Libraries

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

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 21 Exec Tasks

One of the most powerful features of the Amiga operating system is its ability to run and manage multiple independent program tasks, providing each task with processor time based on their priority and activity. These tasks include system device drivers, background utilities, and user interface environments, as well as normal application programs. This multitasking capability is provided by the Exec library's management of task creation, termination, scheduling, event signals, traps, exceptions, and mutual exclusion.

This chapter deals with Exec on a lower level than most applications programmers need and assumes you are already familiar with the Exec basics discussed in the "Introduction to Exec" chapter of this manual.

1.2 21 Exec Tasks / Task Structure

Exec maintains task context and state information in a task-control data structure. Like most Exec structures, Task structures are dynamically linked onto various task queues through the use of an embedded Exec list Node structure (see the "Exec Lists and Queues" chapter). Any task can find its own task structure by calling FindTask(NULL). The C-language form of this structure is defined in the <exec/tasks.h> include file:

```
struct Task  {
    struct Node tc_Node;
    UBYTE       tc_Flags;
    UBYTE       tc_State;
    BYTE        tc_IDNestCnt;   /* intr disabled nesting */
    BYTE        tc_TDNestCnt;   /* task disabled nesting */
    ULONG       tc_SigAlloc;    /* sigs allocated */
    ULONG       tc_SigWait;     /* sigs we are waiting for */
```

```
        ULONG       tc_SigRecvd;    /* sigs we have received */
        ULONG       tc_SigExcept;   /* sigs we will take excepts for */
        UWORD       tc_TrapAlloc;   /* traps allocated */
        UWORD       tc_TrapAble;    /* traps enabled */
        APTR        tc_ExceptData;  /* points to except data */
        APTR        tc_ExceptCode;  /* points to except code */
        APTR        tc_TrapData;    /* points to trap code */
        APTR        tc_TrapCode;    /* points to trap data */
        APTR        tc_SPReg;       /* stack pointer */
        APTR        tc_SPLower;     /* stack lower bound */
        APTR        tc_SPUpper;     /* stack upper bound + 2*/
        VOID      (*tc_Switch)();   /* task losing CPU */
        VOID      (*tc_Launch)();   /* task getting CPU */
        struct List tc_MemEntry;    /* allocated memory */
        APTR        tc_UserData;    /* per task data */
    };
```

A similar assembly code structure is available in the <exec/tasks.i> include file.

Most of these fields are not relevant for simple tasks; they are used by Exec for state and administrative purposes. A few fields, however, are provided for the advanced programs that support higher level environments (as in the case of processes) or require precise control (as in devices). The following sections explain these fields in more detail.

1.3 21 Exec Tasks / Task Creation

To create a new task you must allocate a task structure, initialize its various fields, and then link it into Exec with a call to AddTask(). The task structure may be allocated by calling the AllocMem() function with the MEMF_CLEAR and MEMF_PUBLIC allocation attributes. These attributes indicate that the data structure is to be pre-initialized to zero and that the structure is shared.

The Task fields that require initialization depend on how you intend to use the task. For the simplest of tasks, only a few fields must be initialized:

tc_Node
: The task list node structure. This includes the task's priority, its type, and its name (refer to the chapter "Exec Lists and Queues").

tc_SPLower
: The lower memory bound of the task's stack.

tc_SPUpper
: The upper memory bound of the task's stack.

tc_SPReg
: The initial stack pointer. Because task stacks grow downward in memory, this field is usually set to the same value as tc_SPUpper.

Zeroing all other unused fields will cause Exec to supply the appropriate system default values. Allocating the structure with the MEMF_CLEAR attribute is an easy way to be sure that this happens.

Once the structure has been initialized, it must be linked to Exec. This is done with a call to AddTask() in which the following parameters are specified:

```
AddTask(struct Task *task, APTR initialPC, APTR finalPC )
```

The task argument is a pointer to your initialized Task structure. Set initialPC to the entry point of your task code. This is the address of the first instruction the new task will execute.

