. Remember, too, that you're called *in supervisor mode*, with *all interrupts disabled*. Furthermore, one *very* important point:

> *AbsExecBase is a Big No-No.* Do never ever access **AbsExecBase** in MMU exception code. Even though this access might be tolerated under some conditions, it's the death of proper exception handling. Furthermore, if your code itself generates an exception, the MuLib *will* call the Exec default handler and will not enter your handler recursively. This means, it will "go guru". Quite the same goes for write accesses into **MAPP_ROM** memory, or accesses into the zero page the library would have to emulate.

If you need **SysBase**, use either a private copy, or the copy provided in the **ExceptionData** structure. If you want to make sure whether a specific memory access goes to valid memory, in order to avoid a double access fault, use the *low level* functions like **GetPageProperties()** as they are safe to be called from within interrupts and exceptions.

> *The MuLib's Matter.* There is *no access* to the CPU specific exception stack frame. Exception hooks are — and that's just the point about the MuLib — CPU independent. All data you need is provided in the exception data structure, do *not make any assumptions* about the type of CPU your code runs at, or how the stack frame looks like. This is the matter of the library.

Finally, we discuss the **ExceptionData** structure, defined in `mmu/exceptions.h`. It provides all the information required to handle the exception in a CPU independent way.

```
struct ExceptionData {
        struct  Task           *exd_Task;
        struct MMUContext      *exd_Context;
        ULONG                  *exd_Descriptor;
        ULONG                  *exd_NextDescriptor;
        APTR                    exd_FaultAddress;
        APTR                    exd_NextFaultAddress;
        ULONG                   exd_UserData;
        ULONG                   exd_NextUserData;
        ULONG                   exd_Data;
        APTR                    exd_ReturnPC;
        ULONG                   exd_Flags;
        UBYTE                   exd_internal;
        ULONG                   exd_Properties;
```

```
        ULONG                   exd_NextProperties;
        UBYTE                   exd_FunctionCode;
        BYTE                    exd_Level;
        BYTE                    exd_NextLevel;
        ULONG                   exd_DataRegs[8];
        ULONG                   exd_AddrRegs[7];
        UWORD                  *exd_SSP;
        UWORD                  *exd_USP;
        struct ExecBase        *exd_SysBase;
        struct MMUBase         *exd_MMUBase;
};
```

Let's now discuss the meaning of the components in this structure:

exd_Task contains the pointer to the Task structure of the task that caused the exception. If this hook was added to the supervisor Context, this field is meaningless and must be left alone. This is simply because the library does not distinguish between a supervisor exception in interrupt code, or in supervisor code called from a task.

exd_Context contains the handle of the Context responsible for the fault. This is always the Context the hook was added to in case this is a Context specific hook.

exd_Descriptor contains a pointer to the true hardware MMU descriptor which handles the fault. This pointer should usually be left alone. In case an indirect descriptor caused the access fault, this does *not* point to the page descriptor, but to the indirect descriptor pointing to the page descriptor, i.e. "one level up" to what you might expect. In case you *must* read this descriptor or write a new one, you *absolutely must* call **CacheClearE()** to make sure that the descriptor is really written out.

exd_NextDescriptor In case of a misaligned access, i.e. an access that spawns two pages because the access hit a page boundary, this is the descriptor for the end address of the access, and **exd_Descriptor** is the descriptor for the first address of the access. If the access is aligned, both pointers are identicals. Note, however, that this descriptor *need not to point to a higher address*. For example, a **movem.l regs,-(ax)** could cause a "backwards" misaligned exception. Additionally, read the warnings above about reading and writing of descriptors.

exd_FaultAddress the start address of the access that faulted.

exd_NextFaultAddress The end address of the access that faulted, inclusive. For a long word access, this would be **exd_FaultAddress+3**, for a byte access, this would be indentical to **exd_FaultAddress**. Note that it may well be that **exd_NextFaultAddress** is a *smaller* address than **exd_FaultAdress**. This might, for example, happen for a **movem** with pre-decrement addressing mode, i.e. **movem.l d0-d7/a0-a6,-(a7)**. Exception handlers must be aware of this special situation.

exd_UserData The data provided by **MAPTAG_USERDATA** or **MAPTAG_BLOCKID** for invalid and swapped pages, as defined by **SetProperties()** or **SetPageProperties()**. This will be **NULL** in case no user data is available.

exd_NextUserData In case of a misaligned access, the user data of the second page that was involved in the exception. If the access was aligned, this is identical to **exd_UserData**.

exd_Data If the access was a write access and the **EXDF_WRITEDATAUNKNOWN** flag in the **exd_Flags** field is cleared, then this long word will contain the data the CPU tried to write out. You've to ask the library to provide this data, hence to keep this bit cleared by selecting the **MAPP_REPAIRABLE** property flag. If you do not set **MAPP_REPAIRABLE**, you *might* get data in some situations on some CPUs, but I don't guarantee anything.

The data is right-justified in this field, i.e. bytes will use bits 7...0, words will use bits 15...0 and long word accesses will use the complete field. The library does *not* provide meaningful write data for **double**, **extended**, **movem** or **move16** accesses as this is definitely an advanced feature. Especially, you may see movem faults as a series of long word or word writes on some CPUs, while this instruction is atomic and reported as one big access on other CPUs.

This data should be used only for debugging purposes and "Enforcer like" applications, and for applications where you can guarantee that movems do not occur, i.e. emulators that emulate "computer hardware" by the MMU and might read the data written to the emulated hardware right here.

The size of the access is available by calculating the difference of the **exd_FaultAddress** and the **exd_NextFaultAddress** fields, but this size may have a sign, i.e. will be negative in some cases.

In case you want to allow the CPU to continue execution without retrying a write access, for example because you either managed to complete the write yourself or you want to ignore the access, set the **EXDF_WRITECOMPLETE** bit of the flags field below.

In case you do not set this bit, the CPU will attempt to re-run the faulty instruction. In case you haven't been able to repair the fault using some other techinque, as for example by swapping in the faulty page, your exception handler will be called again.

In case of a read fault, you may place the data to be read back in this field and set the **EXDF_READBACK** flag in the flags value below. The MMU library will then attempt to place this value in the input pipeline of the CPU and to repair the faulty access by providing this dummy data. The data has to be placed right-justified in this field again, and the **MAPP_REPAIRABLE** bit of the page must have been set in order to make this working.

Certain restrictions arise, again: First, a **movem** might show up as several exceptions, hence you might be able to provide data for each register loaded, or it might show up as one single exception. In this case, all registers will be loaded with the same value. It was hard enough to emulate this, so please don't complain...

Instruction faults can't be repaired like this, hence you will not be able to fill the instruction pipeline of the CPU. The only legal way of handling this fault is to provide a spare page or to alter the return PC. Do not try to fill the instruction pipeline, the MuLib will call the Exec exception handler directly in case you try to.

exd_ReturnPC The program counter of the instruction that caused the fault. This is only an approximate value due to the instruction prefetch feature of all CPUs of the 680x0 family. In case a branch was performed after the faulty instruction was loaded, and before it was detected, this PC might be completely useless because program flow after detection of the exception might have taken the branch, causing the PC to be in a completely different part of the program. There's nothing that can be done about it. However, the program is guaranteed to continue "at the right place".

In case you set the **EXDF_CALL** flag, this field should be filled with the address of a small procedure which will be called in user mode, in the Context of the Task that caused the exception. This routine could, for example, send a message to a "daemon" in order to repair the fault, and could run into a **WaitPort()** to wait for the daemon to repair the fault. Additionally, the user code might try to repair the access itself. It has access to all registers of the faulty program, and must therefore preserve all registers unless it attempts to alter them "on purpose". The exception data structure *is no longer available* if your code has been called.

In case you need it, you have to make a backup in the exception handler to a private memory region, and use this backup in your user routine. *This feature is not available for supervisor Contexts.*

Alternatively, use the "message hook" mechanism of the MuLib which provides all this for you already if you don't want to get a headache. Message hooks are described below.

The user routine should then return with an **RTS** to the library code — note that the stacked PC points to some code in the library, not to the faulty code itself, it's truely deep magic to restore all this information — which will then rerun the faulty instruction.

If you haven't been able to repair the cause of the fault, the exception hook will be called again. This means specifically that the exception mechanism will loop forever if you can't repair the access.

exd_Flags A combined input/output flags long word. Amongst private flags you'd better not care about, the following input flags are available:

EXDF_WRITE a write fault if set. If reset, a read fault.

EXDF_INSTRUCTION a fault on fetching an instruction. As a special case, this *could* be a write fault if someone tried to write out instruction data with an instruction function code using the **moves** instruction. This is currently unsupported by the library and should not be tried.

EXDF_WRITEPROTECTED failed due to write protection of the destination.

EXDF_SUPERVISOR failed due to supervisor only protection of the source/destination.

EXDF_WRITEDATAUNKNOWN the write data was lost, **exd_Data** is invalid in this case. This might happen in case a write access hit a non-**MAPP_REPAIRABLE** page. However, you should only set this property flag in case *you really really mean it*. Providing the write data might cause a lot of overhead for the MuLib code on some configurations.

EXDF_MISALIGNED The access was misaligned, i.e. more than one page was involved in the access. Prepare to swap in more than one page, for example. If this bit is set, **exd_FaultAddress** and **exd_NextFaultAddress** should be investigated closely to find the pages that must be repaired.

