

Libraries_Manual

COLLABORATORS

	<i>TITLE :</i> Libraries_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries_Manual	1
1.1	Amiga® RKM Libraries: 17 Introduction to Exec	1
1.2	17 Introduction to Exec / Multitasking	1
1.3	17 Introduction to Exec / Dynamic Memory Allocation	3
1.4	17 Introduction to Exec / Signals	4
1.5	17 / Signals / Looking for Break Keys	5
1.6	17 / Signals / Processing Signals Without Wait()ing	5
1.7	17 Introduction to Exec / Interprocess Communications	6
1.8	17 // Waiting on Message Ports and Signals at the Same Time	8
1.9	17 Introduction to Exec / Libraries and Devices	8
1.10	17 / Libraries and Devices / Library Vector Offsets (LVOs)	10
1.11	17 / Libraries and Devices / Calling a Library Function	11

Chapter 1

Libraries_Manual

1.1 Amiga® RKM Libraries: 17 Introduction to Exec

The Multitasking Executive, better known as Exec, is the heart of the Amiga's operating system. All other systems in the Amiga rely on it to control multitasking, to manage the message-based interprocess communications system, and to arbitrate access to system resources. Because just about every software entity on the Amiga (including application programs) needs to use Exec in some way, every Amiga programmer has to have a basic understanding of its fundamentals.

Multitasking	Interprocess Communications
Dynamic Memory Allocation	Libraries and Devices
Signals	

1.2 17 Introduction to Exec / Multitasking

A conventional micro-computer spends a lot of its time waiting for things to happen. It has to wait for such things as the user to push buttons on the keyboard or mouse, for data to come in through the serial port, and for data to go out to a disk drive. To make efficient use of the CPU's time, an operating system can have the CPU carry out some other task while it is waiting for such events to occur.

A multitasking operating system reduces the amount of time it wastes, by switching to another program when the current one needs to wait for an event. A multitasking operating system can have several programs, or tasks, running at the same time. Each task runs independently of the others, without having to worry about what the other tasks are doing. From a task's point of view, it's as if each task has a computer all to itself.

The Amiga's multitasking works by switching which task is currently using the CPU. A task can be a user's application program, or it can be a task that controls system resources (like the disk drives or the keyboard). Each task has a priority assigned to it. Exec will let the task with the highest priority use the CPU, but only if the task is ready to run. A task can be in one of three states: ready, sleeping, or running.

A ready task is not currently using the CPU but is waiting to use the processor. Exec keeps a list of the tasks that are ready. Exec sorts this list according to task priority, so Exec can easily find the ready task with the highest priority. When Exec switches the task that currently has control of the CPU, it switches to the task at the top of this list.

A sleeping task is not currently running and is waiting for some event to happen. When that event occurs, Exec will move the sleeping task into the list of ready tasks.

A running task is currently using the CPU. It will remain the current task until one of three things occur:

- * A higher priority task becomes ready, so the OS preempts the current task and switches to the higher priority task.
- * The currently running task needs to wait for an event, so it goes to sleep and Exec switches to the highest priority task in Exec's ready list.
- * The currently running task has had control of the CPU for at least a preset time period called a quantum and there is another task of equal priority ready to run. In this case, Exec will preempt the current task for the ready one with the same priority. This is known as time-slicing. When there is a group of tasks of equal priority on the top of the ready list, Exec will cycle through them, letting each one use the CPU for a quantum (a slice of time).

The terms "task" and "process" are often used interchangeably to represent the generic concept of task. On the Amiga, this terminology can be a little confusing because of the names of the data structures that are associated with Exec tasks. Each task has a structure associated with it called a Task structure (defined in <exec/tasks.h>). Most application tasks use a superset of the Task structure called a Process structure (defined in <dos/dosextens.h>). These terms are confusing to Amiga programmers because there is an important distinction between the Exec task with only a Task structure and an Exec task with a Process structure.

