mmu

COLLABORATORS			
	TITLE : mmu		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 25, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 mmu **1**

1.1 mmu.doc . . . 1
1.2 mmu.library/--Background-- . . . 2
1.3 mmu.library/--Patches-- . . . 3
1.4 mmu.library/CreateMMUContext . . . 4
1.5 mmu.library/DeleteMMUContext . . . 10
1.6 mmu.library/EnterMMUContext . . . 11
1.7 mmu.library/LeaveMMUContext . . . 11
1.8 mmu.library/CurrentContext . . . 12
1.9 mmu.library/AddContextHook . . . 13
1.10 mmu.library/RemContextHook . . . 15
1.11 mmu.library/AddMessageHook . . . 15
1.12 mmu.library/RemMessageHook . . . 17
1.13 mmu.library/ActivateException . . . 18
1.14 mmu.library/DeactivateException . . . 18
1.15 mmu.library/GetPageSize . . . 19
1.16 mmu.library/RemapSize . . . 20
1.17 mmu.library/SetProperties . . . 21
1.18 mmu.library/SetPageProperties . . . 26
1.19 mmu.library/RebuildTree . . . 28
1.20 mmu.library/RebuildTrees . . . 29
1.21 mmu.library/GetProperties . . . 31
1.22 mmu.library/GetPageProperties . . . 32
1.23 mmu.library/AllocAligned . . . 34
1.24 mmu.library/LockMMUContext . . . 35
1.25 mmu.library/UnlockMMUContext . . . 35
1.26 mmu.library/AttemptLockMMUContext . . . 36
1.27 mmu.library/LockContextList . . . 37
1.28 mmu.library/UnlockContextList . . . 38
1.29 mmu.library/AttemptLockContextList . . . 38

1.30 mmu.library/AllocLineVec . . . 39
1.31 mmu.library/PhysicalPageLocation . . . 40
1.32 mmu.library/PhysicalLocation . . . 41
1.33 mmu.library/DMAInitiate . . . 42
1.34 mmu.library/DMATerminate . . . 43
1.35 mmu.library/GetMapping . . . 44
1.36 mmu.library/ReleaseMapping . . . 45
1.37 mmu.library/NewMapping . . . 46
1.38 mmu.library/CopyMapping . . . 47
1.39 mmu.library/DupMapping . . . 48
1.40 mmu.library/CopyContextRegion . . . 48
1.41 mmu.library/SetPropertiesMapping . . . 49
1.42 mmu.library/SetMappingProperties . . . 50
1.43 mmu.library/GetMappingProperties . . . 51
1.44 mmu.library/SetPropertyList . . . 52
1.45 mmu.library/GetMMUType . . . 53
1.46 mmu.library/SuperContext . . . 54
1.47 mmu.library/DefaultContext . . . 55
1.48 mmu.library/WithoutMMU . . . 55
1.49 mmu.library/RunOldConfig . . . 56
1.50 mmu.library/SetBusError . . . 57
1.51 mmu.library/GetMMUContextData . . . 58
1.52 mmu.library/SetMMUContextData . . . 59
1.53 mmu.library/BuildIndirect . . . 60
1.54 mmu.library/SetIndirect . . . 63
1.55 mmu.library/SetIndirectArray . . . 64
1.56 mmu.library/GetIndirect . . . 65

Chapter 1

mmu

1.1 mmu.doc

```
--Background--
--Patches--()
ActivateException()
AddContextHook()
AddMessageHook()
AllocAligned()
AllocLineVec()
AttemptLockContextList()
AttemptLockMMUContext()
BuildIndirect()
CopyContextRegion()
CopyMapping()
CreateMMUContext()
CurrentContext()
DeactivateException()
DefaultContext()
DeleteMMUContext()
DMAInitiate()
DMATerminate()
DupMapping()
EnterMMUContext()
GetIndirect()
GetMapping()
GetMappingProperties()
GetMMUContextData()
GetMMUType()
GetPageProperties()
GetPageSize()
GetProperties()
LeaveMMUContext()
LockContextList()
LockMMUContext()
NewMapping()
PhysicalLocation()
PhysicalPageLocation()
RebuildTree()
RebuildTrees()
ReleaseMapping()
```

RemapSize()
RemContextHook()
RemMessageHook()
RunOldConfig()
SetBusError()
SetIndirect()
SetIndirectArray()
SetMappingProperties()
SetMMUContextData()
SetPageProperties()
SetProperties()
SetPropertiesMapping()
SetPropertyList()
SuperContext()
UnlockContextList()
UnlockMMUContext()
WithoutMMU()

1.2 mmu.library/--Background--

PURPOSE

The mmu.library provides functions for MMU related operations as write- or read-protecting certain areas of memory for a given set of tasks, or marking memory regions as "swapped" virtual memory support. It offers an abstraction level on top of the actual MMU and a unified interface for MMU purposes.

The MMU lib does NOT implement virtual memory, that's the purpose of another library - the memory.library. There's no much reason why any application except the memory.library and probably some debugging tools should call this library directly. The memory.library functions on top of this library should suffer for "all day purposes".

The basic object administrated by this library is the "context". A "context" is the software abstraction of a MMU translation tree, it keeps the information of the status of memory. The mmu.library provides a "global" context that represents the MMU translation tree for all tasks. It is build by the init code of the library either by snooping the already existing MMU translation tree or by building an own tree by looking at the available memory and expansion devices.

Programs can build own contexts by using the MMU library functions and may enter or leave the contexts generated in this way. Several tasks may share one context, as they represent, for example, multiple "threads" of one single program. Thus, the notation used here is somehow turned around: A "context" in the amiga world is called "process" in unix, and a "task" or "process" in the amiga world is called a "thread" usually.

Tasks that do not enter an own context explicitly share the common global context. Even though it is possible to change the global context, this technique should be used only by debugging tools as the enforcer. It could, for example, mark unused areas of the memory as well as the first 4K as "invalid".

The MMU library provides two "hooks" for each context, the "bus error hook" and the "segmentation fault" hook. The first hook is called if an access to an invalid memory location is detected, or a write to a read-only memory area. The second is called if an access to a memory page marked as "swapped out" is detected. It's part of the service of the MMU library to keep these two access violations apart.

The default hooks call simply the exception handler of the running task, which generates by default the well known "guru". It's up to programs on top of the MMU library to setup useful hooks. An enforcer like tool might set "bus error hook" to print some useful information about the access violation, the memory.library will set the "segmentation fault" hook to swap in the memory in question.

The hooks are really "low-level" routines and are executed in super-visor mode, and therefore very limited in power.

The library provides itself a ready-for-use hook function which sends a message to a task to be specified and suspends the task that caused the bus error until the message is replied.

MMU trees are managed on two levels, a "software abstraction level" which is comfortable and easy to use. Memory "properties" of large regions of memory can be redefined quite easely with one single call, but rebuilding the MMU table tree from this abstraction layer is a lengthly operation and requires quite a lot of CPU power not available in critical applications.
Therefore, a more "hardware based" approach is available which modifies the existing MMU table "on the fly" and is therefore rather fast. However, memory pages to be handled like this must be marked with the MAPP_SINGLE property to tell the library to build the MMU table in a special way to allow fast modification and to bypass some possible optimizations (e.g. "early page terminators" and "invalid descriptors not at page level").
Modifications on this level are never seen by the abstraction level above and should be used only once the MMU tree is complete.

If that is, too, not fast enough, the MMU library allows building pages using "indirect descriptors". That is, the page property flags are given by a true hardware descriptor you provide. This makes code CPU specific, obviously, but has the advantage that page properties can be modified by a single long word "poke" in your program, and the proper ATC flush. This is almost "hardware banging", with some support.

This method has one important drawback, namely that the library is no longer able to disable caching for DMA transfer to and from this page. Hence, NEVER EVER access an indirect page by DMA.

1.3 mmu.library/--Patches--

To perform its job, the mmu.library has to install a number of system patches. I tried avoid this as best as possible, but some installations have to be done.

```
  The following patches are installed:


exec/CacheControl:      Replaced partially by a function that guarantees
      that the data cache remains off when the mmu.library
      builds descriptors.

exec/ColdReboot:  Replaced partially by a function that removes the
      mmu.library installed MMU trees. This is done to
      restart the processor with a clean MMU setup.

exec/Alert:   Replaced partially. The Alert() replacement routine
      will check whether the alert caused is fatal or not,
      and will Disable() and unload the mmu.library
      installed MMU tree if this is the case. Since a
      deadend alert will go over into a reset without
      calling ColdReboot(), this must be done here
      separately to ensure that the machine restarts
      properly.

exec/AddMem;    Adding memory with a MMU setup active would or would
      not mean that the MMU table must be adjusted to
      reflect the changes. This function should not be
      called once the system is setup and running.
      It is here patched to an Alert().
      The original routine is not called because some
      68040/68060 libraries modify the MMU table on their
      own in case this happens.
      I don't think this routine is really required unless
      on startup, but in case it is I will replace it
      completely on demand.

exec/AddConfigDev:      Patched to an Alert() for just the same reason than
      above.

exec/CachePreDMA:       Replaced completely by a MMU library routine that
      disables copyback mode for non-aligned pages and
      translates the logical address passed in to the
      physical address, by means of the mmu.library
      public context.

exec/CachePostDMA:      Ditto. Replaced completely by the MMU.library.
```

1.4 mmu.library/CreateMMUContext

```
    NAME
  CreateMMUContext - Build a new MMU context.

    SYNOPSIS
  context = CreateMMUContextA ( tags );
  d0              a0
```

```
context = CreateMMUContext ( tag1, ... );


struct MMUContext * CreateMMUContextA ( struct TagItem *);

struct MMUContext * CreateMMUContext ( Tag tag1, ... );

  FUNCTION
This function builds a new MMU context.

  INPUTS
tags    -   Tag items for the context to be created.

Currently defined tags are:


MCXTAG_COPY      -  build a copy of the context * passed in,
        if the pointer is NULL, build a copy of
        the global default context. (RECOMMENDED!)

        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 clone, 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. Even though the library
        itself allocates all its internal structures
        aligned to "worst case" page sizes, this might
        not be true for external user programs.

MCXTAG_EXECBASE  -  allow accesses of AbsExecBase and all accesses to
        valid memory in the first page, even if this
        page in the MMU tree - the "zeropage" - is
        marked as invalid. This is an important feature
        if the mmu.library is used to implement an
        Enforcer type program.
        Accesses will be emulated in software and are
        hence *slow*. They should be avoided, either by
        running Os 3.0 or better, or by using a tool like
        "MuMove4k".

        AbsExecBase accesses are handled with highest
        priority and might be faster than the rest of the
        emulation, even faster than the Enforcer, even
        though they're still slower than "the real thing".
        It is not guaranteed that the library will really
        read AbsExecBase from location four, dependent on
        some internals, it might give you a cached copy
        instead. This means usually no problem since
```

```
        AbsExecBase must not change while the machine is
        up and running.

        This option defaults to TRUE.

        For more quirks about the zeropage, read the
        MCXTAG_ZEROBASE tag documentation below.

MCXTAG_BLANKFILL  - defines a ULONG word fill value for blank
        memory regions.

        This ULONG is read by the CPU in case a program
        tries an access to "empty" memory regions. It
        defaults to 0L, but other values might be use-
        ful for debugging.

MCXTAG_MEMORYATTRS- An exec memory attributes specifier to be used
        for allocating memory for the (hardware) MMU
        tables. Defaults to MEMF_PUBLIC, check
        exec/memory.h for other possibilities.

MCXTAG_PRIVATESUPER A boolean value, either TRUE or FALSE. If TRUE,
        build a private MMU supervisor table for this
        context, independent of the default supervisor
        Context. BE WARNED! This means specifically that
        possible modifications of the default supervisor
        context will not be carried over to your private
        supervisor table.
        Even if you pass in FALSE here, the library might
        still ignore your choice and build a private
        supervisor table anyways. This happens if the
        table layout you've chosen is different to the
        default table layout, making the user tree in-
        compatible to the default supervisor tree.

MCXTAG_ZEROBASE   - This option takes only effect if MCXTAG_EXECBASE
        is TRUE and the first page of the MMU table is
        invalid. This means that the MMU library will
        emulate accesses to valid memory in the first page
        which remains unavailable to implement an
        "Enforcer" function. This tag provides a base
        address to be used by the emulation as physical
        base address.
        This tag defaults to NULL meaning that the
        library will redirect accesses to the true
        zeropage, making it available temporarely, but
        any other memory location - provided it is page
        aligned - can be specified here as well. This
        is important if the zero page gets remapped to
        a different location, and an Enforcer type
        program is run later on. 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 MMU library 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!
THIS IS AN ADVANCED FEATURE.

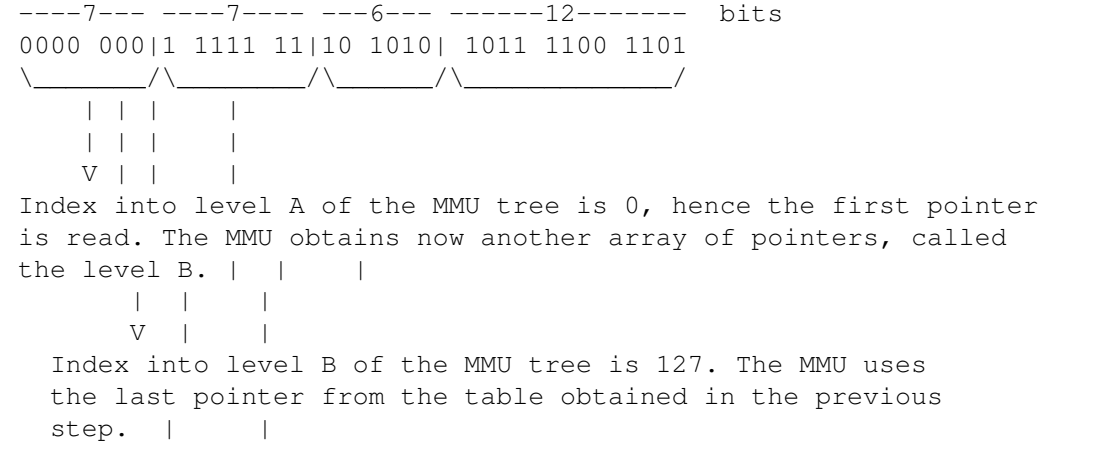
MCXTAG_DISCACHEDES A boolean tag item. If set to TRUE, the memory
that keeps the descriptors is cache-inhibited.
This works around some problems that appear if a
program attempts to hack on the MMU itself. The
mmu.library code has no problems with descriptors
in non-cacheable memory. Note, however, that the
memory will be only non-cacheable "as seen" from
the context itself. It will *NOT* change the
cache mode as seen from other contexts, even from
the supervisor Context of the given Context.

HACKING THE MMU TABLES IS ILLEGAL

MCXTAG_LOWMEMORYLIMIT Define a boundary in the zero page such that
accesses to addresses higher than 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.
(NEW for V42)
The MuLib checks the chip memory base address on
startup and provides a useful default.

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 to define 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 are 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. The address 0x01feabcd would be used like this to find an descriptor:

hex 0x01feabcd is binary



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.

Index into level B of the MMU tree is 127. The MMU uses the last pointer from the table obtained in the previous step.

