

Libraries

COLLABORATORS

	<i>TITLE :</i> Libraries		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries	1
1.1	Amiga® RKM Libraries: 26 Exec Interrupts	1
1.2	26 Exec Interrupts / Introduction	1
1.3	26 / Introduction / Sequence of Events During an Interrupt	1
1.4	26 / Introduction / Interrupt Priorities	3
1.5	26 / Introduction / Nonmaskable Interrupt	4
1.6	26 Exec Interrupts / Servicing Interrupts	4
1.7	26 / Servicing Interrupts / Interrupt Data Structure	4
1.8	26 / Servicing Interrupts / Environment	5
1.9	26 / Servicing Interrupts / Interrupt Handlers	6
1.10	26 // Interrupt Handlers / Interrupt Handler Register Usage	6
1.11	26 / Servicing Interrupts / Interrupt Servers	7
1.12	26 // Interrupt Servers / Interrupt Server Register Usage	8
1.13	26 Exec Interrupts / Software Interrupts	9
1.14	26 Exec Interrupts / Disabling Interrupts	10
1.15	26 Exec Interrupts / Function Reference	10

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 26 Exec Interrupts

Introduction	Software Interrupts	Function Reference
Servicing Interrupts	Disabling Interrupts	

1.2 26 Exec Interrupts / Introduction

Exec manages the decoding, dispatching, and sharing of all system interrupts. This includes control of hardware interrupts, software interrupts, task-relative interrupts (see the discussion of exceptions in the "Exec Tasks" chapter), and interrupt disabling and enabling. In addition, Exec supports a more extended prioritization of interrupts than that provided in the 68000.

The proper operation of multitasking depends heavily on the consistent management of the interrupt system. Task activities are often driven by intersystem communication that is originated by various interrupts.

Sequence of Events During an Interrupt
Interrupt Priorities
Nonmaskable Interrupt

1.3 26 / Introduction / Sequence of Events During an Interrupt

Before useful interrupt handling code can be executed, a considerable amount of hardware and software activity must occur. Each interrupt must propagate through several hardware and software interfaces before application code is finally dispatched:

- * A hardware device decides to cause an interrupt and sends a signal to the interrupt control portions of the 4703 (Paula) custom chip.
- * The 4703 interrupt control logic notices this new signal and performs two primary operations. First, it records that the interrupt has been requested by setting a flag bit in the INTREQ register. Second,

it examines the INTENA register to determine whether the corresponding interrupt and the interrupt master are enabled. If both are enabled, the 4703 generates an interrupt request by placing the priority level of the request onto the three 68000 interrupt control input lines (IPL0, IPL1, IPL2).

- * These three signals correspond to seven interrupt priority levels in the 68000. If the priority of the new interrupt is greater than the current processor priority, an interrupt sequence is initiated. The priority level of the new interrupt is used to index into the top seven words of the processor address space. The odd byte (a vector number) of the indexed word is fetched and then shifted left by two to create an offset into the processor's auto-vector interrupt table. The vector offsets used are in the range of \$064 to \$07C. These are labeled as interrupt autovectors in the 68000 manual. The auto-vector table appears in low memory on a 68000 system, but its location for other 68000 family processors is determined by the processor's CPU Vector Base Register (VBR). VBR can be accessed from supervisor mode with the MOVEC instruction.
- * The processor then switches into supervisor mode (if it is not already in that mode), and saves copies of the status register and program counter (PC) onto the top of the system stack (additional information may be saved by processors other than the 68000). The processor priority is then raised to the level of the active interrupt.
- * From the low memory vector address (calculated in step three above), a 32-bit autovector address is fetched and loaded into the program counter. This is an entry point into Exec's interrupt dispatcher.
- * Exec must now further decode the interrupt by examining the INTREQ and INTENA 4703 chip registers. Once the active interrupt has been determined, Exec indexes into an ExecBase array to fetch the interrupt's handler entry point and handler data pointer addresses.
- * Exec now turns control over to the interrupt handler by calling it as if it were a subroutine. This handler may deal with the interrupt directly or may propagate control further by invoking interrupt server chain processing.

