Default

AmigaGuide version Ponzio

COLLABORATORS

	TITLE : Default	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	AmigaGuide version Ponzio	November 28, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Default **1**

1.1 AmigaDOS Shared Libraries 1

1.2 AmigaDOS Shared Libraries - Index 1

1.3 AmigaDOS Shared Libraries - Shared Library Overview 3

1.4 AmigaDOS Shared Libraries - Calling Shared Library Functions 4

1.5 AmigaDOS Shared Libraries - Parts of an AmigaDOS Shared Library Image 10

1.6 AmigaDOS Shared Libraries - Librares in the System 13

1.7 AmigaDOS Shared Libraries - Programming 17

1.8 AmigaDOS Shared Libraries - Support library calls from ARexx 23

1.9 AmigaDOS Shared Libraries - Appendix 23

Chapter 1

Default

1.1 AmigaDOS Shared Libraries

(Version 1.0 "Mon Aug 1 16:33:46 1994")

Guide to AmigaDOS Shared Libraries
==================================

FidoNet 2:201/328, IRC: 'Bagder'.

An early version of this article was published by the american Amiga technical magazine called "AC's Tech", issue #2 1994. This version is updated, revised and reformatted for ASCII. Major updates to this document will be found in various places, but most like on AmiNet sites.

NOTE:
The examples of this article are written in assembler and C and require some knowledge of the languages to fully understand what they are all about. They are written only to illustrate the explanations and are only parts of larger source codes. They may not be accurate and I take no responsibility for the correctness or function of the examples.

1.2 AmigaDOS Shared Libraries - Index

Table of contents:

1 Shared Library Overview
1.1 Shared Library
1.2 Link Library
1.3 ROM Based/Disk Based Libraries
1.4 Memory Usage
1.5 Other Operating Systems
1.6 Advantages

2 Calling Shared Library Functions

2.1 Address Library Functions
2.2 Library Base
2.3 Index
2.4 Parameters
2.5 Access Libraries
2.5.1 OpenLibrary()
2.5.2 CloseLibrary()
2.5.3 RemLibrary()
2.6 Return Code
2.7 Glue Code
2.8 C and Register Parameters
2.8.1 SAS/Dice pragmas
2.8.2 Aztec/Maxon pragmas
2.8.3 How to create pragma files
2.9 Near Data Effects
2.10 Registers

3 Parts of an AmigaDOS Shared Library Image
3.1 Prevent Execution
3.2 ROMTag Structure
3.3 Init Table
3.4 Function Pointer Table
3.5 Data Table
3.6 Init Routine
3.7 Functions

4 Libraries in the System
4.1 Library Opening Details
4.2 Library List
4.3 Patching Libraries

5 Programming
5.1 Functions
5.1.1 Open()
5.1.2 Close()
5.1.3 Expunge()
5.1.4 Extfunc()
5.2 Function Descriptor File
5.3 Glue Code
5.4 Compiling
5.5 Linking
5.6 Debugging
5.7 Hints

6 Support library calls from ARexx
6.1 How ARexx access the library
6.2 ARexx calls a library function
6.3 ARexx function

Appendix
A. Version numbers and shared libraries
B. Further reading
C. Library source examples

1.3 AmigaDOS Shared Libraries - Shared Library Overview

```
1.                          Shared Library Overview
------------------------------------------------------------------------------
```

To be able to learn how to make a shared library, it's important to have the knowledge about what it is all about. In this article I'll take you through all steps, from the most basic ones down to the ones dealing with low level library programming.

1.1 Shared Library

First, an answer to the question: what is a shared library? As the name says, it is a function library shared by several simultaneous tasks and processes. The shared library code is not present in the executable image on disk, but is a separate file. The shared code is not loaded together with the executable. It is loaded into memory only when a program requires it.

On Amiga, the naming convention says that a shared library should be in lowercase letters with a ".library" ending, and the directory to put public libraries in is "LIBS:".

1.2 Link Library

A link library is not to be mixed up with a shared library. A link library is a function library that is linked into the executable at compile time. A link library becomes a part of an executable image.

1.3 ROM Based/Disk Based Libraries

The AmigaDOS system consists of several shared libraries, whose names you recognize: dos.library, exec.library, graphics.library, only to mention a few. These libraries won't be found in the LIBS: directory, they reside in ROM. Whether in ROM or on disk, shared libraries work and are used the same way.

1.4 Memory Usage

As mentioned, shared libraries are loaded when a program requests, i.e. opens, it. When the program has finished using the library, it closes the library. The library remains in memory even though no process is using it, until the operating system requires the memory it occupies (or is forced to remove itself by a program, such as "avail FLUSH" on the shell prompt in AmigaDOS 2.0 or later).