Set finalPC to the address of the finalization code for your task. This is a code section that will receive control if the initialPC routine ever performs a return (RTS). This exists to prevent your task from being launched into random memory upon an accidental return. The finalPC routine should usually perform various program-related clean-up duties and should then remove the task. If a zero is supplied for this parameter, Exec will use its default finalization code (which simply calls the RemTask() function).

Under Release 2, AddTask() returns the address of the newly added task or NULL for failure. Under 1.3 and older versions of the OS, no values are returned.

Task Creation With Amiga.lib Task Stack Task Priority

1.4 21 / Task Creation / Task Creation With Amiga.lib

A simpler method of creating a task is provided by the amiga.lib Exec support function CreateTask(), which can be accessed if your code is linked with amiga.lib.

```
CreateTask(char *name, LONG priority, APTR initialPC, ULONG stacksize)
```

A task created with CreateTask() may be removed with the amiga.lib DeleteTask() function, or it may simply return when it is finished. CreateTask() adds a MemList to the tc_MemEntry of the task it creates, describing all memory it has allocated for the task, including the task stack and the Task structure itself. This memory will be deallocated by Exec when the task is either explicitly removed (RemTask() or DeleteTask()) or when it exits to Exec's default task removal code (RemTask()).

Note that a bug in the CreateTask() code caused a failed memory allocation to go unnoticed in V33 and early versions of Release 2 amiga.lib.

If your development language is not linkable with amiga.lib, it may provide an equivalent built-in function, or you can create your own based on the createtask.c code in the Amiga ROM Kernel Reference Manual: Includes and Autodocs.

Depending on the priority of a new task and the priorities of other tasks

in the system, the newly added task may begin execution immediately.

Sharing Library Pointers

Although in most cases it is possible for a parent task to pass a library base to a child task so the child can use that library, for some libraries, this is not possible. For this reason, the only library base sharable between tasks is Exec's library base.

Here is an example of simple task creation. In this example there is no coordination or communication between the main process and the simple task it has created. A more complex example might use named ports and messages to coordinate the activities and shutdown of two tasks. Because our task is very simple and never calls any system functions which could cause it to be signalled or awakened, we can safely remove the task at any time.

Keep This In Mind.

Because the simple task's code is a function in our program, we must stop the subtask before exiting.

simpletask.c

1.5 21 / Task Creation / Task Stack

Every task requires a stack. All task stacks are user mode stacks (in the language of the 68000) and are addressed through the A7 CPU register. All normal code execution occurs on this task stack. Special modes of execution (processor traps and system interrupts for example) execute on a single supervisor mode stack and do not directly affect task stacks.

Task stacks are normally used to store local variables, subroutine return addresses, and saved register values. Additionally, when a task loses the processor, all of its current registers are preserved on this stack (with the exception of the stack pointer itself, which must be saved in the task structure).

The amount of stack used by a task can vary widely. The theoretical minimum stack size is 72 bytes, which is the number required to save 17 CPU registers and a single return address. Of course, a stack of this size would not give you adequate space to perform any subroutine calls (because the return address occupies stack space). On the other hand, a stack size of 1K would suffice to call most system functions but would not allow much in the way of local variable storage. Processes that call DOS library functions need an additional 1500 bytes of stack.

Because stack-bounds checking is not provided as a service of Exec, it is important to provide enough space for your task stack. Stack overflows are always difficult to debug and may result not only in the erratic failure of your task but also in the mysterious malfunction of other Amiga subsystems. Some compilers provide a stack-checking option.

You Can't Always Check The Stack.

Such stack-checking options generally cannot be used if part of your

code will be running on the system stack (interrupts, 680x0 exceptions, handlers, servers), or on a different task's stack (libraries, devices, created tasks).

When choosing your stack size, do not cut it too close. Remember that any recursive routines in your code may use varying amounts of stack, and that future versions of system routines may use additional stack variables. By dynamically allocating buffers and arrays, most application programs can be designed to function comfortably within the default process stack size of 4000 bytes.