You can see from the above discussion that the interrupt autovectors should never be altered by the user. If you wish to provide your own system interrupt handler, you must use the Exec SetIntVector() function. You should not change the contents of any autovector location.

Task multiplexing usually occurs as the result of an interrupt. When an interrupt has finished and the processor is about to return to user mode, Exec determines whether task-scheduling attention is required. If a task was signaled during interrupt processing, the task scheduler will be invoked. Because Exec uses preemptive task scheduling, it can be said that the interrupt subsystem is the heart of task multiplexing. If, for some reason, interrupts do not occur, a task might execute forever because it cannot be forced to relinquish the CPU.

Table 26-1: Interrupts by Priority

Hardware Priority	Exec Pseudo- Priority	Description	Label	Type
-----	-----	-----	-----	-----
	1	Serial transmit buffer empty	TBE	H
1 ----	2	disk block complete	DSKBLK	H
	3	software interrupt	SOFTINT	H
2 ----	4	external INT2 & CIAA	PORTS	S
	5	graphics coprocessor	COPER	S
3 ----	6	vertical blank interval	VERTB	S
	7	blitter finished	BLIT	H
	8	audio channel 2	AUD2	H
4 ----	9	audio channel 0	AUD0	H
	10	audio channel 3	AUD3	H
	11	audio channel 1	AUD1	H
5 ----	12	Serial receive buffer full	RBF	H
	13	disk sync pattern found	DSKSYNC	H
6 ----	14	external INT6 & CIAB	EXTER	S
	15	special (master enable)	INTEN	-
7 ----	--	non-maskable interrupt	NMI	S

1.4 26 / Introduction / Interrupt Priorities

Interrupts are prioritized in hardware and software. The 68000 CPU priority at which an interrupt executes is determined strictly by hardware. In addition to this, the software imposes a finer level of pseudo-priorities on interrupts with the same CPU priority. These pseudo-priorities determine the order in which simultaneous interrupts of the same CPU priority are processed. Multiple interrupts with the same CPU priority but a different pseudo-priority will not interrupt one another. Interrupts are serviced by either an exclusive handler or by server chains to which many servers may be attached, as shown in the Type field of the table. The table above summarizes all interrupts by priority.

The 8520s (also called CIAs) are Amiga peripheral interface adapter chips that generate the INT2 and INT6 interrupts. For more information about them, see the Amiga Hardware Reference Manual.

As described in the Motorola 68000 programmer's manual, interrupts may nest only in the direction of higher priority. Because of the time-critical nature of many interrupts on the Amiga, the CPU priority level must never be changed by user or system code. When the system is running in user mode (multitasking), the CPU priority level must remain set at zero. When an interrupt occurs, the CPU priority is raised to the level appropriate for that interrupt. Lowering the CPU priority would permit unlimited interrupt recursion on the system stack and would "short-circuit" the interrupt-priority scheme.

Because it is dangerous on the Amiga to hold off interrupts for any period of time, higher-level interrupt code must perform its business and exit promptly. If it is necessary to perform a time-consuming operation as the result of a high-priority interrupt, the operation should be deferred either by posting a software interrupt or by signalling a task. In this way, interrupt response time is kept to a minimum. Software interrupts are described in a later section.

1.5 26 / Introduction / Nonmaskable Interrupt

The 68000 provides a nonmaskable interrupt (NMI) of CPU priority 7. Although this interrupt cannot be generated by the Amiga hardware itself, it can be generated on the expansion bus by external hardware. Because this interrupt does not pass through the 4703 interrupt controller circuitry, it is capable of violating system code critical sections. In particular, it short-circuits the DISABLE mutual-exclusion mechanism. Code that uses NMI must not assume that it can access system data structures.

1.6 26 Exec Interrupts / Servicing Interrupts

Interrupts are serviced on the Amiga through the use of interrupt handlers and servers. An interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt. An interrupt server is one of possibly many system routines that are invoked as the result of a single 4703 interrupt. Interrupt servers provide a means of interrupt sharing. This concept is useful for general-purpose interrupts such as vertical blanking.