The following flags can be set by your code to let the library know what to do about the fault:

EXDF_READBACK abort a faulty read and provide the **exd_Data** word as input for the CPU. Do not try to rerun the access. This requires a lot of trickery for certain configurations and is not fast at all.

EXDF_WRITECOMPLETE abort a faulty write, and do not try to rerun it. This means that your code somehow managed to complete the write cycle, or that your handler found that it is no longer necessary to complete the cycle. MuForce, for example, sets this flag after having generated the debug output to let the faulty program continue.

EXDF_CALL call a routine whose address is in **exd_ReturnPC**. The procedure will be called in user mode as part of the faulty Task and its Context, as a "subroutine" of the MuLib code. It might be used to sent out a message and wait for its reply to get the missing page fixed by another task. This feature is not available for the supervisor Context, the MuLib will cause a "guru" in case you try. Note that the library does not check for you whether a **Wait()** is useful or even legal, as for example if the faulty task is in **Forbid()** state. It is the matter of your exception hook to check this. The MuLib "message hooks" will reject the exceptions in this case.

exd_Properties The property flags of the page that was responsible for the fault.

exd_NextProperties In case of a misaligned access, the flags of the second page involved in the access. For non-misaligned accesses, this is identical to **exd_Properties**.

exd_internal Leave this alone, it's for internal use of the library.

exd_FunctionCode The function code of the access. The following values are defined for user Contexts:

1 User data access

2 User code access

... and the following for Supervisor Contexts:

5 Supervisor data access

6 Supervisor code access

All remaining function codes relate to physical bus errors and should not be passed thru to your context hook.

exd_Level The level of the MMU tree at which the access fault happend and at which the MMU descriptor resides. This *need not* to be the "page level" for early termination descriptors or invalid descriptors at a higher level. Experts should note that this is not even guaranteed for the 040 or 060! Furthermore, in case of an indirect descriptor, this is the level of the pointer pointing to the final page descriptor, not the level of the real page descriptor.

This is advanced information, you usually should not care about. Level A of the MMU tree is encoded as "0", level B as "1" and so on. Note that this numbering is different from the MC680x0 internal counting which encodes level A as "1".

exd_NextLevel In case of a misaligned access, the level of the second descriptor involved in the fault.

exd_DataRegs A copy of the data registers at the time the fault happened. These images are copied back as soon as the exception terminates, hence a "higher magic" exception handler could alter these...

exd_AddrRegs A copy of the adress registers, but without the stack pointer.

exd_SSP The supervisor stack pointer. This points directly to the processor specific exception stack frame. The first **UWORD** of this stack frame is the copy of the status register at the time of the fault. *Be warned!* Everything else is processor specific, the MuLib might also want to modify this exception stack frame, so hands off!

exd_USP The user stack pointer. Some higher magic could even replace temporarely the USP in the exception handler and run a user routine with a private stack, for example to handle automatic stack enlargement. The MuLib handler will restore the correct USP from this field as soon as it returns.

exd_SysBase A pointer to the library base of the `exec.library`. Do *not* access **AbsExecBase** within the exception handler, use either a private copy or this pointer. *Not following this rule might be fatal.*

exd_MMUBase A pointer to the library base of the `mmu.library`, also found in register **a6**.

In case you're going to write an automatic stack extension program, you should keep in mind that the **EXDF_CALL** mechanism requires about 300 bytes of user stack space. Similar, task switching takes some user stack, too. To be able to swap in stack in case of a stack overflow, you need to provide an alternative user stack, for example by setting the **USP** to a temporary preallocated stack. The library "message hooks" discussed below will handle this automatically and are therefore "safe" for handling this situation.

For the same reason, you might have to add "Switch" or "Launch" handlers or might have to replace the exec scheduler to handle this situation. The MuLib provides an alternative scheduler, but I do not yet want to document how to install it.

There are unfortunately a number of "features" of each CPU which should not go unmentioned:

The 68020/68851 and 68030 Execution of instructions in a access-protected "zero page" is really, really slow. Please keep this in mind! Furthermore, **moves** to instruction space is unsupported. This goes for all other CPUs as well.

FPU related access faults are a dark chapter in exception handling. Of a faulty **fmove**, only the first two long words are checked by the 68030 and therefore at most two hits get reported, all others go unnoticed and will cause a second exception if the instruction is re-run. Of a faulty **fmovem**, only the first two long words hits get reported, too. The CPU might ignore all other hits and continue execution. This means specifically that a debugging tool might not be able to see all hits that happened on these instructions, and you will not be able to capture them for private use. VM systems shouldn't have much problems with this, though, since the invalid page should have been swapped in and each access can hit at most two pages at once if it is misaligned. However, you must be prepared that the mmu.library will not be able to set the **EXDF_MISALIGNED** flag correctly in this case. The instruction handler will be called again, though, in case the first hit did not swap in an adjacent page which was hit by the third or later long word.

The 68040 Repairing of **movem** write accesses is not always available, and certain **movem**s can't be continued safely. In case a **movem** read or write goes to an invalid page other than at level 2 or 3, the MuLib has to abort the movem completely by the "tough way". It won't "guru", though, but you won't be able to get the data that was supposed to be written out. There's nothing I can do against it. To avoid this situation in case you really need the movem data, set the **MAPP_REPAIRABLE** property flag since this will put all descriptors at page level.

Movems to a register included in the register list itself might not to be rerun safely in all circumstances. Avoid them. Readback data is only available for all destination registers at once, you can't specify individual pipeline data for single registers yet.

FPU related access errors cause even more problems than on the 68030. If a **fmove** or **fmovem** moves more than two long words, for example a **fmove.x** or any **fmovem**, the processor will only notice the first access of a faulty page. If the instruction would catch more than one hit, the processor restarts the instruction completely from the beginning. This means for debugging tools that you will *not* be able to fix a faulty instruction of this kind at all by just swapping in a single page at once, the processor would loop forever. The MuLib "fixes" this for **MAPP_REPAIRABLE** pages by replacing all invalid pages by blank dummy pages at once. Note that even disassembling the instruction at the **PC** would not help here because the **PC** might point to an FPU instruction even if the hit was caused by the instruction before. Hence, the following two instructions

```
move.l d0,(a3)
fmovem.x fp0,(a3)
```

with a3 pointing to an invalid page, will both invoke the exception handler with the same PC value, namely the address of the **fmovem**. This is the "fault" of the writeback pipeline of the 68040.

This might be a problem for VMM systems as well: Because there's no single flag that tells whether the hit was caused by an FPU instruction or not, the processor might re-run a FPU instruction as mentioned above again, possibly accessing the two different pages. If the VMM system would have swapped in the second page by the first hit, and removed the first page accessed by this instruction from memory, re-running this instruction will cause another VMM hit, this time from the first page. It is therefore important that a VMM system *must not* swap out a page previously swapped in. At least two adjacent pages must be available at once.

The 68040 has one additional user instruction, **move16**. Due to the tricky nature of this instruction, the library might not be able to handle it correctly, even though I tried my best. As for **movem**s, individiual pipeline access to the data written out or for readback data is not available. Especially, tricky situations may arise if a **move16** accesses hardware registers that do not tolerate double writes. The MuLib can't avoid this case under all circumstances, i.e. double writes may happen in some situations. Since the Amiga hardware does not support **move16** correctly under all circumstances anyhow — since this instruction causes a "burst access" even to non-cacheable memory — it is *highly "recommanded"* not to use it.

The 68060 The library handles branch cache flushes transparently, no need to worry about them. So much for the good news.

Written data is usually not available unless the library emulates this with some heavy magic and is told to do so with the **MAPP_REPAIRABLE** flag. Things are getting even worse if no exception handler wants to make use of the writeback data and returns with the **EXDF_WRITECOMPLETE** flag cleared. In this case, the MuLib tries to rerun the faulty instruction again, even though it has been already executed a first time. It tries to rebuild all the registers except for the FPU registers, the supervisor model and the SSP and then returns to the PC of the faulty instruction. Whether this mechanism is reliable or not has yet to be proven. Avoid this situation.

Except that, all 68040 restrictions regarding **movem**s or **move16** apply here again, see above.

In case of a non-locked read-modify-write access, as for example a

```
addq.l #1,(a0)
```

the library will call the context hooks twice, once with the read data, and then again with the write data. This is an emulation service to guarantee consistency amongst all members of the MC68K family.

However, things are getting rather nasty in case of a misaligned RMW access of which one part of the access is valid and the other part is not. The 060 manual states that these accesses can't be rerun safely because parts of the data might have been written back already, and rerunning the instruction might cause this wrong data to be re-used again. I can't comment on this feature because I haven't tried to reproduce it yet, but it may reduce reliability of the 68060 for virtual memory applications.

As for all other members of the MC68K family, FPU accesses are somewhat critical. This is even worse for the FPU instructions that have to be emulated in software. Unlike the 68040, the 68060 does not prefetch the full instruction before running the emulation exception, which means for you that some exceptions might show up in the FPU emulation core, as supervisor mode exceptions, in the 68060 library. It would be the job of this library to handle them properly, and to emulate a user mode exception, for example to swap in the full instruction before proceeding. The original Motorola FPSP code handled this correctly, but this code has been “optimized” unfortunately by various, if not all authors of 68060.libraries such that *there is currently not a single 68060.library that would support virtual memory correctly.* Sorry, this isn’t my fault, and as I do not own a 68060 based system, there is nothing I can do against it.