1.6 21 / Task Creation / Task Priority

A task's priority indicates its importance relative to other tasks. Higher-priority tasks receive the processor before lower-priority tasks do. Task priority is stored as a signed number ranging from -128 to +127. Higher priorities are represented by more positive values; zero is considered the neutral priority. Normally, system tasks execute somewhere in the range of +20 to -20, and most application tasks execute at priority 0.

It is not wise to needlessly raise a task's priority. Sometimes it may be necessary to carefully select a priority so that the task can properly interact with various system tasks. The SetTaskPri() Exec function is provided for this purpose.

1.7 21 Exec Tasks / Task Termination

Task termination may occur as the result of a number of situations:

* A program returning from its initialPC routine and dropping into its finalPC routine or the system default finalizer.

* A task trap that is too serious for a recovery action. This includes traps like processor bus error, odd address access errors, etc.

* A trap that is not handled by the task. For example, the task might be terminated if your code happened to encounter a processor TRAP instruction and you did not provide a trap handling routine.

* An explicit call to Exec RemTask() or amiga.lib DeleteTask().

Task termination involves the deallocation of system resources and the removal of the task structure from Exec. The most important part of task termination is the deallocation of system resources. A task must return all memory that it allocated for its private use, it must terminate any outstanding I/O commands, and it must close access to any system libraries or devices that it has opened.

It is wise to adopt a strategy for task clean-up responsibility. You should decide whether resource allocation and deallocation is the duty of the creator task or the newly created task. Often it is easier and safer

for the creator to handle the resource allocation and deallocation on behalf of its offspring. In such cases, before removing the child task, you must make sure it is in a safe state such as Wait(0L) and not still using a resources or waiting for an event or signal that might still occur.

NOTE:

Certain resources, such as signals and created ports, must be allocated and deallocated by the same task that will wait on them. Also note that if your subtask code is part of your loaded program, you must not allow your program to exit before its subtasks have cleaned up their allocations, and have been either deleted or placed in a safe state such as Wait(0L).

1.8 21 Exec Tasks / Task Exclusion

From time to time the advanced system program may find it necessary to access global system data structures. Because these structures are shared by the system and by other tasks that execute asynchronously to your task, a task must prevent other tasks from using these structures while it is reading from or writing to them. This can be accomplished by preventing the operating system from switching tasks by forbidding or disabling. A section of code that requires the use of either of these mechanisms to lock out access by others is termed a critical section. Use of these methods is discouraged. For arbitrating access to data between your tasks, semaphores are a superior solution. (See the "Exec Semaphores" chapter)

Forbidding Task Switching Disabling Tasks Task Semaphores

1.9 21 / Task Exclusion / Forbidding Task Switching

Forbidding is used when a task is accessing shared structures that might also be accessed at the same time from another task. It effectively eliminates the possibility of simultaneous access by imposing nonpreemptive task scheduling. This has the net effect of disabling multitasking for as long as your task remains in its running state. While forbidden, your task will continue running until it performs a call to Wait() or exits from the forbidden state. Interrupts will occur normally, but no new tasks will be dispatched, regardless of their priorities.

When a task running in the forbidden state calls the Wait() function, directly or indirectly, it implies a temporary exit from its forbidden state. Since almost all stdio, device I/O, and file I/O functions must Wait() for I/O completion, performing such calls will cause your task to Wait(), temporarily breaking the forbid. While the task is waiting, the system will perform normally. When the task receives one of the signals it is waiting for, it will again reenter the forbidden state. To become forbidden, a task calls the Forbid() function. To escape, the Permit() function is used. The use of these functions may be nested with the expected affects; you will not exit the forbidden mode until you call the outermost Permit().

As an example, the Exec task list should only be accessed when in a Forbid() state. Accessing the list without forbidding could lead to incorrect results or it could crash the entire system. To access the task list also requires the program to disable interrupts which is discussed in the next section.

1.10 21 / Task Exclusion / Disabling Tasks

Disabling is similar to forbidding, but it also prevents interrupts from occurring during a critical section. Disabling is required when a task accesses structures that are shared by interrupt code. It eliminates the possibility of an interrupt accessing shared structures by preventing interrupts from occurring. Use of disabling is strongly discouraged.