1.5 Other Operating Systems

Shared libraries are not AmigaDOS specific. Such are also found under different UNIXes, OS/2, Microsoft Windows, OS-9, OS-9000 and others. They are though not always called "shared libraries", but their concepts are very similar.

1.6 Advantages

The reasons why so many systems are using shared libraries are among others: less disk space is used because the shared library code is not

included in the executable programs, less memory is used because the shared library code is only loaded once, load time may be reduced because the shared library code may already be in memory when a program wants it, and that programs using shared libraries are very easily updated.

1.4 AmigaDOS Shared Libraries - Calling Shared Library Functions

2. Calling Shared Library Functions

We've been looking at what a shared library is, a little about how it works and some of its advantages. Now it's time to see how a library is used and accessed.

2.1 Address Library Functions

To be able to handle library calls, we must know how to call shared library functions. I'll describe it with a small comparsion to standard non-shared functions.

The most significant difference is in the way the functions are addressed. A standard function within a program is more or less an address to which the program counter is set when we want to jump to it. A shared library function is on the other hand addressed by adding a number to the address of the library's base.

When using standard function calls, the compiler or assembler arrange so that e.g. the function "getname" is associated with the particular static address in memory where the "getname" function starts. If the same "getname" function would be a shared library function, the compiler wouldn't know the actual address of it, but dynamically add a certain number (index) to the library's base address to access it.

As you see, we must know the index of the function and the library's base address to be able to call a shared library function.

2.2 Library Base

To find out the library base of a shared library, you must call OpenLibrary() which will return the library base of the specified library in register D0. All library bases are found like that except exec.library's, which is found by reading the pointer stored at the absolute address 4.

2.3 Index

Whenever you want to call a function in a shared library you (or the compiler) have to know the index to add to the library's base address. That 'index' is sometimes reffered to as 'vector' or 'offset'.

E.g., to call OpenLibrary() you must know the index of the function and the library base itself (OpenLibrary() is an exec.library function and we know that exec.library's base address is found at address 4). A call to OpenLibrary()} could look like this in assembler:

```
     move.l  SysBase,a6        ; SysBase is the name of
                               ; exec.library's base pointer
     ; >>> Parameters left out in this example <<<
```

```
     jsr     -552(a6)          ; We'll jump straight into the jump
                               ; table at the certain index. The index
                               ; is -552 in this case
```

2.4 Parameters

Ok, we know how to call a library function and we know that we must call OpenLibrary() to get a library's base address. To inform e.g OpenLibrary() which library we want to open, we must send it some parameters. The documentation tells us that OpenLibrary() wants the library name in A1 and the lowest acceptable version in D0. Parameters to the library functions are always stored in registers. See the library reference documentation for closer information exactly which registers.

This example opens a library with the name at libName with version 33 or higher:

```
     Include "exec/funcdef.i"     ; _LVO macro constructs
     Include "exec/exec_lib.i"    ; exec function index
     VERSION equ 33

     move.l   SysBase,a6          ; exec library base
     lea.l    libName,a1          ; library name
     move.l   #VERSION,d0         ; lowest usable version
     jsr      _LVOOpenLibrary(a6) ; OpenLibrary()
```

2.5 Access Libraries

The operating system provides facilities for the creation, use and access of shared libraries. The functions that let the programmer construct and access libraries are of different levels to give different possibilities. Low level function where you can change every single parameter and more high level functions that do a lot without the programmers exact specification.

I'll describe the functions of the highest level that also are the most frequently used:

2.5.1 OpenLibrary()

Gains access to a named library of a given version number. The library will be search for in ROM, in LIBS: and at last in current directory. You can also specify a library with an absolute path.
Always open libraries with the lowest version which includes the functions you need. To open intuition.library for 2.0+ (version 36) only, try something like:

```
#include <proto/exec.h>
#define LIB_VERSION 36
struct ExecBase *SysBase;
struct IntuitionBase *IntuitionBase;
void main(void)
{
  /*
   * The SysBase should be in order to perform this.
   * (Using any C startup module will do this for you.)
   */
```

```
      OpenLibrary("intuition.library", LIB_VERSION);
    if(!IntuitionBase) {
      printf("Couldn't open intuition version %d+\n", LIB_VERSION);
      exit(10);
    }
    /*
     * The program using intuition.library V36+ follows here!
     */
}
```

AmigaDOS file names are not case sensitive, but Exec lists are. If the library name is specified in a different case than it exists on disk, unexpected results may occur.

