exec

COLLABORATORS

	TITLE : exec		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

exec

1.1 exec.doc

AbortIO()	Debug()	PutMsg()
AddDevice()	DeleteIORequest()	RawDoFmt()
AddHead()	DeleteMsgPort()	ReleaseSemaphore()
AddIntServer()	Disable()	ReleaseSemaphoreList()
AddLibrary()	DoIO()	RemDevice()
AddMemList()	Enable()	RemHead()
AddPort()	Enqueue()	RemIntServer()
AddResource()	FindName()	RemLibrary()
AddSemaphore()	FindPort()	Remove()
AddTail()	FindResident()	RemPort()
AddTask()	FindSemaphore()	RemResource()
Alert()	FindTask()	RemSemaphore()
AllocAbs()	Forbid()	RemTail()
Allocate()	FreeEntry()	RemTask()
AllocEntry()	FreeMem()	ReplyMsg()
AllocMem()	FreeSignal()	SendIO()
AllocSignal()	FreeTrap()	SetExcept()
AllocTrap()	FreeVec()	SetFunction()
AllocVec()	GetCC()	SetIntVector()
AttemptSemaphore()	GetMsg()	SetSignal()
AvailMem()	InitCode()	SetSR()
CacheClearE()	InitResident()	SetTaskPri()
CacheClearU()	InitSemaphore()	Signal()
CacheControl()	InitStruct()	StackSwap()
CachePostDMA()	Insert()	SumKickData()
CachePreDMA()	MakeFunctions()	SumLibrary()
Cause()	MakeLibrary()	SuperState()
CheckIO()	ObtainSemaphore()	Supervisor()
CloseDevice()	ObtainSemaphoreList()	TypeOfMem()
CloseLibrary()	ObtainSemaphoreShared()	UserState()
ColdReboot()	OldOpenLibrary()	Vacate()
CopyMem()	OpenDevice()	Wait()
CopyMemQuick()	OpenLibrary()	WaitIO()
CreateIORequest()	OpenResource()	WaitPort()
CreateMsgPort()	Permit()	
Deallocate()	Procure()	

1.2 exec.library/AbortIO

NAME
    AbortIO - attempt to abort an in-progress I/O request

SYNOPSIS
    AbortIO(iORequest)
            A1

    VOID AbortIO(struct IORequest *);

FUNCTION
    Ask a device to abort a previously started IORequest. This is done
    by calling the device's ABORTIO vector, with your given IORequest.

    AbortIO is a command the device that may or may not grant. If
    successful, the device will stop processing the IORequest, and
    reply to it earlier than it would otherwise have done.

NOTE
    AbortIO() does NOT Remove() the IORequest from your ReplyPort, OR
    wait for it to complete. After an AbortIO() you must wait normally
    for the reply message before actually reusing the request.

    If a request has already completed when AbortIO() is called, no
    action is taken.

EXAMPLE
        AbortIO(timer_request);
        WaitIO(timer_request);
        /* Message is free to be reused */

INPUTS
    iORequest - pointer to an I/O request block (must have been used
            at least once. May be active or finished).

SEE ALSO
    WaitIO, DoIO, SendIO, CheckIO

1.3 exec.library/AddDevice

NAME
    AddDevice -- add a device to the system

SYNOPSIS
    AddDevice(device)
            A1

    void AddDevice(struct Device *);

FUNCTION
    This function adds a new device to the system device list, making
    it available to other programs. The device must be ready to be

```
    opened at this time.
```

INPUTS
```
    device - pointer to a properly initialized device node
```

SEE ALSO
```
    RemDevice, OpenDevice, CloseDevice, MakeLibrary
```

1.4 exec.library/AddHead

NAME
```
    AddHead -- insert node at the head of a list
```

SYNOPSIS
```
    AddHead(list, node)
            A0    A1

    void AddHead(struct List *, struct Node *)
```

FUNCTION
```
    Add a node to the head of a doubly linked list. Assembly
    programmers may prefer to use the ADDHEAD macro from
    "exec/lists.i".
```

WARNING
```
    This function does not arbitrate for access to the list.  The
    calling task must be the owner of the involved list.
```

INPUTS
```
    list - a pointer to the target list header
    node - the node to insert at head
```

SEE ALSO
```
    AddTail, Enqueue, Insert, Remove, RemHead, RemTail
```

1.5 exec.library/AddIntServer

NAME
```
    AddIntServer -- add an interrupt server to a system server chain
```

SYNOPSIS
```
    AddIntServer(intNum, interrupt)
                 D0-0:4  A1

    void AddIntServer(ULONG, struct Interrupt *);
```

FUNCTION
```
    This function adds a new interrupt server to a given server chain.
    The node is located on the chain in a priority dependent position.
    If this is the first server on a particular chain, interrupts will
    be enabled for that chain.
```

Each link in the chain will be called in priority order until the
chain ends or one of the servers returns with the 68000's Z condition
code clear (indicating non-zero). Servers on the chain should return
with the Z flag clear if the interrupt was specifically for that
server, and no one else. VERTB servers should always return Z set.
(Take care with High Level Language servers, the language may not
have a mechanism for reliably setting the Z flag on exit).

Servers are called with the following register conventions:

```
D0 - scratch
D1 - scratch

A0 - scratch
A1 - server is_Data pointer (scratch)

A5 - jump vector register (scratch)
A6 - scratch

all other registers must be preserved
```

INPUTS
    intNum - the Paula interrupt bit number (0 through 14). Processor
            level seven interrupts (NMI) are encoded as intNum 15.
            The PORTS, COPER, VERTB, EXTER and NMI interrupts are
            set up as server chains.
    interrupt - pointer to an Interrupt structure.
            By convention, the LN_NAME of the interrupt structure must
            point a descriptive string so that other users may
            identify who currently has control of the interrupt.

WARNING
    Some compilers or assemblers may optimize code in unexpected ways,
    affecting the conditions codes returned from the function. Watch
    out for a "MOVEM" instruction (which does not affect the condition
    codes) turning into "MOVE" (which does).

BUGS
    The graphics library's VBLANK server, and some user code, currently
    assume that address register A0 will contain a pointer to the custom
    chips. If you add a server at a priority of 10 or greater, you must
    compensate for this by providing the expected value ($DFF000).

SEE ALSO
    RemIntServer, SetIntVector, hardware/intbits.i,exec/interrupts.i

1.6   exec.library/AddLibrary

NAME
    AddLibrary -- add a library to the system

SYNOPSIS
    AddLibrary(library)
            A1

```
        void AddLibrary(struct Library *);
```

FUNCTION
    This function adds a new library to the system, making it available
    to other programs. The library should be ready to be opened at
    this time. It will be added to the system library name list, and
    the checksum on the library entries will be calculated.

INPUTS
    library - pointer to a properly initialized library structure

SEE ALSO
    RemLibrary, CloseLibrary, OpenLibrary, MakeLibrary

1.7 exec.library/AddMemList

NAME
    AddMemList - add memory to the system free pool

SYNOPSIS
    AddMemList(size, attributes, pri, base, name)
                D0   D1   D2   A0   A1

    void AddMemList(ULONG, ULONG, LONG, APTR, STRPTR);

FUNCTION
    Add a new region of memory to the system free pool. The first few
    bytes will be used to hold the MemHeader structure. The remainder
    will be made available to the rest of the world.

INPUTS
    size - the size (in bytes) of the memory area
    attributes - the attributes word that the memory pool will have
    pri - the priority for this memory. CHIP memory has a pri of -10,
           16 bit expansion memory has a priority of 0. The higher the
           priority, the closer to the head of the memory list it will
           be placed.
    base - the base of the new memory area
    name - the name that will be used in the memory header, or NULL
           if no name is to be provided. This name is not copied, so it
           must remain valid for as long as the memory header is in the
           system.

SEE ALSO
    AllocMem, exec/memory.h

1.8 exec.library/AddPort

NAME
    AddPort -- add a public message port to the system

SYNOPSIS

```
AddPort(port)
        A1

void AddPort(struct MsgPort *);
```

FUNCTION
    This function attaches a message port structure to the system's
    public message port list, where it can be found by the FindPort()
    function. The name and priority fields of the port structure must
    be initialized prior to calling this function. If the user does
    not require the priority field, it should be initialized to zero.

    Only ports that will be searched for with FindPort() need to
    be added to the system list. In addition, adding ports is often
    useful during debugging. If the port will be searched for,
    the priority field should be at least 1 (to avoid the large number
    of inactive ports at priority zero). If the port will be searched
    for often, set the proritiry in the 50-100 range (so it will be
    before other less used ports).

    Once a port has been added to the naming list, you must be careful
    to remove the port from the list (via RemPort) before deallocating
    its memory.

NOTE
    A point of confusion is that clearing a MsgPort structure to all
    zeros is not enough to prepare it for use. As mentioned in the
    Exec chapter of the ROM Kernel Manual, the List for the MsgPort
    must be initialized. This is automatically handled by AddPort(),
    and amiga.lib/CreatePort. This initialization can be done manually
    with amiga.lib/NewList or the assembly NEWLIST macro.

    Do not AddPort an active port.

INPUTS
    port - pointer to a message port

SEE ALSO
    RemPort, FindPort, amiga.lib/CreatePort, amiga.lib/NewList

1.9 exec.library/AddResource

NAME
    AddResource -- add a resource to the system

SYNOPSIS
    AddResource(resource)
                A1

    void AddResource(APTR);

FUNCTION
    This function adds a new resource to the system and makes it
    available to other users. The resource must be ready to be called
    at this time.

Resources currently have no system-imposed structure, however they
must start with a standard named node (LN_SIZE), and should with
a standard Library node (LIB_SIZE).

INPUTS
    resource - pointer an initialized resource node

SEE ALSO
    RemResource, OpenResource, MakeLibrary

1.10 exec.library/AddSemaphore

NAME
    AddSemaphore -- initialize then add a signal semaphore to the system

SYNOPSIS
    AddSemaphore(signalSemaphore)
                A1

    void AddSemaphore(struct SignalSemaphore *);

FUNCTION
    This function attaches a signal semaphore structure to the system's
    public signal semaphore list. The name and priority fields of the
    semaphore structure must be initialized prior to calling this
    function. If you do not want to let others rendezvous with this
    semaphore, use InitSemaphore() instead.

    If a semaphore has been added to the naming list, you must be
    careful to remove the semaphore from the list (via RemSemaphore)
    before deallocating its memory.

    Semaphores that are linked together in an allocation list (which
    ObtainSemaphoreList() would use) may not be added to the system
    naming list, because the facilities use the link field of the
    signal semaphore in incompatible ways

INPUTS
   signalSemaphore -- an signal semaphore structure

BUGS
    Does not work in Exec <V36. Instead use this code:

```
#include <exec/execbase.h>
#include <exec/nodes.h>
extern struct ExecBase *SysBase;
    ...
void LocalAddSemaphore(s)
struct SignalSemaphore *s;
{
    s->ss_Link.ln_Type=NT_SIGNALSEM;
    InitSemaphore(s);
    Forbid();
    Enqueue(&SysBase->SemaphoreList,s);
```

```
        Permit();
    }
```

SEE ALSO
    RemSemaphore, FindSemaphore, InitSemaphore

1.11 exec.library/AddTail

NAME
    AddTail -- append node to tail of a list

SYNOPSIS
    AddTail(list, node)
            A0   A1

    void AddTail(struct List *, struct Node *);

FUNCTION
    Add a node to the tail of a doubly linked list. Assembly
    programmers may prefer to use the ADDTAIL macro from
    "exec/lists.i".

WARNING
    This function does not arbitrate for access to the list. The
    calling task must be the owner of the involved list.

INPUTS
    list - a pointer to the target list header
    node - a pointer to the node to insert at tail of the list

SEE ALSO
    AddHead, Enqueue, Insert, Remove, RemHead, RemTail

1.12 exec.library/AddTask

NAME
    AddTask -- add a task to the system

SYNOPSIS
    AddTask(task, initialPC, finalPC)
            A1   A2   A3

    APTR AddTask(struct Task *, APTR, APTR);

FUNCTION
    Add a task to the system. A reschedule will be run; the task with
    the highest priority in the system will start to execute (this may
    or may not be the new task).

    Certain fields of the task control block must be initialized and a
    stack allocated prior to calling this function. The absolute
    smallest stack that is allowable is something in the range of 100
```

bytes, but in general the stack size is dependent on what
subsystems are called. In general 256 bytes is sufficient if only
Exec is called, and 4K will do if anything in the system is called.
DO NOT UNDERESTIMATE.  If you use a stack sniffing utility,
leave a healthy pad above the minimum value.  The system guarantees
that its stack operations will leave the stack longword aligned.

This function will temporarily use space from the new task's stack
for the task's initial set of registers.  This space is allocated
starting at the SPREG location specified in the task control block
(not from SPUPPER).  This means that a task's stack may contain
static data put there prior to its execution.  This is useful for
providing initialized global variables or some tasks may want to
use this space for passing the task its initial arguments.

A task's initial registers are set to zero (except the PC).

The TC_MEMENTRY field of the task structure may be extended by
the user to hold additional MemLists (as returned by AllocEntry()).
These will be automatically be deallocated at RemTask() time.
If the code you have used to start the task has already added
something to the MEMENTRY list, simply use AddHead to add your
new MemLists in.  If no initialization has been done, a NewList will
need to be performed.

INPUTS
    task  - pointer to the task control block (TCB).  All unset fields
            must be zero.
    initialPC - the initial entry point's address
    finalPC - the finalization code entry point's address.  If zero,
            the system will use a general finalizer. This pointer is
            placed on the stack as if it were the outermost return
            address.

RESULTS
    For V36, AddTask returns either a NULL or the address of the new
    task.  Old code need not check this.

WARNING
    Tasks are a low-level building block, and are unable to call
    dos.library, or any system function that might call dos.library.
    See the AmigaDOS CreateProc() for information on Processes.

SEE ALSO
    RemTask, FindTask, amiga.lib/CreateTask, dos/CreateProc,
    amiga.lib/NewList


## 1.13  exec.library/Alert

NAME
    Alert -- alert the user of an error

SYNOPSIS
    Alert(alertNum)
        D7

