

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: 18 Exec Libraries	1
1.2	18 Exec Libraries / What is a Library?	1
1.3	18 / What is a Library? / Using a Library to Reference Data	2
1.4	18 / What is a Library? / Relationship of Libraries to Devices	2
1.5	18 / What is a Library? / Minimum Subset of Library Vectors	2
1.6	18 / What is a Library? / Changing the Contents of a Library	3
1.7	18 Exec Libraries / Adding a Library	4
1.8	18 / Adding a Library / Resident (Romtag) Structure	5

Chapter 1

Libraries_Manual

1.1 Amiga® RKM Libraries: 18 Exec Libraries

Exec maintains lists of libraries and devices. An Amiga library consists of a collection of related functions which can be anywhere in system memory (RAM or ROM). An Amiga device is very similar to an Amiga library, except that a device normally controls some sort of I/O hardware, and generally contains a limited set of standard functions which receive commands for controlling I/O. For more information on how to use devices for I/O, see the "Exec Device I/O" chapter of this book.

Not for Beginners.

This chapter concentrates on the internal workings of Exec libraries (and devices). Most application programmers will not to know the internal workings of libraries to program the Amiga. For an introduction to libraries and how to use them, see chapter one, "Introduction to Amiga System Libraries".

What is a Library? Adding a Library

1.2 18 Exec Libraries / What is a Library?

A library consists of a group of functions somewhere in memory (ROM or RAM), a vector table, and a Library structure which can be followed by an optional private data area for the library. The library's base pointer (as returned by `OpenLibrary()`) points to the library's Library data structure:

```
struct Library
{
    struct Node lib_Node;
    UBYTE lib_Flags;
    UBYTE lib_pad;
    UWORD lib_NegSize;           /* number of bytes before library */
    UWORD lib_PosSize;          /* number of bytes after library */
    UWORD lib_Version;
```

```

    UWORD    lib_Revision;
    APTR     lib_IdString;
    ULONG    lib_Sum;                /* the checksum itself */
    UWORD    lib_OpenCnt;           /* number of current opens */
};
/* Meaning of the flag bits: */
/* A task is currently running a checksum */
#define LIBF_SUMMING (1 << 0) /* on this library (system maintains this */
/* flag) */
#define LIBF_CHANGED (1 << 1) /* One or more entries have been changed */
/* in the library code vectors used by */
/* SumLibrary (system maintains this flag) */
#define LIBF_SUMUSED (1 << 2) /* A checksum fault should cause a system */
/* panic (library flag) */
#define LIBF_DELEXP (1 << 3) /* A user has requested expunge but */
/* another user still has the library */
/* open (this is maintained by library) */

```

Using a Library to Reference Data	Relationship of Libraries to Devices
Minimum Subset of Library Vectors	Changing the Contents of a Library

1.3 18 / What is a Library? / Using a Library to Reference Data

Most libraries (such as Intuition, graphics and Exec) have other data that follows the Library data structure in memory. Although it is not normally necessary, a program can use the library base pointer to access the Library structure and any custom library data.

In general, the system's library base data is read-only, and should be directly accessed as little as possible, primarily because the format of the data may change in future revisions of the library. If the library provides functions to allow access to library data, use those instead.

1.4 18 / What is a Library? / Relationship of Libraries to Devices

A device is a software specification for hardware control based on the Library structure. The structures of libraries and devices are so similar that the routine MakeLibrary() is used to construct both.

Devices require the same initial four code vectors as a library, but must have two additional code vectors for beginning and terminating special device I/O commands. The I/O commands that devices are expected to perform, at minimum, are shown in the "Exec Device I/O" chapter. An example device is listed in the Amiga ROM Kernel Reference Manual: Devices.

1.5 18 / What is a Library? / Minimum Subset of Library Vectors

The first four code vectors of a library must be the following entries:

OPEN

is the entry point called by the function `OpenLibrary()`. In most libraries, OPEN increments the library variable `lib_OpenCnt`. This variable is also used by CLOSE and EXPUNGE.

CLOSE

is the entry point called by the function `CloseLibrary()`. It decrements the library variable `lib_OpenCnt` and may do a delayed EXPUNGE.

EXPUNGE

prepares the library for removal from the system. This often includes deallocating memory resources that were reserved during initialization. EXPUNGE not only frees the memory allocated for data structures, but also the areas reserved for the library node itself.

RESERVED

is a fourth function vector reserved for future use. It must always return zero.

1.6 18 / What is a Library? / Changing the Contents of a Library

The way in which an Amiga library is organized allows a programmer to change where the system looks for a library routine. Exec provides a function to do this: `SetFunction()`. The `SetFunction()` routine redirects a library function call to an application-supplied function. (Although it's not addressed here, `SetFunction()` can also be used on Exec devices.) For instance, the AmigaDOS command `SetPatch` uses `SetFunction()` to replace some OS routines with improved ones, primarily to fix bugs in ROM libraries.

The format of the `SetFunction()` routine is as follows:

```
SetFunction( struct Library *lib, LONG funcOffset, APTR funcEntry)
               A1                A0                D0
```

The `lib` argument is a pointer to the library containing the function entry to be changed. The `funcOffset` is the Library Vector Offset (negative) of the function and `funcEntry` is the address of the new function you want to replace it with. The `SetFunction()` routine replaces the entry in the library's vector table at the given Library Vector Offset with a new address that points to the new routine and returns the old vector address. The old address can be used in the new routine to call the original library function.