At system start, Exec designates certain interrupts as handlers and others as server chains. The PORTS, COPER, VERTB, EXTER, and NMI interrupts are initialized as server chains. Therefore, each of these may execute multiple interrupt routines per each interrupt. All other interrupts are designated as handlers and are always used exclusively.

Interrupt Data Structure	Interrupt Handlers
Environment	Interrupt Servers

1.7 26 / Servicing Interrupts / Interrupt Data Structure

Interrupt handlers and servers are defined by the Exec Interrupt structure. This structure specifies an interrupt routine entry point and data pointer. The C definition of this structure is as follows:

```
struct Interrupt
{
    struct Node is_Node;
    APTR      is_Data;
    VOID      (*is_Code)();
};
```

Once this structure has been properly initialized, it can be used for either a handler or a server.

1.8 26 / Servicing Interrupts / Environment

Interrupts execute in an environment different from that of tasks. All interrupts execute in supervisor mode and utilize the single system stack. This stack is large enough to handle extreme cases of nested interrupts (of higher priorities). Interrupt processing has no effect on task stack usage.

All interrupt processing code, both handlers and servers, is invoked as assembly code subroutines. Normal assembly code register conventions dictate that the D0, D1, A0 and A1 registers be free for scratch use. In the case of an interrupt handler, some of these registers also contain data that may be useful to the handler code. See the section on handlers below.

Because interrupt processing executes outside the context of most system activities, certain data structures will not be self-consistent and must be considered off limits for all practical purposes. This happens because certain system operations are not atomic in nature and might be interrupted only after executing part of an important instruction sequence. For example, memory allocation and deallocation routines do not disable interrupts. This results in the possibility of interrupting a memory-related routine. In such a case, a memory linked list may be inconsistent during an interrupt. Therefore, interrupt routines must not use any memory allocation or deallocation functions.

In addition, interrupts may not call any system function which might allocate memory, wait, manipulate unprotected lists, or modify ExecBase->ThisTask data (for example Forbid(), Permit(), and mathieee libraries). In practice, this means that very few system calls may be used within interrupt code. The following functions may generally be used safely within interrupts:

```
Alert(), Disable(), Enable(), Signal(), Cause(),
GetMsg(), PutMsg(), ReplyMsg(), FindPort(), FindTask()
```

and if you are manipulating your own List structures while in an interrupt:

```
AddHead(), AddTail(), RemHead(), RemTail(), FindName()
```

In addition, certain devices (notably the timer device) specifically allow limited use of `SendIO()` and `BeginIO()` within interrupts.

1.9 26 / Servicing Interrupts / Interrupt Handlers

As described above, an interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt. There can only be one handler per 4703 interrupt. Every interrupt handler consists of an Interrupt structure (as defined above) and a single assembly code routine. Optionally, a data structure pointer may also be provided. This is particularly useful for ROM-resident interrupt code.

An interrupt handler is passed control as if it were a subroutine of `Exec`. Once the handler has finished its business, it must return to `Exec` by executing an `RTS` (return from subroutine) instruction rather than an `RTE` (return from exception) instruction. Interrupt handlers should be kept very short to minimize service-time overhead and thus minimize the possibilities of interrupt overruns. As described above, an interrupt handler has the normal scratch registers at its disposal. In addition, `A5` and `A6` are free for use. These registers are saved by `Exec` as part of the interrupt initiation cycle.

For the sake of efficiency, `Exec` passes certain register parameters to the handler (see the list below). These register values may be utilized to trim a few microseconds off the execution time of a handler. All of the following registers (`D0/D1/A0/A1/A5/A6`) may be used as scratch registers by an interrupt handler, and need not be restored prior to returning.

Don't Make Assumptions About Registers.

Interrupt servers have different register usage rules (see the "Interrupt Servers" section).