```
          |    |
          V    |
    Index into level C of the MMU tree is 42. The MMU
    uses, hence, the 42nd pointer of the array pointed
    to by the 127th pointer of the level B array. This
    is now the "page descriptor", defining the base
    address for the next step.
          |
          |
          V
This is now finally the page offset, 0xbcd in this example. It
is added to the base address from the page descriptor in level C.
If the address is not "remapped", the base address would be
identically to the first 32-12 = 20 bits of the physically address.


MCXTAG_DEPTH      - Depth of the MMU tree to build. Defaults to
        the depth of the global public context.

        Legal values are 1..4 for the 68020/68851 and
        the 68030, but the 68040/68060 supports only
        trees of depth 3.

MCXTAG_LEVELABITS - Number of bits of the logical address that make
        up the level A of the MMU tree. 2^bits is the
        number of entries on 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.

MCXTAG_LEVELBBITS - 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 851 and 030, and
        7 as only possible value for the 040 and 060.

MCXTAG_LEVELCBITS - 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 851 and
        030, and must be either 5 or 6 for the 040 and
        060.

MCXTAG_LEVELDBITS - Number of bits in the level D of the MMU tree,
        only used if the depth is four and therefore
        unused on the 040 and 060.

        Must range from 1..15.

MCXTAG_PAGEBITS   - Number of bits to be used from the logical
        address as the page offset. 2^bits is 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 global
        context.

All the "..."BITS specifications MUST sum up to 32 since a logical
address consists of exactly 32 bits. You may leave out some or all
of these tags, the MMU library will keep care of the rest and will
chose reasonable defaults for you.


MCXTAG_ERRORCODE -  pointer to a ULONG the MMU library will fill
        in with an error code in case building the
        MMU tree failed. It will be set to 0L if the
        function succeeds. The following error codes
        have been defined:

    CCERR_NO_FREE_STORE      -  the library went out of memory.
    CCERR_INVALID_PARAMETERS -  the parameters specified by the
        tags are invalid.
    CCERR_UNSUPPORTED      -  the parameters are valid, but
        not supported by the hardware.
    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
        roudings that have to be performed,
        but the MMU table should work in
        general.

        THIS IS NOT AN ERROR, you will get
        your new context.

    CCERR_UNALIGNED      -  the library had to perform heavy
        rouding in the MMU table passed
        in, it might be unusable. For
        example, remapped pages were mis-
        aligned 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 satis-
        fied.

        Still, THIS IS NOT AN ERROR. You
        get a new context, but possibly an
        usuable one.

  RETURNS
NULL if the new context couldn't be created or a handle to the
new context, as parameter to all MMU library functions.
```

```
  NOTES
This call makes NOT the current task entering the new context,
you've to call EnterMMUContext() explicitly for that purpose.

The context structure is not documented intentionally. It depends
on the implementation.

  BUGS

  SEE ALSO
DeleteMMUContext()

  BUGS
```

1.5 mmu.library/DeleteMMUContext

```
  NAME
DeleteMMUContext - delete an MMU context build with CreateMMUContext.

  SYNOPSIS
DeleteMMUContext ( context );
        a0

void DeleteMMUContext ( struct MMUContext * );

  FUNCTION
This function deletes a context build with CreateMMUContext().


  INPUTS
A handle to the context to be deleted.

  RETURNS
nothing.

  NOTES
This call doesn't remove any task from the context, especially the
current task is NOT removed from the context. You've to call
LeaveMMUContext() before.

Deleting a context with some tasks still in this context will
cause this function to guru.

If your context was created with a private supervisor context
included, you must call this function only *once*, with the
user context created. It will automatically deallocate the
attached supervisor context as well.

  BUGS

  SEE ALSO
CreateMMUContext()
```

1.6 mmu.library/EnterMMUContext

```
  NAME
EnterMMUContext - let a task enter a specific context.

  SYNOPSIS
context = EnterMMUContext( context , task );
d0           a0       a1

struct MMUContext * EnterMMUContext( struct MMUContext * ,
             struct Task * );

  FUNCTION
Add a given task to a context. This makes the MMU settings defined
by the context available for the task in question.

  INPUTS
context - the context to enter or NULL for the global context.
task    - a pointer to the task structure that should enter the
    context.

  RETURNS
a handle to the context the task participated before or NULL if
this function failed. The task will stay in the last context used
in this case.

  NOTES
This call uses the tc_Switch() and tc_Launch() functions of the
task structure. What basically happens here is that these functions
are set to internal procedures that swap the context specific
MMU table or the global MMU in and flush the ATC of the MMU.

This call may fail, check the return code for NULL - either due to
lack of memory or because the task is part of a protected context
the current task hasn't entered.

If you want to use the tc_Switch() and tc_Launch() functions your-
self, you should install a task specific context hook, see
AddContextHook().

This function can be used to change the context of a task by
adding it to a new context. The task specific context switch and
launch hooks will be "carried over" to the new context, but all
other MMU specific exceptions are now the matter of the new context.

  BUGS

  SEE ALSO
LeaveMMUContext(), exec/tasks.h
```

1.7 mmu.library/LeaveMMUContext

```
  NAME
LeaveMMUContext - remove a given task from a context.
```

```
  SYNOPSIS
context = LeaveMMUContext( task );
d0          a1

struct MMUContext * LeaveMMUContext( struct Task *);

  FUNCTION
The specified task leaves its context and enters the global context.

  INPUTS
task - the task that should leave a private context and enter the
    global default context.

  RETURNS
the context the task was added to or NULL on failure.
Might be the global default context if the task did not enter any
context.

  NOTES
It is safe to call this function even if the task wasn't added to
any context. The function returns the global context in this case.

This function must be called to any task participating a given
context to be able to delete that context.

This function is equivalent to EnterMMUContext(NULL,task).

This function makes use of the tc_Switch() and tc_Launch() functions
of the task structure to be able to set the MMU root pointer.

Make sure that you check for failure. This call may return NULL if
the task entered a protected context the current task does not
participate or if switch and launch exceptions are in use.

  BUGS

  SEE ALSO
EnterMMUContext(), DeleteMMUContext(),
RemContextHook(), exec/tasks.h
```

1.8 mmu.library/CurrentContext

```
  NAME
CurrentContext - find out the current context of a task.

  SYNOPSIS
context = CurrentContext( task );
d0         a1

struct MMUContext * CurrentContext( struct Task * );

  FUNCTION
This function is used to get a handle to the context the given
task is added to, or to return the context of the calling task.
```

```
  INPUTS
task - the address of the task structure you like to investigate
or NULL to get a handle to the currently active context.

  RETURNS
a handle to the context the given task is added to, the global
context if the task is not attached to any context or the
currently active context if the argument is NULL.

  NOTES
This call fails only if the given task is part of a protected
context which is not shared by the current task. The NULL
argument is always safe.

  BUGS

  SEE ALSO
FindTask()
```

1.9 mmu.library/AddContextHook

```
  NAME
AddContextHook - set an exception handler to a Context

  SYNOPSIS
hook = AddContextHookA ( tags );
        a0

hook = AddContextHook ( tag1, ... );


struct ExceptionHook * AddContextHookA ( struct TagItem * );

struct ExceptionHook * AddContextHook ( Tag tag1, ... );

  FUNCTION
This call installs an exception hook for a given context for
various exception types the MMU library can provide.

  INPUTS
tags    -   A taglist defining the type of the exception hook
      to be added.

Currently defined are:

MADTAG_CONTEXT  - The context to which this exception hook
      should be added. This MUST be given for
      "swapped" handlers. If it is left blank
      or set to NULL for segmentation fault
      handlers, you define a global segmentation
      fault handler.

MADTAG_TASK - If the hook should be called only if a
      specific task is running, specify a pointer
```

```
      to the task structure here.
      Warning! Adding too many task specific
      hooks slows things down unnecessary.
      Remember that a MMU Context may hold more
      than one task.
      This MUST be given for the switch and launch
      hooks.

MADTAG_TYPE - Type of the exception hook to build. The
      following types are available:

    MMUEH_SEGFAULT  - Called on segmentation fault, i.e. write to
      a write protected page or access of an
      invalid page. Most useful for "Enforcer"
      like tools.

MMUEH_SWAPPED   - Called on access for a "swapped out" page.
      Most useful to implement virtual memory.

MMUEH_SWITCH    - Called when the task looses the CPU.
MMUEH_LAUNCH    - Called when the task gains the CPU.
      Remember that the tc_Switch() and tc_Launch()
      function pointers are no longer available
      if the task has been added to a MMUContext.

MMUEH_PAGEACCESS -  Called whenever a MAPP_SINGLE page gets
      build by the context. This could be used to
      modify the MMU tree "on the fly" if
      necessary, the required parameters are passed
      thru.

MADTAG_CODE  -  A function pointer to the code to be called.
      This should be an assembly language function.
      It is called like this:

Register a0  -  Pointer to the ExceptionData structure or
      the PageAccessData.
Register a1  -  loaded with the MADTAG_DATA provided data.
Register a4  -  Ditto.
Register a5  -  Pointer to the code itself.
Register a6  -  MMUBase. NOT A SCRATCH.

Registers d0-d1/a0-a1/a4-a5 are scratches and are available for the
Exception handler. You *MUST* set the "Z" condition code and
clear d0 on exit in case you handled the exception. Details on
how to write an exception handler are in the "Exception.doc" file.

MADTAG_DATA -   Data to be loaded for the hook function.
MADTAG_NAME - A name for the hook. Currently unused.
MADTAG_PRI  - A priority, ranging from -128...+127.
      Hooks of higher priorities are called first.

  RESULTS
A handle to the exception hook. Do not interpret this handle.
Or NULL on failure.

  NOTES
```

```
this call will be used by the high-level function AddMessageHook().
The global segmentation fault hook may be set by a debugging tool
like the enforcer.

The exception will not be activated, you need to call
ActivateException() explicitly to make the library call it.

Much more needs to be said about this function, see Exception.doc
for details about the exception handlers.

  BUGS

  SEE ALSO
RemContextHook(), AddMessageHook(), ActivateException(),
exec/interrupts.h, Exception.doc
```

1.10 mmu.library/RemContextHook

```
  NAME
RemContextHook - remove an exception handler from a Context

  SYNOPSIS
RemContextHook( hook )
     a1

void RemContextHook( struct ExceptionHook * );

  FUNCTION
This function removes a previously installed context hook
from the hook list.

  INPUTS
The handle of the hook, as obtained by AddContextHook().

  RESULTS
none.

  NOTES
You must call DeactivateException() on your hook before you
remove it.
Be aware that the library will call the exec exception handler,
i.e. will generate a guru in case no exception handler is
available.

  SEE ALSO
AddContextHook(), DeactivateException(), exec/interrupts.h
```

1.11 mmu.library/AddMessageHook

```
  NAME
AddMessageHook - install a high-level hook function.
```

```
  SYNOPSIS
hook = AddMessageHookA ( tags );
        a0

hook = AddMessageHook ( tag1, ... );


struct ExceptionHook * AddMessageHookA ( struct TagItem * );

struct ExceptionHook * AddMessageHook ( Tag tag1, ... );

  FUNCTION
Installs a high-level hook of the tag-given properties.
As soon as an exception of the requested type occurs, an exception
message (see below) will be sent to the port. The task that caused
the exception will be halted until the message gets replied.
BE WARNED: Message hooks perform only operation if task switching
is enabled and interrupts are allowed and the code failed in
User mode. They will just "drop thru" to the next handler if this
is not the case.

  INPUTS
tags    -   A taglist defining the type of the exception hook
      to be added.

Currently defined are:

MADTAG_CONTEXT   -  The context to which this exception hook
      should be added. This MUST be given.

MADTAG_TASK  -  If the hook should be called only if a
      specific task is running, specify a pointer
      to the task structure here.
      Warning! Adding too many task specific
      hooks slows things down unnecessary.
      Remember that a MMU Context may hold more
      than one task.