6.2 Page Access Handlers

The purpose of the page access handler is to notify programs that operate on the low-level side of the MMU programming — namely by **SetPageProperties()** or **GetPageProperties()** — if the high-level functions **RebuildTree()** resp. **RebuildTrees()** try to access or overwrite their setup. It provides a method to modify the MMU tree on the low level transparently to the high level functions. Thus, the low-level program will be able to protect its setup from getting overwritten by a high-level function. Unlike the “segmentation fault” handlers, the page access handler is not a true hardware exception, but the calling rules and restrictions are the same. It is setup by

```
struct MMUContext    *ctx;
struct ExceptionHook *hk;

        hk = AddContextHook(MADTAG_CONTEXT,ctx,
                            MADTAG_TYPE,MMUEH_PAGEACCESS,...);
```

similar to all other exception handlers.

The page access handlers are called whenever **RebuildTree()** tries to reinstall a page descriptor with the **MAPP_SINGLEPAGE** property flag set. If so, your handler is allowed to modify this page descriptor. However, you should remember that the high-level functions are not aware of this modification, so this feature should be used carefully.

A typical application of this feature is the “MuGuardianAngel” type program: The idea is here to mark “free” memory as **MAPP_INVALID** to be able to detect illegal memory accesses. Since an **AllocMem()** has to mark the memory valid it allocated, the MMU tables have to be adjusted for each allocation. Similary, released memory has to be made unavailable on a **FreeMem()**. On the other hand, **AllocMem()** and **FreeMem()** are not allowed to break a **Forbid()** and are hence not able to call the high-level functions — which might do so. They therefore *have* to use the low level **SetPageProperties()**. Since the low level functions do not inform the higher level about the modification, a program adjusting the MMU descriptor with the high-level functions will revert the modifications made by **SetPageProperties()** before. The solution is here to install a page access handler that checks whether the higher level is going to re-install one of the modified page descriptors, to intercept here and to set the **MAPP_INVALID** flag correctly.

The page access handler could either try to generate the new page data itself, for example by using its own database or by investigating system structures; the MuGuardianAngle Page Access Handler scans, for example, the Exec memory lists to find out whether the memory page is “free” or not, and hence to decide whether it should or should not invalidate the page. Alternatively, it would also be permissible to call **GetPageProperties()** to read the old property flags that have been installed before and that are about to be overwritten. In both cases, though, you should be careful to modify only the flags your program really requires or the resulting page layout might be unusable and unexpected to the application that triggered the high-level page rebuild.

Your handler is called with the page access data structure, documented below, in register **a0**, with the provided user data from **MADTAD_DATA** in **a1** and **a4** and the library pointer in **a6**. The calling rules are therefore *identical* to those of all other exception handlers. Registers **d0-d1/a0-a1/a4-a5** are scratch registers, **a6** *is not.* In case your handler returns with the **Z** flag set, the library aborts calling other handlers of lower priority. In case the **Z** flag is found reset, the MuLib will call the next available handler; this is for example desirable if the page to be modified is not one of the pages your handler keeps care of.

Remember that you're called in supervisor mode with all interrupts disabled, as for all other exception handlers. Your code should better be fast, especially if a lot of descriptors have to be rebuild.

Here's the **PageAccessData** structure:

```
struct PageAccessData {
        ULONG                    pgad_Level;
        void                    *pgad_Address;
        ULONG                   *pgad_Destination;
        struct MMUContext       *pgad_Context;
        struct MMUBase          *pgad_MMUBase;
        ULONG                    pgad_Properties;
        ULONG                    pgad_UserData;
};
```

The meaning of the fields in this structure is as follows:

pgad_Level The level at which the MMU descriptor has to be build. This is zero for level A, one for level B and so on. Note that this numbering is different from the internal MC68K scheme which counts level A as one. Since this handler is only called for **MAPP_SINGLEPAGE** descriptors, this will always be the page level of the MMU tree. However, what the page level *is* is up to the depth of the MMU tree and may vary, even though this number is constant for the lifetime of a specific Context.

pgad_Address The logical address the descriptor to be build will handle. Hence, this gives the address that is to be translated or mapped. Your handler should check this field to determinate whether it is responsible for the page that is about to be installed.

pgad_Destination The address the true hardware descriptor will be written to. There's little reason to make use of this entry. Especially, it might not yet be initialized when your code is run, and will be filled in after your code has run.

pgad_Context The Context the MMU tree being rebuild belongs to. This is the Context you installed the page access handler in. Unlike for **MMUEH_SEGFAULT** handlers, there is no global handler list, only Context specific handlers are available.

pgad_MMUBase The library base address.

pgad_Properties The page properties, defined in `mmu/context.h`. This defines which kind of hardware descriptor the library is about to build. In case you want to alter the settings, you need to place the desired new properties of the page here.

pgad_UserData This field will be used if one of the property flags **MAPP_SWAPPED** or **MAPP_INVALID**, **MAPP_REMAPPED** and **MAPP_BUNDLED** or, last but not least, **MAPP_INDIRECT** have been enabled.

It contains property specific information like the block ID for **MAPP_SWAPPED**, the user data for **MAPP_INVALID** pages, the physical destination for **MAPP_REMAPPED** and the physical page address for **MAPP_BUNDLED** pages, and finally the destination descriptor for **MAPP_INDIRECT** pages. It is not available if the **MAPP_REPAIRABLE** bit is set. In case you modify the properties, you possibly need to correct this field, too.

6.3 Switch and Launch Handlers

Since the **tc_Switch()** and **tc_Launch()** fields of the Exec Task structure are no longer available if a Task entered a Context explicitly by means of **EnterMMUContext()**, the MuLib has to offer replacement functions. They are, additionally, more flexible than the Exec function pointers since more than one handler can be installed at a time. Very much like the page access exception, these exceptions are not caused by hardware exceptions, but are called as soon as a Task looses the CPU — for **MMUEH_SWITCH** — or gains the CPU — for **MMUEH_LAUNCH**. Since both exceptions *require* that the MuLib specific switch and launch vectors have been installed in the Task structure itself, your Task has to enter an MMU Context explicitly to make this feature available:

```
struct MMUContext *ctx;
struct Task *task;

        task = FindTask(NULL);
        ctx  = CurrentContext(task);
        if (EnterMMUContext(ctx,task)) {
                /* everything is fine */
        }
```

This will attach the current task explicitly to the public Context if it hasn't been attached to a Context before. Additionally, the Task must be removed from this Context before it quits. This could by done by

```
        LeaveMMUContext(FindTask(NULL));

        /* and say goodbye! */
}
```

before running into the final **RTS**. As for all exceptions, the **AddContextHook()** function will install a new exception handler, but this time the **MADTAG_TASK** tag item *must* be supplied since swap and launch handlers are intrinsically task specific:

```
struct Task          *task;
struct ExceptionHook *hk;

        hk = AddContextHook(MADTAG_TASK,task,
                            MADTAG_TYPE,MMUEH_SWITCH,...);
```

Finally, the hook must be activated by **ActivateException()**. The **MMUEH_SWITCH** hook will be called in supervisor code, immediately before your task looses the CPU. The task specific user and supervisor Contexts will still be active, though. The **MMUEH_LAUNCH** handler will be called right before your task will gain the CPU, with the MMU setup for this task already installed. Calling rules and register usage are as above, and as for all exception hooks.

6.4 Message Hooks

Even though the MuLib exception hooks are very powerful, they are a bit cumbersome to use, especially from high-level languages. Therefore, the MuLib offers for the **MMUEH_SWAPPED** and **MMUEH_SEGFAULT** exceptions another software wrapper which makes exception handling much easier. This wrapper is build on top of the **AddContextHook()** functions, and was designed with "virtual memory management" applications in mind: On installation, you specify a Context and a message port. Then activate the exception. As soon as the MMU detects a page fault, a message will be sent to the provided port, and the faulty task will be halted. Your application has then the chance to fix this problem, for example by swapping in the missing page. As soon as you reply to the message, the faulted instruction will be re-run.

However, due to this mechanism, a task switch must be performed. Since this should not happen if the faulting task is in "Forbid()" or "Disable()" state, the message hooks will reject any faults that happen while the executing task is in one of these states. The fault will then be delivered to the handlers of the next lower priority, or if no other handler is available, will be forwarded to Exec, which in turn would "guru". This is not a restriction of the MuLib, but due to the very design of the Exec kernal.

Message hooks *require* that the tasks whose faults are about to be handled are attached explicitly to a Context, thus message hooks won't work automatically for all public Tasks. They will reject to handle exceptions from Tasks that haven't been attached to any Context, even though these Tasks use the MMU setup of the public Context. The following code would, for example, attach the current Task to the public Context, and hence would allow to handle exceptions of the current Task by means of message hooks:

```
struct MMUContext *ctx;
struct Task *task;

        task = FindTask(NULL);
        ctx  = CurrentContext(task);
        if (EnterMMUContext(ctx,task)) {
                /* everything is fine */
        }
```

Don't forget to **LeaveMMUContext()** when shutting down.