Interrupt Handler Register Usage

1.10 26 // Interrupt Handlers / Interrupt Handler Register Usage

Here are the register conventions for interrupt handlers.

`D0` Contains no valid information.

`D1` Contains the 4703 `INTENAR` and `INTREQR` registers values `AND'`ed together. This results in an indication of which interrupts are enabled and active.

`A0` Points to the base address of the Amiga custom chips. This information is useful for performing indexed instruction access to the chip registers.

`A1` Points to the data area specified by the `is_Data` field of the Interrupt structure. Because this pointer is always fetched (regardless of whether you use it), it is to your advantage to make

some use of it.

A5 Is used as a vector to your interrupt code.

A6 Points to the Exec library base (SysBase). You may use this register to call Exec functions or set it up as a base register to access your own library or device.

Interrupt handlers are established by passing the Exec function `SetIntVector()`, your initialized Interrupt structure, and the 4703 interrupt bit number of interest. The parameters for this function are as follows:

```
SetIntVector(ULONG intNumber, struct Interrupt *interrupt)
```

The first argument is the bit number for which this interrupt server is to respond (example `INTB_VERTB`). The possible bits for interrupts are defined in `<hardware/intbits.h>`. The second argument is the address of an interrupt server node as described earlier in this chapter. Keep in mind that certain interrupts are established as server chains and should not be accessed as handlers.

The following example demonstrates initialization and installation of an assembler interrupt handler. See the "Resources" chapter for more information on allocating resources, and the "Serial Device" chapter in the Amiga ROM Kernel Reference Manual: Devices for the more common method of serial communications.

`rbf.c`

The assembler interrupt handler code, `RBFHandler`, reads the complete word of serial input data from the serial hardware and then separates the character and flag bytes into separate buffers. When the buffers are full, the handler signals the main process causing main to print the character buffer contents, remove the handler, and exit.

`rbfhandler.asm`

NOTE.

The data structure containing the signal to use, task address pointer, and buffers is allocated and initialized in `main()`, and passed to the handler via the `is_Data` pointer of the Interrupt structure.

1.11 26 / Servicing Interrupts / Interrupt Servers

As mentioned above, an interrupt server is one of possibly many system interrupt routines that are invoked as the result of a single 4703 interrupt. Interrupt servers provide an essential mechanism for interrupt sharing.

Interrupt servers must be used for `PORTS`, `COPER`, `VERTB`, `EXTER`, or `NMI` interrupts. For these interrupts, all servers are linked together in a chain. Every server in the chain will be called in turn as long as the

previous server returned with the processor's Z (zero) flag set. If you determine that an interrupt was specifically for your server, you should return with the processor's Z flag cleared (non-zero condition) so that the remaining servers on the chain will be skipped.

Use The Z Flag.

VERTB (vertical blank) servers should always return with the Z (zero) flag set. The processor Z flag is used rather than the normal function convention of returning a result in D0 because it may be tested more quickly by Exec upon the server's return.

The easiest way to set the condition code register is to do an immediate move to the D0 register as follows:

```
SetZflag_Calls_Next:
    MOVEQ    #0,D0
    RTS
```

```
ClrZflag_Ends_Chain:
    MOVEQ    #1,D0
    RTS
```

The same Exec Interrupt structure used for handlers is also used for servers. Also, like interrupt handlers, servers must terminate their code with an RTS instruction.

Interrupt servers are called in priority order. The priority of a server is specified in its `is_Node.ln_Pri` field. Higher-priority servers are called earlier than lower-priority servers. Adding and removing interrupt servers from a particular chain is accomplished with the Exec `AddIntServer()` and `RemIntServer()` functions. These functions require you to specify both the 4703 interrupt number and a properly initialized Interrupt structure.

Servers have different register values passed than handlers do. A server cannot count on the D0, D1, A0, or A6 registers containing any useful information. However, the highest priority system vertical blank server currently expects to receive a pointer to the custom chips A0. Therefore, if you install a vertical blank server at priority 10 or greater, you must place custom (\$DFF000) in A0 before exiting. Other than that, a server is free to use D0-D1 and A0-A1/A5-A6 as scratch.