MADTAG_TYPE  -  Type of the exception hook to build. The
      following types are available:

MMUEH_SEGFAULT   -  Called on segmentation fault, i.e. write to
      a write protected page or access of an
      invalid page. Most useful for "Enforcer"
      like tools.

    MMUEH_SWAPPED    -  Called on access for a "swapped out" page.
      Most useful to implement virtual memory.

MADTAG_CATCHERPORT  -   The port to sent the data to.
MADTAG_NAME     - A name for the hook. Currently unused.
MADTAG_PRI      - A priority, ranging from -128...+127.
      Hooks of higher priorities are called first.

On an exception, the following message will be sent to the port:

struct ExceptionMessage {
```

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

For details about the ExceptionData structure, see Exception.doc.

Once the message gets replied, the faulted task is restarted.

  RESULTS
a handle for the exception that must be passed back to
RemMessageHook() for removal or NULL on failure.
Do not interpret this handle.

  NOTES
The handler must have been added to a context with EnterMMUContext()
before this function can be used. Unlike the AddContextHook()
function, this DOES NOT work for "plain" tasks without a context.

The hook must be activated with ActivateException() before it
gets called.

This function is used by the memory.library to install its
exception handler. The port will be in this case the port of
the swapper daemon that loads swapped out pages from disk.

  BUGS

  SEE ALSO
AddContextHook(), RemContextHook(), RemMessageHook(),
ActivateException(), Exception.doc
```

1.12 mmu.library/RemMessageHook

```
  NAME
RemMessageHook  -   remove a high-level hook from a context.

  SYNOPSIS
RemMessageHook( handle );
     a1

void RemMessageHook( struct ExceptionHook * );


  FUNCTION
This function removed a previously installed Message hook from
the hook list of the context.

  INPUTS
handle - a handle to the message hook as returned by the
    AddMessageHook function.

  RESULTS
none.

  NOTES
```

```
To remove a message hook safely, deactivate it first with
DeactivateException(), then tell the daemon to reply all
exceptions of this hook, then remove it.

Not following these rules may cause deadlocks.

  BUGS

  SEE ALSO
AddMessageHook(), AddContextHook(), RemContextHook(),
DeactivateException(), Exception.doc
```

1.13 mmu.library/ActivateException

```
  NAME
ActivateException   -   enable an exception hook.

  SYNOPSIS
ActivateException( hook );
        a1

void ActivateException( struct ExceptionHook * );

  FUNCTION
Activates a formerly installed exception hook, either a low
level context hook or a high-level message hook.

  RETURNS

  NOTES
Hooks of either kind must be activated before the mmu.library
will call them. Hooks are deactivated after creation and must
be deactivated before they get removed.
This call can be safely used within interrupts and from super-
visor mode.

  BUGS

  SEE ALSO
DeactivateException(), AddContextHook(), AddMessageHook(),
Exception.doc
```

1.14 mmu.library/DeactivateException

```
  NAME
DeactivateException -   enable an exception hook.

  SYNOPSIS
DeactivateException( hook );
          a1

void DeactivateException( struct ExceptionHook * );
```

```
  FUNCTION
Deactivates a formerly installed exception hook, either a low
level context hook or a high-level message hook, i.e. disables
it from being called.

  RETURNS

  NOTES
Hooks of either kind must be activated before the mmu.library
will call them. Hooks are deactivated after creation and must
be deactivated before they get removed.
This call can be safely used within interrupts and from super-
visor mode.

  BUGS

  SEE ALSO
ActivateException(), RemContextHook(), RemMessageHook(),
Exception.doc
```

1.15 mmu.library/GetPageSize

```
  NAME
GetPageSize - return the page size of a context.

  SYNOPSIS
pagesz = GetPageSize( context );
d0      a0

ULONG GetPageSize( struct MMUContext * );

  FUNCTION
This function returns the page size selected by the MMU library for
the given context. Possible page sizes are limited by the hardware
and cannot be adjusted from the outside. The page size is set up
by the MMU library for the global public context, and it can be
selected from the available page sizes for private MMUContexts, see
CreateMMUContext().

  INPUTS
context - a handle to the context to be investigated or NULL for
    the active context.

  RESULTS
the page size in bytes or zero for failure.

  NOTES

  BUGS

  SEE ALSO
CreateMMUContext()
```

1.16 mmu.library/RemapSize

```
  NAME
RemapSize - return the block size for memory remapping.

  SYNOPSIS
remapsize = RemapSize( context );
d0        a0

ULONG RemapSize( struct MMUContext * );

  FUNCTION
This function returns the smallest possible block size, and
therefore the alignment restrictions, for remapping of memory that
should be added to the exec memory pool. Since the MMU tables have
to be placed in non-fragmented memory, certain alignment
restrictions for the memory blocks the MMU tables are placed in
arise.
This harder aligment condition is only required for memory
that is put into the exec free list, but as long as remapped
memory is never returned from AllocMem, the page size is good
enough.

  INPUTS
context - a handle to the context to be investigated or NULL for
    the active context.

  RESULTS
the smallest admissable block size for memory returned by
AllocMem().

  NOTES
The call will fail if the given context is protected, the result
will be zero in this case.

Even though the mmu.library does support memory remapping, this
does not mean all other programs do. For example, remember that
the inputs to the "MAPP_REMAPPED" pages is a physical page size,
hence your program has to translate the logical address obtained
by AllocMem() to a physical address at first. This can be done
with the PhysicalLocation() function.

Additionally, DMA devices might or might not support memory
remapping, just for the same reason: They require physical, not
logical addresses. The MMU library provides a translation mechanism
in form of the ChachePreDMA() and CachePostDMA() functions of
ExecBase, but not all DMA device drivers call these functions
properly. Certain patches might be made available for devices
not following this rule.

  BUGS
Adding remapped memory to the freelist is highly untested and
not recommended because of the quirks of this mechanism.

  SEE ALSO
PhysicalLocation(), GetPageSize(), exec/CachePreDMA(),
```

```
DMAInitiate()
```

1.17 mmu.library/SetProperties

```
  NAME
SetProperties - set memory attributes for a given logical range.

  SYNOPSIS

result = SetPropertiesA( context, flags, mask, lower, size, tags);
d0          a0      d1    d2     a1    d0    a2

BOOL SetPropertiesA( struct MMUContext *, ULONG, ULONG, ULONG, ULONG,
         struct TagItem *);


result = SetProperties( context, flags, mask, lower, size, tag1, ...);

BOOL SetProperties( struct MMUContext *, ULONG, ULONG, ULONG, ULONG,
        Tag tag1, ...);

  FUNCTION
This call sets attributes of a certain memory range of the
software abstraction layer of the MMU tree, aligned to page
boundaries.

  INPUTS
context - a handle to the context to modify or NULL for the active
    context.

flags   - a binary flags field for the attributes to define. The
  following bits have been defined:

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

  MAPP_ROM       - Read only memory, writes tolerated.
      This is almost identically to MAPP_WRITEPROTECTED except
      that writes into this area will not cause a call of the
      segmentation fault handler. The library will filter them
      out.
      This property can be used to simulate a ROM in RAM and
      might be useful for kickstart remappers.

  MAPP_USED      -   The "used" bit of the pages
      will be set. The CPU will set this bit automatically
      as soon as the pages are accessed.

      This flag will turn on the "USED" bit in the hard-
      ware MMU bit. NOT setting this flag means that the
      USED bit in the hardware tree is preserved, regard-
      less of the mask value.

  MAPP_MODIFIED      -   The "modified" bit of the pages
```

will be set. The CPU will set this bit automatically as soon as a write is performed to the page in question. DO NOT SET THIS BIT TOGETHER WITH MAPP_WRITEPROTECTED OR WITHOUT MAPP_USED or the CPU might hang.

This flag will turn on the "MODIFIED" bit in the hardware MMU bit. NOT setting this flag means that the MODIFIED bit in the hardware tree is preserved, regardless of the mask value.

MAPP_PRIVATE - The page will be marked invalid for all but the given contexts.

MAPP_INVALID - The page will be marked as invalid. Accessing it will invoke the bus error hook. User data can be provided for this property mode, provided you don't select MAPP_SINGLEPAGE or MAPP_REPAIRABLE as well.

MAPP_SWAPPED - The page will be marked as swapped out.
A block ID *MUST* be provided for this property mode.

MAPP_CACHEINHIBIT - The page will be marked as non-cacheable.

MAPP_IMPRECISEEXECPTION - The page will be marked as "imprecise exception". MAPP_CACHEINHIBIT is mandatory in this case or this flag does nothing.
Only available for the 68060, but does not harm for other MMUs.

MAPP_NONSERIALIZED - The page will be marked as serialized. MAPP_CACHEINHIBIT is mandatory if this property is selected.
Only available for the 68040, but does not harm for other MMUs.

MAPP_COPYBACK - The page will be marked as "copyback" instead of "writethrough". Generally recommended since this is faster for the '40 and '60.
Only available for 68040 and 68060, but does not harm if selected for other MMUs. MAPP_CACHINHIBIT MUST NOT be selected.

MAPP_REMAPPED - Map the page to a different memory location. Parameters are given in the tag items.

Even though this seems simple, remapping memory is full of quirks. Obviously, DMA devices and the MMU itself see the true physical addresses and not the logical addresses as filtered by the MMU. Therefore, adding remapped memory to the exec.library freelist will cause nothing than trouble and hard to trace disk faults and crashes as soon as this memory is used by a DMA device or the mmu.library itself.
(Note that even though the library is supposed to

```
     support this, this is currently untested)
    Even though there *are* documented methods how to
    prepare a DMA transfer for remapped memory, USING
    these exec function calls is unfortunately the
    exceptions. Therefore, this method is currently un-
    supported, by most (!really!) DMA device drivers.
    Amongst the broken devices are the gvpscsi.device
    and the cybscsi.device, to give just two examples.

    The omniscsi.device (the "Guru ROM") can be fixed
    with the MuOmniScsiPatch.

    If you really *MUST* remap public memory, then align
    it *AT LEAST* to the border given by RemapSize().
    Just page alignment WILL NOT BE ENOUGH due to the way
    how the library works internally.

    YOU HAVE BEEN WARNED!

MAPP_SUPERVISORONLY  -   The page will be not available
    for user programs.

    NOTE: This mode is currently implemented using invalid
    page descriptors for the user pages and is ignored
    when building supervisor tables. This method saves some
    space for the 68040 and 68060 and was the only way how
    it could be done for the 68030 and 68851 without using
    MMU tables twice as large.

MAPP_USERPAGE0       -   Set user page attribute 0.
    This selects the user page attribute 0 for the 68040
    and 68060. "USER" DOES NOT MEAN YOU!
    The status of this bit appears on special pins of the
    CPU and might be required by some hardware, so don't
    play with this. You should not change this bit, by no
    means.

MAPP_USERPAGE1       -   Set user page attribute 1.
    This selects the user page attribute 0 for the 68040
    and 68060. See above for warnings.

MAPP_GLOBAL      -   The pages are part of the
    global (public) memory.
    You should not set this bit manually, it is under
    control of the library to optimize table flushes.

MAPP_BLANK       -   Blank memory.
    The pages are mapped to one special area in RAM so
    erraneous reads and writes to these pages won't harm.
    MAPP_BLANK should ONLY be used to mark special memory
    areas as "non-available" and un-handled by the hardware,
    nothing else.

MAPP_SHARED      -   Properties are identically to
    the public context.
    This tells the library to use the same properties as
    for the global context. However, MAPP_SHARED pages
```

are not automatically updated when the global context changes, it's just a convenient way of saying "I want to uninstall my settings".

MAPP_TRANSLATED - Under control of the TTx registers.
NEVER SET THIS BIT YOURSELF.
Pages with the "MAPP_TRANSLATED" bit set are under control of the "transparent translation registers" of the MMU and are "out of scope" for the mmu.library. Defining any properties for this domain will do nothing (or little, dependent on the TTx register configuration). A virtual memory program MUST NOT use virtual addresses which are transparently translated, this won't work. The mmu.library tries to be smart about the TTx registers and disables "unuseful" TTx settings itself.

MAPP_INDIRECT - Map to a user provided page descriptor.
The provided pages(s) are mapped by a user provided page descriptor. This page descriptor MUST BE aligned to a long word address, and it MUST BE a valid page descriptor for the MMU used.

NEVER EVER attempt DMA, such as harddisk reads or writes to a memory domain marked as MAPP_INDIRECT.

Due to some cache peculatities, the data might be incorrect and the result would be corrupt data.

The library will be able to mark pages as non-cacheable if this is required for the DMA transfer, but this magic does not work for indirect pages.

JUST DON'T DO THAT, MAPP_INIDIRECT is definitely an advanced feature.

MAPP_BUNDLED - Map all pages in range to a single page in RAM.

The main purpose of this function is to provide a MAPP_BLANK property with a user-selectable target page.

MAPP_SINGLEPAGE - Make this page available for SetPageProperties().

WARNING! Setting this bit shortcuts some optimizations the library might perform on the MMU table. The system may easely run out of memory if you select this property for "too many" pages. Use it with care, you have been warned!

MAPP_REPAIRABLE - Inform the exception handler to provide write data and to allow pipeline fill.

If this bit is set, the exception handler gets informed that you want to know the write data in case writes

fail, or you want to provide the read data in case reads fail. The read/write data is available in the ExceptionData structure, read the exception handler documentation.