```
    void Alert(ULONG);
```

FUNCTION
    Alerts the user of a serious system problem. This function will
    bring the system to a grinding halt, and do whatever is necessary
    to present the user with a message stating what happened.
    Interrupts are disabled, and an attempt to post the alert is made.
    If that fails, the system is reset. When the system comes up
    again, Exec notices the cause of the failure and tries again to
    post the alert.

    If the Alert is a recoverable type, this call MAY return.

    This call may be made at any time, including interrupts.

POST-MORTEM DIAGNOSIS
    There are several options for determining the cause of a crash.
    Descriptions of each alert number can be found in the "alerts.h"
    include file. Low numbers not mentioned in the include file
    represent 68000 exceptions, see a 68000 manual for details. The
    most common numbers are:
                    $00000003 - Address Error
                    $00000004 - Illegal Instruction

    A remote terminal can be attached to the Amiga's first built-in
    serial port. Set the communication parameters to 9600 baud, 8 bits,
    no parity. Before resetting the machine, the Alert function will
    blink the power LED 10 times. While the power indicator is flashing,
    pressing DELETE on the remote terminal will invoke the ROMWack
    debugger.

    For Alerts caused by a 68000 exception, all registers are copied
    to a magic low memory location (currently 16 longwords at $180).

INPUT
    alertNum   - a number indicating the particular alert. -1 is
            not a valid input.

NOTE
    Much more needs to be said about this function and its implications.

SEE ALSO
    exec/alerts.h

1.14 exec.library/AllocAbs

NAME
    AllocAbs -- allocate at a given location

SYNOPSIS
    memoryBlock = AllocAbs(byteSize, location)
    D0   D0   A1

    void *AllocAbs(ULONG, APTR);

FUNCTION
    This function attempts to allocate memory at a given absolute
    memory location. Often this is used by boot-surviving entities
    such as recoverable ram-disks. If the memory is already being
    used, or if there is not enough memory to satisfy the request,
    AllocAbs will return NULL.

    This block may not be exactly the same as the requested block
    because of rounding, but if the return value is non-zero, the block
    is guaranteed to contain the requested range.

INPUTS
    byteSize - the size of the desired block in bytes
               This number is rounded up to the next larger
               block size for the actual allocation.
    location - the address where the memory MUST be.

RESULT
    memoryBlock - a pointer to the newly allocated memory block, or
                  NULL if failed.

NOTE
    If the free list is corrupt, the system will panic with alert
    AN_MemCorrupt, $01000005.

    The 8 bytes past the end of an AllocAbs will be changed by Exec
    relinking the next block of memory. Generally you can't trust
    the first 8 bytes of anything you AllocAbs.

SEE ALSO
    AllocMem, FreeMem

1.15 exec.library/Allocate

NAME
    Allocate - allocate a block of memory

SYNOPSIS
    memoryBlock=Allocate(memHeader, byteSize)
    D0   A0   D0

    void *Allocate(struct MemHeader *, ULONG);

FUNCTION
    This function is used to allocate blocks of memory from a given
    private free memory pool (as specified by a MemHeader and its
    memory chunk list). Allocate will return the first free block that
    is greater than or equal to the requested size.

    All blocks, whether free or allocated, will be block aligned;
    hence, all allocation sizes are rounded up to the next block even
    value (e.g. the minimum allocation resolution is currently 8
    bytes. A request for 8 bytes will use up exactly 8 bytes. A

    request for 7 bytes will also use up exactly 8 bytes.).

    This function can be used to manage an application's internal data
    memory. Note that no arbitration of the MemHeader and associated
    free chunk list is done. You must be the owner before calling
    Allocate.

INPUTS
    memHeader - points to the local memory list header.
    byteSize - the size of the desired block in bytes.

RESULT
    memoryBlock - a pointer to the just allocated free block.
          If there are no free regions large enough to satisfy the
          request, return zero.

EXAMPLE
```
    #include <exec/types.h>
    #include <exec/memory.h>
    void *AllocMem();
    #define BLOCKSIZE 4096L /* Or whatever you want */

    void main()
    {
    struct MemHeader *mh;
    struct MemChunk  *mc;
    APTR   block1;
    APTR   block2;

        /* Get the MemHeader needed to keep track of our new block */
        mh = (struct MemHeader *)
            AllocMem((long)sizeof(struct MemHeader), MEMF_CLEAR );
        if( !mh )
            exit(10);

        /* Get the actual block the above MemHeader will manage */
        mc = (struct MemChunk *)AllocMem( BLOCKSIZE, 0L );
        if( !mc )
            {
            FreeMem( mh, (long)sizeof(struct MemHeader) ); exit(10);
            }

        mh->mh_Node.ln_Type = NT_MEMORY;
        mh->mh_Node.ln_Name = "myname";
        mh->mh_First = mc;
        mh->mh_Lower = (APTR) mc;
        mh->mh_Upper = (APTR) ( BLOCKSIZE + (ULONG) mc );
        mh->mh_Free  = BLOCKSIZE;

        /* Set up first chunk in the freelist */
        mc->mc_Next  = NULL;
        mc->mc_Bytes = BLOCKSIZE;

        block1 = (APTR) Allocate( mh, 20L );
        block2 = (APTR) Allocate( mh, 314L );
        printf("mh=$%lx mc=$%lx\n",mh,mc);
        printf("Block1=$%lx, Block2=$%lx\n",block1,block2);
```

```
        FreeMem( mh, (long)sizeof(struct MemHeader) );
        FreeMem( mc, BLOCKSIZE );
    }
```

NOTE
    If the free list is corrupt, the system will panic with alert
    AN_MemCorrupt, $01000005.

SEE ALSO
    Deallocate, exec/memory.h

1.16 exec.library/AllocEntry

NAME
    AllocEntry -- allocate many regions of memory

SYNOPSIS
    memList = AllocEntry(memList)
    D0   A0

    struct MemList *AllocEntry(struct MemList *);

FUNCTION
    This function takes a memList structure and allocates enough memory
    to hold the required memory as well as a MemList structure to keep
    track of it.

    These MemList structures may be linked together in a task control
    block to keep track of the total memory usage of this task. (See
    the description of TC_MEMENTRY under RemTask).

INPUTS
    memList -- A MemList structure filled in with MemEntry structures.

RESULTS
    memList -- A different MemList filled in with the actual memory
        allocated in the me_Addr field, and their sizes in me_Length.
        If enough memory cannot be obtained, then the requirements of
        the allocation that failed is returned and bit 31 is set.

        WARNING: The result is unusual! Bit 31 indicates faulure.

EXAMPLES
    The user wants five regions of 2, 4, 8, 16, and 32 bytes in size
    with requirements of MEMF_CLEAR, MEMF_PUBLIC, MEMF_CHIP!MEMF_CLEAR,
    MEMF_CLEAR, and MEMF_PUBLIC!MEMF_CLEAR respectively. The
    following code fragment would do that:

        MemListDecl:
            DS.B   LN_SIZE   * reserve space for list node
            DC.W   5   * number of entries
            DC.L   MEMF_CLEAR   * entry #0
            DC.L   2
            DC.L   MEMF_PUBLIC   * entry #1
```

```
            DC.L    4
            DC.L    MEMF_CHIP!MEMF_CLEAR            * entry #2
            DC.L    8
            DC.L    MEMF_CLEAR                      * entry #3
            DC.L    16
            DC.L    MEMF_PUBLIC!MEMF_CLEAR          * entry #4
            DC.L    32

        start:
            LEA.L   MemListDecl(PC),A0
            JSR     _LVOAllocEntry(a6)
            BCLR.L  #31,D0
            BEQ.S   success

            ------- Type of memory that we failed on is in D0
```

BUGS
    If any one of the allocations fails, this function fails to back
    out fully. This is fixed by the "SetPatch" program on V1.3
    Workbench disks.

SEE ALSO
    exec/memory.h

1.17   exec.library/AllocMem

NAME
    AllocMem -- allocate memory given certain requirements

SYNOPSIS
    memoryBlock = AllocMem(byteSize, attributes)
    D0   D0   D1

    void *AllocMem(ULONG, ULONG);

FUNCTION
    This is the memory allocator to be used by system code and
    applications. It provides a means of specifying that the allocation
    should be made in a memory area accessible to the chips, or
    accessible to shared system code.

    Memory is allocated based on requirements and options. Any
    "requirement" must be met by a memory allocation, any "option" will
    be applied to the block regardless. AllocMem will try all memory
    spaces until one is found with the proper requirements and room for
    the memory request.

INPUTS
    byteSize - the size of the desired block in bytes. (The operating
            system will automatically round this number to a multiple of
            the system memory chunk size)

    attributes -
        requirements

If no flags are set, the system will return the best
available memory block. For expanded systems, the fast
memory pool is searched first.

MEMF_CHIP: If the requested memory will be used by
                  the Amiga custom chips, this flag *must*
                  be set.

                  Only certain parts of memory are reachable
                  by the special chip sets' DMA circuitry.
                  Chip DMA includes screen memory, images that
                  are blitted, audio data, copper lists, sprites
                  and Pre-V36 trackdisk.device buffers.

MEMF_FAST: This is non-chip memory. If no flag is set
                  MEMF_FAST is taken as the default.

                  DO NOT SPECIFY MEMF_FAST unless you know
                  exactly what you are doing! If MEMF_FAST is
                  set, AllocMem() will fail on machines that
                  only have chip memory! This flag may not
                  be set when MEMF_CHIP is set.

MEMF_PUBLIC: Memory that must not be mapped, swapped,
                  or otherwise made non-addressable. ALL
                  MEMORY THAT IS REFERENCED VIA INTERRUPTS
                  AND/OR BY OTHER TASKS MUST BE EITHER PUBLIC
                  OR LOCKED INTO MEMORY! This includes both
                  code and data.

MEMF_LOCAL: This is memory that will not go away
                  after the CPU RESET instruction. Normally,
                  autoconfig memory boards become unavailable
                  after RESET while motherboard memory
                  may still be available. This memory type
                  is now automatically set in V36. Pre-V36
                  systems may not have this memory type
                  and AllocMem() will then fail.

MEMF_24BITDMA: This is memory that is within the address
                  range of 24-bit DMA devices. (Zorro-II)
                  This is required if you run a Zorro-II
                  DMA device on a machine that has memory
                  beyond the 24-bit addressing limit of
                  Zorro-II. This memory type
                  is now automatically set in V36. Pre-V36
                  systems may not have this memory type
                  and AllocMem() will then fail.

options

MEMF_CLEAR: The memory will be initialized to all

                                                zeros.

                        MEMF_REVERSE: This allocates memory from the top of
                                        the memory pool. It searches the pools
                                        in the same order, such that FAST memory
                                        will be found first. However, the
                                        memory will be allocated from the highest
                                        address available in the pool. This
                                        option is new as of V36.

RESULT
    memoryBlock - a pointer to the newly allocated memory block.
            If there are no free memory regions large enough to satisfy
            the request, zero will be returned. The pointer must be
            checked for zero before the memory block may be used!

WARNING
    The result of any memory allocation MUST be checked, and a viable
    error handling path taken. ANY allocation may fail if memory has
    been filled.

EXAMPLES
    AllocMem(64,0L)   - Allocate the best available memory
    AllocMem(25,MEMF_CLEAR) - Allocate the best available memory, and
                                  clear it before returning.
    AllocMem(128,MEMF_CHIP) - Allocate chip memory
    AllocMem(128,MEMF_CHIP|MEMF_CLEAR) - Allocate cleared chip memory
    AllocMem(821,MEMF_CHIP|MEMF_PUBLIC|MEMF_CLEAR) - Allocate cleared,
            public, chip memory.

NOTE
    If the free list is corrupt, the system will panic with alert
    AN_MemCorrupt, $01000005.

    This function may not be called from interrupts.

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

SEE ALSO
    FreeMem

1.18   exec.library/AllocSignal

NAME
    AllocSignal -- allocate a signal bit

SYNOPSIS
    signalNum = AllocSignal(signalNum)
    D0   D0

    BYTE AllocSignal(BYTE);

FUNCTION
    Allocate a signal bit from the current tasks' pool. Either a
    particular bit, or the next free bit may be allocated. The signal
    associated with the bit will be properly initialized (cleared). At
    least 16 user signals are available per task. Signals should be
    deallocated before the task exits.

    If the signal is already in use (or no free signals are available)
    a -1 is returned.

    Allocated signals are only valid for use with the task that
    allocated them.

WARNING
    Signals may not be allocated or freed from exception handling code.

INPUTS
    signalNum - the desired signal number {of 0..31} or -1 for no
                preference.

RESULTS
    signalNum - the signal bit number allocated {0..31}. If no signals
                are available, this function returns -1.

SEE ALSO
    FreeSignal

1.19 exec.library/AllocTrap

NAME
    AllocTrap -- allocate a processor trap vector

SYNOPSIS
    trapNum = AllocTrap(trapNum)
    D0   D0

    LONG AllocTrap(LONG);

FUNCTION
    Allocate a trap number from the current task's pool. These trap
    numbers are those associated with the 68000 TRAP type instructions.
    Either a particular number, or the next free number may be
    allocated.

    If the trap is already in use (or no free traps are available) a -1
    is returned.

    This function only affects the currently running task.

    Traps are sent to the trap handler pointed at by tc_TrapCode.
    Unless changed by user code, this points to a standard trap
    handler. The stack frame of the exception handler will be:

            0(SP) = Exception vector number. This will be in the

```
                range of 32 to 47 (corresponding to the
                Trap #1...Trap #15 instructions).
        4(SP) = 68000/68010/68020/68030, etc. exception frame
```

    tc_TrapData is not used.

WARNING
    Traps may not be allocated or freed from exception handling code.
    You are not allowed to write to the exception table yourself. In
    fact, on some machines you will have trouble finding it - the VBR
    register may be used to remap its location.

INPUTS
    trapNum - the desired trap number {of 0..15} or -1
              for no preference.

RESULTS
    trapNum - the trap number allocated {of 0..15}. If no traps are
              available, this function returns -1. Instructions of the
              form "Trap #trapNum" will be sent to the task's trap
              handler.

SEE ALSO
    FreeTrap

1.20   exec.library/AllocVec

NAME
    AllocVec -- allocate memory and keep track of the size (V36)

SYNOPSIS
    memoryBlock = AllocVec(byteSize, attributes)
    D0   D0   D1

    void *AllocVec(ULONG, ULONG);

FUNCTION
    This function works identically to AllocMem(), but tracks the size
    of the allocation.

    See the AllocMem() documentation for details.

WARNING
    The result of any memory allocation MUST be checked, and a viable
    error handling path taken. ANY allocation may fail if memory has
    been filled.

SEE ALSO
    FreeVec, AllocMem

1.21   exec.library/AttemptSemaphore

NAME
    AttemptSemaphore -- try to obtain without blocking

SYNOPSIS
    success = AttemptSemaphore(signalSemaphore)
    D0   A0

    LONG AttemptSemaphore(struct SignalSemaphore *);

FUNCTION
    This call is similar to ObtainSemaphore(), except that it will not
    block if the semaphore could not be locked.