To disable interrupts you can call the Disable() function. To enable interrupts again, use the Enable() function. Although assembler DISABLE and ENABLE macros are provided, assembler programmers should use the system functions rather than the macros for upwards compatibility, ease of debugging, and smaller code size.

Like forbidden sections, disabled sections can be nested. To restore normal interrupt processing, an Enable() call must be made for every Disable(). Also like forbidden sections, any direct or indirect call to the Wait() function will enable interrupts until the task regains the processor.

> WARNING:
> --------
> It is important to realize that there is a danger in using disabled sections. Because the software on the Amiga depends heavily on its interrupts occurring in nearly real time, you cannot disable for more than a very brief instant. Disabling interrupts for more than 250 microseconds can interfere with the normal operation of vital system functions, especially serial I/O.
>
> WARNING:
> --------
> Masking interrupts by changing the 68000 processor interrupt priority levels with the MOVE SR instruction can also be dangerous and is very strongly discouraged. The disable- and enable-related functions control interrupts through the 4703 custom chip and not through the 68000 priority level. In addition, the processor priority level can be altered only from supervisor mode (which means this process is much less efficient).

It is never necessary to both Disable() and Forbid(). Because disabling prevents interrupts, it also prevents preemptive task scheduling. When disable is used within an interrupt, it will have the effect of locking out all higher level interrupts (lower level interrupts are automatically disabled by the CPU). Many Exec lists can only be accessed while disabled. Suppose you want to print the names of all system tasks. You would need to access both the TaskReady and TaskWait lists from within a single disabled section. In addition, you must avoid calling system functions that would break a disable by an indirect call to Wait()

(printf() for example). In this example, the names are gathered into a list while task switching is disabled. Then task switching is enabled and the names are printed.

tasklist.c

1.11 21 / Task Exclusion / Task Semaphores

Semaphores can be used for the purposes of mutual exclusion. With this method of locking, all tasks agree on a locking convention before accessing shared data structures. Tasks that do not require access are not affected and will run normally, so this type of exclusion is considered preferable to forbidding and disabling. This form of exclusion is explained in more detail in the "Exec Semaphores" chapter.

1.12 21 Exec Tasks / Task Exceptions

Exec can provide a task with its own task-local "interrupt" called an exception. When some exceptional event occurs, an Exec exception occurs which stops a particular task from executing its normal code and forces it to execute a special, task-specific exception handling routine.

If you are familiar with the 680x0, you may be used to using the term "exceptions" in a different way. The 680x0 has its own form of exception that has nothing to do with an Exec exception. These are discussed in more detail in the "Task Traps" section of this chapter. Do not confuse Exec exceptions with 680x0 exceptions.

To set up an exception routine for a task requires setting values in the task's control structure (the Task structure). The tc_ExceptCode field should point to the task's exception handling routine. If this field is zero, Exec will ignore all exceptions. The tc_ExceptData field should point to any data the exception routine needs.

Exec exceptions work using signals. When a specific signal or signals occur, Exec will stop a task and execute its exception routine. Use the Exec function SetExcept() to tell Exec which of the task's signals should trigger the exception.

When an exception occurs, Exec stops executing the tasks normal code and jumps immediately into the exception routine, no matter what the task was doing. The exception routine operates in the same context the task's normal code; it operates in the CPU's user mode and uses the task's stack.

Before entering the exception routine, Exec pushes the normal task code's context onto the stack. This includes the PC, SR, D0-D7, and A0-A6 registers. Exec then puts certain parameters in the processor registers for the exception routine to use. D0 contains a signal mask indicating which signal bit or bits caused the exception. Exec disables these signals when the task enters its exception routine. If more than one signal bit is set (i.e. if two signals occurred simultaneously), it is up to the exception routine to decide in what order to process the two different

signals. A1 points to the related exception data (from tc_ExceptData), and A6 contains the Exec library base. You can think of an exception as a subtask outside of your normal task. Because task exception code executes in user mode, however, the task stack must be large enough to supply the extra space consumed during an exception.