This flag should be combined with MAPP_INVALID, MAPP_WRITEPROTECTED, MAPP_SWAPPED or MAPP_SUPERVISORONLY

However, this technique requires a lot of "trickery" and should be expected to be slow and to create sub-optimal and over-sized MMU tables. Use it with care, on as few pages as possible and only if your exception handler is "not on a hurry".

MAPP_USERATTRIBUTE0 - This is for your private use. This bit does not have any specific function, it is for your private use.

MAPP_USERATTRIBUTE1 - This is for your private use.
MAPP_USERATTRIBUTE2
MAPP_USERATTRIBUTE3

mask - A bit mask of the attributes to be changed.

Note that the hardware USED and MODIFIED bits will never be cleared, even though the properties say so and the corresponding bits in the mask are set. However, you can force them "ON" if you like, this saves unnecessary write-backs of the MMU.

lower - The lower boundary of the logical address to be modified. This must be aligned to the page size or this call will guru.

size - Size of the region to be modified. Must be a multiple of the page size.

tags - A tag array with additional data. Currently defined:

MAPTAG_DESTINATION - the physical destination of the logical address. Must be provided for the MAPP_MAPPED or MAPP_BUNDLED bits.

MAPTAG_BLOCKID - a unique ID the MMU.library doesn't care about, for external usage of the memory.library. Must be provided for the MAPP_SWAPPED flag and may be used to indicate where on disk the swapped page is kept.

MAPTAG_USERDATA - a unique cookie you might provide for MAPP_INVALID pages and which is passed thru to the segmentation fault exception handler.

MAPTAG_DESCRIPTOR - a pointer to a long word aligned table descriptor for MAPP_INDIRECT.

```
  RESULTS
A boolean success/failure indicator. Might fail if the context
is protected or no memory is available for the modification.

  NOTES
This call adjusts only the abstraction layer of the MMU table
and marks the pages as "dirty". An explicit call to
RebuildTree() is required to make the changes active.
You should bundle changes to the MMU table and call RebuildTree()
once when you're done because rebuilding the MMU tree is a costy
operation.
If you need to modify the MMU table "on the fly" then consider
using "SetPageProperties()", even though its use is restricted
to single pages. Even faster are MAPP_INDIRECT pages, but - to
say that again - DO NOT PERFORM DMA ON THESE PAGES.

The page size can be read with GetPageSize().

SUPERVISORONLY, SWAPPED and INVALID memory are implemented
using the same MMU attributes (invalid, namely), but the
library exception handler will filter them out and call
the appropriate hook.

Write protection goes only for the context specified. It
usually makes sense to mark the memory region as PRIVATE as well,
unless you modify the public hook.
(Note that MAPP_PRIVATE is currently not implemented because it
 will slow down things considerably.)

You may freely mark the first memory page as INVALID provided
the context MCXTAG_EXECBASE flag is set (it usually is).
Long word read accesses to AbsExecBase will be filtered out by
the exception handler of the library and will be satisfied trans-
parently to the program, as well as all other accesses except
those into the first 1024 bytes.

  BUGS

If the mask contains any property that requires user data, i.e.
MAPP_REMAPPED or MAPP_SWAPPED, you *have* to redefine the user
data by tag items and MAY NOT leave them out, as the library
WILL NOT be able to restore the previously defined user data.

  SEE ALSO
GetProperties(), AddContextHook(), GetPageSize(),
RemapSize(), RebuildTree(), SetPageProperties(),
SetMappingProperties()
```

1.18 mmu.library/SetPageProperties

```
  NAME
SetPageProperties - set hardware memory attributes for a single
page.
```

```
  SYNOPSIS

result = SetPagePropertiesA( context, flags, mask, lower, tags);
d0        a0  d1    d2    a1     a2

BOOL SetPagePropertiesA( struct MMUContext *, ULONG, ULONG, ULONG,
         struct TagItem *);


result = SetPageProperties( context, flags, mask, lower, tag1, ...);

BOOL SetPageProperties( struct MMUContext *, ULONG, ULONG, ULONG,
        Tag tag1, ...);

  FUNCTION
This call sets the hardware attributes of a memory page.

  INPUTS
context - a handle to the context to modify or NULL for the active
    context.

flags   -   a binary flags field for the attributes to define. For
      the available attributes, see the SetProperties()
      function.

      Differences:

      MAPP_MODIFIED and MAPP_USED are really set or cleared
      in the true hardware table, the mask is considered
      correctly.

      Note that this function is the only method to clear
      these two hardware flags, SetProperties() or
      RebuildTree() don't do this.

mask    -   a bit mask of all bits to be changed.

tags    -   A tag array with additional data. Check
      SetProperties() for details.

  RESULTS
A boolean success/failure indicator. Might fail if the context
is protected or no memory is available for the modification or
the page is not marked with MAPP_SINGLEPAGE.

  NOTES
BE WARNED! This function is very restricted in its use. It may well
return FALSE even if all parameters are valid due to hardware
restrictions. This function does never ever rebuild an MMU tree,
it just modifies "what is there". If the library choose to optimize
the MMU library tree and to map a couple of pages by one descriptor,
for what reasons ever, this call will fail. The details when this
happens depends not only on the MMU, but on the general system
layout.

The ONLY documented way to get a mapping for a page that can be
```

```
adjusted using this call is to set the page to MAPP_SINGLEPAGE using
SetProperties() before.

This call adjusts the hardware level of the MMU table if a descriptor
is available for a single page. It does not modify more than one
page at once.

It might happen that the library does not satisfy a request
setting a page as "cacheable" if a DMA operation is currently in
progress and the page must remain "nonacheable". However, the
function will not fail in this case, but just delay the operation
until the DMA is complete. The properties will always fall back to
the next available option.

This routine is safe to be called from within interrupts, it does
not break any Forbid() or Disable() and is ideal for
quick-and-dirty repair operations within exceptions handlers,
provided the MAPP_SINGLEPAGE flag has been set.

  BUGS
Note that MAPP_SINGLEPAGE is the flag you want here, not
MAPP_REPAIRABLE. That's something different!

If the mask contains any property that requires user data, i.e.
MAPP_REMAPPED or MAPP_SWAPPED, you *have* to redefine the user
data by tag items and MAY NOT leave them out, as the library
WILL NOT be able to restore the previously defined user data.

  SEE ALSO
    GetPageProperties(), AddContextHook(),
    GetPageSize(), RebuildTree(), SetProperties(),
    SetMappingProperties()
```

1.19 mmu.library/RebuildTree

```
  NAME
RebuildTree - build a MMU hardware tree from the software abstraction
layer.

  SYNOPSIS
result = RebuildTree( context );
d0       a0

BOOL RebuildTree ( struct MMUContext * );

  FUNCTION
This function adjusts the MMU hardware tree to reflect the settings
of the software abstraction layer defined with SetProperties().

  INPUTS
context - a handle to the context to investigate or NULL for the
active context.

  RESULTS
a boolean success/failure indicator. TRUE if the operation was
```

```
performed successfully.

  NOTES
This is the big - and admittedly - slow one.

Rebuilding the MMU tree is a relatively slow operation. The library
tries to be smart about it and rebuilds only the pages whose
mappings have been adjusted, but it's still a heavy beast.

Properties temporarely defined with SetPageProperties() will be
lost after the rebuild, except for the MAPP_USED and MAPP_MODIFIED
bits.

In case of failure, the hardware layer will remain valid and un-
changed, but the context remains marked as "dirty".

BE WARNED! A consistent use of the two flags is only possible
if the pages are marked as MAPP_SINGLE. The page building algorithm
does not guarantee consistent use of these two bits except for
MAPP_SINGLE pages. Even though it *might* look well most the time,
it is not documented that these two bits are kept correctly for non-
SINGLE pages. If you need MODIFIED or USED page information, the only
way to get them is to mark these pages as MAPP_SINGLE. There's no
consistent use for these flags if early termination descriptors
(hence, w/o MAPP_SINGLE) are used by the library.

Even though some pages in the abstraction layer might be marked as
un-USED or un-MODIFIED, this routine NEVER clears the hardware bits.
It requires a call to SetPageProperties(), and hence the MAPP_SINGLE
attribute (once again!) to do this.

  BUGS
Much more should be said about this function.

  SEE ALSO
SetProperties(), SetPageProperties(), GetProperties(),
GetPageProperties(), RebuildTrees()
```

1.20 mmu.library/RebuildTrees

```
  NAME
RebuildTree - build a MMU hardware tree from the software abstraction
layer for several contexts at once.          (V41)

  SYNOPSIS
result = RebuildTreesA( contextptrptr );
d0           a0

result = RebuildTrees( context, context, ... );


BOOL RebuildTreesA ( struct MMUContext ** );

BOOL RebuildTrees ( struct MMUContext, ... );
```

```
  FUNCTION
This function adjusts the MMU hardware tree to reflect the settings
of the software abstraction layer defined with SetProperties().

  INPUTS
contextptr - a pointer to a NULL terminated array of context
handles.

  RESULTS
a boolean success/failure indicator. TRUE if the operation was
performed successfully and all context could have been rebuild.

  NOTES
This is the big - and admittedly - slow one.

Rebuilding the MMU tree is a relatively slow operation. The library
tries to be smart about it and rebuilds only the pages whose
mappings have been adjusted, but it's still a heavy beast.

Properties temporarely defined with SetPageProperties() will be
lost after the rebuild, except for the MAPP_USED and MAPP_MODIFIED
bits.

The advantage of this function is that it guarantees that all
involved contexts remain unmodified and dirty in case one of the
contexts cannot be rebuild. On success, it is guaranteed that all
contexts will have been rebuild successfully.

A special word has to be said about context locking here: This
function will first call LockContextList(), and will then lock the
individiual contexts in the order passed in. Hence, you've either
to lock all contexts as well, and hence have to call
LockContextList() yourself as first step, or you must not lock any
context of the contexts on the list. Everything else is dangerous
and implies the risk of a deadlock.

BE WARNED! A consistent use of the two flags is only possible
if the pages are marked as MAPP_SINGLE. The page building algorithm
does not guarantee consistent use of these two bits except for
MAPP_SINGLE pages. Even though it *might* look well most the time,
it is not documented that these two bits are kept correctly for non-
SINGLE pages. If you need MODIFIED or USED page information, the only
way to get them is to mark these pages as MAPP_SINGLE. There's no
consistent use for these flags if early termination descriptors
(hence, w/o MAPP_SINGLE) are used by the library.

Even though some pages in the abstraction layer might be marked as
un-USED or un-MODIFIED, this routine NEVER clears the hardware bits.
It requires a call to SetPageProperties(), and hence the MAPP_SINGLE
attribute (once again!) to do this.

  BUGS
Much more should be said about this function.

  SEE ALSO
SetProperties(), SetPageProperties(), GetProperties(),
```

```
GetPageProperties(), RebuildTree()
```

1.21 mmu.library/GetProperties

```
  NAME
GetProperties - read memory attributes for a given logical page
from the MMU table abstraction layer.

  SYNOPSIS

flags = GetPropertiesA( context, lower, tags);
d0        a0      a1     a2

ULONG GetPropertiesA( struct MMUContext *, void *, struct TagItem *);


flags = GetProperties( context, lower, tag1, ...);

ULONG GetProperties( struct MMUContext *, void *, Tag tag1, ...);

  FUNCTION
This call reads the page properties of a certain address in
memory from the software abstraction layer. It is the counterpart
of SetProperties().

  INPUTS
context - a handle to the context to investigate or NULL for the
    active context.

lower   - the logical address of the page to investigate. The
    size of the page depends on the hardware and is selected by
    the MMU library. The number of bytes in a page is returned
    by GetPageSize().

tags    - additional tags. Currently defined are:

  MAPTAG_DESTINATION  -   a pointer to a void * where
      the physical destination of the logical address is
      filled in. Only available if the page is physically
      mapped to somewhere. Not filled in otherwise.

  MAPTAG_BLOCKID      -   read the a unique ID for the
      MAPP_SWAPPED property. Untouched if the page isn't
      swapped. The tag data points to a long word which will
      be filled in for swapped out pages.

  MAPTAG_USERDATA     -   read the unique cookie for
      INVALID pages, fill in the long word pointed to by
      the tag data field.

  MAPTAG_DESCRIPTOR   -   fill in the location of the
      indirect descriptor which is used to perform the
      mapping. Only used if MAPP_INDIRECT is used.

  RESULTS
```

```
Returns a binary flags field for the attributes to define. See
SetProperties() for details. Remember that MAPP_USED or
MAPP_MODIFIED reflect the bits on the abstraction layer, not the
true hardware bits. You MUST call GetPageProperties() to read
them, and hence *MUST* use MAPP_SINGLE pages.

  NOTES
The page size can be read with GetPageSize(). Check the return
code of this call!

The flags returned are valid for the given context, a different
context may return a different flag setting and even a different
physical locations.

WARNING: The flags returned DO NOT reflect the hardware flags
in the MMU table for the context. They DO reflect the settings
installed with SetProperties() on the abstraction layer of the
MMU tables.

The hardware table might differ for the following reasons:

    - SetProperties() was called, but the changes haven't been
      made active with RebuildTree() yet.
    - A program modified the hardware layer directly using
      SetPageProperties().
    - DMA is currently active and the page in question has
      therefore been marked as non-cacheable temporarely.

Additionally, the library might have adjusted the abstraction
layer itself by allocating non-cacheabe memory for its
MMU tables.

This routine is *NOT* safe to be called from within interrupts.

  BUGS

  SEE ALSO
SetProperties(), AddContextHook(), GetMappingProperties(),
GetPageSize(), GetPageProperties(), RebuildTree()
```

