Libraries

COLLABORATORS			
	TITLE : Libraries		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 20 Exec Memory Allocation

Exec manages all of the free memory currently available in the system. Using linked list structures, Exec keeps track of memory and provides the functions to allocate and access it.

When an application needs some memory, it can either declare the memory statically within the program or it can ask Exec for some memory. When Exec receives a request for memory, it looks through its list of free memory regions to find a suitably sized block that matches the size and attributes requested.

1.2 20 Exec Memory Allocation / Memory Functions

Normally, an application uses the AllocMem() function to ask for memory:

```
APTR AllocMem(ULONG byteSize, ULONG attributes);
```

The byteSize argument is the amount of memory the application needs and attributes is a bit field which specifies any special memory characteristics (described later). If AllocMem() is successful, it returns a pointer to a block of memory. The memory allocation will fail if the system cannot find a big enough block with the requested attributes. If AllocMem() fails, it returns NULL.

Because the system only keeps track of how much free memory is available and not how much is in use, it has no idea what memory has been allocated by any task. This means an application has to explicitly return, or deallocate, any memory it has allocated so the system can return that memory to the free memory list. If an application does not return a block of memory to the system, the system will not be able to reallocate that memory to some other task. That block of memory will be lost until the Amiga is reset. If you are using AllocMem() to allocate memory, a call to FreeMem() will return that memory to the system:

```
    void FreeMem(APTR mymemblock, ULONG byteSize);
```

Here mymemblock is a pointer to the memory block the application is returning to the system and byteSize is the same size that was passed when the memory was allocated with AllocMem().

Unlike some compiler memory allocation functions, the Amiga system memory allocation functions return memory blocks that are at least longword aligned. This means that the allocated memory will always start on an address which is at least evenly divisible by four. This alignment makes the memory suitable for any system structures or buffers which require word or longword alignment, and also provides optimal alignment for stacks and memory copying.

1.3 20 / Memory Functions / Memory Attributes

When asking the system for memory, an application can ask for memory with certain attributes. The currently supported flags are listed below. Flags marked "V37" are new memory attributes for Release 2. Allocations which specify these new bits may fail on earlier systems.

MEMF_ANY
: This indicates that there is no requirement for either Fast or Chip memory. In this case, while there is Fast memory available, Exec will only allocate Fast memory. Exec will allocate Chip memory if there is not enough Fast memory.

MEMF_CHIP
: This indicates the application wants a block of chip memory, meaning it wants memory addressable by the Amiga custom chips. Chip memory is required for any data that will be accessed by custom chip DMA. This includes screen memory, images that will be blitted, sprite data, copper lists, and audio data, and pre-V37 floppy disk buffers. If this flag is not specified when allocating memory for these types of data, your code will fail on machines with expanded memory.

MEMF_FAST
: This indicates a memory block outside of the range that the special purpose chips can access. "FAST" means that the special-purpose chips do not have access to the memory and thus cannot cause processor bus contention, therefore processor access will likely be faster. Since the flag specifies memory that the custom chips cannot access, this flag is mutually exclusive with the MEMF_CHIP flag. If you specify the MEMF_FAST flag, your allocation will fail on any Amiga that has only CHIP memory. Use MEMF_ANY if you would prefer FAST memory.

```
MEMF_PUBLIC
    This indicates that the memory should be accessible to other tasks.
    Although this flag doesn't do anything right now, using this flag
    will help ensure compatibility with possible future features of the
    OS (like virtual memory and memory protection).

MEMF_CLEAR
    This indicates that the memory should be initialized with zeros.

MEMF_LOCAL (V37)
    This indicates memory which is located on the motherboard which is
    not initialized on reset.

MEMF_24BITDMA (V37)
    This indicates that the memory should be allocated within the 24 bit
    address space, so that the memory can be used in Zorro-II expansion
    device DMA transactions.  This bit is for use by Zorro-II DMA devices
    only.  It is not for general use by applications.

MEMF_REVERSE (V37)
    Indicates that the memory list should be searched backwards for the
    highest address memory chunk which can be used for the memory
    allocation.

If an application does not specify any attributes when allocating memory,
the system tries to satisfy the request with the first memory available on
the system memory lists, which is MEMF_FAST if available, followed by
MEMF_CHIP.

    Make Sure You Have Memory.
    --------------------------
    Always check the result of any memory allocation to be sure the type
    and amount of memory requested is available.  Failure to do so will
    lead to trying to use an non-valid pointer.
```