While processing a given exception, Exec prevents that exception from occurring recursively. At exit from your exception-processing code, you should make sure D0 contains the signal mask the exception routine received in D0 because Exec looks here to see which signals it should reactivate. When the task executes the RTS instruction at the end of the exception routine, the system restores the previous contents of all of the task registers and resumes the task at the point where it was interrupted by the exception signal.

Exceptions Are Tricky.

Exceptions are difficult to use safely. An exception can interrupt a task that is executing a critical section of code within a system function, or one that has locked a system resource such as the disk or blitter (note that even simple text output uses the blitter.) This possibility makes it dangerous to use most system functions within an exception unless you are sure that your interrupted task was performing only local, non-critical operations.

1.13 21 Exec Tasks / Task Traps

Task traps are synchronous exceptions to the normal flow of program control. They are always generated as a direct result of an operation performed by your program's code. Whether they are accidental or purposely generated, they will result in your program being forced into a special condition in which it must immediately handle the trap. Address error, privilege violation, zero divide, and trap instructions all result in task traps. They may be generated directly by the 68000 processor (Motorola calls them "exceptions") or simulated by software.

A task that incurs a trap has no choice but to respond immediately. The task must have a module of code to handle the trap. Your task may be aborted if a trap occurs and no means of handling it has been provided. Default trap handling code (tc_TrapCode) is provided by the OS. You may instead choose to do your own processing of traps. The tc_TrapCode field is the address of the handler that you have designed to process the trap. The tc_TrapData field is the address of the data area for use by the trap handler.

The system's default trap handling code generally displays a Software Error Requester or Alert containing an exception number and the program counter or task address. Processor exceptions generally have numbers in the range hex 00 to 2F. The 68000 processor exceptions of particular interest are as follows.

Table 21-1: Traps (68000 Exception Vector Numbers)

2	Bus error	access of nonexistent memory

```
3       Address error         long/word access of odd address (68000)
4       Illegal instruction   illegal opcode (other than Axxx or Fxxx)
5       Zero divide           processor division by zero
6       CHK instruction       register bounds error trap by CHK
7       TRAPV instruction     overflow error trap by TRAPV
8       Privilege violation   user execution of supervisor opcode
9       Trace                 status register TRACE bit trap
10      Line 1010 emulator    execution of opcode beginning with $A
11      Line 1111 emulator    execution of opcode beginning with $F
32-47   Trap instructions     TRAP N instruction where N = 0 to 15
```

A system alert for a processor exception may set the high bit of the longword exception number to indicate an unrecoverable error (for example $80000005 for an unrecoverable processor exception #5). System alerts with more complex numbers are generally Amiga-specific software failures. These are built from the definitions in the <exec/alerts.h> include file.

The actual stack frames generated for these traps are processor-dependent. The 68010, 68020, and 68030 processors will generate a different type of stack frame than the 68000. If you plan on having your program handle its own traps, you should not make assumptions about the format of the supervisor stack frame. Check the flags in the AttnFlags field of the ExecBase structure for the type of processor in use and process the stack frame accordingly.

Trap Handlers Trap Instructions

1.14 21 / Task Traps / Trap Handlers

For compatibility with the 68000, Exec performs trap handling in supervisor mode. This means that all task switching is disabled during trap handling. At entry to the task's trap handler, the system stack contains a processor-dependent trap frame as defined in the 68000/10/20/30 manuals. A longword exception number is added to this frame. That is, when a handler gains control, the top of stack contains the exception number and the trap frame immediately follows.

To return from trap processing, remove the exception number from the stack (note that this is the supervisor stack, not the user stack) and then perform a return from exception (RTE).

Because trap processing takes place in supervisor mode, with task dispatching disabled, it is strongly urged that you keep trap processing as short as possible or switch back to user mode from within your trap handler. If a trap handler already exists when you add your own trap handler, it is smart to propagate any traps that you do not handle down to the previous handler. This can be done by saving the previous address from tc_TrapCode and having your handler pass control to that address if the trap which occurred is not one you wish to handle.