Message hooks are installed and initialized by

```
struct MMUContext *ctx;
struct MsgPort *port;
struct ExceptionHook *hk;

        hk = AddMessageHook(MADTAG_CONTEXT,ctx,
                            MADTAG_CATCHERPORT,port,
                            MADTAG_TYPE,...);
```

The available tag items are defined in `mmu/mmutags.h` and are identical to those of the low-level **AddContextHook()** call, except that **MADTAG_USERDATA** is not available and a destination port for the exception message must be provided by **MADTAG_CATCHERPORT**. Especially, the following tags are available:

MADTAG_CONTEXT The context to which this exception hook should be added. This tag item *must* be present.

MADTAG_TASK If the hook should be called only if a specific task is running, specify a pointer to the task structure here. Remember that more than one task can be attached to a Context. Be warned, though! Adding too many task specific hooks slows down exception handling considerably.

MADTAG_TYPE The type of the exceptions this hook should handle. Currently, only page faults **MMUEH_SWAPPED** and segmentation faults **MMUEH_SEGFAULT** can be handled by means of message hooks.

MADTAG_CATCHERPORT The port to sent the exception message to. Must point to an initialized exec **MsgPort** structure. This tag item is mandatory, or the function call will fail.

MADTAG_NAME A name for the hook. This name is initialized, but the library makes no use of it. Since the structure of the exception hook is undocumented, there is currently no way of using this name from the outside as well.

MADTAG_PRI A priority, ranging from $-128 \ldots +127$. The higher the priority, the earlier the hook will be called. Higher priority handlers are called first.

Then, as for all other exceptions, the message hook must be activated by calling

```
struct ExceptionHook *hk;

        hk = ActivateException();
```

If the handled task generates an exception, the following message will be sent to the supplied port, documented in `mmu/exceptions.h`:

```
        struct ExceptionMessage {
                struct Message          exm_msg;
                struct ExceptionData    exm_Data;
        };
```

The **ExceptionData** structure is documented above, it contains all the information required to handle the exception. Once you reply to this message, the task that caused the exception will be restarted and will re-run the access cycle.

To disable the exception, the following function should be used:

```
struct ExceptionHook *hk;

        DeactivateException(hk);
```

This function, similar to **ActivateException()** is interrupt-callable and will not break a "Forbid()" or a "Disable()", hence is even available in critical situations. It must be called before you remove the message hook completely with

```
struct ExceptionHook *hk;

        RemMessageHook(hk);
```

However, before you do so, you *must* have replied to all exception messages you received, or the MuLib might deadlock or crash. The following code segment presents a very simple way if only one message hook has been activated:

```
struct ExceptionHook *hk;
struct MsgPort *exceptionport;
struct Message *msg;

        DeactivateException(hk);
        while(msg=GetMsg(exceptionport))
                ReplyMsg(msg);
        RemMessageHook(hk);
```

Note the order of the calls, first disable the hook, then reply to the messages left on the port, then remove the hook.

6.5 Function Reference

To complete this chapter, we give an overview about the exception handling related functions in the MuLib:

Table 5: Exception Control Functions

MuLib function	Description
AddContextHook()	Allocate and install an exception hook
RemContextHook()	Unlink and release an exception hook
AddMessageHook()	Allocate and install a high level hook
RemMessageHook()	Unlink and release a high level hook
ActivateException()	Enable an exception hook
DeactivateException()	Disable an exception hook

7 Building and Adjusting Contexts

The MuLib allows to build several MMU configurations — the Contexts — at once, and to load and unload them automatically as soon as the Tasks attached to these Contexts gain or loose the CPU, as explained in the first chapter. Once the MuLib has been loaded, only two Contexts are available: The public Context, and its supervisor Context. More private Contexts can be build on demand, though. Tasks attached to these Contexts will somehow "detach" from the remaining system as they operate in their private address space. Even though the MuLib offers to use the public Context as a "template" for a private Context, there is, however, currently no mechanism to synchronize a private Context with the public Context, hence to forward some or all adjustments of the public Context to private Contexts. Therefore, tasks running in a private Context currently cannot profit from tools like "MuFastRom" which modify the public Context only.

7.1 Creating a New Context

The following MuLib call will build a new Context; unless you specify some tags to proceed otherwise, the function will create a *completely blank* context with no valid logical addresses at all. Hence, you most likely want to specify tag items to duplicate the public Context and use it as a starting point for further modifications:

```
struct MMUContext *ctx;

        ctx = CreateMMUContext(...);
```

As some other MuLib functions, it is completely tag-based. After the Context has been build, it is ready to accept Tasks for attachment.

The following tags can be passed in, defined in `mmu/mmutags.h`:

MCXTAG_COPY build a copy of the Context passed in as tag data. If this pointer is **NULL**, build a copy of the public user Context. This is highly recommended since if this tag item is not given, all context addresses will be marked as **MAPP_BLANK**. Most likely not what you want.

If you specify a different page size than that of the context you want to copy, you should specify the **MCXTAG_ERRORCODE** as well and study the return code carefully. The mmu.library might have performed some rounding to fit the old table specifications into the new table layout. In worst case, this will make your new table unusable. It is therefore in general not a good idea to specify a page size larger than that of the cloned Context, or even to select a different table size at all. Even though the library itself, and all the MuTools allocates all its internal structures aligned to "worst case" page sizes, this might not be true for external user programs. *(Create only)*

MCXTAG_EXECBASE allow accesses of **AbsExecBase** and of valid chip memory within the first page, even if the first page is marked as **MAPP_INVALID**. This is an important feature if the MuLib is used by debugging tools like "MuForce". These accesses will be emulated in software and are hence *slow*. They should be avoided for that reason. Os 3.0 and up takes this into account already and starts the chip memory pool at a higher address, but on all lower Os releases a tool like "MuMove4K" should be run.

AbsExecBase accesses are handled with highest priority first and might be faster, even faster than with the original "Enforcer". They are still slower than the real thing. It is not guaranteed that the library will really read **AbsExecBase** from the address `4.w`, dependent on some internals, it might give you a cached copy instead. This will usually do only good.

This option defaults to **TRUE**. *(Create, get and set)*

MCXTAG_BLANKFILL defines a **ULONG** fill value for blank memory regions.

This **ULONG** is read by the CPU in case a program tries to access **MAPP_BLANK** It defaults to **0L**, but other values might be useful for debugging. MuForce, for example, can be asked to install here some nastier values. *(Create, get and set)*

MCXTAG_MEMORYATTRS Exec memory attributes as defined in `exec/memory.h`. This memory attribute is used when allocating memory for the hardware MMU tables. This defaults to **MEMF_PUBLIC**, but can be set to other values for special purposes. *(Create and get)*

MCXTAG_PRIVATESUPER A boolean value, either **TRUE** or **FALSE**. If **TRUE**, the MuLib will build a private supervisor Context for your user Context as well, which is independent, but a copy of the public supervisor Context. This means specifically that possible modifications of the public supervisor Context will not be carried over to your private supervisor Context. The default is **FALSE**.

Even if you pass in **FALSE** here, the library might still ignore your choice and might decide to build a private supervisor Context anyways. This happens if the table layout you've chosen is different to the default table layout, making the user tree incompatible to the default supervisor tree. *(Create only)*

MCXTAG_ZEROBASE This option has only an effect if **MCXTAG_EXECBASE** is **TRUE** and the first page of the MMU table is marked as **MAPP_INVALID**; hence, if the MuLib has to emulate accesses into the first page.

This tag provides a base address which will be used as physical address by the software emulation. It defaults to **0L** meaning that the MuLib will emulate accesses into the true zero page. This is important if the zero page gets remapped to a different location in a first step, and an Enforcer type program disabling the access to the zero page is run later on. The MuLib must, in this case, know that it should emulate the accesses into the remapped copy of the zero page instead.

The zero page remapper should specify this tag to redirect accesses transparently, even if an Enforcer type application invalidates the zero page. Failing to do so would make the MuLib emulating the access to the incorrect, non-remapped memory location. Since other programs might want to build a private MMU table with a different table size, it is *not* enough to align the remap destination of the zeropage to **GetPageSize()** boundaries, **RemapSize()** alignment is required! Due to the tricky nature of memory remapping, this is clearly an advanced feature. *(Create, get and set)*

MCXTAG_DISCACHEDES A boolean tag item. If set to **TRUE**, the memory that keeps the descriptors is cache-inhibited. This works around some problems that appear if a program attempts to hack on the MMU tables itself. *Note that this is definitely illegal and unsupported anyways*, the MuLib code has no problems with descriptors in cacheable memory. Note, however, that the descriptors will be only non-cacheable "as seen" from the Context itselt. It will *not* change the cache mode as seen from other Contexts, even from the supervisor Context of the Context passed in.

This means that reading user descriptors from user code will not be cached, and supervisor code reading supervisor descriptors won't fill the cache either, but user code reading the supervisor tables or supervisor code reading the user tables will enter the cache as before. I will not try to improve this compatibility hack further! *(Create and get)*.

Defaults to **FALSE**, but the setting for the public user and supervisor Context can be adjusted by means of

```
DescriptorCacheInhibit ON
```

in the `ENVARC:MMU-Configuration` file.