The library base returned from OpenLibrary() is not sharable between tasks! The only library base allowed to share is Exec's. If your program starts more tasks or processes, they all have to open their own libraries. This is subject to a lot of discussions where people state that all libraries, *except* those that are especially documented as non-shareable, are shareable. Although, this is what the library bible says about it:

"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 shareable between tasks is Exec's library base." (from RKRM Libraries 3rd ed., p. 467)

2.5.2 CloseLibrary()

Concludes access to a library. Whenever your program has finished using the functions of a shared library, there should be a call to CloseLibrary() for every call to OpenLibrary(). Simply like this:

```
CloseLibrary((struct Library *)IntuitionBase);
```

2.5.3 RemLibrary()

Calls the Expunge() function of the specified library. If the library isn't open, it will delete itself from memory. This is not typically called by user code.

```
/* Attempts to flush the named library out of memory. */
#include <exec/types.h>
#include <exec/execbase.h>

void FlushLibrary(STRPTR name)
{
  struct Library *result;

  Forbid();
  if(result=(struct Library *)FindName(&SysBase->LibList,name))
     RemLibrary(result);
  Permit();
}
```

With these three functions in mind, we'll continue.

2.6 Return Code

The return code of a shared library function call is always received in a register. (Today, I don't think there is a single function not using D0 for that purpose.)

2.7 Glue Code

The parameter storage in registers is not that comfortable in all occasions and many compilers (in all kinds of programming languages) don't even have the ability to store parameters in (pre-decided) registers. Then, glue code is required. Glue code (also known as "stub functions" or simply "stubs") is simply a set of functions that you can call instead of the shared library functions. The stub function reads the parameters from the stack and stores them in registers and then calls the shared library function. That makes the use of the glue code functions identical to other functions. Glue code is compiled into a kind of object file, using the suffix ".lib", and is stored in LIB: (not to be mixed up with LIBS: where the shared libraries are stored). All stub functions for the standard AmigaDOS libraries are found in the "amiga.lib" file that comes with most compilers or can be bought straight from Commodore.

2.8 C and Register Parameters

C language compilers are in general using the stack to pass parameters between functions, but to be able to use shared libraries smoothly, several compilers offer ways to force parameters in registers and automatically use the right library base and function index.

The SAS, Dice, Aztec C and Maxon compilers, all provide such solutions by special pragma instructions. A pragma instruction is a line starting with "#pragma", which is a compiler instruction keyword, followed by the compiler specific text. Such a pragma defines the function, which library base it needs and in which registers the parameters must be stored. By using such pragmas you don't have to call or link any glue code within your program.

The GNU C compiler, which is a freely distributable C and C++ compiler, has a very complicated way to solve this problem. It declares and uses inlined functions that use the GNU compiler's own __asm() instruction to set the proper registers to the right values.

When using this information, a compiled result uses SysBase, the index and the parameters in registers just as we did in the assembler examples above. C language usage:

```
#include <pragmas/exec_pragmas.h>
#define libName "foobar.library"
#define VERSION 33
OpenLibrary(libName, VERSION);
```

2.8.1 SAS/Dice pragmas

The library call pragmas available in the SAS compiler are built-up ike this (Dice supports most of this too):

```
#pragma <kind of call> <lib base> <name> <index> <registers>
```

which means:

```
 <kind of call>
     Which kind of library call this pragma should generate. There are
     three different ones:
     'libcall' makes a standard library call
     'tagcall' makes a standard library call where the last parameter
     points to a taglist
     'syscall' makes a call to exec.library

 <lib base>
     The library base name to use. Not specified for 'syscall' calls.
     Example: "DiskfontBase" (The name of diskfont.library's library
     base.)

 <name>
     Function name identifier. Example: "MyFunction".

 <index>
     Function index of the library. A hexadecimal, positive number
     (which is turned negative by the compiler when it generates the
     indexed library call). Example: "1A" (The first library function
     index of all normal libraries.)

 <registers>
     Register/parameter information in a special format, a sequence of
     hexadecimal numbers. Reading from the *right*, each digit has the
     following meaning:
     1. Number of parameters.
     2. Result code register (0-6 means register D0-D6 and 8-9, A-E
     means register A0-A6)
     3+. The parameter registers, read from the left (!). The numbers
     are associated with the same registers as in paragraph 2 above.

     Example:

     #pragma libcall SysBase OpenLibrary 228 0902

2.8.2 Aztec/Maxon pragmas
-------------------------
     The library call pragmas available in the Aztec and Maxon compilers
are built-up ike this:

     #pragma amicall(<lib base>,<index>,<name>(<parameters>))

     which means:

 <lib base>
     The library base name to use.

 <index>
     Function index of the library. A hexadecimal, positive number
     (which is turned negative by the compiler when it generates the
     indexed library call) with a "0x" prefix. Example: "0x1a" (The first
     library function index of all normal libraries.)

 <name>
     Function name identifier. Example: "MyFunction".
```

<parameters>
Register/parameter information. It should be written as 'register, register,register'. Example: "a1,d0"

Example:

```
#pragma amicall(SysBase,0x228,OpenLibrary(a1,d0))
```

2.8.3 How to create pragma files

Most compilers have the pragma files included, and then there's no problem. But if you want to produce them for yourself, for your own library or for new versions of other libraries, most compilers have a utility called 'fd2pragma'. That utility uses function descriptor files (more details about those follow) as sources and generates pragma files. There is also a freely distributable program that can generate pragmas for Aztec, SAS, Dice and Maxon.