The Process structure builds on the Task structure and contains some extra fields which allow the DOS library to associate an AmigaDOS environment to the task. Some elements of a DOS environment include a current input and output stream and a current working directory. These elements are important to applications that need to do standard input and output using functions like printf().

Exec only pays attention to the Task structure portion of the Process structure, so, as far as Exec is concerned, there is no difference between a task with a Task structure and a task with a Process structure. Exec considers both of them to be tasks.

An application doesn't normally worry about which structure their task uses. Instead, the system that launches the application takes care of it. Both Workbench and the Shell (CLI) attach a Process structure to the application tasks that they launch.

1.3 17 Introduction to Exec / Dynamic Memory Allocation

The Amiga has a soft machine architecture, meaning that all tasks, including those that are part of its operating system, do not use fixed memory addresses. As a result, any program that needs to use a chunk of memory must allocate that memory from the operating system.

There are two functions on the Amiga for simple memory allocation: `AllocMem()` and `AllocVec()`. The two functions accept the same parameters, a `ULONG` containing the size of the memory block in bytes followed by 32-bit specifier for memory attributes. Both functions return the address of a longword aligned memory block if they were successful or `NULL` if something went wrong.

`AllocVec()` differs from `AllocMem()` in that it records the size of the memory block allocated so an application does not have to remember the size of a memory block it allocated. `AllocVec()` was introduced in Release 2, so it is not available to the 1.3 developer.

Normally the bitmask of memory attributes passed to these functions will contain any of the following attributes (these flags are defined in `<exec/memory.h>`):

`MEMF_ANY`

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

`MEMF_CHIP`

This indicates the application wants a block of Chip memory, meaning it wants memory addressable by the Amiga custom chips. Chip memory is required for any data that will be accessed by custom chip DMA. This includes floppy disk buffers, screen memory, images that will be blitted, sprite data, copper lists, and audio data. If your application requires a block of Chip RAM, it must use this flag to allocate the Chip RAM. Otherwise, the application will fail on machines with expanded memory.

`MEMF_FAST`

This indicates a memory block outside of the range that the Amiga's custom chips can access. The "FAST" in `MEMF_FAST` has to do with the custom chips and the CPU trying to access the same memory at the same time. Because the custom chips and the CPU both have access to Chip RAM, the CPU may have to wait to access Chip RAM while some custom chip is reading or writing Chip RAM. In the case of Fast RAM, the custom chips do not have access to it, so the CPU does not have to contend with the custom chips access to Fast RAM, making CPU accesses to Fast RAM generally faster than CPU access to Chip RAM.

Since the flag specifies memory that the custom chips cannot access, this flag is mutually exclusive with the `MEMF_CHIP` flag. If you specify the `MEMF_FAST` flag, your allocation will fail on Amigas that have only Chip memory. Use `MEMF_ANY` if you would prefer Fast memory.

`MEMF_PUBLIC`

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

MEMF_CLEAR

This indicates that the memory should be initialized with zeros.

If an application does not specify any attributes when allocating memory, the system first looks for MEMF_FAST, then MEMF_CHIP. There are additional memory allocation flags for Release 2: MEM_LOCAL, MEMF_24BITDMA and MEMF_REVERSE. See the Exec Autodoc for AllocMem() in the Amiga ROM Kernel Reference Manual: Includes and Autodocs or the include file <exec/memory.h> for additional information on these flags. Use of these flags under earlier versions of the operating system will cause your allocation to fail.

Make Sure You Have Memory.

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

When an application is finished with a block of memory it allocated, it must return it to the operating system. There is a function to return memory for both the AllocMem() and the AllocVec() functions. FreeMem() releases memory allocated by AllocMem().