INPUT
    signalSemaphore -- an initialized signal semaphore structure

RESULT
    success -- TRUE if the semaphore was locked, false if some
        other task already possessed the semaphore.

NOTE
    This call does NOT preserve registers.

SEE ALSO
    ObtainSemaphore() ObtainSemaphoreShared(), ReleaseSemaphore(),
    exec/semaphores.h

1.22 exec.library/AvailMem

NAME
    AvailMem -- memory available given certain requirements

SYNOPSIS
    size = AvailMem(attributes)
    D0   D1

    ULONG AvailMem(ULONG);

FUNCTION
    This function returns the amount of free memory given certain
    attributes.

    To find out what the largest block of a particular type is, add
    MEMF_LARGEST into the requirements argument. Returning the largest
    block is a slow operation.

WARNING
    Due to the effect of multitasking, the value returned may not
    actually be the amount of free memory available at that instant.

INPUTS
    requirements - a requirements mask as specified in AllocMem. Any
                   of the AllocMem bits are valid, as is MEMF_LARGEST
                   which returns the size of the largest block matching

                        the requirements.

RESULT
    size - total free space remaining (or the largest free block).

NOTE
    For V36 Exec, AvailMem(MEMF_LARGEST) does a consistency check on
    the memory list. Alert AN_MemoryInsane will be pulled if any mismatch
    is noted.

EXAMPLE
    AvailMem(MEMF_CHIP|MEMF_LARGEST);
    /* return size of largest available chip memory chunk */

SEE ALSO
    exec/memory.h

1.23 exec.library/CacheClearE

NAME
    CacheClearE - Cache clearing with extended control (V37)

SYNOPSIS
    CacheClearE(address,length,caches)
                a0   d0   d1

    void CacheClearE(APTR,ULONG,ULONG);

FUNCTION
    Flush out the contents of the CPU instruction and/or data caches.
    If dirty data cache lines are present, push them to memory first.

    Motorola CPUs have separate instruction and data caches. A data
    write does not update the instruction cache. If an instruction is
    written to memory or modified, the old instruction may still exist
    in the cache. Before attempting to execute the code, a flush of
    the instruction cache is required.

    For most systems, the data cache is not updated by Direct Memory
    Access (DMA), or if some external factor changes shared memory.

    Caches must be cleared after *any* operation that could cause
    invalid or stale data. The most common cases are DMA and modifying
    instructions using the processor.

    Some examples:
                    Self modifying code
                    Building Jump tables
                    Run-time code patches
                    Relocating code for use at different addresses.
                    Loading code from disk

INPUTS
    address - Address to start the operation. This may be rounded
              due to hardware granularity.

```
    length  - Length of area to be cleared, or $FFFFFFFF to indicate all
              addresses should be cleared.
    caches  - Bit flags to indicate what caches to affect.  The current
              supported flags are:
                     CACRF_ClearI    ;Clear instruction cache
                     CACRF_ClearD    ;Clear data cache
              All other bits are reserved for future definition.
```

NOTES
    On systems with a copyback mode cache, any dirty data is pushed
    to memory as a part of this operation.

    Regardless of the length given, the function will determine the most
    efficient way to implement the operation. For some cache systems,
    including the 68030, the overhead partially clearing a cache is often
    too great. The entire cache may be cleared.

    For all current Amiga models, Chip memory is set with Instruction
    caching enabled, data caching disabled. This prevents coherency
    conflicts with the blitter or other custom chip DMA. Custom chip
    registers are marked as non-cacheable by the hardware.

    The system takes care of appropriately flushing the caches for normal
    operations. The instruction cache is cleared by all calls that
    modify instructions, including LoadSeg(), MakeLibrary() and
    SetFunction().

SEE ALSO
    exec/execbase.i, CacheControl, CacheClearU

1.24   exec.library/CacheClearU

NAME
    CacheClearU - User callable simple cache clearing (V37)

SYNOPSIS
    CacheClearU()

    void CacheClearU(void);

FUNCTION
    Flush out the contents of any CPU instruction and data caches.
    If dirty data cache lines are present, push them to memory first.

    Caches must be cleared after *any* operation that could cause
    invalid or stale data. The most common cases are DMA and modifying
    instructions using the processor. See the CacheClearE() autodoc
    for a more complete description.

    Some examples of when the cache needs clearing:
                  Self modifying code
                  Building Jump tables
                  Run-time code patches
                  Relocating code for use at different addresses.
                  Loading code from disk

SEE ALSO
    exec/execbase.i, CacheControl, CacheClearE

1.25   exec.library/CacheControl

NAME
    CacheControl – Instruction & data cache control

SYNOPSIS
    oldBits = CacheControl(cacheBits,cacheMask)
    D0   D0   D1

    ULONG CacheControl(ULONG,ULONG);

FUNCTION
    This function provides global control of any instruction or data
    caches that may be connected to the system. All settings are
    global -- per task control is not provided.

    The action taken by this function will depend on the type of
    CPU installed. This function may be patched to support external
    caches, or different cache architectures. In all cases the function
    will attempt to best emulate the provided settings. Use of this
    function may save state specific to the caches involved.

    The list of supported settings is provided in the exec/execbase.i
    include file. The bits currently defined map directly to the Motorola
    68030 CPU CACR register. Alternate cache solutions may patch into
    the Exec cache functions. Where possible, bits will be interpreted to
    have the same meaning on the installed cache.

INPUTS
    cacheBits – new values for the bits specified in cacheMask.

    cacheMask – a mask with ones for all bits to be changed.

RESULT
    oldBits   – the complete prior values for all settings.

NOTE
    As a side effect, this function clears all caches.

SEE ALSO
    exec/execbase.i, CacheClearU, CacheClearE

1.26   exec.library/CachePostDMA

NAME
    CachePostDMA – Take actions after to hardware DMA (V37)

SYNOPSIS

```
        CachePostDMA(vaddress,&length,flags)
                    a0          a1        d0

        CachePostDMA(APTR,LONG *,ULONG);
```

FUNCTION
    Take all appropriate steps after Direct Memory Access (DMA). This
    function is primarily intended for writers of DMA device drivers. The
    action will depend on the CPU type installed, caching modes, and the
    state of any Memory Management Unit (MMU) activity.

    As implemented
            68000 - Do nothing
            68010 - Do nothing
            68020 - Do nothing
            68030 - Flush the data cache
            68040 - Flush matching areas of the data cache
            ????? - External cache boards, Virtual Memory Systems, or
                    future hardware may patch this vector to best emulate
                    the intended behavior.
                    With a Bus-Snooping CPU, this function my end up
                    doing nothing.

INPUTS
    address - Same as initially passed to CachePreDMA
    length - Same as initially passed to CachePreDMA
    flags   - Values:
                    DMA_NoModify - If the area was not modified (and
                    thus there is no reason to flush the cache) set
                    this bit.

SEE ALSO
    exec/execbase.i, CachePreDMA, CacheClearU, CacheClearE

1.27 exec.library/CachePreDMA

NAME
    CachePreDMA - Take actions prior to hardware DMA (V37)

SYNOPSIS
    paddress = CachePreDMA(vaddress,&length,flags)
    d0   a0   a1   d0

    APTR CachePreDMA(APTR,LONG *,ULONG);

FUNCTION
    Take all appropriate steps before Direct Memory Access (DMA). This
    function is primarily intended for writers of DMA device drivers. The
    action will depend on the CPU type installed, caching modes, and the
    state of any Memory Management Unit (MMU) activity.

    This function supports advanced cache architectures that have
    "copyback" modes. With copyback, write data may be cached, but not
    actually flushed out to memory. If the CPU has unflushed data at the
    time of DMA, data may be lost.

```
    As implemented
            68000 - Do nothing
            68010 - Do nothing
            68020 - Do nothing
            68030 - Do nothing
            68040 - Write any matching dirty cache lines back to memory.
                    As a side effect of the 68040's design, matching data
                    cache lines are also invalidated -- future CPUs may
                    be different.
            ????? - External cache boards, Virtual Memory Systems, or
                    future hardware may patch this vector to best emulate
                    the intended behavior.
                    With a Bus-Snooping CPU, this function my end up
                    doing nothing.
```

INPUTS
    address - Base address to start the action.
    length - Pointer to a longword with a length.
    flags   - Values:
                  DMA_Continue - Indicates this call is to complete
                  a prior request that was broken up.

RESULTS
    paddress- Physical address that coresponds to the input virtual
            address.
    &length - This length value will be updated to reflect the contiguous
            length of physical memory present at paddress. This may
            be smaller than the requested length. To get the mapping
            for the next chunk of memory, call the function again with
            a new address, length, and the DMA_Continue flag.

NOTE
    Due to processor granularity, areas outside of the address range
    may be affected by the cache flushing actions. Care has been taken
    to ensure that no harm is done outside the range, and that activities
    on overlapping cache lines won't harm data.

SEE ALSO
    exec/execbase.i, CachePostDMA, CacheClearU, CacheClearE

1.28 exec.library/Cause

NAME
   Cause -- cause a software interrupt

SYNOPSIS
   Cause(interrupt)
       A1

   void Cause(struct Interrupt *);

FUNCTION
    This function causes a software interrupt to occur. If it is
    called from user mode (and processor level 0), the software

interrupt will preempt the current task. This call is often used
by high-level hardware interrupts to defer medium-length processing
down to a lower interrupt level. Note that a software interrupt is
still a real interrupt, and must obey the same restrictions on what
system function it may call.

Currently only 5 software interrupt priorities are implemented:
-32, -16, 0, +16, and +32. Priorities in between are truncated,
values outside the -32/+32 range are not allowed.

NOTE
    When setting up the Interrupt structure, set the node type to
    NT_INTERRUPT, or NT_UNKOWN.

IMPLEMENTATION
    1> Checks if the node type is NT_SOFTINT. If so does nothing since
       the softint is already pending. No nest count is maintained.
    2> Sets the node type to NT_SOFTINT.
    3> Links into one of the 5 priority queues.
    4> Pokes the hardware interrupt bit used for softints.

    The node type returns to NT_INTERRUPT after removal from the list.

INPUTS
    interrupt - pointer to a properly initialized interrupt node

BUGS
    Unlike other Interrupts, SoftInts must preserve the value of A6.

1.29   exec.library/CheckIO

NAME
    CheckIO -- get the status of an IORequest

SYNOPSIS
    result = CheckIO(iORequest)
    D0   A1

    BOOL CheckIO(struct IORequest *);

FUNCTION
    This function determines the current state of an I/O request and
    returns FALSE if the I/O has not yet completed. This function
    effectively hides the internals of the I/O completion mechanism.

    CheckIO() will NOT remove the returned IORequest from the reply port.
    This is best performed with WaitIO(). If the request has already
    completed, WaitIO() will return quickly. Use of the Remove()
    function is dangerous, since other tasks may still be adding things
    to your message port; a Disable() would be required.

    This function should NOT be used to busy loop (looping until IO is
    complete). WaitIO() is provided for that purpose.

INPUTS

```
    iORequest - pointer to an I/O request block
```

RESULTS
```
    result - NULL if I/O is still in progress.  Otherwise
             D0 points to the IORequest block.
```

NOTE
```
    CheckIO can hang if called on an IORequest that has never been used.
    This occurs if LN_TYPE of the IORequest is set to "NT_MESSAGE".
    Instead simply set LN_TYPE to 0.
```

SEE ALSO
```
    DoIO, SendIO, WaitIO, AbortIO
```

1.30   exec.library/CloseDevice

NAME
```
    CloseDevice -- conclude access to a device
```

SYNOPSIS
```
    CloseDevice(iORequest)
                A1

    void CloseDevice(struct IORequest *);
```

FUNCTION
```
    This function informs the device that access to a device/unit
    previously opened has been concluded.  The device may perform
    certain house-cleaning operations.

    The user must ensure that all outstanding IORequests have been
    returned before closing the device.  The AbortIO function can kill
    any stragglers.

    After a close, the I/O request structure is free to be reused.
    Starting with V36 exec it is safe to CloseDevice() with an
    IORequest that is either cleared to zeros, or failed to
    open.
```

INPUTS
```
    iORequest - pointer to an I/O request structure
```

SEE ALSO
```
    OpenDevice
```

1.31   exec.library/CloseLibrary

NAME
```
    CloseLibrary -- conclude access to a library
```

SYNOPSIS
```
    CloseLibrary(library)
```

```
            A1

    void CloseLibrary(struct Library *);
```

FUNCTION
    This function informs the system that access to the given library
    has been concluded. The user must not reference the library or any
    function in the library after this close.

    Starting with V36, it is safe to pass a NULL instead of
    a library pointer.

INPUTS
    library - pointer to a library node

NOTE
    Library writers must pass a SegList pointer or NULL back from their
    open point. This value is used by the system, and not visible as
    a return code from CloseLibrary.

SEE ALSO
    OpenLibrary

1.32   exec.library/ColdReboot

NAME
    ColdReboot - reboot the Amiga (V36)

SYNOPSIS
    ColdReboot()

    void ColdReboot(void);

FUNCTION
    Reboot the machine. All external memory and periperals will be
    RESET, and the machine will start its power up diagnostics.

    This function never returns.

INPUT
    A chaotic pile of disoriented bits.

RESULTS
    An altogether totally integrated living system.

1.33   exec.library/CopyMem

NAME
    CopyMem - general purpose memory copy function

SYNOPSIS
    CopyMem(source, dest, size)

```
              A0        A1      D0

    void CopyMem(APTR,APTR,ULONG);
```

FUNCTION
    CopyMem is a general purpose, fast memory copy function. It can
    deal with arbitrary lengths, with its pointers on arbitrary
    alignments. It attempts to optimize larger copies with more
    efficient copies, it uses byte copies for small moves, parts of
    larger copies, or the entire copy if the source and destination are
    misaligned with respect to each other.

    Arbitrary overlapping copies are not supported.

    The internal implementation of this function will change from
    system to system, and may be implemented via hardware DMA.

INPUTS
    source – a pointer to the source data region
    dest – a pointer to the destination data region
    size – the size (in bytes) of the memory area. Zero copies
             zero bytes

SEE ALSO
    CopyMemQuick

1.34   exec.library/CopyMemQuick

NAME
    CopyMemQuick – optimized memory copy function

SYNOPSIS
    CopyMemQuick(source, dest, size)
                  A0   A1   D0

    void CopyMemQuick(ULONG *,ULONG *,ULONG);

FUNCTION
    CopyMemQuick is a highly optimized memory copy function, with
    restrictions on the size and alignment of its arguments. Both the
    source and destination pointers must be longword aligned. In
    addition, the size must be an integral number of longwords (e.g.
    the size must be evenly divisible by four).

    Arbitrary overlapping copies are not supported.