The following example installs a simple trap handler which intercepts processor divide-by-zero traps, and passes on all other traps to the previous default trap code. The example has two code modules which are linked together. The trap handler code is in assembler. The C module

installs the handler, demonstrates its effectiveness, then restores the previous tc_TrapCode.

trap_c.c example

1.15 21 / Task Traps / Trap Instructions

The TRAP instructions in the 68000 generate traps 32-47. Because many independent pieces of system code may desire to use these traps, the AllocTrap() and FreeTrap() functions are provided. These work in a fashion similar to that used by AllocSignal() and FreeSignal(), mentioned in the "Exec Signals" chapter.

Allocating a trap is simply a bookkeeping job within a task. It does not affect how the system calls the trap handler; it helps coordinate who owns what traps. Exec does nothing to determine whether or not a task is prepared to handle a particular trap. It simply calls your code. It is up to your program to handle the trap.

To allocate any trap, you can use the following code:

```
    if (-1 == (trap = AllocTrap(-1)))
        printf("all trap instructions are in use\n");
```

Or you can select a specific trap using this code:

```
    if (-1 == (trap = AllocTrap(3)))
        printf("trap #3 is in use\n");
```

To free a trap, you use the FreeTrap() function passing it the trap number to be freed.

1.16 21 Exec Tasks / Processor and Cache Control

Exec provides a number of to control the processor mode and, if available, the caches. All these functions work independently of the specific M68000 family processor type. This enables you to write code which correctly controls the state of both the MC68000 and the MC68040. Along with processor mode and cache control, functions are provided to obtain information about the condition code register (CCR) and status register (SR). No functions are provided to control a paged memory management unit (PMMU) or floating point unit (FPU).

Table 21-2: Processor and Cache Control Functions

Function	Description
GetCC()	Get processor condition codes.
SetSR()	Get/set processor status register.
SuperState()	Set supervisor mode with user stack.

Supervisor()	Execute a short supervisor mode function.
UserState()	Return to user mode with user stack.
CacheClearE()	Flush CPU instruction and/or data caches (V37).
CacheClearU()	Flush CPU instruction and data caches (V37).
CacheControl()	Global cache control (V37).
CachePostDMA()	Perform actions prior to hardware DMA (V37).
CachePreDMA()	Perform actions after hardware DMA (V37).

Supervisor Mode Status Register Condition Code Register Cache Functions DMA Cache Functions The 68040 and CPU Caches

1.17 21 / Processor and Cache Control / Supervisor Mode

While in supervisor mode, you have complete access to all data and registers, including those used for task scheduling and exceptions, and can execute privileged instructions. In application programs, normally only task trap code is directly executed in supervisor mode, to be compatible with the MC68000. For normal applications, it should never be necessary to switch to supervisor mode itself, only indirectly through Exec function calls. Remember that task switching is disabled while in supervisor mode. If it is absolutely needed to execute code in supervisor mode, keep it as brief as possible.

Supervisor mode can only be entered when a 680x0 exception occurs (an interrupt or trap). The Supervisor() function allows you to trap an exception to a specified assembly function. In this function your have full access to all registers. No registers are saved when your function is invoked. You are responsible for restoring the system to a sane state when you are done. You must return to user mode with an RTE instruction. You must not return to user mode by executing a privileged instruction which clears the supervisor bit in the status register. Refer to a manual on the M68000 family of CPUs for information about supervisor mode and available privileged instructions per processor type.

The MC68000 has two stacks, the user stack (USP) and supervisor stack (SSP). As of the MC68020 there are two supervisor stacks, the interrupt stack pointer (ISP) and the master stack pointer (MSP). The SuperState() function allows you to enter supervisor mode with the USP used as SSP. The function returns the SSP, which will be the MSP, if an MC68020 or greater is used. Returning to user mode is done with the UserState() function. This function takes the SSP as argument, which must be saved when SuperState() is called. Because of possible problems with stack size, Supervisor() is to be preferred over SuperState().