It takes two parameters, a pointer to a memory block and the size of the memory block. FreeVec() releases memory allocated by AllocVec(). It takes only one parameter, a pointer to a memory block allocated by AllocVec(). The following example shows how to allocate and deallocate memory.

```
APTR  my_mem;

if (my_mem = AllocMem(100, MEMF_ANY))
{
    /* Your code goes here */
    FreeMem(my_mem, 100);
}
else { /* couldn't get memory, exit with an error */ }
```

1.4 17 Introduction to Exec / Signals

The Amiga uses a mechanism called signals to tell a task that some event occurred. Each task has its own set of 32 signals, 16 of which are set aside for system use. When one task signals a second task, it asks the OS to set a specific bit in the 32-bit long word set aside for the second task's signals.

Signals are what makes it possible for a task to go to sleep. When a task goes to sleep, it asks the OS to wake it up when a specific signal bit gets set. That bit is tied to some event. When that event occurs, that signal bit gets set. This triggers the OS into waking up the sleeping

task.

To go to sleep, a task calls a system function called `Wait()`. This function takes one argument, a bitmask that tells Exec which of the task's signal bits to "listen to". The task will only wake up if it receives one of the signals whose corresponding bit is set in that bitmask. For example, if a task wanted to wait for signals 17 and 19, it would call `Wait()` like this:

```
mysignals = Wait(1L<<17 | 1L<<19);
```

`Wait()` puts the task to sleep and will not return until some other task sets at least one of these two signals. When the task wakes up, `mysignals` will contain the bitmask of the signal or signals that woke up the task. It is possible for several signals to occur simultaneously, so any combination of the signals that the task `Wait()`ed on can occur. It is up to the waking task to use the return value from `Wait()` to figure out which signal or signals occurred.

Looking for Break Keys Processing Signals Without `Wait()`ing

1.5 17 / Signals / Looking for Break Keys

One common usage of signals on the Amiga is for processing a user break. As was mentioned earlier, the OS reserves 16 of a task's 32 signals for system use. Four of those 16 signals are used to tell a task about the Control-C, D, E, and F break keys. An application can process these signals. Usually, only CLI-based programs receive these signals because the Amiga's console handler is about the only user input source that sets these signals when it sees the Control-C, D, E, and F key presses.

The signal masks for each of these key presses are defined in `<dos/dos.h>`:

```
SIGBREAKF_CTRL_C  
SIGBREAKF_CTRL_D  
SIGBREAKF_CTRL_E  
SIGBREAKF_CTRL_F
```

Note that these are bit masks and not bit numbers.

1.6 17 / Signals / Processing Signals Without `Wait()`ing

In some cases an application may need to process signals but cannot go to sleep to wait for them. For example, a compiler might want to check to see if the user hit Control-C, but it can't go to sleep to check for the break because that will stop the compiler. In this case, the task can periodically check its own signal bits for the Ctrl-C break signal using the Exec library function, `SetSignal()`:

```
oldsignals = ULONG SetSignal(ULONG newsignals, ULONG signalmask);
```

Although `SetSignal()` can change a task's signal bits, it can also monitor

them. The following fragment illustrates using `SetSignal()` to poll a task's signal bits for a Ctrl-C break:

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

/* check for Ctrl-C */
if (signals & SIGBREAKF_CTRL_C)
{
    /* The Ctrl-C signal has been set, take care of processing it... */