    The internal implementation of this function will change from system
    to system, and may be implemented via hardware DMA.

INPUTS
    source – a pointer to the source data region, long aligned
    dest – a pointer to the destination data region, long aligned
    size – the size (in bytes) of the memory area. Zero copies
             zero bytes.

SEE ALSO
    CopyMem

1.35   exec.library/CreateIORequest

NAME
    CreateIORequest() -- create an IORequest structure (V36)

SYNOPSIS
    ioReq = CreateIORequest(ioReplyPort, size);
                                 A0   D0

    struct IORequest *CreateIORequest(struct MsgPort *, ULONG);

FUNCTION
    Allocates memory for and initializes a new IO request block
    of a user-specified number of bytes. The number of bytes
    must be at least as large as a "struct Message".

INPUTS
    ioReplyPort - Pointer to a port for replies (an initialized message
            port, as created by CreateMsgPort()). If NULL, this
            function fails.
    size - the size of the IO request to be created.

RESULT
    ioReq - A pointer to the new IORequest block, or NULL.

SEE ALSO
    DeleteIORequest, CreateMsgPort(), amiga.lib/CreateExtIO()

1.36   exec.library/CreateMsgPort

NAME
    CreateMsgPort - Allocate and initialize a new message port (V36)

SYNOPSIS
    CreateMsgPort()

    struct MsgPort * CreateMsgPort(void);

FUNCTION
    Allocates and initializes a new message port. The message list
    of the new port will be prepared for use (via NewList). A signal
    bit will be allocated, and the port will be set to signal your
    task when a message arrives (PA_SIGNAL).

    You *must* use DeleteMsgPort() to delete ports created with
    CreateMsgPort()!

RESULT
    MsgPort - A new MsgPort structure ready for use, or NULL if out of

                    memory or signals. If you wish to add this port to the public
                    port list, fill in the ln_Name and ln_Pri fields, then call
                    AddPort(). Don't forget RemPort()!

SEE ALSO
    DeleteMsgPort(), exec/AddPort(), exec/ports.h, amiga.lib/CreatePort()

1.37   exec.library/Deallocate

NAME
    Deallocate -- deallocate a block of memory

SYNOPSIS
    Deallocate(memHeader, memoryBlock, byteSize)
              A0   A1   D0

    void Deallocate(struct MemHeader *,APTR,ULONG);

FUNCTION
    This function deallocates memory by returning it to the appropriate
    private free memory pool. This function can be used to free an
    entire block allocated with the above function, or it can be used
    to free a sub-block of a previously allocated block. Sub-blocks
    must be an even multiple of the memory chunk size (currently 8
    bytes).

    This function can even be used to add a new free region to an
    existing MemHeader, however the extent pointers in the MemHeader
    will no longer be valid.

    If memoryBlock is not on a block boundary (MEM_BLOCKSIZE) then it
    will be rounded down in a manner compatible with Allocate(). Note
    that this will work correctly with all the memory allocation
    functions, but may cause surprises if one is freeing only part of a
    region. The size of the block will be rounded up, so the freed
    block will fill to an even memory block boundary.

INPUTS
    memHeader - points to the memory header this block is part of.
    memoryBlock - address of memory block to free.
    byteSize - the size of the block in bytes. If NULL, nothing
              happens.

SEE ALSO
    Allocate, exec/memory.h

1.38   exec.library/Debug

NAME
    Debug -- run the system debugger

SYNOPSIS

```
    Debug(flags)
          D0

    void Debug(ULONG);
```

FUNCTION
    This function calls the system debugger. By default this debugger
    is "ROM-WACK". Other debuggers are encouraged to take over this
    entry point (via SetFunction()) so that when an application calls
    Debug(), the alternative debugger will get control. Currently a
    zero is passed to allow future expansion.

NOTE
    The Debug() call may be made when the system is in a questionable
    state; if you have a SetFunction() patch, make few assumptions, be
    prepared for Supervisor mode, and be aware of differences in the
    Motorola stack frames on the 68000,'10,'20, and '30.

SEE ALSO
    SetFunction
    your favorite debugger's manual
    the ROM-WACK chapter of the ROM Kernel Manual

1.39   exec.library/DeleteIORequest

NAME
    DeleteIORequest() - Free a request made by CreateIORequest() (V36)

SYNOPSIS
    DeleteIORequest(ioReq);
                       a0

    void DeleteIORequest(struct IORequest *);

FUNCTION
    Frees up an IO request as allocated by CreateIORequest().

INPUTS
    ioReq - A pointer to the IORequest block to be freed, or NULL.
            This function uses the mn_Length field to determine how
            much memory to free.

SEE ALSO
    CreateIORequest(), amiga.lib/DeleteExtIO()

1.40   exec.library/DeleteMsgPort

NAME
    DeleteMsgPort - Free a message port created by CreateMsgPort (V36)

SYNOPSIS
    DeleteMsgPort(msgPort)
```

```
            a0

    void DeleteMsgPort(struct MsgPort *);
```

FUNCTION
    Frees a message port created by CreateMsgPort(). All messages that
    may have been attached to this port must have already been
    replied to.

INPUTS
    msgPort - A message port. NULL for no action.

SEE ALSO
    CreateMsgPort(), amiga.lib/DeletePort()

1.41 exec.library/Disable

NAME
    Disable -- disable interrupt processing.

SYNOPSIS
    Disable();

    void Disable(void);

FUNCTION
    Prevents interrupts from being handled by the system, until a
    matching Enable() is executed. Disable() implies Forbid().

    DO NOT USE THIS CALL WITHOUT GOOD JUSTIFICATION. THIS CALL IS
    VERY DANGEROUS!

RESULTS
    All interrupt processing is deferred until the task executing makes
    a call to Enable() or is placed in a wait state. Normal task
    rescheduling does not occur while interrupts are disabled. In order
    to restore normal interrupt processing, the programmer must execute
    exactly one call to Enable() for every call to Disable().

    IMPORTANT REMINDER:

    It is important to remember that there is a danger in using
    disabled sections. Disabling interrupts for more than ~250
    microseconds will prevent vital system functions (especially serial
    I/O) from operating in a normal fashion.

    Think twice before using Disable(), then think once more.
    After all that, think again. With enough thought, the need
    for a Disable() can often be eliminated. For the user of many
    device drivers, a write to disable *only* the particular interrupt
    of interest can replace a Disable(). For example:
                    MOVE.W #INTF_PORTS,_intena
    Do not use a macro for Disable(), insist on the real thing.

    This call may be made from interrupts, it will have the effect
```

    of locking out all higher-level interrupts (lower-level interrupts
    are automatically disabled by the CPU).

    Note: In the event of a task entering a Wait() after disabling
          interrupts, the system "breaks" the disabled state and runs
          normally until the task which called Disable() is rescheduled.

NOTE
    This call is guaranteed to preserve all registers.

SEE ALSO
    Forbid, Permit, Enable

1.42 exec.library/DoIO

NAME
    DoIO -- perform an I/O command and wait for completion

SYNOPSIS
    error = DoIO(iORequest)
    D0   A1

    BYTE DoIO(struct IORequest *);

FUNCTION
    This function requests a device driver to perform the I/O command
    specified in the I/O request. This function will always wait until
    the I/O request is fully complete.

    DoIO() handles all the details, including Quick I/O, waiting for
    the request, and removing the reply message, etc..

IMPLEMENTATION
    This function first tries to complete the IO via the "Quick I/O"
    mechanism. The io_Flags field is always set to IOF_QUICK (0x01)
    before the internal device call.

    The LN_TYPE field is used internally to flag completion. Active
    requests have type NT_MESSAGE. Requests that have been replied
    have type NT_REPLYMSG. It is illegal to start IO using a
    still active IORequest, or a request with type NT_REPLYMSG.

INPUTS
    iORequest - pointer to an IORequest initialized by OpenDevice()

RESULTS
    error - a sign-extended copy of the io_Error field of the
            IORequest. Most device commands require that the error
            return be checked.

SEE ALSO
    SendIO, CheckIO, WaitIO, AbortIO, amiga.lib/BeginIO

1.43 exec.library/Enable

NAME
    Enable -- permit system interrupts to resume.

SYNOPSIS
    Enable();

    void Enable(void);

FUNCTION
    Allow system interrupts to again occur normally, after a matching
    Disable() has been executed.

RESULTS
    Interrupt processing is restored to normal operation. The
    programmer must execute exactly one call to Enable() for every call
    to Disable().

NOTE
    This call is guaranteed to preserve all registers.

SEE ALSO
    Forbid, Permit, Disable

1.44 exec.library/Enqueue

NAME
    Enqueue -- insert or append node to a system queue

SYNOPSIS
    Enqueue(list, node)
            A0   A1

    void Enqueue(struct List *, struct Node *);

FUNCTION
    Insert or append a node into a system queue. The insert is
    performed based on the node priority -- it will keep the list
    properly sorted. New nodes will be inserted in front of the first
    node with a lower priority.   Hence a FIFO queue for nodes of equal
    priority

WARNING
    This function does not arbitrate for access to the list.   The
    calling task must be the owner of the involved list.

INPUTS
    list - a pointer to the system queue header
    node - the node to enqueue. This must be a full featured list
         with type, priority and name fields.

SEE ALSO

AddHead, AddTail, Insert, Remove, RemHead, RemTail

1.45 exec.library/FindName

```
NAME
    FindName -- find a system list node with a given name

SYNOPSIS
    node = FindName(start, name)
    D0,Z             A0      A1

    struct Node *FindName(struct List *, STRPTR);

FUNCTION
    Traverse a system list until a node with the given name is found.
    To find multiple occurrences of a string, this function may be
    called with a node starting point.

    No arbitration is done for access to the list!  If multiple tasks
    access the same list, an arbitration mechanism such as
    SignalSemaphores must be used.

INPUTS
    start - a list header or a list node to start the search
            (if node, this one is skipped)
    name - a pointer to a name string terminated with NULL

RESULTS
    node - a pointer to the node with the same name else
        zero to indicate that the string was not found.
```

1.46 exec.library/FindPort

```
NAME
    FindPort -- find a given system message port

SYNOPSIS
    port = FindPort(name)
    D0              A1

    struct MsgPort *FindPort(STRPTR);

FUNCTION
    This function will search the system message port list for a port
    with the given name.  The first port matching this name will be
    returned.  No arbitration of the port list is done.  This function
    MUST be protected with A Forbid()/Permit() pair!

EXAMPLE
    #include <exec/types.h>
    struct MsgPort *FindPort();
```

```
    ULONG SafePutToPort(message, portname)
    struct Message *message;
    STRPTR          portname;
    {
    struct MsgPort *port;

        Forbid();
            port = FindPort(portname);
            if (port)
                PutMsg(port,message);
        Permit();
        return((ULONG)port); /* If zero, the port has gone away */
    }

INPUT
    name - name of the port to find

RETURN
    port - a pointer to the message port, or zero if
            not found.
```

1.47   exec.library/FindResident

```
NAME
    FindResident - find a resident module by name

SYNOPSIS
    resident = FindResident(name)
    D0                      A1

    struct Resident *FindResident(STRPTR);

FUNCTION
    Search the system resident tag list for a resident tag ("ROMTag") with
    the given name.  If found return a pointer to the resident tag
    structure, else return zero.

    Resident modules are used by the system to pull all its parts
    together at startup.  Resident tags are also found in disk based
    devices and libraries.

INPUTS
    name - pointer to name string

RESULT
    resident - pointer to the resident tag structure or
            zero if none found.

SEE ALSO
    exec/resident.h, InitResident
```

1.48   exec.library/FindSemaphore

```
NAME
    FindSemaphore -- find a given system signal semaphore

SYNOPSIS
    signalSemaphore = FindSemaphore(name)
    D0                                A1

    struct SignalSemaphore *FindSemaphore(STRPTR);

FUNCTION
    This function will search the system signal semaphore list for a
    semaphore with the given name.  The first semaphore matching this
    name will be returned.

    This function does not arbitrate for access to the semaphore list,
    surround the call with a Forbid()/Permit() pair.

INPUT
    name - name of the semaphore to find

RESULT
    semaphore - a pointer to the signal semaphore, or zero if not
                found.
```

1.49   exec.library/FindTask

```
NAME
    FindTask -- find a task with the given name or find oneself

SYNOPSIS
    task = FindTask(name)
    D0             A1

    struct Task *FindTask(STRPTR);

FUNCTION
    This function will check all task queues for a task with the given
    name, and return a pointer to its task control block.  If a NULL
    name pointer is given a pointer to the current task will be
    returned.

    Finding oneself with a NULL for the name is very quick.  Finding a
    task by name is very system expensive, and will disable interrupts
    for a long time.  Since a task may remove itself at any time,
    a Forbid()/Permit() pair may be needed to ensure the pointer
    returned by FindTask() is still valid when used.

INPUT
    name - pointer to a name string

RESULT
    task - pointer to the task (or Process)
```

1.50 exec.library/Forbid

NAME
    Forbid -- forbid task rescheduling.

SYNOPSIS
    Forbid()

    void Forbid(void);

FUNCTION
    Prevents other tasks from being scheduled to run by the dispatcher,
    until a matching Permit() is executed, or this task is scheduled to
    Wait(). Interrupts are NOT disabled.

    DO NOT USE THIS CALL WITHOUT GOOD JUSTIFICATION. THIS CALL IS
    DANGEROUS!

RESULTS
    The current task will not be rescheduled as long as it is ready to
    run. In the event that the current task enters a wait state, other
    tasks may be scheduled. Upon return from the wait state, the original
    task will continue to run without disturbing the Forbid().

    Calls to Forbid() nest. In order to restore normal task rescheduling,
    the programmer must execute exactly one call to Permit() for every
    call to Forbid().

WARNING
    In the event of a task entering a Wait() after a Forbid(), the system
    "breaks" the forbidden state and runs normally until the task which
    called Forbid() is rescheduled. If caution is not taken, this can
    cause subtle bugs, since any device or DOS call will (in effect)
    cause your task to wait.

    Forbid() is not useful or safe from within interrupt code
    (All interrupts are always higher priority than tasks, and
    interrupts are allowed to break a Forbid()).

NOTE
    This call is guaranteed to preserve all registers.

SEE ALSO
    Permit, Disable, ObtainSemaphore, ObtainSemaphoreShared

1.51 exec.library/FreeEntry

NAME
    FreeEntry -- free many regions of memory

SYNOPSIS
    FreeEntry(memList)
              A0
    void FreeEntry(struct MemList *);

FUNCTION
    This function takes a memList structure (as returned by AllocEntry)
    and frees all the entries.