1.22 mmu.library/GetPageProperties

```
  NAME
GetPropertiesA - read memory attributes from the hardware level
for a given logical page.

  SYNOPSIS

flags = GetPagePropertiesA( context, lower, tags);
d0            a0       a1    a2

ULONG GetPagePropertiesA( struct MMUContext *, void *,

      struct TagItem *);
```

```
result = GetPageProperties( context, lower, tag1, ...);

BOOL GetPageProperties( struct MMUContext *, void *, Tag tag1, ...);

  FUNCTION
This call reads the page properties of a certain address in
memory directly from the hardware. It is the counterpart
of SetPageProperties().

  INPUTS
context - a handle to the context to investigate or NULL for the
    active context. The library might use the MMU hardware directly
    if NULL is passed in, this call might be faster therefore.

lower   - the logical address of the page to investigate. The
    size of the page depends on the hardware and is selected by
    the MMU library. The number of bytes in a page is returned
    by GetPageSize().

tags    - additional tags. See GetProperties() for details.


  RESULTS
Returns a binary flags field for the attributes to define. See
SetProperties() for details.

  NOTES
The page size can be read with GetPageSize().

The flags returned are valid for the given context, a different
context may return a different flag setting and even a different
physical location.

WARNING: The flags returned reflect the NOT the hardware flags
in the MMU table for the context except for the MODIFIED and USED
properties, even though the hardware level is *almost* consistent
with these flags.

The hardware table might differ slightly in the following
situations:

    - DMA is currently active and the page in question has
      therefore been marked as non-cacheable temporarely.
      Therefore, the cache settings returned are what will be
      re-installed here when DMA is finished. The library
      will "fake" the flags you have installed for the page
      investigated.
    - The library will use invalid descriptors to implement
      supervisor only or swapped pages.

However, even though the flags might differ from the hardware
flags, you're always safe to re-install the properties with
SetPageProperties, there's no need to keep track of pecularities
like cache disabling for DMA pages. The library does this for you.

MAPP_MODIFIED and MAPP_USED are always read from the hardware
```

```
directly.

KEEP IN MIND that these two bits are only set and handled
consistently for MAPP_SINGLE pages. You MUST NOT interpret
them in all other cases, their values might get lost on a
RebuildTree() call.

This routine is safe to be called from within interrupts, most
useful within exception handlers.

  BUGS

  SEE ALSO
    GetProperties(), AddContextHook(),
    GetPageSize(), SetPageProperties(), RebuildTree()
```

1.23 mmu.library/AllocAligned

```
  NAME
AllocAligned    -   allocate memory aligned to a memory border.

  SYNOPSIS
mem = AllocAligned( bytesize, reqments, align );
d0      d0   d1 a0

void * AllocAligned( ULONG, ULONG, ULONG);

  FUNCTION
Allocate memory aligned to certain boundaries.

  INPUTS
bytesize - the size of the memory to allocate.

reqments -  exec style memory attributes

align   -   the alignment restrictions of the page.
      MUST be a power of two.

  RETURNS
a pointer to the allocated memory, aligned to the given border or
NULL if no free physical memory could be found.

  NOTES
Examples of how to use the "align" parameter:

mem = AllocAligned(123,MEMF_PUBLIC|MEMF_CLEAR,1024);

will allocate 123 bytes starting at a 1024 byte border, i.e.
the address returned will be divisible by 1024. The call will
clear the 123 bytes, NOT MORE.

This is a service routine for the memory.library and shouldn't be
used for all-day purposes.

A DOS process will have its pr_Result2 field set to
```

```
ERROR_NO_FREE_STORE if the memory allocation fails.

The mmu.library calls this function by using its LVO library entry,
so it can be patched to a smarter implementation if desired.

  BUGS

  SEE ALSO
GetPageSize(), exec/memory.h
```

1.24 mmu.library/LockMMUContext

```
  NAME
LockMMUContext      -   lock a MMU context

  SYNOPSIS
LockMMUContext( context );
      a0

void LockMMUContext( struct MMUContext * );

  FUNCTION
Lock the software abstraction layer of the MMU table against
modifications from other tasks.

  INPUTS
A handle to a MMUContext or NULL for the active context.

  RETURNS

  NOTES
This mechanism DOES NOT avoid changes of the MMU table on a lower
level by SetPageProperties(), only SetProperties() from other
tasks will be locked.
Hence, it locks the abstraction layer, but not the hardware
level.

DO NOT lock more than one context at once, unless you locked
also the context list with LockContextList(). Not following
this rule might cause deadlocks.

  BUGS

  SEE ALSO
UnlockMMUContext(), SetPageProperties(), SetProperties(),
LockContextList().
```

1.25 mmu.library/UnlockMMUContext

```
  NAME
UnlockMMUContext    -   release a MMU context
```

```
  SYNOPSIS
UnlockMMUContext( context );
        a0

void UnlockMMUContext( struct MMUContext * );

  FUNCTION
Release the software abstraction layer of the MMU table, allow
modifications from other tasks.

  INPUTS
A handle to a MMUContext or NULL for the active context.

  RETURNS

  NOTES
This mechanism DOES NOT avoid changes of the MMU table on a lower
level by SetPageProperties(), only SetProperties() from other
tasks will be locked.
Hence, it locks the abstraction layer, but not the hardware
level.

  BUGS

  SEE ALSO
LockMMUContext(), SetPageProperties(), SetProperties(),
AttemptLockMMUContext()
```

1.26 mmu.library/AttemptLockMMUContext

```
  NAME
AttemptLockMMUContext       -   attempt to lock a MMU context

  SYNOPSIS
ok = AttemptLockMMUContext( context );
            a0

struct MMUContext * AttemptLockMMUContext( struct MMUContext * );

  FUNCTION
Grants non-blocking access to a MMU context.
Attempts to lock the software abstraction layer of the MMU
table against modifications from other tasks.

  INPUTS
A handle to a MMUContext or NULL for the active context.

  RETURNS
TRUE in case of success - the context is then locked for you
and this lock must be released with UnlockMMUContext().
FALSE in case any other task holds a lock.

  NOTES
This mechanism DOES NOT avoid changes of the MMU table on a lower
level by SetPageProperties(), only SetProperties() from other
```

```
tasks will be locked.
Hence, it locks the abstraction layer, but not the hardware
level.

DO NOT lock more than one context at once, unless you locked
also the context list with LockContextList(). Not following
this rule might cause deadlocks.

  BUGS
In pre-V39 machines, this call does not lock the context again
in case you already hold a lock. This is a bug of the pre-V39
AttemptSemaphore(), read the exec autodocs for a workaround.

  SEE ALSO
UnlockMMUContext(), SetPageProperties(), SetProperties(),
LockContextList(), AttemptSemaphore()
```

1.27 mmu.library/LockContextList

```
  NAME
LockContextList     -   arbitrate a master lock.

  SYNOPSIS
LockContextList( );

void LockContextList( void );

  FUNCTION
Arbitrates a master lock that allows locking more than one context
at once to avoid deadlocks.

  INPUTS

  RETURNS

  NOTES
This lock grants access for locking more than one context at once,
to avoid deadlocks. I.e. in case you need to lock more than one
context at a time, get this lock FIRST, then lock the contexts
in any order you prefer.

This call DOES NOT avoid modification of the context list or
individual contexts at all, i.e. other tasks are still able
to create and to dispose contexts. To avoid this, you must lock
the contexts afterwards.

When you're done with the contexts, unlock the contexts first,
THEN release this lock with UnlockContextList(). NOTE THE ORDER!

  BUGS

  SEE ALSO
UnlockContextList(), LockMMUContext(), AttemptLockContextList()
```

1.28 mmu.library/UnlockContextList

```
   NAME
UnlockContextList    -    release the master context lock

   SYNOPSIS
UnlockContextList( );

void UnlockContextList( void );

   FUNCTION
Releases the master lock that allows locking more than one context
at once to avoid deadlocks.

   INPUTS

   RETURNS

   NOTES
This lock grants access for locking more than one context at once,
to avoid deadlocks. I.e. in case you need to lock more than one
context at a time, get this lock FIRST, then lock the contexts
in any order you prefer.

This call DOES NOT avoid modification of the context list or
individual contexts at all, i.e. other tasks are still able
to create and to dispose contexts. To avoid this, you must lock
the contexts afterwards.

When you're done with the contexts, unlock the contexts first,
THEN release this lock with UnlockContextList(). NOTE THE ORDER!

   BUGS

   SEE ALSO
LockContextList(), LockMMUContext(), AttemptLockContextList()
```

1.29 mmu.library/AttemptLockContextList

```
   NAME
AttemptLockContextList      -    attempt to arbitrate the master lock.

   SYNOPSIS
ok = AttemptLockContextList( );

LONG AttemptLockContextList( void );

   FUNCTION
Attempts granting the context master lock in a non-blocking
fashion.

   INPUTS

   RETURNS
```

```
TRUE in case the master lock could be arbitrated. You have then to
release it with UnlockContextList().
FALSE in case it is already locked and access could not be granted.

  NOTES
This lock grants access for locking more than one context at once,
to avoid deadlocks. I.e. in case you need to lock more than one
context at a time, get this lock FIRST, then lock the contexts
in any order you prefer.

This call DOES NOT avoid modification of the context list or
individual contexts at all, i.e. other tasks are still able
to create and to dispose contexts. To avoid this, you must lock
the contexts afterwards.

When you're done with the contexts, unlock the contexts first,
THEN release this lock with UnlockContextList(). NOTE THE ORDER!

  BUGS
In pre-V39 machines, this call does not lock the context again
in case you already hold a lock. This is a bug of the pre-V39
AttemptSemaphore(), read the exec autodocs for a workaround.

  SEE ALSO
UnlockContextList(), LockMMUContext(), LockContextList(),
AttemptLockSemaphore()
```

1.30 mmu.library/AllocLineVec

```
  NAME
AllocLineVec  -   allocate cache line aligned, keep size

  SYNOPSIS
AllocLineVec ( bytesize , attributes );
      d0        d1

void * AllocLineVec ( ULONG, ULONG );

  FUNCTION
Allocates memory like AllocVec(), but the memory is guaranteed
to be aligned to cache lines of the processors in the system,
even though the pointer returned IS NOT.
Minimal guaranteed alignment is currently 32 bytes, i.e. a
PPC cache line. Future hardware may require stricter alignments.

AllocLineVec'd memory is released with FreeVec() from the
exec.library.

  INPUTS
bytesize    -   the size of the memory block in bytes.
attributes  -   memory attributes, see exec.library/AllocMem.

  RETURNS
a pointer to the memory allocated or NULL on failure.
```

```
A DOS process will have its pr_Result2 field set to
ERROR_NO_FREE_STORE if the memory allocation fails.

  NOTES
BIG WARNING: The pointer returned IS NEVER cache line aligned
itself, but the complete memory block toghether with the
vector size is kept in a cache line, regardless of the size
passed in. Due to alignment restrictions, the routine might
allocate a larger memory block than requested. It is however
guaranteed that AT LEAST the size requested is returned, and
that a FreeVec() will, indeed, free all memory.

If MEMF_CLEAR is requested, the memory is cleared on the
MC68K side, but the "zeros written" might be still in the
cache. Hence, it is a good idea to flush the cache if the
memory is passed over to the PPC.

However, the vector size itself is always "pushed" to
memory, it is therefore guaranteed to be properly written
back to the memory.

The memory allocated this way is released by FreeVec() of
the exec.library.

  BUGS

  SEE ALSO
exec.library/AllocVec(), exec.library/FreeVec(), AllocLineVec()
```

1.31 mmu.library/PhysicalPageLocation

```
  NAME
PhysicalPageLocation    -   translate logical address to physical

  SYNOPSIS
addr = PhysicalPageLocation( context , addr );
d0        a0      a1

void * PhysicalPageLocation( struct MMUContext * , void * );

  FUNCTION
This function finds the physical address for the logical address
passed in by scanning the MMU hardware table.
If the physical address is not available, NULL is returned.

  INPUTS
context - the context to enter or NULL for the current context.
addr    - the logical address to be translated.

  RETURNS
the physical address of the logical address passed in, or NULL
in case the logical address is not swapped in or otherwise out of
control of the library.

  NOTES
```

```
This is the low-level function, consider using the high-level
function PhysicalLocation() when possible.
This call can be safely used within interrupts.

  BUGS
The function will also return NULL in case the logical address
is translated to the address 0L. However, 0L should never be
used as physical address anyhow.

  SEE ALSO
GetPageProperties(), PhysicalLocation()
```

1.32 mmu.library/PhysicalLocation

```
  NAME
PhysicalLocation    -   translate logical address to physical

  SYNOPSIS
props = PhysicalLocation( context, addrptr, lenptr );
d0          d1       a0       a1

ULONG PhysicalLocation( struct MMUContext * , void ** , ULONG * );

  FUNCTION
This function finds the physical address for the logical address
passed in by scanning the software abstraction layer.
If the physical address is not available, NULL is returned.

  INPUTS
context - the context to enter or NULL for the current context.
addrptr - points to the logical address to be translated.
    The physical address is filled in here, or NULL in case
    the logical address passed in is not available.
lenptr  - Points to the length of the address range to be trans-
    lated. The function returns the length of the largest
    possible continous memory range contained in the memory
    range passed in. Hence, this function may shorten the
    memory block for fragmentized memory models. It will
    NULL in case the memory is not available.

  RETURNS
the properties of the memory range.

  NOTES
This is the high-level function, it is not callable from within
interrupts.

In case you've to operate on a range of physical memory, start
the translation with this call, then compare the size returned
with the size of the memory block passed in. Because this
function may shorten the memory size in case the physical
memory is fragmentated, you should be prepared that the size
returned is smaller than what was passed in. In this case, operate
on the memory region returned, then add the returned size to
the original logical address and call this function again to
```

```
get the physical location of the next chunk.

  BUGS
The function will also return NULL in case the logical address
is translated to the address 0L. However, 0L should never be
used as physical address anyhow.

  SEE ALSO
GetProperties(), PhysicalPageLocation()
```