    /* ...then clear the Ctrl-C signal */
    SetSignal(0L, SIGBREAKF_CTRL_C);
}
```

If your task is waiting for signals, but is also waiting for other events that have no signal bit (such as input characters from standard input), you may need to use `SetSignal()`. In such cases, you must be careful not to poll in a tight loop (also known as busy-waiting). Busy-waiting hogs CPU time and degrades the performance of other tasks. One easy way around this is for a task to sleep briefly within its polling loop by using the `timer.device`, or the graphics function `WaitTOF()`, or (if the task is a Process) the DOS library `Delay()` or `WaitForChar()` functions.

For more information on signals, see the "Exec Signals" chapter of this manual.

1.7 17 Introduction to Exec / Interprocess Communications

Another feature of the Amiga OS is its system of message-based interprocess communication. Using this system, a task can send a message to a message port owned by another task. Tasks use this mechanism to do things like trigger events or share data with other tasks, including system tasks. Exec's message system is built on top of Exec's task signaling mechanism. Most Amiga applications programming (especially Intuition programming) relies heavily upon this message-based form of interprocess communication.

When one task sends a message to another task's message port, the OS adds the message to the port's message queue. The message stays in this queue until the task that owns the port is ready to check its port for messages. Typically, a task has put itself to sleep while it is waiting for an event, like a message to arrive at its message port. When the message arrives, the task wakes up to look in its message port. The messages in the message port's queue are arranged in first-in-first-out (FIFO) order so that, when a task receives several messages, it will see the messages in the order they arrived at the port.

A task can use a message to share any kind of data with another task. This is possible because a task does not actually transmit an entire message, it only passes a pointer to a message. When a task creates a message (which can have many Kilobytes of data attached to it) and sends it to another task, the actual message does not move.

Essentially, when task A sends a message to task B, task A lends task B a

chunk of its memory, the memory occupied by the message. After task A sends the message, it has relinquished that memory to task B, so it cannot touch the memory occupied by the message. Task B has control of that memory until task B returns the message to task A with the ReplyMsg() function.

Let's look at an example. Many applications use Intuition windows as a source for user input. Without getting into too much detail about Intuition, when an application opens a window, it can set up the window so Intuition will send messages about certain user input events. Intuition sends these messages to a message port created by Intuition for use with this window. When an application successfully opens a window, it receives a pointer to a Window structure, which contains a pointer to this message port (Window.UserPort). For this example, we'll assume the window has been set up so Intuition will send a message only if the user clicks the window's close gadget.

When Intuition opens the window in this example, it creates a message port for the task that opened the Window. Because the most common message passing system uses signals, creating this message port involves using one of the example task's 32 signals. The OS uses this signal to signal the task when it receives a message at this message port. This allows the task to sleep while waiting for a "Close Window" event to arrive. Since this simple example is only waiting for activity at one message port, it can use the WaitPort() function. WaitPort() accepts a pointer to a message port and puts a task to sleep until a message arrives at that port.

This simple example needs two variables, one to hold the address of the window and the other to hold the address of a message.

```
struct Message *mymsg; /*defined in <exec/ports.h> */
struct Window *mywin; /* defined in <intuition/intuition.h> */
...

/* at some point, this application would have successfully opened a */
/* window and stored a pointer to it in mywin. */
...

/* Here the application goes to sleep until the user clicks */
/* the window's close gadget. This window was set up so */
/* that the only time Intuition will send a message to this */
/* window's port is if the user clicks the window's close */
/* gadget. */

WaitPort(mywin->UserPort);
while (mymsg = GetMsg(mywin->UserPort))
    /* process message now or copy information from message */
    ReplyMsg(mymsg);
...