INPUTS
    memList -- pointer to structure filled in with MemEntry
               structures

SEE ALSO
    AllocEntry

1.52   exec.library/FreeMem

NAME
    FreeMem -- deallocate with knowledge

SYNOPSIS
    FreeMem(memoryBlock, byteSize)
            A1   D0

    void FreeMem(void *,ULONG);

FUNCTION
    Free a region of memory, returning it to the system pool from which
    it came. Freeing partial blocks back into the system pool is
    unwise.

NOTE
    If a block of memory is freed twice, the system will Guru. The
    Alert is AN_FreeTwice ($01000009).   If you pass the wrong pointer,
    you will probably see AN_MemCorrupt $01000005. Future versions may
    add more sanity checks to the memory lists.

INPUTS
    memoryBlock - pointer to the memory block to free
    byteSize - the size of the desired block in bytes. (The operating
            system will automatically round this number to a multiple of
            the system memory chunk size)

SEE ALSO
    AllocMem

1.53   exec.library/FreeSignal

NAME
    FreeSignal -- free a signal bit

SYNOPSIS
    FreeSignal(signalNum)
               D0

```
    void FreeSignal(BYTE);
```

FUNCTION
    This function frees a previously allocated signal bit for reuse.
    This call must be performed while running in the same task in which
    the signal was allocated.

WARNING
    Signals may not be allocated or freed from exception handling code.

NOTE
    Starting with V37, an attempt to free signal -1 is harmless.

INPUTS
    signalNum - the signal number to free {0..31}.

1.54   exec.library/FreeTrap

NAME
    FreeTrap -- free a processor trap

SYNOPSIS
    FreeTrap(trapNum)
            D0

    void FreeTrap(ULONG);

FUNCTION
    This function frees a previously allocated trap number for reuse.
    This call must be performed while running in the same task in which
    the trap was allocated.

WARNING
    Traps may not be allocated or freed from exception handling code.

INPUTS
    trapNum - the trap number to free {of 0..15}

1.55   exec.library/FreeVec

NAME
    FreeVec -- return AllocVec() memory to the system (V36)

SYNOPSIS
    FreeVec(memoryBlock)
            A1

    void FreeVec(void *);

FUNCTION
    Free an allocation made by the AllocVec() call. The memory will
    be returned to the system pool from which it came.

NOTE
    If a block of memory is freed twice, the system will Guru. The
    Alert is AN_FreeTwice ($01000009).   If you pass the wrong pointer,
    you will probably see AN_MemCorrupt $01000005. Future versions may
    add more sanity checks to the memory lists.

INPUTS
    memoryBlock - pointer to the memory block to free, or NULL.

SEE ALSO
    AllocVec

1.56 exec.library/GetCC

NAME
    GetCC -- get condition codes in a 68010 compatible way.

SYNOPSIS
    conditions = GetCC()
      D0

    UWORD GetCC(void);

FUNCTION
    The 68000 processor has a "MOVE SR,<ea>" instruction which gets a
    copy of the processor condition codes.

    On the 68010,20 and 30 CPUs, "MOVE SR,<ea>" is privileged. User
    code will trap if it is attempted. These processors need to use
    the "MOVE CCR,<ea>" instruction instead.

    This function provides a means of obtaining the CPU condition codes
    in a manner that will make upgrades transparent. This function is
    VERY short and quick.

RESULTS
    conditions - the 680XX condition codes

NOTE
    This call is guaranteed to preserve all registers. This function
    may be implemented as code right in the jump table.

1.57 exec.library/GetMsg

NAME
    GetMsg -- get next message from a message port

SYNOPSIS
    message = GetMsg(port)
    D0   A0

```
struct Message *GetMsg(struct MsgPort *);
```

FUNCTION
    This function receives a message from a given message port. It
    provides a fast, non-copying message receiving mechanism. The
    received message is removed from the message port.

    This function will not wait. If a message is not present this
    function will return zero. If a program must wait for a message,
    it can Wait() on the signal specified for the port or use the
    WaitPort() function. There can only be one task waiting for any
    given port.

    Getting a message does not imply to the sender that the message is
    free to be reused by the sender. When the receiver is finished
    with the message, it may ReplyMsg() it back to the sender.

    Getting a signal does NOT always imply a message is ready. More
    than one message may arrive per signal, and signals may show up
    without messages. Typically you must loop to GetMsg() until it
    returns zero, then Wait() or WaitPort().

INPUT
    port – a pointer to the receiver message port

RESULT
    message – a pointer to the first message available. If
              there are no messages, return zero.
              Callers must be prepared for zero at any time.

SEE ALSO
    PutMsg, ReplyMsg, WaitPort, Wait, exec/ports.h

1.58 exec.library/InitCode

NAME
    InitCode – initialize resident code modules (internal function)

SYNOPSIS
    InitCode(startClass, version)
            D0   D1

    void InitCode(ULONG,ULONG);

FUNCTION
    (This function may be ignored by application programmers)

    Call InitResident() for all resident modules in the ResModules array
    with the given startClass and with versions equal or greater than
    that specified. The segList parameter is passed as zero.

    Resident modules are used by the system to pull all its parts
    together at startup. Modules are initialized in a prioritized order.

```
    Modules that do not have a startclass should be of priority -120.
    RTF_AFTERDOS modues should start at -100 (working down).

INPUTS
    startClass - the class of code to be initialized:
            BITDEF RT,COLDSTART,0
            BITDEF RT,SINGLETASK,1  ;ExecBase->ThisTask==0 (V36 only)
            BITDEF RT,AFTERDOS,2    ;(V36 only)
    version - a major version number

SEE ALSO
    ResidentTag (RT) structure definition (resident.h)
```

1.59 exec.library/InitResident

```
NAME
    InitResident - initialize resident module

SYNOPSIS
    object = InitResident(resident, segList)
    D0                              A1          D1

    APTR InitResident(struct Resident *,ULONG);

FUNCTION
    Initialize a ROMTag.  ROMTags are used to link system modules
    together.  Each disk based device or library must contain a
    ROMTag structure in the first code hunk.

    Once the validity of the ROMTag is verified, the RT_INIT pointer
    is jumped to  with the following registers:
            D0 = 0
            A0 = segList
            A6 = ExecBase

INPUTS
    resident - Pointer to a ROMTag
    segList  - SegList of the loaded object, if loaded from disk.
               Libraries & Devices will cache this value for later
               return at close or expunge time.  Pass NULL for ROM
               modules.

RESULTS
    object  - Return value from the init code, usually the library
               or device base.  NULL for failure.

AUTOINIT FEATURE
    An automatic method of library/device base and vector table
    initialization is also provided by InitResident().  The initial code
    hunk of the library or device should contain "MOVEQ #-1,d0; RTS;".
    Following that must be an initialized Resident structure with
    RTF_AUTOINIT set in rt_Flags, and an rt_Init pointer which points
    to four longwords.  These four longwords will be used in a call
    to MakeLibrary();
```

         - The size of your library/device base structure including initial
           Library or Device structure.

         - A pointer to a longword table of standard, then library
           specific function offsets, terminated with -1L.
           (short format offsets are also acceptable)

         - Pointer to data table in exec/InitStruct format for
           initialization of Library or Device structure.

         - Pointer to library initialization function, or NULL.
           Calling sequence:
                  D0 = library base
                  A0 = segList
                  A6 = ExecBase
        This function must return in D0 the library/device base to be
        linked into the library/device list. If the initialization
        function fails, the device memory must be manually deallocated,
        then NULL returned in D0.

SEE ALSO
    exec/resident.i, FindResident

1.60 exec.library/InitSemaphore

NAME
    InitSemaphore -- initialize a signal semaphore

SYNOPSIS
    InitSemaphore(signalSemaphore)
              A0

    void InitSemaphore(struct SignalSemaphore *);

FUNCTION
    This function initializes a signal semaphore and prepares it for
    use. It does not allocate anything, but does initialize list
    pointers and the semaphore counters.

    Semaphores are often used to protect critical data structures
    or hardware that can only be accessed by one task at a time.
    After initialization, the address of the SignalSemaphore may be
    made available to any number of tasks. Typically a task will
    try to ObtainSemaphore(), passing this address in. If no other
    task owns the semaphore, then the call will lock and return
    quickly. If more tasks try to ObtainSemaphore(), they will
    be put to sleep. When the owner of the semaphore releases
    it, the next waiter in turn will be woken up.

    Semaphores are often preferable to the old-style Forbid()/Permit()
    type arbitration. With Forbid()/Permit() *all* other tasks are
    prevented from running. With semaphores, only those tasks that
    need access to whatever the semaphore protects are subject
    to waiting.

```
INPUT
    signalSemaphore -- a signal semaphore structure (with all fields
                       set to zero before the call)

SEE ALSO
    ObtainSemaphore(), ObtainSemaphoreShared(), AttemptSemaphore(),
    ReleaseSemaphore(), exec/semaphores.h
```

1.61 exec.library/InitStruct

```
NAME
    InitStruct - initialize memory from a table

SYNOPSIS
    InitStruct(initTable, memory, size);
              A1          A2      D0

    void InitStruct(struct InitStruct *, APTR, ULONG);

FUNCTION
    Clear a memory area, then set up default values according to
    the data and offset values in the initTable.  Typically only assembly
    programs take advantage of this function, and only with the macros
    defined in "exec/initializers.i".

    The initialization table has byte commands to

         |a      ||byte|       |given||byte|         |once         |
    load |count||word| into |next ||rptr| offset, |repetitively |
              |long|

    Not all combinations are supported.  The offset, when specified, is
    relative to the memory pointer provided (Memory), and is initially
    zero.  The initialization data (InitTable) contains byte commands
    whose 8 bits are interpreted as follows:

    ddssnnnn
        dd  the destination type (and size):
            00  no offset, use next destination, nnnn is count
            01  no offset, use next destination, nnnn is repeat
            10  destination offset is in the next byte, nnnn is count
            11  destination offset is in the next 24-bits, nnnn is count
        ss  the size and location of the source:
            00  long, from the next two aligned words
            01  word, from the next aligned word
            10  byte, from the next byte
            11  ERROR - will cause an ALERT (see below)
      nnnn  the count or repeat:
          count  the (number+1) of source items to copy
         repeat  the source is copied (number+1) times.

    initTable commands are always read from the next even byte. Given
    destination offsets are always relative to the memory pointer (A2).

    The command %00000000 ends the InitTable stream: use %00010001 if you
```

        really want to copy one longword without a new offset.

        24 bit APTR not supported for 68020 compatibility -- use long.

INPUTS
    initTable - the beginning of the commands and data to init
            Memory with. Must be on an even boundary unless only
            byte initialization is done. End table with "dc.b 0"
            or "dc.w 0".
    memory - the beginning of the memory to initialize. Must be
            on an even boundary if size is specified.
    size - the size of memory, which is used to clear it before
            initializing it via the initTable. If Size is zero,
            memory is not cleared before initializing.

            size must be an even number.

SEE ALSO
    exec/initializers.i

1.62 exec.library/Insert

NAME
    Insert -- insert a node into a list

SYNOPSIS
    Insert(list, node, listNode)
          A0   A1   A2

    void Insert(struct List *, struct Node *, struct Node *);

FUNCTION
    Insert a node into a doubly linked list AFTER a given node
    position. Insertion at the head of a list is possible by passing a
    zero value for listNode, though the AddHead function is slightly
    faster for that special case.

WARNING
    This function does not arbitrate for access to the list. The
    calling task must be the owner of the involved list.

INPUTS
    list - a pointer to the target list header
    node - the node to insert
    listNode - the node after which to insert

SEE ALSO
    AddHead, AddTail, Enqueue, RemHead, Remove, RemTail

1.63 exec.library/MakeFunctions

NAME
    MakeFunctions -- construct a function jump table

SYNOPSIS
    tableSize = MakeFunctions(target, functionArray, funcDispBase)
    D0   A0   A1   A2

    ULONG MakeFunctions(APTR,APTR,APTR);

FUNCTION
    A low level function used by MakeLibrary to build jump tables of
    the type used by libraries, devices and resources. It allows the
    table to be built anywhere in memory, and can be used both for
    initialization and replacement. This function also supports function
    pointer compression by expanding relative displacements into absolute
    pointers.

    The processor instruction cache is cleared after the table building.

INPUT
    destination – the target address for the high memory end of the
            function jump table. Typically this will be the library
            base pointer.

    functionArray – pointer to an array of function pointers or
            function displacements. If funcDispBase is zero, the array
            is assumed to contain absolute pointers to functions. If
            funcDispBase is not zero, then the array is assumed to
            contain word displacements to functions. In both cases,
            the array is terminated by a –1 (of the same size as the
            actual entry.

    funcDispBase – pointer to the base about which all function
            displacements are relative. If zero, then the function
            array contains absolute pointers.

RESULT
    tableSize – size of the new table in bytes (for LIB_NEGSIZE).

SEE ALSO
    exec/MakeLibrary

1.64   exec.library/MakeLibrary

NAME
    MakeLibrary -- construct a library

SYNOPSIS
    library = MakeLibrary(vectors, structure, init, dSize, segList)
    D0   A0   A1   A2   D0   D1

    struct Library *MakeLibrary
                        (APTR,struct InitStruct *,APTR,ULONG,BPTR);

FUNCTION
    This function is used for constructing a library vector and data
    area. The same call is used to make devices. Space for the library
    is allocated from the system's free memory pool. The data portion of
    the library is initialized. init may point to a library specific
    entry point.

NOTE
    Starting with V36, the library base is longword adjusted. The
    lib_PosSize and lib_NegSize fields of the library structure are
    adjusted to match.

INPUTS
    vectors – pointer to an array of function pointers or function
            displacements. If the first word of the array is -1, then
            the array contains relative word displacements (based off
            of vectors); otherwise, the array contains absolute
            function pointers. The vector list is terminated by a -1
            (of the same size as the pointers).

    structure – points to an "InitStruct" data region. If NULL,
            then it will not be used.

    init – If non-NULL, an entry point that will be called before adding
            the library to the system. Registers are as follows:
                    d0 = libAddr   ;Your Library Address
                    a0 = segList   ;Your AmigaDOS segment list
                    a6 = ExecBase   ;Address of exec.library
            The result of the init function must be the library address,
            or NULL for failure.   If NULL, the init point must manually
            deallocate the library base memory (based on the sizes stored
            in lib_PosSize and lib_NegSize).

    dSize – the size of the library data area, including the
            standard library node data. This must be at leas
            sizeof(struct Library).

    segList – pointer to an AmigaDOS SegList (segment list).
             This is passed to a library's init code, and is used later
             for removing the library from memory.

RESULT
    library – the reference address of the library. This is the
             address used in references to the library, not the
             beginning of the memory area allocated. If the library
             vector table require more system memory than is
             available, this function will return NULL.