2.9 Near Data Effects

Compilers of different programming languages often create machine language instructions that address data indexed by a 16-bit register, instead of straight 32-bit addressing, to increase execution speed and decrease the code size.

Some libraries might request or offer a "callback function", a function supplied by you in the form of a function pointer that might get called from inside the library. A call from within a library may not have that index register set properly and therefore you must set it before you can access any data that requires that register!

In SAS/C, this is simply done by defining the function like:

```
void __saveds callback( void );
```

if using DICE, __saveds must be replaced with __geta4.

(In the SAS and Aztec compilers, it can also be done by calling geta4() first in the callback function.)

From version 36 some of the AmigaDOS system library functions feature hook abilities, which is a kind of callback function. They are also called from inside a library and then of course also demand loading of the index register the same way.

2.10 Registers

Library functions should preserve the a2-a7 and d2-d7 registers. The rest must be stored in a safe place and then brought back after the library call if you want to be sure of their contents.

1.5 AmigaDOS Shared Libraries - Parts of an AmigaDOS Shared Library Image

```
3.                  Parts of an AmigaDOS Shared Library Image
------------------------------------------------------------------------------
```

If we were content with only using shared libraries, we would have enough information by now to use all kinds of library calls.

Only scratching the surface isn't enough if we want to create someting by ourselves. We must instead start digging into detailed information. How is a shared library constructed? Of which parts? How do you combine those parts to make your own shared library?

A shared library image is built up by a few different parts:
- Code preventing execution
- ROMTag structure with sub data
- Init table
- Function pointer table
- Data table
- Init routine
- Functions

3.1 Prevent Execution

The first thing the disk image contains is a piece of code that prevent users from trying to execute the library as an executable file. That piece of code should preferably return an error code to the calling environment (that most possibly is a shell).

Example:

```
moveq     #-1,d0
rts
```

3.2 ROMTag Structure

Coming up next is a ROMTag structure. ROMTags are used to link system resident modules together. The ROMTag looks like:

(found in <exec/resident.h>)

```
struct Resident {
  UWORD rt_MatchWord;
  struct Resident *rt_MatchTag;
  APTR  rt_EndSkip;
  UBYTE rt_Flags;
  UBYTE rt_Version;
  UBYTE rt_Type;
  BYTE  rt_Pri;
  char  *rt_Name;
  char  *rt_IdString;
  APTR  rt_Init;
};
```

`rt_MatchWord` - Used by exec to find this structure when it is about to link us into the ROMTag list. This must contain RTC_MATCHWORD (the hexadecimal number 4AFC, which is

```
                    a MC68000 "ILLEGAL" instruction).
  rt_MatchTag -     This must contain a pointer to this struct.
  rt_EndSkip -      Pointer to end of library init code.
  rt_Flags -        RTF_AUTOINIT informs exec that this structure's
                    'rt_Init' member points to an init table.
  rt_Version -      Library version number
  rt_Type -         Should contain NT_LIBRARY (found in <exec/nodes.h>),
                    which informs exec about the fact that this is a
                    shared library image.
  rt_Pri -          Initialization priority. 0 (zero) is perfectly ok.
  rt_Name -         Pointer to the zero terminated library name.
  rt_IdString -     Standard name/version/date ID string. Example:
                    "myown.library 1.0 (21.11.93)"
  rt_Init -         This data points to an init table if
                    RTF_AUTOINIT is set in structure member
                    rt_Flags.
```

As you can see, this structure requires some more information stored. You must have the library name and a standard ID string stored, and the last structure member should point to a "init table".

3.3 Init Table

The init table is a table of four longwords. I try to visualize them in a structure like this:

(A struct of this kind is not found in any standard include file, this is written by me.)

```
struct InitTable {
  ULONG it_LibBaseSize;
  APTR  it_FuncTable;
  ULONG *it_DataTable;
  APTR  it_InitRoutine;
};
```

```
  it_LibBaseSize -  Size of your library base structure. In common
                    situations it is no point in using anything else but
                    a straight struct Library as library base. It must not
                    be smaller than that!
  it_FuncTable -    This should contain a pointer to an array of
                    function pointers.
  it_DataTable -    Pointer to a data table in exec/InitStruct format for
                    initialization of the Library base structure.
  in_InitRoutine -  Pointer to a library initialization routine or NULL.
```

Once again we have a structure that needs more data. The function pointer table, the data table and the init routine is left.