/* Close window, clean up */
```

The Exec function GetMsg() is used to extract messages from the message port. Since the memory for these messages belongs to Intuition, the example relinquishes each message as it finishes with them by calling ReplyMsg(). Notice that the example keeps on trying to get messages from the message port until mymsg is NULL. This is to make sure there are no

messages left in the message port's message queue. It is possible for several messages to pile up in the message queue while the task is asleep, so the example has to make sure it replies to all of them. Note that the window should never be closed within this GetMsg() loop because the while statement is still accessing the window's UserPort.

Note that each task with a Process structure (sometimes referred to as a process) has a special process message port, Process.pr_MsgPort. This message port is only for use by Workbench and the DOS library itself. No application should use this port for its own use!

Waiting on Message Ports and Signals at the Same Time

1.8 17 // Waiting on Message Ports and Signals at the Same Time

Most applications need to wait for a variety of messages and signals from a variety of sources. For example, an application might be waiting for Window events and also timer.device messages. In this case, an application must Wait() on the combined signal bits of all signals it is interested in, including the signals for the message ports where any messages might arrive.

The MsgPort structure, which is defined in <exec/ports.h>, is what Exec uses to keep track of a message port. The UserPort field from the example above points to one of these structures. In this structure is a field called mp_SigBit, which contains the number (not the actual bit mask) of the message port's signal bit. To Wait() on the signal of a message port, Wait() on a bit mask created by shifting 1L to the left mp_SigBit times ($1L \ll \text{msgport} \rightarrow \text{mp_SigBit}$). The resulting bit mask can be OR'd with the bit masks for any other signals you wish to simultaneously wait on.

1.9 17 Introduction to Exec / Libraries and Devices

One of the design goals for the Amiga OS was to make the system dynamic enough so that the OS could be extended and updated without effecting existing applications. Another design goal was to make it easy for different applications to be able to share common pieces of code. The Amiga accomplished these goals through a system of libraries. An Amiga library consists of a collection of related functions which can be anywhere in system memory (RAM or ROM).

Devices are very similar to libraries, except they usually control some sort of I/O device and contain some extra standard functions. Although this section does not really discuss devices directly, the material here applies to them. For more information on devices, see the "Exec Device I/O" section of this manual or the Amiga ROM Kernel Reference Manual: Devices.

An application accesses a library's functions through the library's jump, or vector, table. Before a task can use the functions of a particular library, it must first acquire the library's base pointer by calling the Exec OpenLibrary() function:

```
struct Library *OpenLibrary( UBYTE *libName,  
                             unsigned long mylibversion );
```

where libName is a string naming the library and mylibversion is a version number for the library. The version number reflects a revision of the system software. The chart below lists the specific Amiga OS release versions that system libraries versions correspond to:

- 30 Kickstart 1.0 - This revision is obsolete.
- 31 Kickstart 1.1 - This was an NTSC only release and is obsolete.
- 32 Kickstart 1.1 - This was a PAL only release and is obsolete.
- 33 Kickstart 1.2 - This is the oldest revision of the OS still in use.

- 34 Kickstart 1.3 - This is almost the same as release 1.2 except it has an Autobooting expansion.library

- 35 This is a special RAM-loaded version of the 1.3 revision, except that it knows about the A2024 display modes. No application should need this library unless they need to open an A2024 display mode under 1.3.

- 36 Kickstart 2.0 - This is the original Release 2 revision that was initially shipped on early Amiga 3000 models.

- 37 Kickstart 2.04 - This is the general Release 2 revision for all Amiga models.

The OpenLibrary() function looks for a library with a name that matches libName and also with a version at least as high as mylibversion. For example, to open version 33 or greater of the Intuition library:

```
IntuitionBase = OpenLibrary("intuition.library", 33L);
```

In this example, if version 33 or greater of the Intuition library is not available, OpenLibrary() returns NULL. A version of zero in OpenLibrary() tells the OS to open any version of the library. Unless your code requires Release 2 features, it should specify a version number of 33 to remain backwards compatible with Kickstart 1.2.

When OpenLibrary() looks for a library, it first looks in memory. If the library is not in memory, OpenLibrary() will look for the library on disk. If libName is a library name with an absolute path (for example, "myapp:mylibs/mylib.library"), OpenLibrary() will follow that absolute path looking for the library. If libName is only a library name ("diskfont.library"), OpenLibrary() will look for the library in the directory that the LIBS: logical assign currently references.

If OpenLibrary() finds the library on disk, it takes care of loading it and initializing it. As part of the initialization process, OpenLibrary() dynamically creates a jump, or vector, table. There is a vector for each function in the library. Each entry in the table consists of a 680x0 jump instruction (JMP) to one of the library functions. The OS needs to create the vector table dynamically because the library functions can be anywhere

in memory.

After the library is loaded into memory and initialized, `OpenLibrary()` can actually "open" the library. It does this by calling the library's Open function vector. Every library has a standard vector set aside for an OPEN function so the library can set up any data or processes that it needs. Normally, a library's OPEN function increments its open count to keep track of how many tasks currently have the library opened.

If any step of `OpenLibrary()` fails, it returns a NULL value. If `OpenLibrary()` is successful, it returns the address of the library base. The library base is the address of this library's Library structure (defined in `<exec/libraries.h>`). The Library structure immediately follows the vector table in memory.

After an application is finished with a library, it must close it by calling `CloseLibrary()`:

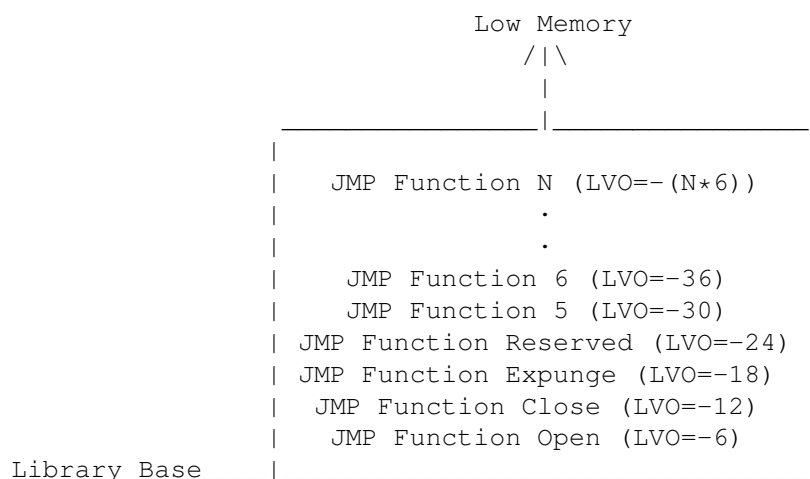
```
void CloseLibrary(struct Library *libPtr);
```

where `libPtr` is a pointer to the library base returned when the library was opened with `OpenLibrary()`.

Library Vector Offsets (LVOs) Calling a Library Function

1.10 17 / Libraries and Devices / Library Vector Offsets (LVOs)

After an application has successfully opened a library, it can start using the library's functions. To access a library function, an application needs the library base address returned by `OpenLibrary()` and the function's Library Vector Offset (LVO). A function's LVO is the offset from the library's base address to the function's vector in the vector table, which means an LVO is a negative number (the vectors precedes the library base in memory). Application code enters a library function by doing a jump to subroutine (the 680x0 instruction JSR) to the proper negative offset (LVO) from the address of the library base. The library vector itself is a jump instruction (the 680x0 instruction JMP) to the actual library function which is somewhere in memory.



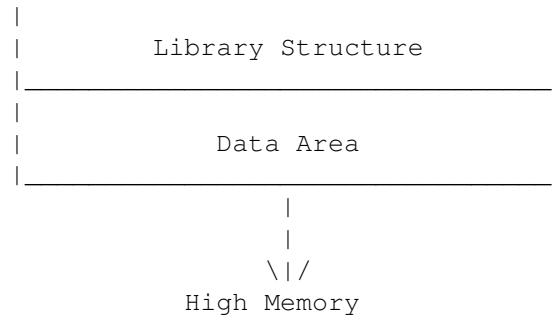


Figure 17-1: An Exec Library Base in RAM

The only legal way for an application to access a library function is through the vector table. A function's LVO is always the same on every system and is not subject to change. A function's jump vector can, and does, change. Assuming that a function's jump vector is static is very bad, so don't do it.

Each library has four vectors set aside for library housekeeping: OPEN, CLOSE, EXPUNGE, and RESERVED. The OPEN vector points to a function that performs any custom initialization that this library needs, for example, opening other libraries that this library uses. The CLOSE function takes care of any clean up necessary when an application closes a library. The EXPUNGE vector points to a function that prepares the library for removal from the system. Normally the OS will not remove a library from memory until the system needs the memory the library occupies. The other vector, RESERVED, does not do anything at present and is reserved for future system use.

1.11 17 / Libraries and Devices / Calling a Library Function

To call a function in an Amiga system library, an assembler application must put the library's base address in register A6 and JSR to the function's LVO relative to A6. For example to call the Intuition function DisplayBeep():