SEE ALSO
    InitStruct, InitResident, exec/initializers.i

1.65 exec.library/ObtainSemaphore

NAME
    ObtainSemaphore -- gain exclusive access to a semaphore

SYNOPSIS
    ObtainSemaphore(signalSemaphore)
                    A0

    void ObtainSemaphore(struct SignalSemaphore *);

FUNCTION
    Signal semaphores are used to gain exclusive access to an object.
    ObtainSemaphore is the call used to gain this access. If another
    user currently has the semaphore locked the call will block until
    the object is available.

    If the current task already has locked the semaphore and attempts to
    lock it again the call will still succeed. A "nesting count" is
    incremented each time the current owning task of the semaphore calls
    ObtainSemaphore(). This counter is decremented each time
    ReleaseSemaphore() is called. When the counter returns to zero the
    semaphore is actually released, and the next waiting task is called.

    A queue of waiting tasks is maintained on the stacks of the waiting
    tasks. Each will be called in turn as soon as the current task
    releases the semaphore.

    Signal Semaphores are different than Procure()/Vacate() semaphores.
    The former requires less CPU time, especially if the semaphore is
    not currently locked. They require very little set up and user
    thought. The latter flavor of semaphore make no assumptions about
    how they are used -- they are completely general. Unfortunately
    they are not as efficient as signal semaphores, and require the
    locker to have done some setup before doing the call.

INPUT
    signalSemaphore -- an initialized signal semaphore structure

NOTE
    This function preserves all registers (see BUGS).

BUGS
    Until V37, this function could destroy A0.

SEE ALSO
    ObtainSemaphoreShared(), InitSemaphore(), ReleaseSemaphore(),
    AttemptSemaphore(), ObtainSemaphoreList()

1.66   exec.library/ObtainSemaphoreList

NAME
    ObtainSemaphoreList -- get a list of semaphores.

SYNOPSIS
    ObtainSemaphoreList(list)
                        A0

    void ObtainSemaphoreList(struct List *);

FUNCTION
    Signal semaphores may be linked together into a list. This function
    takes a list of these semaphores and attempts to lock all of them at
    once. This call is preferable to applying ObtainSemaphore() to each
    element in the list because it attempts to lock all the elements
    simultaneously, and won't deadlock if someone is attempting to lock
    in some other order.

    This function assumes that only one task at a time will attempt to
    lock the entire list of semaphores. In other words, there needs to
    be a higher level lock (perhaps another signal semaphore...) that is
    used before someone attempts to lock the semaphore list via
    ObtainSemaphoreList().

    Note that deadlocks may result if this call is used AND someone
    attempts to use ObtainSemaphore() to lock more than one semaphore on
    the list. If you wish to lock more than semaphore (but not all of
    them) then you should obtain the higher level lock (see above)

INPUT
    list -- a list of signal semaphores

SEE ALSO
    InitSemaphore(), ReleaseSemaphoreList()

1.67   exec.library/ObtainSemaphoreShared

NAME
    ObtainSemaphoreShared -- gain shared access to a semaphore (V36)

SYNOPSIS
    ObtainSemaphoreShared(signalSemaphore)
                          a0

    void ObtainSemaphoreShared(struct SignalSemaphore *);

FUNCTION
    A lock on a signal semaphore may either be exclusive, or shared.
    Exclusive locks are granted by the ObtainSemaphore() and
    AttemptSemaphore() functions. Shared locks are granted by
    ObtainSemaphoreShared(). Calls may be nested.

    Any number of tasks may simultaneously hold a shared lock on a
    semaphore. Only one task may hold an exclusive lock. A typical
    application is a list that is often read, but only occasionally
    written to.

    Any exlusive locker will be held off until all shared lockers
    release the semaphore. Likewise, if an exlusive lock is held,
    all potential shared lockers will block until the exclusive lock
    is released. All shared lockers are restarted at the same time.

EXAMPLE
            ObtainSemaphoreShared(ss);

```
            /* read data */
            ReleaseSemaohore(ss);

            ObtainSemaphore(ss);
            /* modify data */
            ReleaseSemaohore(ss);
```

NOTES
    While this function was added for V36, the feature magically works
    with all older semaphore structures.

    A task owning a shared lock must not attempt to get an exclusive
    lock on the same semaphore.

INPUT
    signalSemaphore -- an initialized signal semaphore structure

NOTE
    This call is guaranteed to preserve all registers, starting with
    V37 exec.

RESULT

SEE ALSO
    InitSemaphore(), ReleaseSemaphore()

1.68 exec.library/OldOpenLibrary

NAME
    OldOpenLibrary -- obsolete OpenLibrary

SYNOPSIS
    library = OldOpenLibrary(libName)
    D0   A1

    struct Library *OldOpenLibrary(APTR);

FUNCTION
    The 1.0 release of the Amiga system had an incorrect version of
    OpenLibrary that did not check the version number during the
    library open. This obsolete function is provided so that object
    code compiled using a 1.0 system will still run.

    This exactly the same as "OpenLibrary(libName,0L);"

INPUTS
    libName - the name of the library to open

RESULTS
    library - a library pointer for a successful open, else zero

SEE ALSO
    CloseLibrary

1.69 exec.library/OpenDevice

```
NAME
    OpenDevice -- gain access to a device

SYNOPSIS
    error = OpenDevice(devName, unitNumber, iORequest, flags)
    D0                 A0       D0          A1         D1

    BYTE OpenDevice(STRPTR,ULONG,struct IORequest *,ULONG);

FUNCTION
    This function opens the named device/unit and initializes the given
    I/O request block.  Specific documentation on opening procedures
    may come with certain devices.

    The device may exist in memory, or on disk; this is transparent to
    the OpenDevice caller.

    A full path name for the device name is legitimate.  For example
    "test:devs/fred.device".  This allows the use of custom devices
    without requiring the user to copy the device into the system's
    DEVS: directory.

NOTES
    All calls to OpenDevice should have matching calls to CloseDevice!

    Devices on disk cannot be opened until after DOS has been
    started.

    As of V36 tasks can safely call OpenDevice, though DOS may open
    system requesters (e.g., asking the user to insert the Workbench
    disk if DEVS: is not online).  You must call this function from a
    DOS Process if you want to turn off DOS requesters.

INPUTS
    devName - requested device name

    unitNumber - the unit number to open on that device.  The format of
            the unit number is device specific.  If the device does
            not have separate units, send a zero.

    iORequest - the I/O request block to be returned with
            appropriate fields initialized.

    flags - additional driver specific information.  This is sometimes
            used to request opening a device with exclusive access.

RESULTS
    error - Returns a sign-extended copy of the io_Error field
            of the IORequest. Zero if successful, else an error code
            is returned.

BUGS
    AmigaDOS file names are not case sensitive, but Exec lists are.  If
    the library name is specified in a different case than it exists on
```

```
        disk, unexpected results may occur.

        Prior to V36, tasks could not make OpenDevice calls requiring disk
        access (since tasks are not allowed to make dos.library calls).
        Now OpenDevice is protected from tasks.

SEE ALSO
        CloseDevice, DoIO, SendIO, CheckIO, AbortIO, WaitIO
```

1.70   exec.library/OpenLibrary

```
NAME
        OpenLibrary -- gain access to a library

SYNOPSIS
        library = OpenLibrary(libName, version)
        D0                    A1       D0

        struct Library *OpenLibrary(STRPTR, ULONG);

FUNCTION
        This function returns a pointer to a library that was previously
        installed into the system.  If the requested library is exists, and
        if the library version is greater than or equal to the requested
        version, then the open will succeed.

        The device may exist in memory, or on disk; this is transparent to
        the OpenDevice caller.  Only Processes are allowed to call
        OpenLibrary (since OpenLibrary may in turn call dos.library).

        A full path name for the library name is legitimate.  For example
        "wp:libs/wp.library".  This allows the use of custom libraries
        without requiring the user to copy the library into the system's
        LIBS: directory.

NOTES
        All calls to OpenLibrary should have matching calls to CloseLibrary!

        Libraries on disk cannot be opened until after DOS has been
        started.

        As of V36 tasks can safely call OpenLibrary, though DOS may open
        system requesters (e.g., asking the user to insert the Workbench
        disk if LIBS: is not online).  You must call this function from a
        DOS Process if you want to turn off DOS requesters.

INPUTS
        libName - the name of the library to open

        version - the version of the library required.

RESULTS
        library - a library pointer for a successful open, else zero

BUGS
```

AmigaDOS file names are not case sensitive, but Exec lists are. If
the library name is specified in a different case than it exists on
disk, unexpected results may occur.

Prior to V36, tasks could not make OpenLibrary calls requiring disk
access (since tasks are not allowed to make dos.library calls).
Now OpenLibrary is protected from tasks.

The version number of the resident tag in disk based library must
match the version number of the library, or V36 may fail to load it.

SEE ALSO
    CloseLibrary

1.71 exec.library/OpenResource

NAME
    OpenResource -- gain access to a resource

SYNOPSIS
    resource = OpenResource(resName)
    D0   A1

    APTR OpenResource(STRPTR);

FUNCTION
    This function returns a pointer to a resource that was previously
    installed into the system.

    There is no CloseResource() function.

INPUTS
   resName - the name of the resource requested.

RESULTS
    resource - if successful, a resource pointer, else NULL

1.72 exec.library/Permit

NAME
    Permit -- permit task rescheduling.

SYNOPSIS
    Permit()

    void Permit(void);

FUNCTION
    Allow other tasks to be scheduled to run by the dispatcher, after a
    matching Forbid() has been executed.

RESULTS

Other tasks will be rescheduled as they are ready to run. In order
to restore normal task rescheduling, the programmer must execute
exactly one call to Permit() for every call to Forbid().

NOTE
    This call is guaranteed to preserve all registers.

SEE ALSO
    Forbid, Disable, Enable

1.73 exec.library/Procure

NAME
    Procure -- bid for a message lock (semaphore)

SYNOPSIS
    result = Procure(semaphore, bidMessage)
    D0   A0   A1

    BYTE Procure(struct Semaphore *, struct Message *);

FUNCTION
    This function is used to obtain a message based semaphore lock. If
    the lock is immediate, Procure() returns a true result, and the
    bidMessage is not used.   If the semaphore is already locked,
    Procure() returns false, and the task must wait for the bidMessage
    to arrive at its reply port.

    Straight "Semaphores" use the message system. They are therefore
    queueable, and users may wait on several of them at the same time.
    This makes them more powerful than "Signal Semaphores"

INPUT
    semaphore - a semaphore message port. This port is used to queue
    all pending lockers. This port should be initialized with the
    PA_IGNORE option, as the MP_SigTask field is used for a pointer to
    the current locker message (not a task). New semaphore ports must
    also have the SM_BIDS word initialized to -1. If the semaphore is
    public, it should be named, its priority set, and the added with
    AddPort. Message port priority is often used for anti-deadlock
    locking conventions.

RESULT
    result - true when the semaphore is free. In such cases no waiting
    needs to be done. If false, then the task should wait at its
    bidMessage reply port.

BUGS
    Procure() and Vacate() do not have proven reliability.

SEE ALSO
    Vacate()

1.74   exec.library/PutMsg

NAME
    PutMsg -- put a message to a message port

SYNOPSIS
    PutMsg(port, message)
           A0   A1

    void PutMsg(struct MsgPort *, struct Message *);

FUNCTION
    This function attaches a message to the end of a given message port.
    It provides a fast, non-copying message sending mechanism.

    Messages can be attached to only one port at a time. The message
    body can be of any size or form. Because messages are not copied,
    cooperating tasks share the same message memory. The sender task
    must not recycle the message until it has been replied by the
    receiver. Of course this depends on the message handling conventions
    setup by the involved tasks. If the ReplyPort field is non-zero,
    when the message is replied by the receiver, it will be sent back to
    that port.

    Any one of the following actions can be set to occur when a message
    is put:

            1. no special action
            2. signal a given task (specified by MP_SIGTASK)
            3. cause a software interrupt (specified by MP_SIGTASK)

    The action is selected depending on the value found in the MP_FLAGS
    of the destination port.

IMPLEMENTATION
    1.   Sets the LN_TYPE field to "NT_MESSAGE".
    2.   Attaches the message to the destination port.
    3.   Performs the specified arrival action at the destination.

INPUT
    port - pointer to a message port
    message - pointer to a message

SEE ALSO
    GetMsg, ReplyMsg, exec/ports.h

1.75   exec.library/RawDoFmt

NAME
    RawDoFmt -- format data into a character stream.

SYNOPSIS
    NextData = RawDoFmt(FormatString, DataStream, PutChProc, PutChData);
    d0   a0   a1   a2   a3

```
    APTR RawDoFmt(STRPTR,APTR,void (*)(),APTR);
```

FUNCTION
    perform "C"-language-like formatting of a data stream, outputting
    the result a character at a time. Where % formatting commands are
    found in the FormatString, they will be replaced with the
    corresponding element in the DataStream. %% must be used in the
    string if a % is desired in the output.

    Under V36, RawDoFmt() returns a pointer to the end of the DataStream
    (The next argument that would have been processed). This allows
    multiple formatting passes to be made using the same data.

INPUTS
    FormatString - a "C"-language-like NULL terminated format string,
    with the following supported % options:

     %[flags][width.limit][length]type

    flags - only one allowed. '-' specifies left justification.
    width - field width. If the first character is a '0', the
             field will be padded with leading 0's.
       .   - must follow the field width, if specified
    limit - maximum number of characters to output from a string.
             (only valid for %s).
    length - size of input data defaults to WORD for types d, x,
             and c, 'l' changes this to long (32-bit).
    type   - supported types are:
                        b - BSTR, data is 32-bit BPTR to byte count followed
                            by a byte string, or NULL terminated byte string.
                            A NULL BPTR is treated as an empty string.
                            (Added in V36 exec)
                        d - decimal
                        u - unsigned decimal (Added in V37 exec)
                        x - hexadecimal
                        s - string, a 32-bit pointer to a NULL terminated
                            byte string. In V36, a NULL pointer is treated
                            as an empty string
                        c - character

    DataStream - a stream of data that is interpreted according to
            the format string. Often this is a pointer into
            the task's stack.
    PutChProc - the procedure to call with each character to be
            output, called as:

    PutChProc(Char, PutChData);
             D0-0:8 A3

            the procedure is called with a NULL Char at the end of
            the format string.

    PutChData - a value that is passed through to the PutChProc
            procedure. This is untouched by RawDoFmt, and may be
            modified by the PutChProc.