1.4 20 / Memory Functions / Allocating System Memory

```
The following examples show how to allocate memory.

    APTR  apointer,anotherptr, yap;

    if (!(apointer = AllocMem(100, MEMF_ANY)))
        {  /* COULDN'T GET MEMORY, EXIT */ }

AllocMem() returns the address of the first byte of a memory block that is
at least 100 bytes in size or NULL if there is not that much free memory.
Because the requirement field is specified as MEMF_ANY (zero), memory will
be allocated from any one of the system-managed memory regions.

    if (!(anotherptr = (APTR)AllocMem(1000, MEMF_CHIP | MEMF_CLEAR)))
        {  /* COULDN'T GET MEMORY, EXIT */ }

The example above allocates only chip-accessible memory, which the system
fills with zeros before it lets the application use the memory.  If the
system free memory list does not contain enough contiguous memory bytes in
```

an area matching your requirements, AllocMem() returns a zero. You must check for this condition.

If you are using Release 2, you can use the AllocVec() function to allocate memory. In addition to allocating a block of memory, this function keeps track of the size of the memory block, so your application doesn't have to remember it when it deallocates that memory block. The AllocVec() function allocates a little more memory to store the size of the memory allocation request.

```
if (!(yap = (APTR)AllocVec(512, MEMF_CLEAR)))
    {  /* COULDN'T GET MEMORY, EXIT */ }
```

1.5 20 / Memory Functions / Freeing System Memory

The following examples free the memory chunks shown in the previous calls to AllocMem().

```
FreeMem(apointer, 100);
FreeMem(anotherptr, 1000);
```

A memory block allocated with AllocVec() must be returned to the system pool with the FreeVec(). This function uses the stored size in the allocation to free the memory block, so there is no need to specify the size of the memory block to free.

```
FreeVec(yap);
```

FreeMem() and FreeVec() return no status. However, if you attempt to free a memory block in the middle of a chunk that the system believes is already free, you will cause a system crash. Applications must free the same size memory blocks that they allocated. An allocated block may not be deallocated as smaller pieces. Due to the internal way the system rounds up and aligns allocations. Partial deallocations can corrupt the system memory list.

```
Leave Memory Allocations Out Of Interrupt Code.
-----------------------------------------------
```

Do not allocate or deallocate system memory from within interrupt code. The "Exec Interrupts" chapter explains that an interrupt may occur at any time, even during a memory allocation process. As a result, system data structures may not be internally consistent at this time.

1.6 20 / Memory Functions / Memory Information Functions

The memory information routines AvailMem() and TypeOfMem() can provide the amount of memory available in the system, and the attributes of a particular block of memory.

Memory Requirements Calling Memory Information Functions

1.7 20 / / Memory Information Functions / Memory Requirements

The same attribute flags used in memory allocation routines are valid for the memory information routines. There is also an additional flag, MEMF_LARGEST, which can be used in the AvailMem() routine to find out what the largest available memory block of a particular type is. Specifying the MEMF_TOTAL flag will return the total amount of memory currently available.