```

;*****
;      xref      _LVODisplayBeep
;...
;      move.l    A6, -(sp)          ;save the current contents of A6
;---- intuition.library must already be opened
;---- and IntuitionBase must contain its base address
;      move.l    IntuitionBase, A6 ;put intuition base pointer in A6
;      jsr       _LVODisplayBeep(A6) ;call DisplayBeep()
;      move.l    (SP)+, A6          ;restore A6 to its original value

```

IntuitionBase contains a pointer to the Intuition library's library base and _LVODisplayBeep is the LVO for the DisplayBeep() function. The external reference (xref) to _LVODisplayBeep is resolved from the linker library, amiga.lib. This linker library contains the LVO's for all of the

A0 D0 D1

The stub for `MoveWindow()` in `amiga.lib` has to copy window to register A0, `deltaX` to register D0, and `deltaY` to register D1.

The stub also copies Intuition library base into A6 and does an address-relative JSR to `MoveWindow()`'s LVO (relative to A6). The stub gets the library base from a global variable in your code called `IntuitionBase`. If you are using the stubs in `amiga.lib` to call Intuition library functions, you must declare a global variable called `IntuitionBase`. It must be called `IntuitionBase` because `amiga.lib` is specifically looking for the label `IntuitionBase`.

```
/* This global declaration is here so amiga.lib can find
   the intuition.library base pointer.
*/
struct Library *IntuitionBase;
    ...

void main(void)
{
    ...

    /* initialize IntuitionBase */
    if (IntuitionBase = OpenLibrary("intuition.library", 33L))
    {
        ...

        /* When this code gets linked with amiga.lib, the
           linker extracts the DisplayBeep() stub routine from
           from amiga.lib and copies it into the executable.
           The stub copies whatever is in the variable
           IntuitionBase into A6, and JSRs to
           _LVODisplayBeep(A6).
        */
        DisplayBeep();
        ...

        CloseLibrary(IntuitionBase);
    }
    ...
}
```

There is a specific label in `amiga.lib` for the library base of every library in the Amiga operating system. The chart below lists the names of the library base pointer `amiga.lib` associates with each Amiga OS library. The labels for library bases are also in the "Function Offsets Reference" list in the Amiga ROM Kernel Reference Manual: Includes and Autodocs.

Table 17-1: `Amiga.lib` Library Base Labels

Library Name	Library Base Pointer Name
asl.library	AslBase
commodities.library	CxBase

diskfont.library	DiskfontBase	
* dos.library	DOSBase	
* exec.library	SysBase	
expansion.library	ExpansionBase	
gadtools.library	GadToolsBase	
graphics.library	GfxBase	
icon.library	IconBase	
iffparse.library	IFFParseBase	
intuition.library	IntuitionBase	
keymap.library	KeyMapBase	
layers.library	LayersBase	
mathffp.library	MathBase	
mathieeedoubbas.library	MathIeeeDoubBasBase	
mathieeedoubtrans.library	MathIeeeDoubTransBase	
mathieeesingbas.library	MathIeeeSingBasBase	
mathieeesingtrans.library	MathIeeeSingTransBase	
mathtrans.library	MathTransBase	
rexxsys.library	RexxSysBase	
rexxsupport.library	RexxSupBase	
translator.library	TranslatorBase	
utility.library	UtilityBase	
version .library	(system private)	
workbench.library	WorkbenchBase	
Library Name	Library Base Pointer Name	

* Automatically opened by the standard C startup module		

The chart mentions that SysBase and DOSBase are already set up by the standard C startup module. For more information on the startup module,

You May Not Need amiga.lib.

Many C compilers provide ways of using pragmas or registerized parameters, so that a C program does not have to link with an amiga.lib stub to access a library function. See your compiler documentation for more details.