EXAMPLE
```
    ;
    ; Simple version of the C "sprintf" function.  Assumes C-style
    ; stack-based function conventions.
    ;
    ;   long eyecount;
    ;   eyecount=2;
    ;   sprintf(string,"%s have %ld eyes.","Fish",eyecount);
    ;
    ; would produce "Fish have 2 eyes." in the string buffer.
    ;
            XDEF _sprintf
            XREF _AbsExecBase
            XREF _LVORawDoFmt
    _sprintf:       ; ( ostring, format, {values} )
            movem.l a2/a3/a6,-(sp)

            move.l  4*4(sp),a3      ;Get the output string pointer
            move.l  5*4(sp),a0      ;Get the FormatString pointer
            lea.l   6*4(sp),a1      ;Get the pointer to the DataStream
            lea.l   stuffChar(pc),a2
            move.l  _AbsExecBase,a6
            jsr     _LVORawDoFmt(a6)

            movem.l (sp)+,a2/a3/a6
            rts

    ;------ PutChProc function used by RawDoFmt -----------
    stuffChar:
            move.b  d0,(a3)+        ;Put data to output string
            rts
```

WARNING
    This Amiga ROM function formats word values in the data stream. If
    your compiler defaults to longs, you must add an "l" to your
    % specifications. This can get strange for characters, which might
    look like "%lc".

    The result of RawDoFmt() is *ONLY* valid in V36 and later releases
    of EXEC. Pre-V36 versions of EXEC have "random" return values.

SEE ALSO
    Documentation on the C language "printf" call in any C language
    reference book.

1.76 exec.library/ReleaseSemaphore

NAME
    ReleaseSemaphore -- make signal semaphore available to others

SYNOPSIS
    ReleaseSemaphore(signalSemaphore)
                     A0

    void ReleaseSemaphore(struct SignalSemaphore *);

FUNCTION
    ReleaseSemaphore() is the inverse of ObtainSemaphore(). It makes
    the semaphore lockable to other users. If tasks are waiting for
    the semaphore and this this task is done with the semaphore then
    the next waiting task is signalled.

    Each ObtainSemaphore() call must be balanced by exactly one
    ReleaseSemaphore() call. This is because there is a nesting count
    maintained in the semaphore of the number of times that the current
    task has locked the semaphore. The semaphore is not released to
    other tasks until the number of releases matches the number of
    obtains.

    Needless to say, havoc breaks out if the task releases more times
    than it has obtained.

INPUT
   signalSemaphore -- an initialized signal semaphore structure

NOTE
    This call is guaranteed to preserve all registers.

SEE ALSO
    InitSemaphore(), ObtainSemaphore(), ObtainSemaphoreShared()

1.77 exec.library/ReleaseSemaphoreList

NAME
    ReleaseSemaphoreList -- make a list of semaphores available

SYNOPSIS
    ReleaseSemaphoreList(list)
                         A0

    void ReleaseSemaphoreList(struct List *);

FUNCTION
    ReleaseSemaphoreList() is the inverse of ObtainSemaphoreList(). It
    releases each element in the semaphore list.

    Needless to say, havoc breaks out if the task releases more times
    than it has obtained.

INPUT
   list -- a list of signal semaphores

SEE ALSO
    ObtainSemaphoreList()

1.78 exec.library/RemDevice

NAME
    RemDevice -- remove a device from the system

SYNOPSIS
    RemDevice(device)
             A1

    void RemDevice(struct Device *);

FUNCTION
    This function calls the device's EXPUNGE vector, which requests
    that a device delete itself. The device may refuse to do this if
    it is busy or currently open. This is not typically called by user
    code.

    There are certain, limited circumstances where it may be
    appropriate to attempt to specifically flush a certain device.
    Example:

     /* Attempts to flush the named device out of memory. */
     #include <exec/types.h>
     #include <exec/execbase.h>

     void FlushDevice(name)
     STRPTR name;
     {
     struct Device *result;

         Forbid();
         if(result=(struct Device *)FindName(&SysBase->DeviceList,name))
             RemDevice(result);
         Permit();
     }

INPUTS
    device - pointer to a device node

SEE ALSO
    AddLibrary

1.79   exec.library/RemHead

NAME
    RemHead -- remove the head node from a list

SYNOPSIS
    node = RemHead(list)
    D0   A0

    struct Node *RemHead(struct List *);

FUNCTION
    Get a pointer to the head node and remove it from the list.
    Assembly programmers may prefer to use the REMHEAD macro from

```
    "exec/lists.i".
```

WARNING
    This function does not arbitrate for access to the list. The
    calling task must be the owner of the involved list.

INPUTS
    list - a pointer to the target list header

RESULT
    node - the node removed or zero when empty list

SEE ALSO
    AddHead, AddTail, Enqueue, Insert, Remove, RemTail

1.80   exec.library/RemIntServer

NAME
    RemIntServer -- remove an interrupt server from a server chain

SYNOPSIS
    RemIntServer(intNum, interrupt)
                 D0   A1

    void RemIntServer(ULONG,struct Interrupt *);

FUNCTION
    This function removes an interrupt server node from the given
    server chain.

    If this server was the last one on this chain, interrupts for this
    chain are disabled.

INPUTS
    intNum - the Paula interrupt bit (0..14)
    interrupt - pointer to an interrupt server node

BUGS
    Before V36 Kickstart, the feature that disables the interrupt
    would not function. For most server chains this does not
    cause a problem.

SEE ALSO
    AddIntServer, hardware/intbits.h

1.81   exec.library/RemLibrary

NAME
    RemLibrary -- remove a library from the system

SYNOPSIS
    RemLibrary(library)

```
              A1

    void RemLibrary(struct Library *);
```

FUNCTION
    This function calls the library's EXPUNGE vector, which requests
    that a library delete itself. The library may refuse to do this if
    it is busy or currently open. This is not typically called by user
    code.

    There are certain, limited circumstances where it may be
    appropriate to attempt to specifically flush a certain Library.
    Example:

```
     /* Attempts to flush the named library out of memory. */
     #include <exec/types.h>
     #include <exec/execbase.h>

     void FlushLibrary(name)
     STRPTR name;
     {
     struct Library *result;

        Forbid();
        if(result=(struct Library *)FindName(&SysBase->LibList,name))
            RemLibrary(result);
        Permit();
     }
```

INPUTS
    library - pointer to a library node structure

1.82 exec.library/Remove

NAME
    Remove -- remove a node from a list

SYNOPSIS
    Remove(node)
          A1

    void Remove(struct Node *);

FUNCTION
    Unlink a node from whatever list it is in. Nodes that are not part
    of a list must not be passed to this funcion! Assembly programmers
    may prefer to use the REMOVE macro from "exec/lists.i".

WARNING
    This function does not arbitrate for access to the list. The
    calling task must be the owner of the involved list.

INPUTS
    node - the node to remove

SEE ALSO
    AddHead, AddTail, Enqueue, Insert, RemHead, RemTail

1.83   exec.library/RemPort

NAME
    RemPort -- remove a message port from the system

SYNOPSIS
    RemPort(port)
            A1

    void RemPort(struct MsgPort *);

FUNCTION
    This function removes a message port structure from the system's
    message port list. Subsequent attempts to rendezvous by name with
    this port will fail.

INPUTS
    port - pointer to a message port

SEE ALSO
    AddPort, FindPort

1.84   exec.library/RemResource

NAME
    RemResource -- remove a resource from the system

SYNOPSIS
    RemResource(resource)
                A1

    void RemResource(APTR);

FUNCTION
    This function removes an existing resource from the system resource
    list. There must be no outstanding users of the resource.

INPUTS
    resource - pointer to a resource node

SEE ALSO
    AddResource

1.85   exec.library/RemSemaphore

NAME
    RemSemaphore -- remove a signal semaphore from the system

SYNOPSIS
    RemSemaphore(signalSemaphore)
                  A1

    void RemSemaphore(struct SignalSemaphore *);

FUNCTION
    This function removes a signal semaphore structure from the
    system's signal semaphore list. Subsequent attempts to
    rendezvous by name with this semaphore will fail.

INPUTS
    signalSemaphore -- an initialized signal semaphore structure

SEE ALSO
    AddSemaphore, FindSemaphore

1.86   exec.library/RemTail

NAME
    RemTail -- remove the tail node from a list

SYNOPSIS
    node = RemTail(list)
    D0   A0

    struct Node *RemTail(struct List *);

FUNCTION
    Remove the last node from a list, and return a pointer to it. If
    the list is empty, return zero. Assembly programmers may prefer to
    use the REMTAIL macro from "exec/lists.i".

WARNING
    This function does not arbitrate for access to the list. The
    calling task must be the owner of the involved list.

INPUTS
    list - a pointer to the target list header

RESULT
    node - the node removed or zero when empty list

SEE ALSO
    AddHead, AddTail, Enqueue, Insert, Remove, RemHead, RemTail

1.87   exec.library/RemTask

```
NAME
    RemTask -- remove a task from the system

SYNOPSIS
    RemTask(task)
            A1

    void RemTask(struct Task *);

FUNCTION
    This function removes a task from the system.  Deallocation of
    resources should have been performed prior to calling this
    function.  Removing some other task is very dangerous.  Generally
    is is best to arrange for tasks to call RemTask(0L) on themselves.

    RemTask will automagically free any memory lists attached to the
    task's TC_MEMENTRY list.

INPUTS
    task - pointer to the task node representing the task to be
            removed.  A zero value indicates self removal, and will
            cause the next ready task to begin execution.

BUGS
    Before V36 if RemTask() was called on a task other than the current
    task, and that task was created with amiga.lib/CreateTask, there was
    a slight chance of a crash.  The problem can be hidden by bracketing
    RemTask() with Forbid()/Permit().

SEE ALSO
    AddTask, exec/AllocEntry, amiga.lib/DeleteTask
```

1.88 exec.library/ReplyMsg

```
NAME
    ReplyMsg -- put a message to its reply port

SYNOPSIS
    ReplyMsg(message)
            A1

    void ReplyMsg(struct Message *);

FUNCTION
    This function sends a message to its reply port.  This is usually
    done when the receiver of a message has finished and wants to
    return it to the sender (so that it can be re-used or deallocated,
    whatever).

    This call may be made from interrupts.

INPUT
    message - a pointer to the message
```

IMPLEMENTATION
    1> Places "NT_REPLYMSG" into LN_TYPE.
    2> Puts the message to the port specified by MN_REPLYPORT
       If there is no replyport, sets LN_TYPE to "NT_FREEMSG" (use this
       feature only with extreeme care).

SEE ALSO
    GetMsg, PutMsg, exec/ports.h

1.89 exec.library/SendIO

NAME
    SendIO -- initiate an I/O command

SYNOPSIS
    SendIO(iORequest)
            A1

    void SendIO(struct IORequest *);

FUNCTION
    This function requests the device driver start processing the given
    I/O request. The device will return control without waiting for
    the I/O to complete.

    The io_Flags field of the IORequest will be set to zero before the
    request is sent. See BeginIO() for more details.

INPUTS
    iORequest - pointer to an I/O request, or a device specific
                extended IORequest.

SEE ALSO
    DoIO, CheckIO, WaitIO, AbortIO

1.90 exec.library/SetExcept

NAME
    SetExcept -- define certain signals to cause exceptions

SYNOPSIS
    oldSignals = SetExcept(newSignals, signalMask)
    D0   D0   D1

    ULONG SetExcept(ULONG,ULONG);

FUNCTION
    This function defines which of the task's signals will cause a
    private task exception. When any of the signals occurs the task's
    exception handler will be dispatched. If the signal occurred prior
    to calling SetExcept, the exception will happen immediately.

The user function pointed to by the task's tc_ExceptCode gets
called as:

```
newExcptSet = <exceptCode>(signals, exceptData),SysBase
D0                          D0       A1           A6
```

signals - The set of signals that caused this exception. These
    Signals have been disabled from the current set of signals
    that can cause an exception.

exceptData - A copy of the task structure tc_ExceptData field.

newExcptSet - The set of signals in NewExceptSet will be re-
    enabled for exception generation. Usually this will be the
    same as the Signals that caused the exception.

INPUTS
    newSignals - the new values for the signals specified in
        signalMask.
    signalMask - the set of signals to be effected

RESULTS
    oldSignals - the prior exception signals

EXAMPLE
    Get the current state of all exception signals:
        SetExcept(0,0)
    Change a few exception signals:
        SetExcept($1374,$1074)

SEE ALSO
    Signal, SetSignal

1.91   exec.library/SetFunction

NAME
    SetFunction -- change a function vector in a library

SYNOPSIS
    oldFunc = SetFunction(library, funcOffset, funcEntry)
    D0   A1   A0.W   D0

    APTR SetFunction(struct Library *,LONG,APTR);

FUNCTION
    SetFunction is a functional way of changing where vectors in a
    library point. They are changed in such a way that the
    checksumming process will never falsely declare a library to be
    invalid.

WARNING
    If you use SetFunction on a function that can be called from
    interrupts, you are obligated to provide your own arbitration.

NOTE

SetFunction cannot be used on non-standard libraries like pre-V36
dos.library. Here you must manually Forbid(), preserve all 6
original bytes, set the new vector, SumLibrary(), then Permit().

INPUTS
    library   - a pointer to the library to be changed
    funcOffset - the offset of the function to be replaced
    funcEntry - pointer to new function

RESULTS
    oldFunc   - pointer to the old function that was just replaced

1.92 exec.library/SetIntVector

NAME
    SetIntVector -- set a new handler for a system interrupt vector

SYNOPSIS
    oldInterrupt = SetIntVector(intNumber, interrupt)
    D0   D0   A1

    struct Interrupt *SetIntVector(ULONG, struct Interrupt *);

FUNCTION
    This function provides a mechanism for setting the system interrupt
    vectors. These are non-sharable; setting a new interrupt handler
    disconnects the old one. Installed handlers are responsible for
    processing, enabling and clearing the interrupt. Note that interrupts
    may have been left in any state by the previous code.

    The IS_CODE and IS_DATA pointers of the Interrupt structure will
    be copied into a private place by Exec. A pointer to the previously
    installed Interrupt structure is returned.

    When the system calls the specified interrupt code, the registers are
    setup as follows:

        D0 - scratch
        D1 - scratch (on entry: active
                      interrupts -> equals INTENA & INTREQ)

        A0 - scratch (on entry: pointer to base of custom chips
                      for fast indexing)
        A1 - scratch (on entry: Interrupt's IS_DATA pointer)

        A5 - jump vector register (scratch on call)
        A6 - Exec library base pointer (scratch on call)

        all other registers must be preserved

INPUTS
    intNum - the Paula interrupt bit number (0..14). Only non-chained
             interrupts should be set. Use AddIntServer() for server
             chains.
    interrupt - a pointer to an Interrupt structure containing the

                    handler's entry point and data segment pointer. A NULL
                    interrupt pointer will remove the current interrupt and
                    set illegal values for IS_CODE and IS_DATA.