3.4 Function Pointer Table

This should be a table of function pointers to the different functions in the library. It can be specified in two ways:

1) By setting the first word (16 bits) in the list to -1, you specify that the table is a list with 16-bit addresses relative to the start of the list. End the table with a -1 word.

2) By storing absolute 32-bit pointers to the functions and ending
with a -1 longword (32 bits).

My examples will use the second way.

The pointers should point to the functions of the library. All libraries should still have a few standard functions used by exec and must not be left out. The first four entries are dedicated to such functions.

The list must look like:

- Open() - Open library routine.
- Close() - Close library routine.
- Expunge() - Delete library from memory routine.
- Extfunc() - Reserved for future expansion.
- own1() - Our first function
- own2() - Our second function
- ... - The rest of our functions
- -1 - End of table

How to program such functions is discussed further on. Let's continue, we have the data table and the init routine left to look at.

3.5 Data Table

The data table is used to initialize the library base structure when it's linked into the system list of shared libraries. The table is in the so called "exec/InitStruct" format. A data table is controlling a number of different initializing methods. In our case we just use a number of offsets (relative to the library base) and their initialization values.

```
#include <exec/libraries.i>
#include <exec/initializers.i>
#include <exec/nodes.i>

INITBYTE     LN_TYPE,NT_LIBRARY
     ; Init type: Library.
INITLONG     LN_NAME,LibName
     ; Init name of the library
INITBYTE     LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
     ; Set the flags that tells exec we have changed
     : the library and that we allow checksumming.
INITWORD     LIB_VERSION,VERSION
     ; Init version
INITWORD     LIB_REVISION,REVISION
     ; Init revision
INITLONG     LIB_IDSTRING,IDString
     ; Init IDString
DC.L 0     ; End of InitStruct() command table
```

If you have a larger library base than a Library struct, you might want to add more initialize entries to this table. The only thing left now to complete our ROMTag structure is the init routine.

3.6 Init Routine

This routine gets called after the library has been allocated by exec. The library base pointer is in D0, the segment list is in A0 and SysBase in A6. This function must return the library base in D0 to be linked into the library list. If this initialization function fails, the library memory must be manually deallocated, then NULL returned in D0.

Deallocate library memory by using something like:

```
move.l    d0,a5
moveq     #0,d0
move.l    a5,a1
move.w    LIB_NEGSIZE(a5),d0

sub.l     d0,a1
add.w     LIB_POSSIZE(a5),d0

jsr       _LVOFreeMem(a6)
```

The segment list, that we receive in A0, should be stored somewhere for later access. We'll need it when the library is to be removed from memory. Note that this routine will be called only once for every time the library is being loaded into memory. That makes it perfectly ok to store the segment list simply like:

```
lea       anywhere(pc),a1
move.l    a0,(a1)
rts

anywhere: DC.L 0
```

A nice way to store this data is to extend the library base structure to hold the segment list pointer too.

This was the last of the initialization part. The ROMTag structure is complete. Left in the library are the functions that it should contain.

3.7 Functions

As mentioned before, there are four required functions that should be in all shared libraries. The rest of the functions are up to you to decide, design and make sure they receive proper data. How to code the functions and what to think of when doing so, is discussed in a chapter below.

1.6 AmigaDOS Shared Libraries - Librares in the System

4. Libraries in the System

We know what shared libraries are and we are familiar with all data stored in the library image. We know what functions to use when we want to access libraries and we know how to call library functions. What about low level information? What is done in the system when we call OpenLibrary()? How can I check if library already is loaded and which version number that library has? How can I patch a function of an already loaded library?

4.1 Library Opening Details

When a single OpenLibrary() is called, a lot of things happen:

1. Exec checks the already loaded libraries to see if the requested library is there. If it is, go to step 6.

2. If the library name is specified without path, it is searched for in ROM, LIBS: and then current directory, otherwise simply in the specified path. The first directory that holds a library with the name it searches for, will be the one it loads from. If the library wasn't found, return NULL. If the library was found anywhere else but in ROM, it's LoadSeg()'ed into memory. ROM libraries are already accessible.

3. Exec scans the library for the 4AFC word with a following 32-bit pointer back to it. That word is the beginning of a ROMTag structure!

4. InitResident() is called, which hopefully finds the RTF_AUTOINIT flag set in the rt_Init member of the ROMTag structure and therefore calls MakeLibrary() which performs:

   Memory is allocated to fit a jump table and the library base structure. The size of the library base structure is found in the first longword of the data table. The jump table is created by a call to MakeFunctions() and is placed just before the library base in memory. The size of the allocation can be read in the library base structure (lib_NegSize + lib_PosSize).

   The library base structure is initialized using the data table list and an InitStruct() call.

   The init routine is called with the library base pointer in D0, SysBase in A6 and the segment list pointer in A0. If NULL is returned, the entire OpenLibrary() fails and returns NULL.