1.8 20 / / Memory Info Functions / Calling Memory Information Functions

The following example shows how to find out how much memory of a particular type is available.

```
    ULONG size;

    size = AvailMem(MEMF_CHIP|MEMF_LARGEST);
```

AvailMem() returns the size of the largest chunk of available chip memory.

```
    AvailMem() May Not Be Totally Accurate.
    ---------------------------------------
    Because of multitasking, the return value from AvailMem() may be
    inaccurate by the time you receive it.
```

The following example shows how to determine the type of memory of a specified memory address.

```
    ULONG memtype;

    memtype = TypeOfMem((APTR)0x090000);
    if ((memtype & MEMF_CHIP) == MEMF_CHIP) {/* ...It's chip memory... */}
```

TypeOfMem() returns the attributes of the memory at a specific address. If it is passed an invalid memory address, TypeOfMem() returns NULL. This routine is normally used to determine if a particular chunk of memory is in chip memory.

1.9 20 / Memory Functions / Using Memory Copy Functions

For memory block copies, the CopyMem() and CopyMemQuick() functions can be used.

Copying System Memory

1.10 20 / / Using Memory Copy Functions / Copying System Memory

The following samples show how to use the copying routines.

```
    APTR source, target;

    source = AllocMem(1000, MEMF_CLEAR);
    target = AllocMem(1000, MEMF_CHIP);
    CopyMem(source, target, 1000);
```

CopyMem() copies the specified number of bytes from the source data region to the target data region. The pointers to the regions can be aligned on arbitrary address boundaries. CopyMem() will attempt to copy the memory as efficiently as it can according to the alignment of the memory blocks, and the amount of data that it has to transfer. These functions are optimized for copying large blocks of memory which can result in unnecessary overhead if used to transfer very small blocks of memory.

```
    CopyMemQuick(source, target, 1000);
```

CopyMemQuick() performs an optimized copy of the specified number of bytes from the source data region to the target data region. The source and target pointers must be longword aligned and the size (in bytes) must be divisible by four.

```
    Not All Copies Are Supported.
    -----------------------------
    Neither CopyMem() nor CopyMemQuick() supports copying between
    regions that overlap.
```

1.11 20 / / Summary of System Controlled Memory Handling Routines

```
AllocMem() and FreeMem()
    These are system-wide memory allocation and deallocation routines.
    They use a memory free-list owned and managed by the system.

AvailMem()
    This routine returns the number of free bytes in a specified type of
    memory.

TypeOfMem()
    This routine returns the memory attributes of a specified memory
    address.

CopyMem()/CopyMemQuick()
    CopyMem() is a general purpose memory copy routine. CopyMemQuick() is
    an optimized version of CopyMemQuick(), but has restrictions on the
    size and alignment of the arguments.
```

1.12 20 Exec Memory Allocation / Allocating Multiple Memory Blocks

Exec provides the routines AllocEntry() and FreeEntry() to allocate multiple memory blocks in a single call. AllocEntry() accepts a data structure called a MemList, which contains the information about the size of the memory blocks to be allocated and the requirements, if any, that you have regarding the allocation. The MemList structure is found in the

include file <exec/memory.h> and is defined as follows:

```
    struct MemList
    {
        struct Node     ml_Node;
        UWORD           ml_NumEntries;      /* number of MemEntrys */
        struct MemEntry ml_ME[1];           /* where the MemEntrys begin*/
    };
```

Node
    allows you to link together multiple MemLists. However, the node is ignored by the routines AllocEntry() and FreeEntry().

ml_NumEntries
    tells the system how many MemEntry sets are contained in this MemList. Notice that a MemList is a variable-length structure and can contain as many sets of entries as you wish.

The MemEntry structure looks like this:

```
    struct MemEntry
    {
        union {
            ULONG   meu_Reqs;   /* the AllocMem requirements */
            APTR    meu_Addr;   /* address of your memory */
            } me_Un;
        ULONG   me_Length;      /* the size of this request */
    };
```


1.13 20 / Allocating Multiple Memory Blocks / Sample Code

Here's an example of showing how to use the AllocEntry() with multiple blocks of memory.

allocentry.c

AllocEntry() returns a pointer to a new MemList of the same size as the MemList that you passed to it. For example, ROM code can provide a MemList containing the requirements of a task and create a RAM-resident copy of the list containing the addresses of the allocated entries. The pointer to the MemList is used as the argument for FreeEntry() to free the memory blocks.

