

Libraries

COLLABORATORS

	<i>TITLE :</i> Libraries		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		March 28, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries	1
1.1	Amiga® RKM Libraries: 22 Exec Signals	1
1.2	22 Exec Signals / The Signal System	1
1.3	22 / The Signal System / Signal Allocation	2
1.4	22 / The Signal System / Waiting for a Signal	3
1.5	22 / The Signal System / Generating a Signal	4
1.6	22 Exec Signals / Function Reference	5

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 22 Exec Signals

Tasks often need to coordinate with other concurrent system activities (like other tasks and interrupts). This coordination is handled by Exec through the synchronized exchange of specific event indicators called signals.

This is the primary mechanism responsible for all intertask communication and synchronization on the Amiga. This signal mechanism operates at a low level and is designed for high performance. Signals are used extensively by the Exec message system as a way to indicate the arrival of an inter-task message. The message system is described in more detail in the "Exec Messages and Ports" chapter.

Not for Beginners.

This chapter concentrates on details about signals that most applications do not need to understand for general Amiga programming. For a general overview of signals, see the "Introduction to Exec" chapter of this manual.

The Signal System Function Reference

1.2 22 Exec Signals / The Signal System

The signal system is designed to support independent simultaneous events, so several signals can occur at the same time. Each task has 32 independent signals, 16 of which are pre-allocated for use by the operating system. The signals in use by a particular task are represented as bits in a 32-bit field in its Task structure (<exec/tasks.h>). Two other 32-bit fields in the Task structure indicate which signals the task is waiting for, and which signals have been received.

Signals are task relative. A task can only allocate its own signals, and may only wait on its own signals. In addition, a task may assign its own significance to a particular signal. Signals are not broadcast to all tasks; they are directed only to individual tasks. A signal has meaning to

the task that defined it and to those tasks that have been informed of its meaning.

For example, signal bit 12 may indicate a timeout event to one task, but to another task it may indicate a message arrival event. You can never wait on a signal that you did not directly or indirectly allocate yourself, and any other task that wishes to signal you must use a signal that you allocated.

Signal Allocation Waiting for a Signal Generating a Signal

1.3 22 / The Signal System / Signal Allocation

As mentioned above, a task assigns its own meaning to a particular signal. Because certain system libraries may occasionally require the use of a signal, there is a convention for signal allocation. It is unwise ever to make assumptions about which signals are actually in use.

Before a signal can be used, it must be allocated with the `AllocSignal()` function. When a signal is no longer needed, it should be freed for reuse with `FreeSignal()`.

```
BYTE AllocSignal( LONG signalNum );
VOID FreeSignal( LONG signalNum );
```

`AllocSignal()` marks a signal as being in use and prevents the accidental use of the same signal for more than one event. You may ask for either a specific signal number, or more commonly, you would pass `-1` to request the next available signal. The state of the newly allocated signal is cleared (ready for use). Generally it is best to let the system assign you the next free signal. Of the 32 available signals, the lower 16 are reserved for system use. This leaves the upper 16 signals free for application programs to allocate. Other subsystems that you may call depend on `AllocSignal()`.

The following C example asks for the next free signal to be allocated for its use:

```
if (-1 == (signal = AllocSignal(-1)))
    printf("no signal bits available\n");
else
{
    printf("allocated signal number %ld\n", signal);
    /* Other code could go here */
    FreeSignal(signal)
}
```

The value returned by `AllocSignal()` is a signal bit number. This value cannot be used directly in calls to signal-related functions without first being converted to a mask:

```
mask = 1L << signal;
```

It is important to realize that signal bit allocation is relevant only to the running task. You cannot allocate a signal from another task. Note

that functions which create a signal `MsgPort` will allocate a signal from the task that calls the function. Such functions include `OpenWindow()`, `CreatePort()`, and `CreateMsgPort()`. For this reason, only the creating task may `Wait()` (directly or indirectly) on the `MsgPort`'s signal. Functions which call `Wait()` include `DoIO()`, `WaitIO()` and `WaitPort()`.

1.4 22 / The Signal System / Waiting for a Signal

Signals are most often used to wake up a task upon the occurrence of some external event. Applications call the `Exec Wait()` function, directly or indirectly, in order to enter a wait state until some external event triggers a signal which awakens the task.