1.18 21 / Processor and Cache Control / Status Register

The processor status register bits can be set or read with the SetSR() function. This function operates in supervisor mode, thus both the upper and lower byte of the SR can be read or set. Be very sure you know what

you are doing when you use this function to set bits in the SR and above all never try to use this function to enter supervisor mode. Refer to the M68000 Programmers Reference Manual by Motorola Inc. for information about the definition of individual SR bits per processor type.

1.19 21 / Processor and Cache Control / Condition Code Register

On the MC68000 a copy of the processor condition codes can be obtained with the MOVE SR,<ea> instruction. On MC68010 processors and up however, the instruction MOVE CCR,<ea> must be used. Using the specific MC68000 instruction on later processors will cause a 680x0 exception since it is a privileged instruction on those processors. The GetCC() function provides a processor independent way of obtaining a copy of the condition codes. For all processors there are 5 bits which can indicate the result of an integer or a system control instruction:

```
    X - extend    N - negative    Z - zero    V - overflow    C - carry
```

The X bit is used for multiprecision calculations. If used, it is copy of the carry bit. The other bits state the result of a processor operation.

1.20 21 / Processor and Cache Control / Cache Functions

As of the MC68020 all processors have an instruction cache, 256 bytes on the MC68020 and MC68030 and 4 KBytes on a MC68040. The MC68030 and MC68040 have data caches as well, 256 bytes and 4 KBytes respectively. All the processors load instructions ahead of the program counter (PC), albeit it that the MC68000 and MC68010 only prefetch one and two words respectively. This means the CPU loads instructions ahead of the current program counter. For this reason self-modifying code is strongly discouraged. If your code modifies or decrypts itself just ahead of the program counter, the pre-fetched instructions may not match the modified instructions. If self-modifying code must be used, flushing the cache is the safest way to prevent this.

1.21 21 / Processor and Cache Control / DMA Cache Functions

The CachePreDMA() and CachePostDMA() functions allow you to flush the data cache before and after Direct Memory Access. Typically only DMA device drivers benefit from this. These functions take the processor type, possible MMU and cache mode into account. When no cache is available they end up doing nothing. These functions can be replaced with ones suitable for different cache hardware. Refer to the ROM Kernel Reference Manual: Includes and Autodocs for implementation specifics.

Since DMA device drivers read and write directly to memory, they are effected by the CopyBack feature of the MC68040 (explained below). Using DMA with CopyBack mode requires a cache flush. If a DMA device needs to read RAM via DMA, it must make sure that the data in the caches has been written to memory first, by calling CachePreDMA(). In case of a write to

memory, the DMA device should first clear the caches with CachePreDMA(), write the data and flush the caches again with CachePostDMA().

1.22 21 / Processor and Cache Control / The 68040 and CPU Caches

The 68040 is a much more powerful CPU than its predecessors. It has 4K of cache memory for instructions and another 4K cache for data. The reason for these two separate caches is so that the CPU core can access data and CPU instructions at the same time.

Although the 68040 provides greater performance it also brings with it greater compatibility problems. Just the fact that the caches are so much larger than Motorola's 68030 CPU can cause problems. However, this is not its biggest obstacle.

The 68040 data cache has a mode that can make the system run much faster in most cases. It is called CopyBack mode. When a program writes data to memory in this mode, the data goes into the cache but not into the physical RAM. That means that if a program or a piece of hardware were to read that RAM without going through the data cache on the 68040, it will read old data. CopyBack mode effects two areas of the Amiga: DMA devices and the CPU's instruction reading.

CopyBack mode effects DMA devices because they read and write data directly to memory. Using DMA with CopyBack mode requires a cache flush. If a DMA device needs to read RAM via DMA, it must first make sure that data in the caches has been written to memory. It can do this by calling the Exec function CachePreDMA(). If a DMA device is about to write to memory, it should call CachePreDMA() before the write, do the DMA write, and then call CachePostDMA(), which makes sure that the CPU uses the data just written to memory.