   Notice that any kind of failure in InitResident() means that the library is never added to the system.

5. AddLibrary() adds the library to the system list, making it available to programs. The checksum of the library entries will be calculated.

6. The OpenLibrary() call's version number parameter is checked against the version number of the library base (lib_Version). If the requested number is higher than the library version, OpenLibrary() fails and returns NULL.

7. The open function of the library is called. If that fails NULL is returned, otherwise the library base is returned in D0.

If the same library exists in LIBS: with one version and in current directory with a later version, OpenLibrary() will always go for the one that it finds first. In this case that is the library in LIBS:. If that library has a too low version number, OpenLibrary() fails.

As you can see, OpenLibrary() is a rather high level function. By using the other mentioned functions you can add a library to the system without going the way I describe in this article. But that wouldn't make it a standard shared library.

4.2 Library List

Exec keeps track of all libraries that are opened. We can take part of exec's library list information by studying the linked list starting at SysBase->LibList. That pointer points to a 'struct List', whose 'struct Node' pointers point to the 'struct Library' of all libraries that are currently in memory. This sounds more difficult than it is. Take a look at this small example.

To find a certain library name in the library list, we can write:

```
struct Library *findlib(char *name)
{
  struct Library *lib;
  Forbid();
  lib = (struct Library *)FindName( SysBase->LibList, name );
  Permit();
  return( lib );
}
```

4.3 Patching Libraries

All libraries that are opened get a jump table created. That means that even ROM based libraries get a jump table in RAM. When using functions in any library, we always go through that jump table which consists of nothing but a number of JMP #ADDRESS instructions. As you understand, these jumps are supposed to jump into the library to perform whatever they are to perform. By changing an entry in that jump table, we can make a certain library call to call our own function instead of the original! But to change an entry is more than just storing in the list (since there are checksums and things that have to be correct). The correct way to do it, is to use SetFunction(), which can make one of those JMPs jump to our own code.

To replace OpenLibrary() with our own function, we can do it like:

```
#include <exec/types.h>
#include <exec/protos.h>

#ifdef SAS
/*
 * Things to set for the SAS/C compiler:
 */
#define ASM __asm
#define DREG(x) register __d ## x
#define AREG(x) register __a ## x

#else
/*
 * Defines for the Dice compiler:
 */
```

```
#define ASM /* not used */
#define DREG(x) __D ## x
#define AREG(x) __A ## x

#endif

int ASM OurOpenLibrary(AREG(1) char *, DREG(0) int);
void patch(void)
{
  APTR oldfunc;

  oldfunc = SetFunction((struct Library *)SysBase,
                        -552,
                        (APTR)OurOpenLibrary );

  /*
   * Now, all following calls to OpenLibrary() will
   * call our own function instead.
   */

  /*
   * To swap back, we simply use SetFunction()
   * again. We really should be careful before
   * doing so, because someone else might have
   * patched the function after us, and if we
   * simply restore our original we would ruin
   * that patch!
   */

  SetFunction( SysBase, -552, oldfunc );
}

int ASM OurOpenLibrary(AREG(1) char *libName,
                       DREG(0) int version)
{
/*
 * Code our own library opener. Do remember that
 * our index register is not initialized now, and
 * if you want it, make sure you can restore the
 * previous value before returning from this
 * function. We don't want to crash any programs,
 * do we?
 */

 /* Preserve used registers! */

}
```

Patching libraries are often used when creating debugging tools (such as the well-known 'Mungwall' which patches AllocMem and FreeMem, 'Snoopdos' which hangs on to most of dos.library's functions and others) and for programs that enhances or somehow changes the functionality of a function system wide (such as 'Explodewindows' which patches OpenWindow() and beautifies window openings, 'RTpatch' and 'reqchange' which patches different requester calls to bring up reqtools.library requesters instead).

NOTE: SetFunction() cannot be used on non-standard libraries like pre-V36
dos.library! If you want to patch such a library, you must manually Forbid(),
preserve all 6 original bytes of the jump table entry, SumLibrary() (to
evaluate the new checksum) and then Permit().

1.7 AmigaDOS Shared Libraries - Programming

5. Programming

5.1 Functions

Shared libraries must be programmed by someone. Until now, you've learned how to control, play around and change already existing libraries. Now, we'll check out more of what there is to know to be able to program a library. The ROMTag initializing is of course required when programming a library, but the biggest part and the part that really makes the library, is still the functions.

You're not restricted to anything when it comes to the function of the routines you want to put in a shared library. What must be thought of when creating functions for a shared library using a compiler, is that there is no main function and no startup modules, and therefore no one of the symbols declared in those modules will be declared if you don't do it yourself.