1.33 mmu.library/DMAInitiate

```
  NAME
DMAInitiate     -   start a DMA transport given a logical address.

  SYNOPSIS
fine = DMAInitiate( context, addrptr, lenptr, write );
         d1       a0   a1     d0

BOOL DMAInitiate( struct MMUContext * , void ** , ULONG * , BOOL );

  FUNCTION
This function finds the physical address for the logical address
passed in by scanning a backup of the MMU translation tree.
It ignores modifications made by the high-level and low-level
functions unless RebuildTree() is called.

  INPUTS
context - the context to enter or NULL for the current context.
    NOTE: This parameter is currently a dummy and should be
    set to NULL. The mmu.library will always use the public
    context for translation.
addrptr - points to the logical address to be translated.
    The physical address is filled in here.
lenptr  - Points to the length of the address range to be trans-
    lated. The function returns the length of the largest
    possible continous memory range contained in the memory
    range passed in. Hence, this function may shorten the
    memory block for fragmentized memory models.
write   - set this to TRUE for transports from a DMA device INTO
    the memory, i.e. device reads. Set this to FALSE for
    writes from memory to the device.

  RETURNS
fine    - TRUE if the address and length passed in pointed to available
    memory. FALSE if the requested DMA transfer was invalid.

The result code is new for V42, do not check it for V41 or below.

  NOTES
The function checks whether the memory range passed in is available
for DMA. It will guru in case it is not, i.e. the page is either
swapped out, invalid, indirect, or write protected for
DMA device reads. Reads into ROM addresses are silently tolerated,
and, hence, are translations to and from blank dummy pages.
```

```
This function is callable from within interrupts, but does only
use a backup of the high-level table for its translation.
Changes to the software abstraction level are not visible for
this function unless RebuildTree() is called. Changes to the
hardware level are not at all visible to this (and all other
high-level) functions.

In case you've to operate on a range of physical memory, start
the translation with this call, then compare the size returned
with the size of the memory block passed in. Because this
function may shorten the memory size in case the physical
memory is fragmentated, you should be prepared that the size
returned is smaller than what was passed in. With the physical
address returned, start the DMA and call DMATerminate() when
done.
In case the returned size is smaller than the block passed in,
add the returned size to the original logical address and
call this function again to get the physical location of the
next chunk.

EACH CALL TO DMAInitiate() must be matched by ONE AND PRECISELY
ONE call to DMATerminate().

Even though this function does not require locking the context,
I highly recommend doing so. It won't crash if you don't, but
someone else could modify the MMU translation table in between.
The library can deal with that, but the result of the DMA
operation might be different than what you expect.

The pre-V42 releases did not return a result code but generated
an alert in case illegal memory has been passed in. This changed
for V42. DO NOT assume a meaningful result code for V41 or
below.

  BUGS
This function should really use the context passed in, but
since most (if not all) DMA device drivers do not keep the
context of the task that actually initiated the transfer and
hence would use the wrong context anyhow, DMA is currently limited
to the public context.

  SEE ALSO
DMATerminate(), PhysicalPageLocation(), exec/CachePreDMA()
```

1.34 mmu.library/DMATerminate

```
  NAME
DMATerminate    -   end a DMA transfer initiated by DMAInitiate.

  SYNOPSIS
DMATerminate( context );
    d1

void DMATerminate( struct MMUContext * );
```

```
  FUNCTION
This function ends a DMA transfer initiated by DMAInitate. It
releases the resources by the first call.

  INPUTS
context - the context to enter or NULL for the current context.
    NOTE: This parameter is currently a dummy and should be
    set to NULL. The mmu.library will always use the public
    context for translation.

  RETURNS

  NOTES
This function is callable from within interrupts, but does only
use a backup of the high-level table for its translation.
Changes to the software abstraction level are not visible for
this function unless RebuildTree() is called. Changes to the
hardware level are not at all visible to this (and all other
high-level) functions.

EACH CALL TO DMAInitiate() must be matched by ONE AND PRECISELY
ONE call to DMATerminate().

For details, check the DMAInitiate() function.

  BUGS
This function should really use the context passed in, but
since most (if not all) DMA device drivers do not keep the
context of the task that actually initiated the transfer and
hence would use the wrong context anyhow, DMA is currently limited
to the public context.

  SEE ALSO
DMAInitiate(), PhysicalPageLocation(), exec/CachePostDMA()
```

1.35 mmu.library/GetMapping

```
  NAME
GetMapping  -   get access to the memory map of a MMUContext

  SYNOPSIS
list = GetMapping( context );
d0         a0

struct MinList * GetMapping( struct MMUContext * );

  FUNCTION
This function makes a copy of the MapNodes for the given context.
The nodes in this list describe the memory map as seen from tasks
attached to this context, sorted by logical addresses.
The list must be released afterwards with ReleaseMapping().

  INPUTS
context - the context to enter or NULL for the current context.
```

```
  RETURNS
a pointer to a struct MinList which contains the MapNodes for this
context, sorted by physical address, or NULL in case of failure.

  NOTES
The nodes are just a copy of the real nodes within the context.

This function is most useful to make a backup of the context
memory map before altering it. In case any of the modifications
fail, you are able to undo all modifications completely with
a call to SetPropertyList() - which can't fail.

To give an example:

/* make a backup of the context how it looks now */

LockMMUContext(ctx);

if (list=GetMapping(ctx)) {
    fine=TRUE;

    /* Try to alter it, step by step. */

    if (!SetProperties(...))
  fine=FALSE;

    if (!SetProperties(...))
  fine=FALSE;

    /* etc, etc.... */

    /* Oops, we failed! Re-install the old setup. */
    if (!fine)
  SetPropertyList(ctx,list);
}

ReleaseMapping(ctx,list);
UnlockMMUContext(ctx);
/* and so on... */

Note that you've still to call ReleaseContextList(), even in
case of failure when you've already re-installed backup property
list.

  BUGS

  SEE ALSO
ReleaseMapping(), SetPropertyList(), mmu/context.h
```

1.36 mmu.library/ReleaseMapping

```
  NAME
ReleaseMapping  -   get access to the memory map
```

```
  SYNOPSIS
ReleaseMapping( context , list );
     a0 a1

void ReleaseMapping( struct MMUContext * , struct MinList * );

  FUNCTION
This function releases the list of MapNodes arbitrated by
GetMapping.

  INPUTS
context - the context the nodes where taken from.
list    - the backup property list to release.

  RETURNS

  NOTES
This function *MUST* be called, even in case the property list
was re-installed with SetPropertyList().

  BUGS

  SEE ALSO
NewMapping(), GetMapping(), SetPropertyList()
```

1.37 mmu.library/NewMapping

```
  NAME
NewMapping    -   build a new memory map

  SYNOPSIS
list = NewMapping ( );
d0

struct MinList * NewMapping ( void );

  FUNCTION
Build and initialize a new memory map list. All addresses in this
list will be marked as MAPP_BLANK.

  INPUTS
nothing.

  RESULTS
a MinList structure, initialized with MapNodes representing a
completely blank memory layout or NULL on failure. You either need
to copy the layout from a context with CopyContextRegion(), or
define your own layout with calls to SetMappingProperties().

  NOTES
Don't forget to release the list (and its contents) with
ReleaseMapping() when you're done.

  BUGS
```

```
  SEE ALSO
CopyContextRegion(), SetMappingProperties(), ReleaseMapping(),
GetMapping()
```

1.38 mmu.library/CopyMapping

```
  NAME
CopyMapping  -   transfer memory properties between lists

  SYNOPSIS
fine = CopyMapping ( from , to , base , length , mask );
d0          a0    a1    d0      d1      d2


BOOL CopyMapping ( struct MinList * , struct MinList * ,

         ULONG , ULONG , ULONG );

  FUNCTION
Copy the memory properties from one memory map to another, thru
a mask.

  INPUTS
from    -   the memory map which is (partially) to be transfered.
to      -   the destination of the copy operation.
base    -   base address. Memory properties will be copied starting
      at this address.
length  -   length of the memory region in bytes whose properties
      shall be transfered.
mask    -   a mask of property bits which are to be transfered. A
      zero bit in this mask indicates that the corresponding
      property in the destination will not be touched. For
      all the properties, check the SetProperties() function.

  RESULTS
a boolean success/failure indicator. It is TRUE in case the
operation was performed, FALSE otherwise. The destination will
not have been touched at all in this case.

  NOTES
this call does not copy memory. It just copies memory attributes
from one memory map to another.

Check CopyContextRegion() to copy the properties from a context
instead from a list.

Since this call is not context based, the library will not be able
to perform checks for correct page alignment, you have to do that
yourself. Especially, note that SetPropertyList() - which attaches
a memory map to a context - does not perform any check on this
list either. Hence, *NOT* checking for page alignment here might
result in an invalid context if you try to attach an incorrectly
aligned list to a context later on.

  BUGS
```

```
  SEE ALSO
SetPropertyList(), ReleaseMapping(), DupMapping(), SetProperties(),
CopyContextRegion()
```

1.39 mmu.library/DupMapping

```
  NAME
DupMapping    -   make a one-to-one copy of a memory map

  SYNOPSIS
dup = DupMapping ( list );
d0         a0

struct MinList * DupMapping( struct MinList * );

  FUNCTION
this call builds an identical copy of the memory map passed in.

  INPUTS
list    -   the memory map to be copied.

  RESULTS
another memory list, identical to the list passed in, or NULL on
failure.

  NOTES
Don't forget to release the memory list with ReleaseMapping()
if you're done with it. You need to release both, the original as
well as the duplicate.
In case you want to make a copy of the memory map of a context,
use GetMapping() instead.

  BUGS

  SEE ALSO
GetMapping(), ReleaseMapping()
```

1.40 mmu.library/CopyContextRegion

```
  NAME
CopyContextRegion   -   transfer properties from a context to a list

  SYNOPSIS
fine = CopyContextRegion ( ctx, list, base, length, mask );
d0           a0   a1    d0    d1      d2

BOOL CopyContextRegion ( struct MMUContext *, struct MinList *,

       ULONG, ULONG, ULONG );

  FUNCTION
```

```
Copy the properties of a memory region defined by a context to
another memory map, thru a mask.

  INPUTS
ctx     -   source context whose memory map shall be transfered.
list    -   the destination memory map which is to be altered.
base    -   base address of the memory region whose properties are
      to be copied.
length  -   length of the memory region in bytes whose properties
      will be transfered.
mask    -   a mask of property bits, see the SetProperties()
      function for a detailed explanation. A zero bit in this
      mask means that the corresponding property in the
      destination will be left alone and will remain unchanged.

  RESULTS
a boolean success/failure indicator, TRUE for success. On failure,
the destination memory map will not have been touched at all.

  NOTES
This call does not copy memory at all, it just defines the memory
properties of a given memory map from that of a given context.

Check CopyMapping() to transport memory properties from one list
to another.

Since this call is not context based, the library will not be able
to perform checks for correct page alignment, you have to do that
yourself. Especially, note that SetPropertyList() - which attaches
a memory map to a context - does not perform any check on this
list either. Hence, *NOT* checking for page alignment here might
result in an invalid context if you try to attach an incorrectly
aligned list to a context later on.

  BUGS

  SEE ALSO
SetPropertyList(), ReleaseMapping(), SetProperties(),
SetPropertiesMapping()
```

1.41 mmu.library/SetPropertiesMapping

```
  NAME
SetPropertiesMapping    -   transfer properties from a map list
          to a context

  SYNOPSIS
fine = SetPropertiesMapping ( ctx, list, base, length, mask );
d0          a0   a1    d0     d1      d2

BOOL SetPropertiesMapping ( struct MMUContext *, struct MinList *,

       ULONG, ULONG, ULONG );

  FUNCTION
```

```
Copy the properties of a memory map to a context, thru a mask.
This is equivalent to SetProperties(), except that the source
data is contained in a memory map instead given as function
arguments. This function is reverse to CopyContextRegion().

  INPUTS
ctx     -   destination context whose memory map shall be set.
list    -   the source memory map, containing the data to be
      transfered.
base    -   base address of the memory region whose properties are
      to be copied.
length  -   length of the memory region in bytes whose properties
      will be transfered.
mask    -   a mask of property bits, see the SetProperties()
      function for a detailed explanation. A zero bit in this
      mask means that the corresponding property in the
      destination will be left alone and will remain unchanged.

  RESULTS
a boolean success/failure indicator, TRUE for success. On failure,
the destination context will not have been touched at all.

  NOTES
This call does not copy memory at all, it just defines the memory
properties of the context passed in.

Check CopyMapping() to transport memory properties from one list
to another, or CopyContextRegion() to transfer properties from a
context to a list (the other direction).

The library will not be able to perform checks for correct page
alignment, you have to do that yourself.
  BUGS

  SEE ALSO
SetPropertyList(), ReleaseMapping(), SetProperties(),
CopyContextRegion()
```