Though signals are usually not used to interrupt an executing task, they can be used this way. Task exceptions, described in the "Exec Tasks" chapter, allow signals to act as a task-local interrupt.

The `Wait()` function specifies the set of signals that will wake up the task and then puts the task to sleep (into the waiting state).

```
ULONG Wait( ULONG signalSet );
```

Any one signal or any combination of signals from this set are sufficient to awaken the task. `Wait()` returns a mask indicating which signals satisfied the `Wait()` call. Note that when signals are used in conjunction with a message port, a set signal bit does not necessarily mean that there is a message at the message port. See the "Exec Messages and Ports" chapter for details about proper handling of messages.

Because tasks (and interrupts) normally execute asynchronously, it is often possible to receive a particular signal before a task actually `Wait()`s for it. In such cases the `Wait()` will be immediately satisfied, and the task will not be put to sleep.

The `Wait()` function implicitly clears those signal bits that satisfied the wait condition. This effectively resets those signals for reuse. However, keep in mind that a task might get more signals while it is still processing the previous signal. If the same signal is received multiple times and the signal bit is not cleared between them, some signals will go unnoticed.

Be aware that using `Wait()` will break a `Forbid()` or `Disable()` state. `Wait()` cannot be used in supervisor mode or within interrupts.

A task may `Wait()` for a combination of signal bits and will wake up when any of the signals occur. `Wait()` returns a signal mask specifying which signal or signals were received. Usually the program must check the returned mask for each signal it was waiting on and take the appropriate action for each that occurred. The order in which these bits are checked is often important.

Here is a hypothetical example of a process that is using the console and timer devices, and is waiting for a message from either device and a possible break character issued by the user:

```

consoleSignal = 1L << ConsolePort->mp_SigBit;
timerSignal   = 1L << TimerPort->mp_SigBit;
userSignal    = SIGBREAKF_CTRL_C;          /* Defined in <dos/dos.h> */

signals = Wait(consoleSignal | timerSignal | userSignal);

if (signals & consoleSignal)
    printf("new character\n");

if (signals & timeOutSignal)
    printf("timeout\n");

if (signals & userSignal)
    printf("User Ctrl-C Abort\n");

```

This code will put the task to sleep waiting for a new character, or the expiration of a time period, or a Ctrl-C break character issued by the user. Notice that this code checks for an incoming character signal before checking for a timeout. Although a program can check for the occurrence of a particular event by checking whether its signal has occurred, this may lead to busy wait polling. Such polling is wasteful of the processor and is usually harmful to the proper function of the Amiga system. However, if a program needs to do constant processing and also check signals (a compiler for example) `SetSignal(0,0)` can be used to get a copy of your task's current signals.

```

ULONG SetSignal( ULONG newSignals, ULONG signalSet );

```

`SetSignal()` can also be used to set or clear the state of the signals. Implementing this can be dangerous and should generally not be done. The following fragment illustrates a possible use of `SetSignal()`.

```

signals = SetSignal(0,0);          /* Get current state of signals */

if (signals & SIGBREAKF_CTRL_C)    /* Check for Ctrl-C.          */
{
    printf("Break\n");             /* Ctrl-C signal has been set. */
    SetSignal(0, SIGBREAKF_CTRL_C) /* Clear Ctrl-C signal.        */
}

```

1.5 22 / The Signal System / Generating a Signal

Signals may be generated from both tasks and system interrupts with the `Signal()` function.

```

VOID Signal( struct Task *task, ULONG signalSet );

```

For example `Signal(tc,mask)` would signal the task with the specified mask signals. More than one signal can be specified in the mask. The following example code illustrates `Wait()` and `Signal()`.

```

signals.c

```

1.6 22 Exec Signals / Function Reference

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

Table 22-1: Exec Signal Functions

Exec Signal Function	Description
AllocSignal()	Allocate a signal bit.
FreeSignal()	Free a signal bit allocated with AllocSignal().
SetSignal()	Query or set the state of the signals for the current task.
Signal()	Signal a task by setting signal bits in its Task structure.
Wait()	Wait for one or more signals from other tasks or interrupts.