Nevertheless, *hacking the MMU has to stop, I do not guarantee for anything.*

MCXTAG_LOWMEMORYLIMIT Define a boundary in the zero page such that accesses to addresses higher or equal to this boundary will be emulated in software. This is mainly for 68060/68040 support under Os V37 and V38 where chip memory started at 0x400 inside the zero page. The MuLib checks the chip memory base address on startup and provides this as default value. *(Create, get and set).*

MCXTAG_ERRORCODE Defines a pointer to a **ULONG** the MuLib will fill with an error or warning code. It will be set to **0L** in case the operation succeeds. *(Create only).* The following error codes are defined in `mmu/context.h`:

CCERR_NO_FREE_STORE The operation failed due to lack of memory.

CCERR_INVALID_PARAMETERS The parameters specified by the tags are invalid or out of range.

CCERR_UNSUPPORTED The parameters are valid, but not supported by the hardware the program currently runs on.

CCERR_TRIMMED The library performed some minor adjustments on the MMU table passed in for cloning. The cache modes might not be optimal due to some roundings that have to be performed, but the MMU table should work in general.

This is not an error, building the Context succeeded.

CCERR_UNALIGNED The library had to perform heavy rouding in the MMU table passed in, it might be unusable. For example, remapped pages were misaligned and due to the rounding accesses might go to wrong locations. If you get this return code, you should possibly deallocate the new Context and inform the user that the request could not be satisfied.

Still, *this is not an error.* The function will return a new Context, but possibly an unusable one.

The next tag items define the MMU table layout. A logical address, used as input for the MMU, consists of exactly 32 bits. These bits are now split from the left to the right into groups, defining a "path" in the MMU tree. Each "level" of the MMU tree should be considered as an array of pointers, pointing to the next lower level of the tree. The nodes of the tree contain the descriptors that define how the address belonging to the path from the root to this descriptor should be handled. For example, consider a three level tree with 7 bits for level A and B, 6 bits for level C and 12 bits for the "page level". The address `0x01feabcd` would be used like this to find a descriptor:

$$\texttt{0x01feabcd} = \underbrace{\overbrace{0000\ 000}^{7}}_{A}\underbrace{\overbrace{1\ 1111\ 11}^{7}}_{B}\underbrace{\overbrace{10\ 1010}^{6}}_{C}\ \underbrace{\overbrace{1011\ 1100\ 1101}^{12}}_{D}$$

The index into level A of the MMU tree is 0, hence the first pointer is read. The MMU obtains now another array of pointers, called the level B.

The index into level B is, as we see above, 127. The MMU uses now the 127th entry of the table at level B to obtain a pointer to the level C table.

The index into level C is here 101010, binary for 42. Hence, the 42th pointer of the level C table will be used, pointing to the page in memory and defining the base address for the next step.

The page offset, indicated by D is 1011 1100 1101 in this example, or short `0xbcd` in hex notation. This number is added to the base address obtained from the descriptor in level C. If the address is not "re-mapped", the base address would be identically to the first $32 - 12 = 20$ bits of the physical address.

Generic Page Formats are Cheaper. Except for special applications, it is usually not a good idea to build a Context with a different MMU table organization than that used by the MuLib. If you build a new Context which does not use the same page organization as the public Context, the MuLib has to build a private supervisor Context for you. Moreover, task switches between tasks using different table layouts are considerably slower because the MuLib has to load more MMU registers, and has to flush the CPU caches. The overhead is usually not worth the effort.

Nevertheless, now for the tag items defining the table layout. The first group of tag items specify the number of bits to be used for each level of the MMU tree. They *must* sum up to 32, as explained above. You need not to specify all of them, the MuLib will calculate reasonable defaults if you don't.

MCXTAG_DEPTH The depth of the MMU tree to build. Defaults to the depth of the public Context. In the example above, the depth is three, which is also the depth used by the MuLib on startup.

Legal values range from 1 . . . 4 for the 68020/68851 and the 68030, but the 68040 and 68060 support only one value, namely 3. *(Create and get).*

MCXTAG_LEVELABITS The number of bits of the logical address that make up the level A of the MMU tree. 2^{bits} is the number of entries in this level of the tree.

The 68020/68851 and the 68030 support here values from 1 . . . 15, the only legal value for the 68040 and 68060 is 7.

The MMU library will pick a reasonable and system dependent default for you if you don't specify this tag item. *(Create and get).*

MCXTAG_LEVELBBITS The number of bits for the level B of the MMU tree, unused if the depth is smaller than two.

Legal values are 1..15 for the 68851 and the 68030, and 7 as only possible value for the 68040 and 68060. *(Create and get).*

MCXTAG_LEVELCBITS The number of bits of the level C of the MMU tree, unused if the depth is smaller than three.

Arguments may range from 1 . . . 15 for the 68851 and 68030, and must be either 5 or 6 for the 68040 and 68060. *(Create and get).*

MCXTAG_LEVELDBITS The number of bits in the level D of the MMU tree, only used if the depth is four and therefore unused on the 68040 and 68060; must range from 1 . . . 15. *(Create and get).*

MCXTAG_PAGEBITS The number of bits to be used from the logical address as the page offset, therefore 2^{bits} will be the page size.

Legal values are 8 . . . 15, giving 256 bytes up to 32K pages for the 68020/68851 and 68030, or 12 . . . 13 defining 4K resp. 8K pages for the 68040 and 68060.