Assembly Does Not Have MemEntry.

The MemList structure used by assembly programmers is slightly different; it has only a label for the start of the MemEntry array. See the Exec AllocEntry() Autodoc for an example of using AllocEntry() from assembler.

1.14 20 / Allocating Multiple Memory Blocks / Result

The MemList created by AllocEntry() contains MemEntry entries. MemEntrys are defined by a union statement, which allows one memory space to be defined in more than one way.

If AllocEntry() returns a value with bit 31 clear, then all of the meu_Addr positions in the returned MemList will contain valid memory addresses meeting the requirements you have provided. To use this memory area, you would use code similar to the following:

```
    #define ALLOCERROR 0x80000000
    struct  MemList *ml;
    APTR    data, moredata;

    if ( ! ((ULONG)ml & ALLOCERROR)))    /* After calling AllocEntry to */
                                         /* allocate ml */
    {
        data     = ml->ml_ME[0].me_Addr;
        moredata = ml->ml_ME[1].me_Addr;
    }
    else  exit(200);                     /* error during AllocEntry */
```

If AllocEntry() has problems while trying to allocate the memory you have requested, instead of the address of a new MemList, it will return the memory requirements value with which it had the problem. Bit 31 of the value returned will be set, and no memory will be allocated. Entries in the list that were already allocated will be freed. For example, a failed allocation of cleared Chip memory (MEMF_CLEAR | MEMF_CHIP) could be indicated with 0x80010002, where bit 31 indicates failure, bit 16 is the MEMF_CLEAR flag and bit 1 is the MEMF_CHIP flag.

1.15 20 / Allocating Multiple Memory / Multiple Memory Blocks and Tasks

If you want to take advantage of Exec's automatic cleanup, use the MemList and AllocEntry() facility to do your dynamic memory allocation.

In the Task control block structure, there is a list header named tc_MemEntry. This is the list header that you initialize to include MemLists that your task has created by call(s) to AllocEntry(). Here is a short program segment that handles task memory list header initialization only. It assumes that you have already run AllocEntry() as shown in the simple AllocEntry() example above.

```
    struct Task *tc;
    struct MemList *ml;

    /* First initialize the task pointer and AllocEntry() the memlist ml */

    if(!tc->tc_MemEntry)
        NewList(tc->tc_MemEntry);  /* Initialize the task's memory    */
                                   /* list header. Do this once only! */
    AddTail(tc->tc_MemEntry, ml);
```

Assuming that you have only used the AllocEntry() method (or AllocMem() and built your own custom MemList), the system now knows where to find the blocks of memory that your task has dynamically allocated. The RemTask() function automatically frees all memory found on tc_MemEntry.

CreateTask() Sets Up A MemList.

The amiga.lib CreateTask() function, and other system task and process creation functions use a MemList in tc_MemEntry so that the Task structure and stack will be automatically deallocated when the Task is removed.

1.16 20 / Allocating Multiple Memory / Summary of Allocation Routines

AllocEntry() and FreeEntry()
These are routines for allocating and freeing multiple memory blocks with a single call.

InitStruct()
This routine initializes memory from data and offset values in a table. Typically only assembly language programs benefit from using this routine. For more details see the Amiga ROM Kernel Reference Manual: Include & Autodocs.

1.17 20 Exec Memory Allocation / Other Memory Functions

Allocate() and Deallocate() use a memory region header, called MemHeader, as part of the calling sequence. You can build your own local header to manage memory locally. This structure takes the form:

```
struct MemHeader {
    struct Node       mh_Node;
    UWORD             mh_Attributes; /* characteristics of region  */
    struct  MemChunk *mh_First;      /* first free region          */
    APTR              mh_Lower;      /* lower memory bound         */
    APTR              mh_Upper;      /* upper memory bound + 1     */
    ULONG             mh_Free;       /* total number of free bytes */
};
```

mh_Attributes
is ignored by Allocate() and Deallocate().