1.42 mmu.library/SetMappingProperties

```
  NAME
SetMappingPropertiesA   -   set memory attributes in a memory map.

  SYNOPSIS

result = SetMappingPropertiesA( list, flags, mask, lower, size, tags);
d0         a0     d1    d2    a1     d0    a2

int SetMappingPropertiesA( struct MinList *, ULONG, ULONG,
         ULONG, ULONG, struct TagItem *);


result = SetMappingProperties( list, flags, mask,
             lower, size, tag1, ...);
```

```
int SetMappingProperties( struct MinList *, ULONG, ULONG,
        ULONG, ULONG, Tag tag1, ...);

  FUNCTION
This call sets attributes of a certain memory range of a given
memory map.

  INPUTS
list    - a minlist structure keeping the memory map to be altered.
flags   - a binary flags field for the attributes to define. Check
    SetProperties() for details about the defined bits.
mask    - A bit mask of the attributes to be changed.
lower   - The lower boundary of the logical address to be modified.
size    - Size of the region to be modified.
tags    - A tag array with additional data, identical to the
    tags defined for SetProperties().

  RESULTS
Unlike SetProperties() or SetPageProperties(), this does not
return a boolean value! The result code is 0 on failure, and
different from zero on success, though. To be more precise, this
routine will return "1" in case of success, and "2" in case the
memory map was really altered and is now "dirty", hence upper
software layers might require a "rebuild".

  NOTES
This call really doesn't do anything to the MMU, it is just an
administration call to modify a memory map - a handy data structure
you might want to use for your own memory administration. Its
context-based equivalent SetProperties() will, hence, adjust the
memory map which is kept by a context, and SetPageProperties()
will perform the same operation truely on the hardware.

Since this call is not context based, the library will not be able
to perform checks for correct page alignment, you have to do that
yourself. Especially, note that SetPropertyList() - which attaches
a memory map to a context - does not perform any check on this
list either. Hence, *NOT* checking for page alignment here might
result in an invalid context if you try to attach an incorrectly
aligned list to a context later on.

  BUGS

  SEE ALSO
SetPropertyList(), ReleaseMapping(), SetProperties(),
SetPageProperties(), GetMappingProperties()
```

1.43 mmu.library/GetMappingProperties

```
  NAME
GetMappingPropertiesA   - read memory attributes from a memory map.

  SYNOPSIS

flags = GetMappingPropertiesA( list, lower, tags);
```

```
d0        a0    a1     a2

ULONG GetMappingPropertiesA( struct MinList *, ULONG,
           struct TagItem *);

result = GetMappingProperties( list, lower, tag1, ...);

ULONG GetMappingProperties( struct MinList *, ULONG, Tag tag1, ...);

  FUNCTION
This call reads the page properties of a certain address in
memory from a memory map. It is the counterpart of
SetMappingProperties() and the memory map analogue of
GetProperties().

  INPUTS
list    -   a MinList holding the memory map, obtained from either
      GetMapping(), DupMapping() or NewMapping().
lower   -   the logical address of the address to investigate.
tags    -   additional tags, identical to those defined for
      GetProperties(). Check the documentation of this
      function for details.

  RESULTS
Returns a binary flags field for the attributes to define. See
SetProperties() for details.

  NOTES
This call is the analogue of the GetProperties() call. It operates
directly on memory maps, unlike the former which operates only on
the memory map of a context. This function does not require page
alignment because the mmu.library does not have a context to check
the alignment restrictions, but you should note that a memory map
that is to be attached to a context with DefineMapping() *HAS* to
be correctly aligned.

This routine is *NOT* safe to be called from within interrupts.

  BUGS

  SEE ALSO
SetMappingProperties(), SetProperties(), GetPageProperties(),
GetProperties()
```

1.44 mmu.library/SetPropertyList

```
  NAME
SetPropertyList     -   re-install a backup memory map

  SYNOPSIS
SetPropertyList ( context, list );
        a0      a1

void SetPropertyList ( struct MMUContext * , struct MinList * );
```

```
  FUNCTION
This call re-installes a property list, i.e. a complete memory
map of a context, obtained from GetMapping() before.

  INPUTS
context     -   the context the list should be installed in.
    This should be the same context the list was
    taken from.
list  -   the property list to install.

This list *MUST* have been obtained with GetMapping() before.

  RESULTS
Nothing. The big advantage of this call is that it cannot fail.

  NOTES
The property list will become part of the context and is empty
after this call. You can't re-use it for that reason. However,
you still need to call ReleaseMapping() with the list pointer
you've obtained before.

For additional tips how this function should be used, see the
GetMapping() function; especially, you can only un-do changes
to the software abstraction level of a MMU-tree, and only as
long as you haven't called RebuildTree() to translate these
into hardware MMU tables. Trying to un-do these changes with
SetPropertyList() will fail, and it will even fail if you call
RebuildTree() afterwards. SetPropertyList() *does not* inform
the software abstraction level about any changes, it is just a
quick un-do operation. (For the experts: It even re-installs
the "dirty" flags).

  BUGS

  SEE ALSO
GetMapping(), ReleaseMapping(), RebuildTree()
```

1.45 mmu.library/GetMMUType

```
  NAME
GetMMUType - return the type of the MMU available in the system.

  SYNOPSIS
mmu = GetMMUType( );


char GetMMUType( void );

  FUNCTION
Returns an identifier for the MMU available in the system or
NUL in case no MMU is installed.

  INPUTS

  RETURNS
```

```
a character identifying the MMU type:

MUTYPE_NONE   no working MMU detected.
MUTYPE_68851    a 68020 system with an external 68851
      MMU.
MUTYPE_68030    a 68030 MMU.
MUTYPE_68040    the internal 68040 MMU.
MUTYPE_68060    the 68060 MMU.

  NOTES
The mmu library is smart enough to detect EC processors without a
working MMU, but the library does not detect multiple CPUs in the
system. (How?)

  BUGS

  SEE ALSO
mmu/mmubase.h
```

1.46 mmu.library/SuperContext

```
  NAME
SuperContext - find the supervisor context for a given context.

  SYNOPSIS
super = SuperContext( context );
d0      a0

struct MMUContext * SuperContext( struct MMUContext * );

  FUNCTION
Returns the context that manages the supervisor mode for the user
mode context passed in.

  INPUTS
A user mode context or NULL for the current context.

  RETURNS
A pointer to the context managing the supervisor type accesses with-
in the current context.

  NOTES
All contexts build by CreateMMUContext are by default user mode
contexts. The current version of the library manages one global
supervisor tree, and optionally private supervisor trees for private
contexts if you ask for one on context creation. This is different
to former releases!

To find the public supervisor mode context, call DefaultContext()
first and pass in its return value to this function.

  BUGS

  SEE ALSO
```

```
DefaultContext(), CreateMMUContext()
```

1.47 mmu.library/DefaultContext

```
  NAME
DefaultContext  -   get the global default context

  SYNOPSIS
public = DefaultContext( );
d0

struct MMUContext * DefaultContext( void );

  FUNCTION
Returns the global default user mode context which is used for
tasks that are not attached to any other private context.

  INPUTS

  RETURNS
A pointer to the context managing the user mode accesses for
"context less" tasks.

  NOTES
A task is by default part of this default context unless you
call EnterMMUContext() and attach it to a different context.

Note that you might have to enter even the default context
explicitly to be able to use certain features of the exception
hook mechanism.

  BUGS

  SEE ALSO
SuperContext()
```

1.48 mmu.library/WithoutMMU

```
  NAME
WithoutMMU - execute a short subroutine with the MMU disabled.

  SYNOPSIS
result = WithoutMMU( userFunc );
d0      a5

ULONG WithoutMMU(void *);

  FUNCTION
Executes a small assembly language routine pointed to in a5
in supervisor mode, with all interrupts disabled, and the MMU
disabled. All registers are preserved by this call.
The function must end with an RTS instruction.
```

```
  INPUTS
userFunc    -   A pointer to a *short* assembly language routine,
    ending with RTS. The function has full access to
    all registers.

  RETURNS
whatever was left in register d0 by the called function.

  NOTES
This is a low-level function. Remember that disabling the MMU
might or might not be what you want, especially if memory is
remapped.

Note that this function works even without a MMU. It just calls the
routine in a5 in this case.

  BUGS
Big trouble if the supervisor stack is in remapped memory.

  SEE ALSO
exec/Supervisor(), RunOldConfig()
```

1.49 mmu.library/RunOldConfig

```
  NAME
RunOldConfig - execute a short subroutine with the old MMU configuration.   (V42 ↩
   )

  SYNOPSIS
result = RunOldConfig( userFunc );
d0        a5

ULONG RunOldConfig(void *);

  FUNCTION
Executes a small assembly language routine pointed to in a5
in supervisor mode, with all interrupts disabled, and the MMU
configuration the library found when it was started. This could
or could not be a disabled MMU.
The function must end with an RTS instruction.

  INPUTS
userFunc    -   A pointer to a *short* assembly language routine,
    ending with RTS. The function has full access to
    all registers.

  RETURNS
whatever was left in register d0 by the called function.

  NOTES
This is a low-level function. Remember that reloading the MMU
with the last MMU configuration might or might not be what
you want, especially if memory is remapped or the MMU tables of
the old configuration has been released.
```

```
Note that this function works even without a MMU. It just calls the
routine in a5 in this case.

  BUGS
Big trouble if the supervisor stack is in remapped memory.

  SEE ALSO
exec/Supervisor(), WithoutMMU()
```

1.50 mmu.library/SetBusError

```
  NAME
SetBusError - define the bus error handler.

  SYNOPSIS
SetBusError ( newfuncptr , oldfuncptrptr );
d0    a0        a1


void SetBusError ( void (*)() , void (**)() );

  FUNCTION
This defines the bus error handler which is called in case the
MMU library exception handler was not able to handle the fault.

This happens for true physical bus errors, errors initiated by
the unsupported TAS, CAS and CAS2 instructions using "locked
transfers" not supported by the amiga hardware, and MOVE16 in-
structions causing an access fault.

The default bus error handler is whatever the library finds in
the autovectors of the CPU when starting up. It is usually the
exec fault handler which presents the nice guru 80000002.

  INPUTS
newfuncptr    - A pointer to the new bus error handler. It is
    called without any parameters in supervisor state,
    with the exception stack frame on the stack.
oldfuncptrptr - A pointer to a pointer filled in with the previously
    defined handler, or NULL.

  RETURNS

  NOTES
The function pointer is guaranteed to be flushed to memory by
the library, the function pointer pointer could be used, for
example, to modify the destination of a JMP instruction.

The bus error handler *MUST BE IMMEDIATELY* ready for run after
this function has to be called.

This is a low-level function. You do not want to call it.

  BUGS
```

```
  SEE ALSO
```

1.51 mmu.library/GetMMUContextData

```
  NAME
GetMMUContextData - read MMUContext specific data

  SYNOPSIS
GetMMUContextData ( ctx , id );
d0        a0    d0

ULONG GetMMUContextData ( struct MMUContext * , ULONG );

  FUNCTION
This function reads various parameters of the MMU context,
indexed by an ID from mmu/mmutags.h. All legal tag items
of CreateMMContext() are available, plus the following:

    MGXTAG_PAGESIZE     -       Return the page size in bytes
        of the input context. Unlike
        MCXTAG_PAGEBITS, this is *NOT*
        an exponent. This is identical
        to GetPageSize().

    MGXTAG_REMAPSIZE    -       Returns the alignment restriction
        for remapped memory to be added to
        the exec memory free list. This is
        identical to RemapSize().

    MGXTAG_ROOT   -       Return the pointer to the root of
        the MMU tree. You have to cast the
        result to an ULONG *.
        Note that you DO NOT WANT to modify
        this tree directly.

    MGXTAG_CONFIG       -       Returns the pointer to a
        struct MMUConfig * specifying the
        complete MMU configuration for
        this context. You usually DO NOT
        NEED to touch this.

  INPUTS
ctx     -       A pointer to a struct MMUContext *.

id      -       An ID specifying which data you want to read.
    NOTE THAT THIS IS NOT A TAG LIST, but just the
    tag item id from mmutags.h.
  RETURNS
the data requested. You need to cast it to the correct type.

  NOTES

  BUGS
```

```
  SEE ALSO
CreateMMUContext(), mmu/config.h
```

1.52 mmu.library/SetMMUContextData

```
  NAME
SetMMUContextData - define MMU context specifications on the fly

  SYNOPSIS
SetMMUContextDataA ( ctx , tags );
         a0    a1

SetMMUContextData ( ctx , ... );


void SetMMUContextDataA ( struct MMUContext * , struct TagItem * );

void SetMMUContextData ( struct MMUContext * , Tag tag1 , ... );

  FUNCTION
This function allows to adjust *some* of the MMUContext
specifications on the fly. The following tag items of the
CreateMMUContext() call are supported:

    MCXTAG_BLANKFILL    -       define the data to read from the
        dummy "blank" pages. Note that
        a *working* program should never
        read this data.

    MCXTAG_EXECBASE     -       specify whether or not page 0 will
        be threated specially and accesses
        to the first Kbyte should be
        emulated. If set to FALSE, the
        first page will be threated as all
        other pages.

    MCXTAG_ZEROBASE     -       Specify a base address for where
        possibly emulated zero page accesses
        have to be redirected to. This address
        is usually ignored unless the zero-
        page is invalidated and MCXTAG_EXEBASE
        is TRUE. For the messy details, check
        the CreateMMUContext() autodocs.

    MCXTAG_LOWMEMORYLIMIT -     Define the lower boundary of valid
        memory, used to emulate access to the
        zero page. Defaults to the lower
        boundary of chip memory. (V42)

No other tag items are valid here.

  INPUTS
ctx     -       A pointer to a struct MMUContext *.

tags    -       A tag list containing the parameters to be
```

```
    adjusted.
  RETURNS

  NOTES

  BUGS

  SEE ALSO
CreateMMUContext(), exec/memory.h
```