Normally, programs should not attempt to "improve" library functions. Because most programmers do not know exactly what system library functions do internally, OS patches can do more harm than good. However, a legitimate use for `SetFunction()` is in a debugger utility. Using `SetFunction()`, a debugger could reroute system calls to a debugging routine. The debugging routine can inspect the arguments to a library function call before calling the original library function (if everything is OK). Such a debugger doesn't do any OS patching, it merely inspects.

SetFunction() is for Advanced Users Only.

It is very difficult to cleanly exit after performing SetFunction() because other tasks may be executing your code and also because additional SetFunction()'s may have occurred on the same function. Also note that certain libraries (for example the V33 version of DOS library) and some individual library function vectors are of non-standard format and cannot be replaced via SetFunction().

Although useful, performing SetFunction() on a library routines poses several problems. If a second task performs SetFunction() on the same library entry, SetFunction() returns the address of the new routine to the second task, not the original system vector. In that case, the first task can no longer exit cleanly since that would leave the second task with an invalid pointer to a function which it could be relying on.

You also need to know when it is safe to unload your replacement function. Removing it while another task is executing it will quickly lead to a crashed system. Also, the replacement function will have to be re-entrant, like all Exec library functions.

Don't Do This!

For those of you who might be thinking about writing down the ROM addresses returned by SetFunction() and using them in some other programs: Forget It. The address returned by SetFunction() is only good on the current system at the current time.

1.7 18 Exec Libraries / Adding a Library

Exec provides several ways to add your own libraries to the system library list. One rarely used way is to call LoadSeg() (a DOS library function) to load your library and then use the Exec MakeLibrary() and AddLibrary() functions to initialize your library and add it to the system.

MakeLibrary() allocates space for the code vectors and data area, initializes the library node, and initializes the data area according to your specifications, returning to you a library base pointer. The base pointer may then be passed to AddLibrary() to add your library to the system.

Another way to initialize and add a library or device to the system is through the use of a Resident structure or romtag (see <exec/resident.h>). A romtag allows you to place your library or device in a directory (default LIBS: for libraries, DEVS: for devices) and have the OS automatically load and initialize it when an application tries to open it with OpenLibrary() or OpenDevice().

Two additional initialization methods exist for a library or device which is bound to a particular Amiga expansion board. The library or device (containing a romtag) may be placed in the SYS:Expansion drawer, along with an icon containing the Manufacturer and Product number of the board it requires. If the startup-sequence BindDrivers command finds that board in the system, it will load and initialize the matching Expansion drawer

device or library. In addition, since 1.3, the Amiga system software supports ROM drivers on expansion boards. See the "Expansion Library" chapter for additional information on ROM drivers and Expansion drawer drivers. The sample device code in the Amiga ROM Kernel Reference Manual: Devices volume of this manual set may be conditionally assembled as an Expansion drawer driver.

Resident (Romtag) Structure

1.8 18 / Adding a Library / Resident (Romtag) Structure

A library or device with a romtag should start with MOVEQ #-1,D0 (to safely return an error if a user tries to execute the file), followed by a Resident structure:

```

STRUCTURE RT,0
    UWORD RT_MATCHWORD    * romtag identifier (==$4AFC)
    APTR  RT_MATCHTAG     * pointer to the above UWORD (RT_MATCHWORD)
    APTR  RT_ENDSKIP      * usually ptr to end of your code
    UBYTE RT_FLAGS        * usually RTF_AUTOINIT
    UBYTE RT_VERSION      * release version number (for example: 37)
    UBYTE RT_TYPE         * type of module (NT_LIBRARY)
    BYTE  RT_PRI          * initialization priority (for example: 0)
    APTR  RT_NAME         * pointer to node name ("my.library")
    APTR  RT_IDSTRING     * pointer to id string ("name ver.rev (date)")
    APTR  RT_INIT         * pointer to init code or AUTOINIT tables
    LABEL RT_SIZE         * size of a Resident structure (romtag)

```

If you wish to perform MakeLibrary() and AddLibrary() yourself, then your RT_FLAGS will not include RTF_AUTOINIT, and RT_INIT will be simply be a pointer to your own initialization code. To have Exec automatically load and initialize the library, set the RTF_AUTOINIT flag in the Resident structure's RT_FLAGS field, and point RT_INIT to a set four longwords containing the following:

dataSize

This is the size of your library data area, i.e., the combined size of the standard Library node structure plus your own library-specific data.

vectors

This is a pointer to a table of pointers to your library's functions, terminated with a -1. If the first word of the table is -1, then the table is interpreted as a table of words specifying the relative displacement of each function entry point from the start of the table. Otherwise it is treated as a table of longword address pointers to the functions. vectors must specify a valid table address.

structure

This parameter points to the base of an InitStruct() data region. That is, it points to the first location within a table that the InitStruct() routine can use to initialize your Library node structure, library-specific data, and other memory areas. InitStruct() will typically be used to initialize the data segment of

the library, perhaps forming data tables, task control blocks, I/O control blocks, etc. If this entry is a 0, then InitStruct() is not called.

initFunction

This points to a routine that is to be executed after the library (or device) node has been allocated and the code and data areas have been initialized. When the routine is called, the base address of the newly created library is passed in D0. If initFunction is zero, no initialization routine is called.

Complete source code for an RT_AUTOINIT library may be found in the appendix C of this book.