The default is the page size of the public Context. *(Create and get).*

The following program gives an example how to build a new Context and how to attach a new task to it:

```
/***********************************************
 ** MuContextTest                             **
 **                                           **
 ** Build a task with a private context       **
```

```
 **                                              **
 ** (c) 1999-2000 THOR-Software                  **
 ** Version 1.03        19.03.2000               **
 *************************************************/

/* Includes */
#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <dos/dostags.h>
#include <dos/dosextens.h>

/* MMU specific includes */
#include <mmu/mmutags.h>
#include <mmu/context.h>

#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/mmu.h>

#include <string.h>

/* Defines */

/* This is the location we will re-map accesses to. Should be
** available on all systems.
*/
#define TESTLOCATION 0x80000000

/* Protos */

long __saveds main(void);
void MMUTaskTest(void);
void RunTests(struct MMUContext *privctx,UBYTE *testpage,UBYTE *pother);
void Sync(struct MsgPort *destination,struct Message *msg);
void __saveds TestProc(void);

/* Statics */

/* Just the library bases we need */
char version[]="$VER: MuContextTest 1.03 (19.3.2000) (c) THOR";

struct MMUBase *MMUBase;
struct DosLibrary *DOSBase;
struct ExecBase *SysBase;

/* The main program */
long __saveds main(void)
{
long err,rc;
```

```
/* This program is able to compile without startup code, hence we have
   to setup ourselfs */

SysBase=*((struct ExecBase **)(4L));
rc=20;

/* open the required libraries */

if (DOSBase=(struct DosLibrary *)
                     OpenLibrary("dos.library",37)) {

        if (MMUBase=(struct MMUBase *)
                     OpenLibrary("mmu.library",40L)) {

                err=ERROR_REQUIRED_ARG_MISSING;

                /* Check for a valid MMU.
                ** The mmu.library will also
                ** open without!
                */

                if (!GetMMUType()) {
                        Printf("MuContextTest requires "
                               "a working MMU.\n");
                        err=10;
                } else {
                        /* Run the tests */
                        MMUTaskTest();
                        err=0;
                }

                /* Check for error codes. Everything
                ** below 64 is considered to be a custom
                ** error and passed thru as primary
                ** result code.
                */
                if (err<64) {
                        rc=err;
                        err=0;
                } else {
                        PrintFault(err,"MuContextTest failed");
                        rc=10;
                }
                SetIoErr(err);

                /* Shut down: Close libraries */
                CloseLibrary((struct Library *)MMUBase);
        } else PrintFault(ERROR_OBJECT_NOT_FOUND,"MuContextTest");
        CloseLibrary((struct Library *)DOSBase);
}
```

```
        return rc;
}

/* MMUTaskTest */
void MMUTaskTest(void)
{
struct MMUContext *ctx,*privctx;
UBYTE *testpage,*physical;
ULONG size,psize;
ULONG pother=TESTLOCATION;
ULONG error=0;

        /* This is the TRUE test, finally. */


        /* Get the public default context as template for the new
           context */
        Printf("Locating the default context...\n");
        ctx=DefaultContext();

        Printf("Building a new context...\n");
        if (privctx=CreateMMUContext(MCXTAG_COPY,ctx,
         /* make a copy of the already existing context */

         /* Just stay to plain 4K or 1K pages. In case
         ** your are in an experimental mood, remove
         ** these comments... (-:
         **                   MCXTAG_PAGEBITS,13,
         */

                              MCXTAG_ERRORCODE,&error,
         /* and deliver an error code */

                            TAG_DONE)) {

          /* I don't check here for an error, even though
          ** I should. The library will build the context,
          ** provided there is enough memory and the
          ** parameters are valid for the hardware, but
          ** "error" should be checked for problems the
          ** library found. This is only required if you
          ** tried to make a table setup different to the
          ** default - here the 8K pages. "error" should
          ** be checked for CCERR_UNALIGNED. In this case,
          ** the mmu.library had to round some descriptors
          ** heavely to be 8K aligned and the resulting
          ** page setup is most likely not what you want.
          ** For example, MAPP_REMAPPED pages have been
          ** trimmed, and the setup is therefore incorrect
          ** at the boundary.
          */
```

```
/* Find out the page size of this Context. */

size=GetPageSize(privctx);
Printf("Getting the new page size. "
       "It is 0x%lx bytes.\n",size);

/* allocate a test page */
testpage=AllocAligned(size,MEMF_PUBLIC,size);
if (testpage) {

 /* Find out the physical location of
 ** this page. Note that we use the
 ** public context since this is the
 ** context we're running in. The other
 ** context has not yet been loaded.
 */

 physical=testpage;
 psize=size;
 Printf("Allocating a test page.\n");
 PhysicalLocation(ctx,(void **)&physical,&psize);

 if (psize==size) {

  /* remap (mirror) it to pother. This is just
  ** for demonstrational purposes.
  */
  Printf("Mirroring the page at 0x%08lx "
         "(0x%08lx phys.) to 0x%08lx\n",
         testpage,physical,pother);

  if (SetProperties(privctx,MAPP_COPYBACK|MAPP_REMAPPED,
                           ~0,pother,size,
                            MAPTAG_DESTINATION,physical,
                            TAG_DONE)) {

   /* the above call modified only the software abstraction
   ** level. Now rebuild the MMU tree for the private
   ** context to reflect the changes
   */

   Printf("Building a new MMU tree for the "
          "private context...\n");
   if (RebuildTree(privctx)) {

    /* and run the test */

    RunTests(privctx,testpage,(UBYTE *)pother);

    /* all the rest is shutdown code */
```

```
          } else Printf("Can't rebuild the tree.\n");
         } else Printf("Failed to setup memory remapping.\n");
        } else Printf("Can't handle fragmented memory.\n");

        /* release the test page */

        Printf("Releasing the test page.\n");
        FreeMem(testpage,size);
       } else Printf("Failed to allocate a test page.\n");

       /* ... and the context */

       Printf("Releasing the private context.\n");
       DeleteMMUContext(privctx);

     } else Printf("Failed to build the MMUTaskTest.\n");

}

/* RunTests */
void RunTests(struct MMUContext *privctx,
              UBYTE *testpage,UBYTE *pother)
{
struct Process *proc;
struct Message *msg;
struct Task *testtask,*mytask;
struct MsgPort *testport;
int i;

        /* given the MMU context created above,
        ** create a new task
        ** and run it in this context
        */

        /* Build a message with our process port
        ** as reply port. I'm here to lazy to
        ** setup a message port since we already
        ** have one.
        */

        mytask=FindTask(NULL);
        Printf("Building a new IO request for the test.\n");
        msg=CreateIORequest(
                &(((struct Process *)mytask)->pr_MsgPort),
                sizeof(struct IORequest));

        if (msg) {

         /* build a new process. It will start in the
         ** default public context, but we will attach
```

```
** it to the private context as soon as it
** is set up.
*/

Printf("Creating a new task, in the public context.\n");

if (proc = CreateNewProcTags(   NP_Entry,&TestProc,
                                NP_CurrentDir,NULL,
                                NP_StackSize,512,
                                NP_Name,"MuContextTest.task",
                                NP_Priority,0,
                                NP_ConsoleTask,NULL,
                                NP_HomeDir,NULL,
                                NP_CopyVars,FALSE,
                                TAG_DONE)) {

 /* Get the task (uhm, complicated) and its
 ** process message port we use here for
 ** communications
 */

 testtask=&(proc->pr_Task);
 testport=&(proc->pr_MsgPort);

 /* This is the trick: Let the tast enter the
 ** private context. From now on, the library will
 ** exchange MMU trees on task switches,
 ** performing TRUE "context switches".
 */

 Printf("Let the task enter the private context.\n");

 if (EnterMMUContext(privctx,testtask)) {

  /* This demonstrates that the library keeps
  ** caches consistently across contexts. They
  ** will be flushed correctly on a context
  ** switch. We pass a stupid message to the
  ** testtask, get it modified there and print
  ** it here.
  */

  Printf("Setup a test string.\n");
  strcpy(testpage,"A silly test.\n");

  /* print the original */
  Printf("%s",testpage);

  msg->mn_Node.ln_Name=pother;
  for (i=0;i<10;i++) {
   /* pass over the message to the test task */
```

```
            Sync(testport,msg);

            /* print the result */
            Printf("%s",testpage);

            /* and restore the final A */
            *testpage='A';
           }

          } else Printf("Failed to add the test "
                       "task to the context.\n");

          /* tell the task to commit suicide. It will
          ** remove itself from the private context.
          ** This step is important and must be performed
          ** somewhere, or you'll have a memory leak.
          */

          Printf("Signalling the task to unload.\n");

          msg->mn_Node.ln_Name=NULL;
          Sync(testport,msg);

         } else Printf("Can't run child task.\n");

         Printf("Clean up the message.\n");
         DeleteIORequest((struct IORequest *)msg);
        } else Printf("Can't build communication message.\n");

}

/* Sync */
void Sync(struct MsgPort *destination,struct Message *msg)
{
struct Task *mytask;
struct MsgPort *port;

        /* naive sync between the calling task and the
        ** background task
        */

        mytask=FindTask(NULL);
        port=&(((struct Process *)mytask)->pr_MsgPort);

        PutMsg(destination,msg);
        WaitPort(port);
        GetMsg(port);

}
```

```
/* TestProc: The main loop of the detached task */
void __saveds TestProc(void)
{
int i=0;
struct Message *msg=NULL;
struct MsgPort *port;

        /* this is now the test task. Note that we have
        ** here our own MMU table.
        */

        port=&(((struct Process *)(FindTask(NULL)))->pr_MsgPort);
        for(;;) {
                WaitPort(port);
                msg=GetMsg(port);
                /* get the next message */

                /* end? If so, commit suicide */
                if (msg->mn_Node.ln_Name==NULL)
                        break;

                /* if not, just do something to make us known
                ** to the user
                */
                (*(msg->mn_Node.ln_Name)) += i;
                i++;
                ReplyMsg(msg);
        }

        /* The next step is important: We shut down,
        ** and hence have to
        ** leave the private context.
        */
        LeaveMMUContext(FindTask(NULL));

        /* We're done. Make sure main doesn't unload
        ** us before we're shut down.
        */
        Forbid();
        ReplyMsg(msg);
}
```

7.2 Adjusting an Existing Context

Most of the Context parameters can be read back after the Context has been created, and some of them are even adjustable after creation. It's the job of **GetMMUContextData()** and **SetMMUContextData()** to grant access to these internals of the Context. The function

```
struct MMUContext *ctx;

        SetMMUContextData(ctx,...);
```

takes a pointer to the Context, and a list of tag items. It accepts a subset of the tag items defined for **CreateMMUContext()** in the last chapter, namely those which are marked as *set*-able.

Its counterpart is **GetMMUContextData()**: It reads Context specific parameters, one at a time:

```
struct MMUContext *ctx;
ULONG parameter;

        option = GetMMUContextData(ctx,parameter);
```

The **parameter** argument is one of the *get*-able tag ids defined for **CreateMMUContext()**, as in the chapter above. Note that this call is not tag-based. It accepts a single tag id and returns a single result, namely the current setting of the specified option.

Some additional tag ids are accepted as well. Some of them provide some kind of internal information you really shouldn't care about, and which might be of some interest for debugging software only; as all other tag ids, they are defined in `mmu/mmutags.h`:

MGXTAG_PAGESIZE Returns the page size in bytes. Therefore, the function calls

```
GetMMUContextData(ctx,MGXTAG_PAGESIZE) == GetPageSize(ctx)
```

are identical. Unlike **MCXTAG_PAGEBITS**, this is not an exponent. The former specifies the page size in bytes, the latter the number of bits required to address one page. Obviously, we have $pagesize = 2^{bits}$.

MGXTAG_REMAPSIZE Returns the worst-case alignment restrictions for remapped memory in exec memory free lists. This is identical to the **RemapSize()** function we will discuss in detail below.

MGXTAG_ROOT Returns the pointer to the root level of the true hardware MMU descriptors. The result is a **ULONG ***. This tag item is only provided for debugging software, you *never need to touch this yourself.*

MGXTAG_CONFIG Returns a poiner to the **MMUConfig** structure which is *strictly read only.* This structure is defined in `mmu/config.h` and contains the setup for all MMU registers in case this Context gains the CPU (or rather, the MMU). Since this structure is only of some interest for debugging software, it is not explained here in detail. Just leave it alone.

7.3 Function Reference

As usual, we conclude this chapter with the functions explained so far:

Table 6: Context Setup and Manipulation Functions

MuLib function	Description
CreateMMUContext()	Build a new Context
DeleteMMUContext()	Dispose a Context
GetMMUContextData()	Read Context parameters
SetMMUContextData()	Define Context parameters

8 Mapping Lists

The MuLib does not provide functions to reserve or allocate system addresses, for example to reserve them as addresses for "virtual memory". It does not implement these functions because it is not the job of this library to do so; the MuLib is considered as a low-level interface to the MMU, and it is up to a software wrapper on top of the MuLib to do so. However, the MuLib offers functions to simplify the implementation of these features by making its own "MMU list" management functions available to the outside. These so called "mapping lists" do not have any influence on the MMU setup at all, they are a purely administrational tool for effective MMU list management. The same functions are used by the MuLib itself to implement the high-level MMU setup calls like **SetProperties()**. Mapping lists can be seen as the basic tool to hold the "properties" of the address space, hence they implement a memory map in an abstract way. A memory map assigns a set of "property flags" to each address in the memory map, very much like the property flags used in Context manipulation, however, the meaning of most of these flags is up to you because they are never interpreted by the MuLib mapping list management functions themselves.