1.53 mmu.library/BuildIndirect

```
  NAME
BuildIndirect       -   build a true hardware page descriptor.

  SYNOPSIS
descr = BuildIndirect ( ctx , address , props );
d0      a0      d0  d1

ULONG   BuildIndirect ( struct MMUContext * , ULONG , ULONG );

  FUNCTION
This function builds a true hardware page descriptor, to be used for
the MAPP_INDIRECT property.

  INPUTS
ctx     -   the MMUContext handle in which this descriptor is to
      be used as the destination of a MAPP_INDIRECT descriptor.

address -   in case this descriptor is "valid", this is the
      PHYSICAL destination address, to be told to the MMU, to
      which accesses will be redirected to. Unlike
      SetProperties(), there is *NO* MAPP_REMAPPED bit. In
      case you do not want remapping, set this to the logical
      address. DO NOT LEAVE THIS BLANK or accesses will be
      redirected to address 0 - which is most likely not want
      you want. Note that this address must be, as all other
      addresses the MMU cares about, page aligned!
      In case this descriptor is of invalid type, the address
      can be used for your purposes, but note that this MUST
      STILL be a number which is page aligned for the given
      context. It *MAY NOT* be arbitrary.

props   -   Properties for the descriptor to be defined.
      NOTE THAT THIS IS A TRUE HARDWARE DESCRIPTOR! The
      library WILL NOT BE ABLE to help you out by emulating
      missing features of one MMU or another. Instead, it will
      just ignore them. This means for you, specifically,
      that only a minor subset of the usual properties may be
      used safely here, and that parsing the hardware
      properties later on with GetIndirect() might result in
      different properties than you intended because of
      missing features of the MMU in use. To be precise, the
      following properties *might* be available - meaning that
      all others ARE NOT!
```

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

MAPP_USED      -   The "used" bit of the pages
    will be set. The CPU will set this bit automatically
    as soon as the pages are accessed.

MAPP_MODIFIED      -   The "modified" bit of the pages
    will be set. The CPU will set this bit automatically
    as soon as a write is performed to the page in question.
    DO NOT SET THIS BIT TOGETHER WITH MAPP_WRITEPROTECTED
    OR WITHOUT MAPP_USED or the CPU might hang.

MAPP_INVALID       -   The page will be marked as
    invalid. Accessing it will invoke the bus error hook.
    See below for how to mark this page as REMAIRABLE. Note
    that the MAPP_REPAIRABLE bit is *here* not available.

MAPP_CACHEINHIBIT    -   The page will be marked as non-
    cacheable.

MAPP_IMPRECISE       -   The page will be marked as
    "imprecise exception". MAPP_CACHEINHIBIT is mandatory
    in this case or this flag does nothing. Only avail-
    able for the 060, ignored and read as zero
    by all others.

MAPP_NONSERIALIZED   -   The page will be marked as
    serialized. MAPP_CACHEINHIBIT is mandatory if this
    property is selected. Only available for the 040,
    ignored and read as zero by all others.

MAPP_COPYBACK      -   The page will be marked as
    "copyback" instead of "writethrough". Generally re-
    commended since this is faster for the '40 and '60.
    MAPP_CACHEINHIBIT *MUST* be disabled for this to work.
    Only available for the 040 and 060, ignored and read
    as zero by others.

MAPP_USERPAGE0       -   Set user page attribute 0,
    only available for the 040 and 060, ignored and read
    as zero by all others.
    The status of this bit appears on special pins of the
    CPU and might be required by some hardware, so don't
    play with this.

MAPP_USERPAGE1       -   Set user page attribute 1,
    see above for details.

MAPP_GLOBAL      -   DIFFERENT TO SetProperties()
    and others!
    Set the GLOBAL bit in the page descriptor, only avail-
    able for the 040 and 060, ignored and read as zero by
    all others.
    Setting this bit means that certain specialized
    instructions will not flush this descriptor from the
```

cache (the ATC) of the MMU.
The mmu.library writes only descriptors without this
bit set and does not use these instructions. It will
always flush descriptors independent of the G bit.
There is little use of this bit.

RESULTS
a page descriptor for the current MMU in use, designed and to be used for as the destination of an MAPP_INDIRECT descriptor. NOT to be used as a true page descriptor, and NOT to be used as a table descriptor.
In case the library finds no MMU or the alignment restrictions aren't satisfied, it will return BAD_DESCRIPTOR (0x03), it WILL NOT return NULL as this is a "valid invalid" descriptor.

NOTES
Note specifically that MAPP_SUPERVISORONLY *IS NOT* supported. The mmu.library enforces a distinct user/supervisor model and as such you might want to install an invalid descriptor into the user table and a valid descriptor into the supervisor table to emulate this feature. True "supervisor only" descriptors are available for the 040 and 060 anyways, but this "emulation" works for all MMUs.

MAPP_REMAPPED is not supported either because you have to specify a physical destination address in all cases, even if no remapping has to be performed. Use the logical address as physical address in case remapping is not desired.

MAPP_REPAIRABLE is not available because this flag is in fact an emulation provided by the library. However, you may make access faults to this page repairable by setting the MAPP_REPAIRABLE bit for the MAPP_INDIRECT descriptor that POINTS to the descriptor you're building by this call. This will be enough to inform the library about how to treat access faults.

How to use this function:

Allocate four bytes of memory, long-word aligned, or even 16 bytes line (16 byte) aligned in case you want to read back the descriptor later by GetIndirect() and calculate its TRUE PHYSICAL location with PhysicalLocation(). Use the return code of this function to mask in your properties, DO NOT ASSUME fixed properties. Build a descriptor with BuildIndirect() and install it into this memory by calling SetIndirect(). *DO NOT* write it to the memory yourself!
Due to CPU caching effects, this must be done by the library.
Then build a MAPP_INDIRECT descriptor, and tell the library with the MAPTAG_DESCRIPTOR tag of SetProperties() to make it point to your memory. In case you want access faults on this to be repairable, set the MAPP_REPAIRABLE bit for this call. In case you want it write protected but want to ignore write accesses, set MAPP_ROM, too. Then call RebuildTree() as usual. In case you want to exchange descriptors really fast - this is after all what indirect descriptors are designed for - build all descriptors you require in a first step and keep them. A single call to SetIndirect() will exchange them VERY RAPIDLY which is ideal for certain applications.

```
  BUGS
Much more must be said about this function. It is definitely an
advanced feature, so don't play with this in case you don't know
what it does.

  SEE ALSO
SetIndirect(), SetIndirectArray(), GetIndirect()
```

1.54 mmu.library/SetIndirect

```
  NAME
SetIndirect     -   Write a page descritor to memory

  SYNOPSIS
SetIndirect ( destination , logical , descriptor );
    a0        a1    d0

void SetIndirect ( ULONG *, ULONG, ULONG );

  FUNCTION
Write a page descriptor, used as the destination of one or
several MAPP_INDIRECT descriptors, out to memory and make the MMU
aware of the change.

  INPUTS
destination     -   the memory location to which the descriptor
        should be written to. This is the same address
        specified by MAPTAD_DESCRIPTOR in the corresponding
        SetProperties() call. NOTE THAT THIS IS A
        PHYSICAL, NOT A LOGICAL ADDRESS.
        This must be long-word aligned, or even 16 bytes
        line-aligned (16 bytes aligned) in case you want
        to read the descriptor later with GetIndirect().

logical   -   The logical address covered by this descriptor,
        i.e. the address of the page which this
        descriptor is managing. In case you installed
        the same descriptor for several logical addresses,
        specify -1L (this will be slower, though.)

descriptor      -   The descriptor to install.

  RESULTS
doesn't return a result.

  NOTES
Do NOT try to install a descriptor yourself, even though this
*seems* to be more effective. Somewhat more must be done than just
writing the descriptor to memory. This call ensures that the
descriptor is really written out to memory, and really fetched by
the MMU. Specifying -1L as logical address is possible and supported
but *slightly* less efficient than giving the correct
logical address.
This call DOES NOT ensure that all data in the page to be modified
is really written out. Hence, if you change the cache mode, the
```

```
protection status or the validity status of a page, you should call
CacheClearU() or CacheClearE() before. This step is, however, not
required required in case you just changed the physical destination.
The library keeps then care about the necessary cache operations
itself.

This call is *very* effective, there is little reason to try this
yourself, not counting the portability problems.

In case you want to install more than about four descriptors at once,
you should consider using SetIndirectArray() which causes even less
overhead in this situation.

  BUGS

  SEE ALSO
SetIndirectArray(), GetIndirect(), BuildIndirect()
```

1.55 mmu.library/SetIndirectArray

```
  NAME
SetIndirectArray    -   Write multiple page descritors to memory

  SYNOPSIS
SetIndirectArray ( destination , descriptors , number );
      a0    a1    d0

void SetIndirectArray ( ULONG *, ULONG *, ULONG );

  FUNCTION
Write a complete array of page descriptors at once, re-defining
the mapping for more than one page, and make the MMU aware of the
changes.

  INPUTS
destination     -   an array in memory which keeps the true hardware
        MMU descriptors. This address, and all subsequent
        addresses must have been used in the corresponding
        SetProperties() call to setup the descriptor.
        NOTE THAT THIS IS A PHYSICAL ADDRESS. Note further
        that it is up to you to ensure that the
        (obviously continuous) array of logical addresses
        is not fragmentated into several physical memory
        blocks. If this happens, you would have to call
        this routine several times, once for each dis-
        continuous block.
        This must be long-word aligned, or even a multiple
        of a 16 byte line-aligned array in case you want
        to read the descriptor later with GetIndirect().

descriptors     -   An array of pre-calculated descriptor values to be
        filled in. This is a logical address. Effectively,
        this call copies this array to its first argument.

number      -   The number of the descriptors to be defined. Zero
```

```
        is allowed here, and is a no-op.

  RESULTS
doesn't return a result.

  NOTES
Do NOT try to install descriptors yourself, even though this
*seems* to be more effective. Somewhat more must be done than just
writing the descriptors to memory. This call ensures that the
descriptors are really written out to memory, and really fetched by
the MMU. Note that this call does not require the logical address
of the page(s) which are about to be changed, quite different to
SetIndirect(). It will be therefore slower than SetIndirect() if
only very descriptors are to be written.
This call DOES NOT ensure that all data in the page(s) to be modified
is really written out. Hence, if you change the cache mode, the
protection status or the validity status of a page, you should call
CacheClearU() or CacheClearE() before.
This step is not required in case you just change the physical
destination of the pages involved, the library is able to handle
this transparently.

This call is *very* effective, there is little reason to try this
yourself, not counting the portability problems.

If you install only one descriptor, SetIndirect() is more effective.
Note further that this call does not require the logical address of
the page(s) to be provided, it will flush the complete ATC of the
MMU and is therefore slower if only a few pages or even a single page
has to be modified.

  BUGS

  SEE ALSO
SetIndirect(), GetIndirect(), BuildIndirect()
```

1.56 mmu.library/GetIndirect

```
  NAME
GetIndirect     -   Read a hardware page descritor from memory

  SYNOPSIS
GetIndirect ( ctx , adt , address );
        a0    a1      d0

void GetIndirect ( struct MMUContext *,
       struct AbstractDescriptor *, ULONG * );

  FUNCTION
Reads a true hardware page descriptor, used as the destination
of one or several MAPP_INDIRECT descriptors, and places its data
in the AbstractDescriptor structure.

  INPUTS
ctx   -   the MMU context handle this page descriptor
```

```
        belongs.
adt   -   An abstract table descriptor, to be filled
        out with the descriptor data.
address   -   the address from where the descriptor is to
        be read. This is the same address which has been
        passed as argument to the MAPTAG_DESCRIPTOR
        tag of the SetProperties() call when building
        the indirect descriptor.
        All descriptors to be read by this function must
        reside in a separate cache line, i.e. must be
        part of a line (16 byte) aligned array which is a
        multiple of 16 bytes long. In all other cases,
        this call could return improper results due to
        cache clashes.

  RESULTS
no direct result code, but the AbstractDescriptor is filled
in as follows:

struct AbstractDescriptor {
    ULONG       atd_Pointer;
    ULONG       atd_Properties;
    UWORD       atd_LowerLimit;
    UWORD       atd_UpperLimit;
    UBYTE       atd_ThisType;
    UBYTE       atd_NextType;
    UWORD       atd_reserved;
};

atd_Pointer is either the physical address the accesses to the
page(s) this descriptor is installed for are redirected to, or
the user data if this descriptor is of invalid type. This is the
same value that was passed in as "address" argument to the
corresponding BuildIndirect() call.

atd_Properties is the set of MMU properties read from the
descriptor. This NEED NOT to be identical to the properties setup
by BuildIndirect(), for two reasons: First, the MMU sets the
USED and MODIFIED attributes as soon as any access or a write
access happens to the page(s) handled by the descriptor. Second,
not all MMUs support all properties. Unavailable properties are
ignored by BuildIndirect(), and read as zero by this function.

All other fields are currently not documented and should not be
read.

  NOTES
Do NOT try to read a descriptor yourself, even though this
*seems* to be more effective. Somewhat more care must be kept for
doing this. NOT following this rule might even lock up your
machine!

This call is *very* effective, there is little reason to try this
yourself, not counting the portability problems.

  BUGS
```

```
    SEE ALSO
SetIndirect(), BuildIndirect(), mmu/descriptor.h
```