There are always four functions required that have to be in every library. They are Open(), Close(), Expunge() and Extfunc() and are called by exec when the library is to be opened, closed and removed from memory (the fourth is reserved for future use). Exec turns off task switching while executing these routines (via Forbid), so we should make them not take too long. (When using SAS/C these functions won't be necessary to code, see the "Compiling" and "Linking" chapters.)

5.1.1 Open() - (Library base:a6, version:d0)

This routine is called by exec when OpenLibrary() (or more correct InitResident()) is called. Open should return the library pointer in D0 if the open was successful. If the open fails, NULL should be returned. It might fail in cases where we allocate memory on each open, or if the library only can be open once at a time...
Example:

```
; Increase the library's open counter
addq.w     #1,LIB_OPENCNT(a6)

; Switch off delayed expunge
bclr       #LIBB_DELEXP,LIB_FLAGS(a6)

; Return library base
move.l     a6,d0
rts
```

5.1.2 Close() - (Library base:a6)

This routine is called by exec when CloseLibrary() is called. If the library is no longer open and there is a delayed expunge, then Expunge! Otherwise Close should return NULL.
Example:

```
; Decrease the library's open counter
subq.w     #1,LIB_OPENCNT(a6)

; If there is anyone still open, return
bne.s      retlabel

; Is there a delayed expunge waiting?
btst       #LIBB_DELEXP,LIB_FLAGS(a6)
beq.s      retlabel

; Do the expunge!
bsr        Expunge

retlabel:
; set the return value
moveq      #0,d0

rts
```

5.1.3 Expunge() - (Library base:a6)

This routine is called by exec when RemLibrary() is called, or from Close when there was a delayed expunge. If the library is no longer open then Expunge should Remove() itself from the library list, FreeMem() the InitResident()'s allocation and return the segment list (which was given to the Init routine). Otherwise Expunge should set the delayed expunge flag and return NULL.

Because Expunge might be called from the memory allocator, it may NEVER Wait() or otherwise take long time to complete.

Example:

```
; Is the library still open?
tst.w      LIB_OPENCNT(a6)
beq        notopen

; It is still open. set the delayed expunge flag
; and return zero
bset       #LIBB_DELEXP,LIB_FLAGS(a6)
moveq      #0,d0
rts

notopen: ; Get rid of us!

movem.l    d2/a5/a6,-(sp)      ; save some registers
move.l     a6,a5

; Store our segment list in d2
lea      anywhere(pc),a6
move.l     (a6),d2

move.l     4,a6      ; get SysBase
```

```
     ; Unlink from library list
     move.l    a5,a1
     jsr       _LVORemove(a6)     ; This removes our node from the list

     ; Free our memory
     moveq     #0,d0
     move.l    a5,a1
     move.w    LIB_NEGSIZE(a5),d0 ; jump table size

     sub.l     d0,a1
     add.w     LIB_POSSIZE(a5),d0 ; the size of the rest of the library.

     jsr       _LVOFreeMem(a6)

     ; Return the segment list
     move.l    d2,d0

     movem.l   (sp)+,d2/a5/a6     ; Get back the registers
     rts
```

5.1.4 Extfunc() - (we don't know about any registers!)

This routine is reserved for future use and should return 0 in register D0.

Example:

```
moveq #0,d0
rts
```

5.2 Function Descriptor File

To easily use the SAS/C options for creating a shared library or using the pragma construction utilities, a standard AmigaDOS function descriptor file is required. It describes the functions in a library like:

```
##base _OwnBase
##bias 30
##public
* ---- Here follows the public functions ----
OwnFunction(name,age)(A0/D2)
OwnFoobar(daynumber,dayname)(D1/A3)
##private
* ---- Private! Hands off
OwnPrivate(things)(a1)
##end
```

Where:

##base - The library base identifier

##bias - Index base position. The first function specified will use this index, which should be positive (turned negative later by the compiler) and in all normal cases starts on the first free jump table entry: 30.

```
     ##public -    The functions following are public functions that
                   everybody is allowed to use.

     ##private -   Private functions follow. Such functions should
                   not be messed with and we won't get any information
                   on those.

     * -           All lines starting with an asterisc '*' are treated
                   as comments.

     functionname(name1,name2)(register1/register2) -
          Describes the parameters and in which registers the function
          received the parameters in. The entire line should be written
          without whitespaces as in the example above. The registers
          should be written like: d0/a1/d2. The parameter names are only
          for documentation use.

     ##end -   end of function descriptor file
```

5.3 Glue Code

Glue code is written to be called with the parameters on the stack instead of the registers as it should. The glue functions should pick parameters from the stack and assign to the proper registers.