Interrupt Server Register Usage

1.12 26 // Interrupt Servers / Interrupt Server Register Usage

D0 Scratch.

D1 Scratch.

A0 Scratch except in certain cases (see note above).

A1 Points to the data area specified by the `is_Data` field of the Interrupt structure. Because this pointer is always fetched

(regardless of whether you use it), it is to your advantage to make some use of it (scratch).

A5 Points to your interrupt code (scratch).

A6 Scratch.

In a server chain, the interrupt is cleared automatically by the system. Having a server clear its interrupt is not recommended and not necessary (clearing could cause the loss of an interrupt on PORTS or EXTER).

Here is an example of a program to install and remove a low-priority vertical blank interrupt server:

vertb.c

This is the assembler VertBServer installed by the C example:

vertbserver.asm

1.13 26 Exec Interrupts / Software Interrupts

Exec provides a means of generating software interrupts. Software interrupts execute at a priority higher than that of tasks but lower than that of hardware interrupts, so they are often used to defer hardware interrupt processing to a lower priority. Software interrupts use the same Interrupt data structure as hardware interrupts. As described above, this structure contains pointers to both interrupt code and data, and should be initialized as node type NT_INTERRUPT (not NT_SOFTINT which is an internal Exec flag).

A software interrupt is usually activated with the Cause() function. If this function is called from a task, the task will be interrupted and the software interrupt will occur. If it is called from a hardware interrupt, the software interrupt will not be processed until the system exits from its last hardware interrupt. If a software interrupt occurs from within another software interrupt, it is not processed until the current one is completed. However, individual software interrupts do not nest, and will not be caused if already running as a software interrupt.

Don't Trash A6!

Software interrupts execute in an environment almost identical to that of hardware interrupts, and the same restrictions on allowable system function calls (as described earlier) apply to both. Note however that, unlike other interrupts, software interrupts must preserve A6.

Software interrupts are prioritized. Unlike interrupt servers, software interrupts have only five allowable priority levels: -32, -16, 0, +16, and +32. The priority should be put into the ln_Pri field prior to calling Cause().

Software interrupts can also be generated by message arrival at a PA_SOFTINT message port. The applications of this technique are limited

since it is not permissible, with most devices, to send IO requests from within interrupt code. However, the timer.device does allow such interactions, so a self-perpetuating PA_SOFTINT timer port can provide an application with quite consistent timing under varying multitasking loads. The following example demonstrates use of a software interrupt and a PA_SOFTINT port. See the "Exec Messages and Ports" chapter for more information about messages and ports.

```
timersoftint.c
```

1.14 26 Exec Interrupts / Disabling Interrupts

As mentioned in the "Exec Tasks" chapter, it is sometimes necessary to disable interrupts when examining or modifying certain shared system data structures. However, for proper system operation, interrupts should never be disabled unless absolutely necessary, and never for more than 250 microseconds. Interrupt disabling is controlled with the Disable() and Enable() functions. Although assembler DISABLE and ENABLE macros are provided, we strongly suggest that you use the system functions rather than the macros for upwards compatibility and smaller code size.

In some system code, there are nested disabled sections. Such code requires that interrupts be disabled with the first Disable() and not re-enabled until the last Enable(). The system Enable() and Disable() functions are designed to permit this sort of nesting.

Disable() increments a counter to track how many levels of disable have been issued. Only 126 levels of nesting are permitted. Enable() decrements the counter, and reenables interrupts when the last disable level has been exited.

1.15 26 Exec Interrupts / Function Reference

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

Table 26-2: Exec Interrupt Functions

Interrupt Function	Description
=====	=====
AddIntServer()	Add an interrupt server to a system server chain.
Cause()	Cause a software interrupt.
Disable()	Disable interrupt processing.
Enable()	Restart system interrupt processing.
RemIntServer()	Remove an interrupt server from a system server chain.
SetIntVector()	Set a new handler for a system interrupt vector.