The mapping lists are handled by two basic objects: The **MappingList** and the **MappingNode**, the contents of the **MappingList**. Both objects are extensions to the exec lists as defined in `exec/nodes.h` and `exec/lists.h`.

The MuLib includes do not define a structure for the first object; just use it as a **struct MinList ***, but never allocate or release these lists yourself because the MuLib might define some internal fields in these lists which are intentionally not documented, and it might use a private memory pool to optimize memory management. Your program should simply use a **struct MinList *** whereever you need to hold one of these lists.

8.1 Creation and Deletion of Mapping Lists

The MuLib offers three functions to build mapping lists:

```
struct MinList *maplist;

        maplist = NewMapping();
```

creates a new mapping list which is completely empty. The complete 4GB address space managed by the list will be set to **MAPP_BLANK**.

The function call

```
struct MinList *maplist,*source;

        maplist = DupMapping(source);
```

however, creates a full duplicate of an already existing mapping list. It comes in a special form for Contexts:

```
struct MinList *maplist;
struct MMUContext *ctx;

        maplist = GetMapping(ctx);
```

This function call runs a **DupMapping()** on the Context implemented mapping list, hence creates a duplicate of the high-level memory map defined by the Context passed in. We discussed this function already in the "Working on Cotexts" chapter.

All mapping lists are released in the same way, namely by

```
struct Context *ctx;
struct MinList *maplist;

        ReleaseMapping(ctx,maplist);
```

The Context argument of this function is no longer used, though.

In fact, these functions have been defined already when discussing Context manipulation. They have been used to make a backup of the high-level setup of the Context setup, and this is what **GetMapping()** does, indeed: It creates a duplicate of the mapping list within the Context which defines the property flags for each address in the address space. Finally, **ReleaseMapping()** has been used to dispose this backup again.

> *What You Get is not What You See.* Mapping lists are, even though they appear as **MinList**s, not **MinList** structures. Please never ever try to allocate or release these structures yourself, or you will mess up the internal memory management of the MuLib — and cause a "guru" most likely.

8.2 Mapping Nodes

A mapping list holds nodes much like Exec lists hold nodes, namely as doubly linked list with "header node", which is the list structure itself. Each node on this list is called a "mapping node", and its structure is defined in `mmu/context.h`:

```
struct MappingNode {
        struct MappingNode     *map_succ;
        struct MappingNode     *map_pred;

        ULONG           map_Lower;      /* lower address */
        ULONG           map_Higher;     /* higher address, inclusive */
        ULONG           map_Flags;      /* internal use only. */
        ULONG           map_Properties;/* your property flags */
        union {
                void    *map_UserData;  /* your data if invalid or swapped */
                void    *map_Page;      /* destination page if bundled */
                LONG    *map_Descriptor;/* ptr to a descriptor */
                LONG     map_Delta;     /* added to logical if remapped */
        }               map_un;
};
```

As for mapping lists, *never allocate this structure yourself.* The **MappingNode** structure might hold more components than defined above. Furthermore, it is stricly *read only.* The only way to access these nodes is by calling MuLib functions.

A mapping list contains therefore an arbitrary number of mapping nodes, defining the properties of each byte in the 4GB address space the MC68K family has to offer. This list is ordered by the **map_Lower** field in the structure above, defining the lower end of the address region each mapping node keeps care of. The list is not allowed to contain overlaps or holes, i.e. each byte in the 4GB address space must be handled by one and exactly one mapping node.

Let's start with a brief description of the components of this structure:

map_succ,map_pred Used for linking the mapping nodes together in the same way nodes on the Exec lists are linked.

map_Lower The lower base address of the range of addresses that is managed by this node. Mapping nodes are sorted by this lower address.

map_Higher The upper end of the range of addresses managed by this node. This address is *inclusive*. Hence, a mapping node responsible for 4K of memory starting at `0x1000` will have a **map_Lower** of `0x1000` and a **map_Higher** of `0x1fff`. Therefore, the last mapping node in a mapping list has **map_Higher** set to `0xffffffff`.

map_Flags These flags are for internal use only. Do not touch them in any way. They are intentionally undocumented.

map_Properties This is the place where the "property flags" like **MAPP_CACHEINHIBIT** are stored. As long as you never want to transfer these settings to a true Context, the meaning of most of these flags is up to yourself.

map_un This union keeps some user specific data for some special pre-defined property flags. Namely,

map_UserData keeps user specific data in case the **MAPP_INVALID** or, alternatively, the **MAPP_SWAPPED** property flag is set.

map_Page keeps the physical address of the destination page for **MAPP_BUNDLED** pages.

map_Descriptor keeps the address of the page descriptor for **MAPP_INDIRECT** pages.

map_Delta This is a special case for **MAPP_REMAPPED** pages. Unlike what you might expect, the property node does not keep the physical destination for **MAPP_REMAPPED** pages. Instead, it keeps the difference between the logical source address and the physical destination address. You get the physical destination address by the formula

```
physical = node->map_un.map_Delta + node->map_Lower
```

This might seem strange, but it was really easier to manage remapping like this.

The **map_un** union is unused for all other property flags.

To give an example for the application of these mapping nodes: The "MuScan" program copies the Context mapping list by **GetMapping()**, and then prints the contents of all mapping nodes in this duplicate to the console by analyzing the structure above.

The MuLib implements the following functions to work on property lists:

```
struct MinList *from,*to;
ULONG base,length,mask;
BOOL fine;

        fine = CopyMapping(from,to,base,length,mask);
```

This function transfers properties from one mapping list to another and filters them thru a mask. A 1-bit in the mask copies the corresponding property bit from the source to the destination, a 0 bit leaves the destination intact. The length parameter indicates the size of the memory block whose properties are to be transfered. As a special case, **length = 0** and **base = 0** means to transfer the full list.

This function has an analog which takes a Context instead of a mapping list as source. It is otherwise identical:

```
struct MMUContext *ctx;
struct MinList *to;
ULONG base,length,mask;
BOOL fine;

        fine = CopyContextRegion(ctx,to,base,length,mask);
```

Like all other MuLib calls, this and the previous functions don't leave you with an unusable mapping list in case it failed. Either, the resulting list will reflect all the requested changes, or the destination list remains unmodified.

And last but not least, another analog which takes a mapping list as source and a Context as destination:

```
struct MMUContext *ctx;
struct MinList *from;
ULONG base,length,mask;
BOOL fine;

        fine = SetPropertiesMapping(ctx,from,base,length,mask);
```

This is therefore the list-based analog of **SetProperties()**; instead taking parameters and tag items to define the property flags, they are read from the list passed in. There is, however, also a parameter and tag based function which operates on mapping lists instead of Contexts. It is almost identical to **SetProperties()**, it just takes a mapping list instead of a Context as destination operand, but returns a different result code:

```
struct MinList *to;
ULONG flags,mask,lower,size;
int result;

        result = SetMappingProperties(to,flags,mask,lower,size,
                                      TAG_DONE);
```

The result codes are as follows: It returns 0 on failure, much like **SetProperties()** but either 1 or 2 on success. The special result code 2 means that the mapping list was really altered, whereas a result code of 1 means that the operation was performed successfully, but the resulting list was identical to the list the operation started with. This happens, for example, if you tried to set some property bits which have been set already before.

There is of course a similar call to extract information from a mapping list. It is the list-based analogue of the Context based **GetProperties()**:

```
struct MinList *maplist;
ULONG address;
ULONG flags;

        flags = GetMappingProperties(maplist,address,TAG_DONE);
```

Very much like **GetProperties()**, this call returns the property flags for the logical address passed in. Additionally, you may provide pointers to more variables to be filled in by passing the very same tag items defined for **GetProperties()**.

8.3 Function Reference

Let's give a brief overview about the mapping list functions:

Table 7: Memory Map Adminstration Functions

MuLib function	Description
NewMapping()	Allocate a new mapping list
GetMapping()	Make a copy of a Context mapping list
DupMapping()	Duplicate an existing mapping list
ReleaseMapping()	Release a mapping list
CopyMapping()	Copy a region from one list to another
CopyContextMapping()	Copy a region from a Context to a list
SetPropertiesMapping()	Copy a region from a list to a Context
SetMappingProperties()	Define the mapping of a property list
GetMappingProperties()	Read the flags from a property list

9 Miscellaneous Functions

In this section, we're going to describe the remaining miscellaneous MuLib functions that do not belong to any other class. Most of them are related low-level MMU programming and are usually not required by the arbitrary program.

9.1 Aligned Memory Allocation

Even though the Exec memory allocation functions like **AllocMem()** guarantee to return **LONG** or even quad word aligned memory blocks, this is sometimes not enough. Most MMU functions operate on "pages" which start at multiples of the page size in memory, and the CPU caches divide the memory in "cache lines" to four long words each. Therefore, it is desirable to have a function that allocates memory according to these stricter requirements:

```
ULONG bytesize,requirements,alignment;
void *mem;

        mem = AllocAligned(bytesize,requirements,alignment);
```

This MuLib function, **AllocAligned()** is similar to its Exec counterpart **AllocMem()** except that it takes one additional parameter defining the required alignment: The returned (logical) address is guaranteed to be a multiple of the **alignment** parameter passed in. This parameter *must* be a power of two, though. As all MMU alignment restrictions fulfill this requirement, this is not a restriction for the purpose of the MuLib.