An added advantage of using the CachePreDMA() and CachePostDMA() functions is that they give the OS the chance to tell the DMA device that the physical addresses and memory sizes are not the same. This will make it possible in the future to add features such as virtual memory. See the Autodocs for more information on these calls.

The other major compatibility problem with the 68040's CopyBack mode is with fetching CPU instructions. CPU instructions have to be loaded into memory so the CPU can copy them into its instruction cache. Normally, instructions that will be executed are written to memory by the CPU (i.e., loading a program from disk). In CopyBack mode, anything the CPU writes to memory, including CPU instructions, doesn't actually go into memory, it goes into the data cache. If instructions are not flushed out of the data cache to RAM, the 68040 will not be able to find them when it tries to copy them into the instruction cache for execution. It will instead find and attempt to execute whatever garbage data happened to be left at that location in RAM.

To remedy this, any program that writes instructions to memory must flush the data cache after writing. The V37 Exec function CacheClearU() takes care of this. Release 2 of the Amiga OS correctly flushes the caches as needed after it does the LoadSeg() of a program (LoadSeg() loads Amiga executable programs into memory from disk). Applications need to do the

same if they write code to memory. It can do that by calling CacheClearU() before the call to CreateProc(). In C that would be:

```
    extern struct ExecBase *SysBase;
    . . .

    /* If we are in 2.0, call CacheClearU() before CreateProc() */
    if (SysBase->LibNode.lib_Version >= 37) CacheClearU();

    /* Now do the CreateProc() call... */
    proc=CreateProc(... /* whatever your call is like */ ...);
    . . .
```

For those of you programming in assembly language:

```
    ************************************************************************
    * Check to see if we are running in V37 ROM or better.  If so, we want
    * to call CacheClearU() to make sure we are safe on future hardware
    * such as the 68040.  This section of code assumes that a6 points at
    * ExecBase.  a0/a1/d0/d1 are trashed in CacheClearU()
    *
            cmpi.w  #37,LIB_VERSION(a6)     ; Check if exec is >= V37
            bcs.s   TooOld                  ; If less than V37, too old...
            jsr     _LVOCacheClearU(a6)     ; Clear the cache...
    TooOld:                                 ; Exit gracefully.
    ************************************************************************
```

Note that CreateProc() is not the only routine where CopyBack mode could be a problem. Any program code copied into memory for execution that is not done via LoadSeg() will need to call CacheClearU(). Many input device handlers have been known to allocate and copy the handler code into memory and then exit back to the system. These programs also need to have this call in them. The above code will work under older versions of the OS, and will do the correct operations in Release 2 (and beyond).

1.23 21 Exec Tasks / Function Reference

The following chart gives a brief description of the Exec functions that control tasks. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details about each call.

Table 21-3: Exec Task, Processor and Cache Control Functions

Exec Task Function	Description
AddTask()	Add a task to the system.
AllocTrap()	Allocate a processor trap vector.
Disable()	Disable interrupt processing.
Enable()	Enable interrupt processing.
FindTask()	Find a specific task.
Forbid()	Forbid task rescheduling.
FreeTrap()	Release a process trap.

Permit()	Permit task rescheduling.
SetTaskPri()	Set the priority of a task.
RemTask()	Remove a task from the system.
CacheClearE()	Flush CPU instruction and/or data caches (V37).
CacheClearU()	Flush CPU instruction and data caches (V37).
CacheControl()	Global cache control (V37).
CachePostDMA()	Perform actions prior to hardware DMA (V37).
CachePreDMA()	Perform actions after hardware DMA (V37).
GetCC()	Get processor condition codes.
SetSR()	Get/set processor status register.
SuperState()	Set supervisor mode with user stack.
Supervisor()	Execute a short supervisor mode function.
UserState()	Return to user mode with user stack.
CreateTask()	Amiga.lib function to setup and add a new task.
DeleteTask()	Amiga.lib function to delete a task created with CreateTask().