                    By convention, the LN_NAME of the interrupt structure must
                    point a descriptive string so that other users may
                    identify who currently has control of the interrupt.

RESULT
    A pointer to the prior interrupt structure which had control
    of this interrupt.

SEE ALSO
    AddIntServer(),exec/interrupts.i,hardware/intbits.i

1.93   exec.library/SetSignal

NAME
    SetSignal -- define the state of this task's signals

SYNOPSIS
    oldSignals = SetSignal(newSignals, signalMask)
    D0   D0   D1

    ULONG SetSignal(ULONG,ULONG);

FUNCTION
    This function can query or modify the state of the current task's
    received signal mask. Setting the state of signals is considered
    dangerous. Reading the state of signals is safe.

INPUTS
    newSignals - the new values for the signals specified in
                 signalSet.
    signalMask - the set of signals to be affected.

RESULTS
    oldSignals - the prior values for all signals

EXAMPLES
    Get the current state of all signals:
        SetSignal(0L,0L);
    Clear the CTRL-C signal:
        SetSignal(0L,SIGBREAKF_CTRL_C);

    Check if the CTRL-C signal was pressed:

        #include <libraries/dos.h>

        /* Check & clear CTRL_C signal */
        if(SetSignal(0L,SIGBREAKF_CTRL_C) & SIGBREAKF_CTRL_C)
            {
            printf("CTRL-C pressed!\n");
            }

SEE ALSO
    Signal, Wait

1.94   exec.library/SetSR

NAME
    SetSR -- get and/or set processor status register

SYNOPSIS
    oldSR = SetSR(newSR, mask)
    D0   D0   D1

    ULONG SetSR(ULONG, ULONG);

FUNCTION
    This function provides a means of modifying the CPU status register
    in a "safe" way (well, how safe can a function like this be
    anyway?). This function will only affect the status register bits
    specified in the mask parameter. The prior content of the entire
    status register is returned.

INPUTS
    newSR - new values for bits specified in the mask.
        All other bits are not effected.
    mask - bits to be changed

RESULTS
    oldSR - the entire status register before new bits

EXAMPLES
    To get the current SR:
        currentSR = SetSR(0,0);
    To change the processor interrupt level to 3:
        oldSR = SetSR($0300,$0700);
    Set processor interrupts back to prior level:
        SetSR(oldSR,$0700);

1.95   exec.library/SetTaskPri

NAME
    SetTaskPri -- get and set the priority of a task

SYNOPSIS
    oldPriority = SetTaskPri(task, priority)
    D0-0:8   A1   D0-0:8

    BYTE SetTaskPri(struct Task *,LONG);

FUNCTION
    This function changes the priority of a task regardless of its
    state. The old priority of the task is returned. A reschedule is

```
    performed, and a context switch may result.

    To change the priority of the currently running task, pass the
    result of FindTask(0); as the task pointer.

INPUTS
    task - task to be affected
    priority - the new priority for the task

RESULT
    oldPriority - the tasks previous priority
```

1.96 exec.library/Signal

```
NAME
    Signal -- signal a task

SYNOPSIS
    Signal(task, signals)
          A1     D0

    void Signal(struct Task *,ULONG);

FUNCTION
    This function signals a task with the given signals.  If the task
    is currently waiting for one or more of these signals, it will be
    made ready and a reschedule will occur. If the task is not waiting
    for any of these signals, the signals will be posted to the task
    for possible later use. A signal may be sent to a task regardless
    of whether its running, ready, or waiting.

    This function is considered "low level".  Its main purpose is to
    support multiple higher level functions like PutMsg.

    This function is safe to call from interrupts.

INPUT
    task - the task to be signalled
    signals - the signals to be sent

SEE ALSO
    Wait, SetSignal
```

1.97 exec.library/StackSwap

```
NAME
    StackSwap - EXEC supported method of replacing task's stack       (V37)

SYNOPSIS
    StackSwap(newStack)
              A0
```

```
    VOID StackSwap(struct StackSwapStruct *);
```

FUNCTION
    This function will, in an EXEC supported manner, swap the
    stack of your task with the given values in StackSwap.
    The StackSwapStruct structure will then contain the values
    of the old stack such that the old stack can be restored.
    This function is new in V37.

NOTE
    If you do a stack swap, only the new stack is set up.
    This function does not copy the stack or do anything else
    other than set up the new stack for the task. It is
    generally required that you restore your stack before
    exiting.

INPUTS
    newStack - A structure that contains the values for the
            new upper and lower stack bounds and the new stack
            pointer. This structure will have its values
            replaced by those in you task such that you can
            restore the stack later.

RESULTS
    newStack - The structure will now contain the old stack.
            This means that StackSwap(foo); StackSwap(foo);
            will effectively do nothing.

SEE ALSO
    AddTask, RemTask, exec/tasks.h

1.98   exec.library/SumKickData

NAME
    SumKickData -- compute the checksum for the Kickstart delta list

SYNOPSIS
    checksum = SumKickData()
    D0

    ULONG SumKickData(void);

FUNCTION
    The Amiga system has some ROM (or Kickstart) resident code that
    provides the basic functions for the machine. This code is
    unchangeable by the system software. This function is part of a
    support system to modify parts of the ROM.

    The ROM code is linked together at run time via ROMTags (also known
    as Resident structures, defined in exec/resident.h). These tags tell
    Exec's low level boot code what subsystems exist in which regions of
    memory. The current list of ROMTags is contained in the ResModules
    field of ExecBase. By default this list contains any ROMTags found
    in the address ranges $F80000-$FFFFFF and $F00000-$F7FFFF.

There is also a facility to selectively add or replace modules to the
ROMTag list. These modules can exist in RAM, and the memory they
occupy will be deleted from the memory free list during the boot
process. SumKickData() plays an important role in this run-time
modification of the ROMTag array.

Three variables in ExecBase are used in changing the ROMTag array:
KickMemPtr, KickTagPtr, and KickCheckSum. KickMemPtr points to a
linked list of MemEntry structures. The memory that these MemEntry
structures reference will be allocated (via AllocAbs) at boot time.
The MemEntry structure itself must also be in the list.

KickTagPtr points to a long-word array of the same format as the
ResModules array. The array has a series of pointers to ROMTag
structures. The array is either NULL terminated, or will have an
entry with the most significant bit (bit 31) set. The most
significant bit being set says that this is a link to another
long-word array of ROMTag entries. This new array's address can be
found by clearing bit 31.

KickCheckSum has the result of SumKickData(). It is the checksum of
both the KickMemPtr structure and the KickTagPtr arrays. If the
checksum does not compute correctly then both KickMemPtr and
KickTagPtr will be ignored.

If all the memory referenced by KickMemPtr can't be allocated then
KickTagPtr will be ignored.

There is one more important caveat about adding ROMTags. All this
ROMTag magic is run very early on in the system -- before expansion
memory is added to the system. Therefore any memory in this
additional ROMTag area must be addressable at this time. This means
that your ROMTag code, MemEntry structures, and resident arrays
cannot be in expansion memory. There are two regions of memory that
are acceptable: one is chip memory, and the other is "Ranger" memory
(memory in the range between $C00000-$D80000).

Remember that changing an existing ROMTag entry falls into the
"heavy magic" category -- be very careful when doing it. The odd are
that you will blow yourself out of the water.

NOTE
    SumKickData was introduced in the 1.2 release

RESULT
    Value to be stuffed into ExecBase->KickCheckSum.

WARNING
    After writing to KickCheckSum, you should push the data cache.
    This prevents potential problems with large copyback style caches.
    A call to CacheClearU will do fine.

SEE ALSO
    InitResident, FindResident

1.99 exec.library/SumLibrary

NAME
    SumLibrary -- compute and check the checksum on a library

SYNOPSIS
    SumLibrary(library)
              A1

    void SumLibrary(struct Library *);

FUNCTION
    SumLibrary computes a new checksum on a library. It can also be
    used to check an old checksum. If an old checksum does not match,
    and the library has not been marked as changed, then the system
    will call Alert().

    This call could also be periodically made by some future
    system-checking task.

INPUTS
    library - a pointer to the library to be changed

NOTE
    An alert will occur if the checksum fails.

SEE ALSO
    SetFunction

1.100 exec.library/SuperState

NAME
    SuperState -- enter supervisor state with user stack

SYNOPSIS
    oldSysStack = SuperState()
    D0

    APTR SuperState(void);

FUNCTION
    Enter supervisor mode while running on the user's stack. The user
    still has access to user stack variables. Be careful though, the
    user stack must be large enough to accommodate space for all
    interrupt data -- this includes all possible nesting of interrupts.
    This function does nothing when called from supervisor state.

RESULTS
    oldSysStack - system stack pointer; save this. It will come in
                  handy when you return to user state. If the system
                  is already in supervisor mode, oldSysStack is zero.

SEE ALSO
    UserState/Supervisor

1.101   exec.library/Supervisor

NAME
    Supervisor -- trap to a short supervisor mode function

SYNOPSIS
    result = Supervisor(userFunc)
    Rx   A5

    ULONG Supervisor(void *);

FUNCTION
    Allow a normal user-mode program to execute a short assembly language
    function in the supervisor mode of the processor. Supervisor() does
    not modify or save registers; the user function has full access to the
    register set.   All rules that apply to interrupt code must be
    followed. In addition, no system calls are permitted. The function
    must end with an RTE instruction.

EXAMPLE
            ;Obtain the Exception Vector base. 68010 or greater only!
            MOVECtrap: movec.l VBR,d0 ;$4e7a,$0801
                             rte

INPUTS
    userFunc - A pointer to a short assembly language function ending
               in RTE. The function has full access to the register set.

RESULTS
    result   - Whatever values the userFunc left in the registers.

SEE ALSO
    SuperState,UserState

1.102   exec.library/TypeOfMem

NAME
    TypeOfMem -- determine attributes of a given memory address

SYNOPSIS
    attributes = TypeOfMem(address)
    D0   A1

    ULONG TypeOfMem(void *);

FUNCTION
    Given a RAM memory address, search the system memory lists and
    return its memory attributes. The memory attributes are similar to
    those specified when the memory was first allocated: (eg. MEMF_CHIP
    and MEMF_FAST).

This function is usually used to determine if a particular block of
memory is within CHIP space.

If the address is not in known-space, a zero will be returned.
(Anything that is not RAM, like the ROM or expansion area, will
return zero. Also the first few bytes of a memory area are used up
by the MemHeader.)

INPUT
    address - a memory address

RESULT
    attributes - a long word of memory attribute flags.
    If the address is not in known RAM, zero is returned.

SEE ALSO
    AllocMem()

1.103   exec.library/UserState

NAME
    UserState -- return to user state with user stack

SYNOPSIS
    UserState(sysStack)
              D0

    void UserState(APTR);

FUNCTION
    Return to user state with user stack, from supervisor state with
    user stack. This function is normally used in conjunction with the
    SuperState function above.

    This function must not be called from the user state.

INPUT
    sysStack - supervisor stack pointer

BUGS
    This function is broken in V33/34 Kickstart. Fixed in V1.31 setpatch.

SEE ALSO
    SuperState/Supervisor

1.104   exec.library/Vacate

NAME
    Vacate -- release a message lock (semaphore)

SYNOPSIS

```
    Vacate(semaphore)
          A0

    void Vacate(struct Semaphore *);
```

FUNCTION
    This function releases a previously locked semaphore (see
    the Procure() function).
    If another task is waiting for the semaphore, its bidMessage
    will be sent to its reply port.

INPUT
    semaphore - the semaport message port representing the
    semaphore to be freed.

BUGS
    Procure() and Vacate() do not have proven reliability.

SEE ALSO
    Procure

1.105 exec.library/Wait

NAME
    Wait -- wait for one or more signals

SYNOPSIS
    signals = Wait(signalSet)
    D0   D0

    ULONG Wait(ULONG);

FUNCTION
    This function will cause the current task to suspend waiting for
    one or more signals. When one or more of the specified signals
    occurs, the task will return to the ready state, and those signals
    will be cleared.

    If a signal occurred prior to calling Wait(), the wait condition will
    be immediately satisfied, and the task will continue to run without
    delay.

CAUTION
    This function cannot be called while in supervisor mode or
    interrupts! This function will break the action of a Forbid() or
    Disable() call.

INPUT
    signalSet - The set of signals for which to wait.
                Each bit represents a particular signal.

RESULTS
    signals - the set of signals that were active

1.106 exec.library/WaitIO

NAME
    WaitIO -- wait for completion of an I/O request

SYNOPSIS
    error = WaitIO(iORequest)
    D0   A1

    BYTE WaitIO(struct IORequest *);

FUNCTION
    This function waits for the specified I/O request to complete, then
    removes it from the replyport. If the I/O has already completed,
    this function will return immediately.

    This function should be used with care, as it does not return until
    the I/O request completes; if the I/O never completes, this
    function will never return, and your task will hang. If this
    situation is a possibility, it is safer to use the Wait() function.
    Wait() will return return when any of a specified set of signal is
    received. This is how I/O timeouts can be properly handled.

WARNING
    If this IORequest was "Quick" or otherwise finished BEFORE this
    call, this function drops though immediately, with no call to
    Wait(). A side effect is that the signal bit related the port may
    remain set. Expect this.

    When removing a known complete IORequest from a port, WaitIO() is the
    preferred method. A simple Remove() would require a Disable/Enable
    pair!

INPUTS
    iORequest - pointer to an I/O request block

RESULTS
    error - zero if successful, else an error is returned
            (a sign extended copy of io_Error).

SEE ALSO
    DoIO, SendIO, CheckIO, AbortIO

1.107 exec.library/WaitPort

NAME
    WaitPort -- wait for a given port to be non-empty

SYNOPSIS
    message = WaitPort(port)
    D0   A0

    struct Message *WaitPort(struct MsgPort *);

FUNCTION
    This function waits for the given port to become non-empty. If
    necessary, the Wait() function will be called to wait for the port
    signal. If a message is already present at the port, this function
    will return immediately. The return value is always a pointer to
    the first message queued (but it is not removed from the queue).

CAUTION
    More than one message may be at the port when this returns. It is
    proper to call the GetMsg() function in a loop until all messages
    have been handled, then wait for more to arrive.

    To wait for more than one port, combine the signal bits from each
    port into one call to the Wait() function, then use a GetMsg() loop
    to collect any and all messages. It is possible to get a signal
    for a port WITHOUT a message showing up. Plan for this.

INPUT
    port - a pointer to the message port

RETURN
    message - a pointer to the first available message

SEE ALSO
    GetMsg