The **bytesize** and **requirements** arguments are identical to the corresponding arguments of **AllocMem()** and are described in more detail in the Exec autodocs and in the file `exec/memory.h`.

Similar to **AllocMem()**, **AllocAligned()** returns **NULL** in case of failure, and memory allocated by means of **AllocAligned()** is released by **FreeMem()**.

This call requires currently some trickery, its implementation is not "too nice", but there is currently not other way, unfortunately. Therefore, to allow future enhancements in the form of external patches, the MuLib code calls this function over the `_LVO` library vector meaning that the internal MuLib memory allocation can be transparently enhanced.

The MuLib provides a second specialized memory allocation function which might prove useful in a mixed MC68K/PPC environment:

```
ULONG bytesize,requirements;
void *mem;

        mem = AllocLineVec(bytesize,requirements);
```

This call allocates a memory vector similar to the Exec function **AllocVec()** which must be released with **FreeVec()** afterwards. The memory vector returned is guaranteed to reside in its own cache line to avoid cache interactions in a two-processor system. However, the address returned **is not guaranteed** to be aligned to a cache line, even though the full vector including the vector length count and the remaining vector are guaranteed to fill an even multiple of the cache line size. Before returning, this function ensures that the vector length count is successfully written out to the memory such that this information is consistently available to a second CPU. However, if you specify **MEMF_CLEAR**, this function does not guarantee that the "zeros" written out by this function in order to clear the memory region arrived in physical memory. Instead, they may reside in the cache of the CPU that called this function. If this is a problem for your application, you have to call the Exec function **CacheClearE()** or **CacheClearU()** to ensure that the caches are properly pushed.

The line alignment restrictions are currently hard-coded into this function, the cache line size is fixed to 32 bytes. This happens to be the cache line size of the PPC processors, and twice the cache line size of the MC68K family.

9.2 Public Memory Remapping

Some special race conditions arise in case you want to link remapped memory back into the exec free list. It is in general *not a good idea* to try this as some programs, mainly device drivers, do not know how to handle this case correctly, even though Exec offers support functions for this case. Therefore, this matter is clearly an advanced feature and should not be tried without all the necessary precaution.

> *Remap in Private.* If you remap or touch memory, you are urged to allocate this memory first to "reserve" the address range for your purpose. If you really want to setup an Exec memory pool containing remapped memory, make sure that the attribute flags of this memory pool do not have the **MEMF_PUBLIC** attribute bit set.

Especially, it is no good idea to "defragmentize" the Exec memory pool by remapping two non-adjoint physical memory blocks to one continuous logical memory block. Even though this technique would work provided all DMA device drivers would have been written according to the Os rules, namely by using **CachePreDMA()** and **CachePostDMA()**, this protocol is hardly implemented correctly at all and one shouldn't expect this trick to work at all. Note however that the MuLib itself could happily work in this environment as it knows, internally, the difference between physical and logical addresses.

However, the hardware MMU descriptors have to fulfill even stronger alignment restrictions than the MMU pages, and they usually can't be fragmentated. Hence, if you really really have to remap parts of the exec free memory pool, it is not enough to align the boundaries of this pool to the page size returned by **GetPageSize()**. This would break the MuLib MMU descriptor allocation routines heavely — they will cause "guru" in case this situation is detected. Instead, the tougher alignment restrictions must be obtained by

```
ULONG remapsize;
struct MMUContext *ctx;

        remapsize = RemapSize(ctx);
```

Please be prepared that this will return a size considerably larger than a page size.

Another restriction when remapping public memory is that *the remapped memory must remain available under its physical address as well.* This is, the physical address of the remapped memory *must* remain valid, because the MuLib has to setup and write to the physical addresses from time to time — as for example when constructing descriptors — and does not disable the MMU to avoid the overhead.

9.3 Determinating the MMU Type

For miscellaneous purposes, the MuLib offers a function to check which MMU type is available, and whether a MMU is available at all:

```
char type;

        type = GetMMUType();
```

This returns a single character specifying the MMU type detected in the running machine.

> *Happy Without MMU?* Unlike common believe, the MuLib *will* open even on systems without any MMU. Some of its function calls will even work correctly, as for example the memory map administration functions which do not require a working MMU in hardware. However, the low-level MMU functions will return a failure code. It is therefore important to check return codes properly, or to check whether a MMU is available in first place.

This function is currently able to detect the following MMU types, defined in `mmu/mmubase.h`:

MUTYPE_NONE No working MMU has been detected.

MUTYPE_68851 The library detected a 68020 with an external 68851 MMU.

MUTYPE_68030 A 68030 CPU with internal MMU was found in the system.

MUTYPE_68040 The internal 68040 MMU.

MUTYPE_68060 The 68060 MMU.

This function does currently not care about the PPC processors and their internal MMUs.

> *Beware of Economy Class.* It is unfortuntaly not enough to check the processor type in `SysBase->AttnFlags` to second-guess the type of the MMU which might be available. Motorola produced so called "EC" processors — where "EC" is short for "Embedded Controller" — which lack the build-in MMU of the corresponding full processors. Exec is not able to tell them apart, but the MuLib is. Hence, it is indeed possible that the **AttnFlag** field indicates a 68030 but **GetMMUType()** returns **MUTYPE_NONE**. This is no contradiction. In principle, it is even possible to equip an 68EC030 with an external 68851 MMU, even though the library does currently not handle this configuration. In principle, a 68020 or 68030 based system could even provide multiple external MMUs, but there is no Amiga board that makes use of this possibility, and the MuLib does not support it at all.

9.4 Reprogramming the MMU Temporarely

For some low-level operations it is desireable to disable or reprogram the MMU temporarely. However, note that you really can't run large subroutines in this state as *even the Os might be absent*. You are only able to call a tiny subroutine in supervisor mode, with all interrupts disabled. Therefore, the routine has to be better quick!

```
ULONG result;
void *proc;

        result = WithoutMMU(proc);
```

This function calls the subroutine passed in as **proc** with the MMU and all interrupts disabled. Therefore, **proc** has to be a physical address. The subroutine must end with an **RTS** to return to the calling code.

The called routine has full access to all registers, it will be completely register-transparent. Whatever was left in the CPU registers when calling **WithoutMMU()** will be availble to this routine, and whatever will be left in the CPU registers by this routine will remain in the registers after **WithoutMMU()** when your routine returned. Note that this is the only way to guarantee correct parameter passing as it is not clear whether the logical addresses used by the caller are identical to the physical addresses seen with the MMU disabled.

Due to the very nature of this routine, the CPU caches have to be disabled temporarely as well.

Where's the Kickstart? Because the MMU is disabled by this call, it is not at all clear that the Os image doesn't vanish as well, or is maybe exchanged by a completely different Os image. Therefore, not a single Os function can be safely called within your subroutine.

The MuLib offers a second function of the same flavour, namely

```
ULONG result;
void *proc;

        result = RunOldConfig(proc);
```

It is identical to **WithoutMMU()**, including all the restrictions, except that it doesn't clear the MMU setup, but it loads the MMU with the setting the MuLib found active when it was loaded. This might have been an empty MMU setup as well, dependent on how the user installed the MuLib.

9.5 Setting the Physical Bus Error Handler

It is not the purpose of the MuLib to handle physical bus errors, i.e. exceptions generated by external hardware due to access violations. The MuLib cares only about the MMU generated exceptions and passes all other exceptions thru to the exception handler which have been installed before the MuLib has been loaded. This is typcially the Exec exception handler which will sooner or later run into a "guru". In case this is not desirable, you can tell the MuLib to call a different exception handler instead:

```
void (*NewExcept)();
void (**oldexcept)();

        SetBusError(NewExcept,oldexcept);
```

The **SetBusError()** function installs the **newexcept** handler and keeps the address of the old handler in **oldexcept**. The new handler must be ready to run immediately. Furthermore, the MuLib establishes no protocol how physical bus errors should be shared amongst several applications, if this is desirable at all. This is not the purpose of the library.

The "handler" is not an MMU-type exception handler at all; the MuLib jumps into this handler if it detects an exception it can't handle or it is not responsible for, restoring the exception stack frame and all CPU registers. Therefore, your code will be called "as if" you installed your handler directly in the exception vector base of the MC68K processor. This means, specifically, you have to save the registers you have to touch, and you have to exit either by an **RTE** or by calling the old exception handler. **Avoid hacking into the CPU exception vector base directly** because some MMU related exceptions should be handled quickly, for example **AbsExecBase** access emulation. The MuLib exception handler must be called first.

9.6 Function Reference

We present again the reference for the miscellaneous functions:

Table 8: Misellaneous Functions

AllocAligned()	Allocate a memory block aligned
AllocLineVec()	Allocate a vector aligned to cache lines
RemapSize()	Get alignment for remapped public memory
GetMMUType()	Check for the available MMU
WithoutMMU()	Call a subroutine with the MMU disabled
RunOldConfig()	Call a routine with the previous MMU setup
SetBusError()	Install a physical bus error handler

References

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

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

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

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

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

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

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

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