mh_First
is the pointer to the first MemChunk structure.

mh_Lower
is the lowest address within the memory block. This must be a multiple of eight bytes.

mh_Upper
is the highest address within the memory block + 1. The highest address will itself be a multiple of eight if the block was allocated

```
    to you by AllocMem().

mh_Free
    is the total free space.

This structure is included in the include files <exec/memory.h> and
<exec/memory.i>.

The following sample code fragment shows the correct initialization of a
MemHeader structure.  It assumes that you wish to allocate a block of
memory from the global pool and thereafter manage it yourself using
Allocate() and Deallocate().

     allocate.c

    How Memory Is Tagged.
    ---------------------
    Only free memory is "tagged" using a MemChunk linked list. Once
    memory is allocated, the system has no way of determining which task
    now has control of that memory.

If you allocate memory from the system, be sure to deallocate it when your
task exits.  You can accomplish this with matched deallocations, or by
adding a MemList to your task's tc_MemEntry, or you can deallocate the
memory in the finalPC routine (which can be specified if you perform
AddTask() yourself).

 Allocating Memory at an Absolute Address
 Adding Memory to the System Pool
```

1.18 20 / Other Memory Functions / Allocating Memory at an Absolute Address

```
For special advanced applications, AllocAbs() is provided. Using
AllocAbs(), an application can allocate a memory block starting at a
specified absolute memory address.  If the memory is already allocated or
if there is not enough memory available for the request, AllocAbs()
returns a zero.

Be aware that an absolute memory address which happens to be available on
one Amiga may not be available on a machine with a different configuration
or different operating system revision, or even on the same machine at a
different times.  For example, a piece of memory that is available during
expansion board configuration might not be available at earlier or later
times.  Here is an example call to AllocAbs():

    APTR absoluteptr;

    absoluteptr = (APTR)AllocAbs(10000, 0x2F0000);
    if (!(absoluteptr))
        { /* Couldn't get memory, act accordingly. */  }

    /* After we're done using it, we call FreeMem() to free the memory */
    /* block. */
    FreeMem(absoluteptr, 10000);
```

1.19 20 / Other Memory Functions / Adding Memory to the System Pool

When non-Autoconfig memory needs to be added to the system free pool, the AddMemList() function can be used. This function takes the size of the memoryblock, its type, the priority for the memory list, the base address and the name of the memory block. A MemHeader structure will be placed at the start of the memory block, the remainder of the memory block will be made available for allocation. For example:

```
AddMemList(0x200000, MEMF_FAST, 0, 0xF00000, "FZeroBoard");
```

will add a two megabyte memory block, starting at $F00000 to the system free pool as Fast memory. The memory list entry is identified with "FZeroBoard".

1.20 20 Exec Memory Allocation / Function Reference

The following are brief descriptions of the Exec functions that handle memory management. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details on each call.

Table 20-1: Exec Memory Functions

Memory Function	Description
AllocMem()	Allocate memory with specified attributes. If an application needs to allocate some memory, it will usually use this function.
AddMemList()	Add memory to the system free pool.
AllocAbs()	Allocate memory at a specified location.
Allocate()	Allocate memory from a private memory pool.
AllocEntry()	Allocate multiple memory blocks.
AllocVec()	Allocate memory with specified attributes and keep track of the size (V36).
AvailMem()	Return the amount of free memory, given certain conditions.
CopyMem()	Copy memory block, which can be non-aligned and of arbitrary length.
CopyMemQuick()	Copy aligned memory block.
Deallocate()	Return memory block allocated, with Allocate() to the private memory pool.
FreeEntry()	Free multiple memory blocks, allocated with AllocEntry().
FreeMem()	Free a memory block of specified size, allocated with AllocMem().
FreeVec()	Free a memory block allocated with AllocVec().
InitStruct()	Initialize memory from a table.
TypeOfMem()	Determine attributes of a specified memory address.