Example:

```
move.l     a1,-(sp)  ; Store register A1 on stack
move.l     a6,-(sp)  ; Store register A6 on stack
move.l     12(sp),a1 ; Get first argument from stack to a1
move.l     16(sp),d0 ; Get second argument from stack to d0
move.l     4,a6      ; Get SysBase in A6.
jsr     _LVOOpenLibrary(a6) ; Call OpenLibrary()
     ; Now d0 contains the result code from the
     ; library call
move.l     (sp)+,a6  ; Restore A6 from stack
move.l     (sp)+,a1  ; Restore A1 from stack
rts
```

5.4 Compiling

Things to think of when compiling library code:

* Always make the function called from another process (the outside) a "__saveds" function as the index register has to be properly initialized before continuing. __saveds should be replaced with __geta4 when using Dice and an initial 'geta4()' call when using Aztec C.

* Whether to use global symbols unique or shared by every task. SAS/C features easy changing between these two, but other compilers might have trouble creating unique global variables for each library open.

* Options when compiling a library may include some of the following. (These are the SAS/C options, but all compilers of

```
     today offer simlar functionalities.):

     LIBCODE        Forces all index addressing to use
                    the library base pointer (a6) instead
                    of the standard a4.

     NOSTANDARDIO   Do not use any of the C standard io
                    functions such as printf() or
                    fprintf(stderr, ...) since they rely on
                    global symbols declared and
                    initialized in the startup module.

     OPTIMIZE       Optimize the output code.
```

5.5 Linking

Linking a library often causes many problems, at least it has done so for me. You must remember that no compiler startup symbols will exist unless you declare them (or use a compiler that enables such things, like SAS/C v6.50 and later)! Things like stack expansions can't be made to work, and routines like fopen() and others are using startup module symbols (which can be declared by us though).

With the symbols in mind, we continue! All the talk about the library initializer structures is no problem of a SAS/C programmer's mind. By including the following flags in your 'slink' line, all such problems are solved:

* LIBPREFIX <prefix>
  Default is '_' (underscore). This is the prefix added to the functions specified in the function descriptor file to match the symbols of the object file(s).

* LIBFD <function desc file>
  Tells where the function descriptor file is.

* FROM lib:libent.o lib:libinit.o
  Two nice object files holding code that we would have to code by ourselves otherwise. If you are using global variables in your code, "libinit.o" will make all currently open sessions of the library access the same, shared, variable. By using "libinitr.o" all globals will be copied at the library open, thus each open library has its own global variables.

* LIBID
  Sets the IdString of the library

* LIBVERSION <number>
  Sets the version number of the library

* LIBREVISION <number>
  Sets the revision number of the library

5.6 Debugging

Using SAS/C, shared libraries can be run time debugged (including variable checking, break-pointing and so on) just like any other program

using the "step into reslib" option in ‘cpr’. Break any library function by writing "b myown.library:foobar" (where foobar is the name of the function we want cpr to stop in when we enter) on the command prompt of ‘cpr’. When creating debug code, remember to debug the library that exists in the same directory as the code does, or specify the compiler flag SOURCEIS= and the name of your source file.

5.7 Hints

I have been programming and developing shared libraries for some time by now, and there are a few things to pay certain attention to when dealing with this stuff.

- Flush before retry
  Libraries don’t go away simply because you close them, you know that. If you run your library once, close it and recompile it with a few changes, there will still be the older version remaining in memory that will be opened. When debugging libraries, always make sure that your library isn’t already in memory before debugging a new version!

  I made a small program that resets the open counter and then RemLibrary() a named library. It is not at all a nice thing to do, but there really is a problem when you open your library and something crashes before you have had the chance to close it. There is no "nice" way of removing such a library from memory!

- Globals
  By using the SAS/C object files libinit.o or libinitr.o you can make your global variables to be shared by all processes or unique for each OpenLibrary() call. If you want to mix the two versions or create something different, I advise you to code the library initial code by yourself.

- Stack usage
  When your library is called and runs, it uses the same stack as the caller. If the caller has a very small stack, so do you. Built-in stack check routines are not available since they need irreplaceable symbols. For advanced users, allocating and using an own stack while the library is running could be the only and best way to solve a problem like this.

- Symbols
  I’ve written it earlier and I do it again: high level language functions often use symbols initialized and declared in the startup modules. Declare them by yourself if possible or avoid using such functions!

- Register preservation
  I think it’s a good habit to always preserve all registers (except for D0 that holds the return code) when your library routines are called. Remember that your library code index register is un-initialized when called from the library opener.

1.8 AmigaDOS Shared Libraries - Support library calls from ARexx

```
6                        Support library calls from ARexx
------------------------------------------------------------------------------
```

ARexx is since the introduction of AmigaDOS 2.0 a part of the operating system, and is for earlier releases available as a separate product.

ARexx can access and call functions in shared libraries, if the shared libraries support it. This section will describe the actions that have to be taken to make your library support function invokes from ARexx.

6.1 How ARexx access the library

To access a custom shared library from ARexx, the ARexx program must call 'addlib' specifying library, version and "ARexx entry index". That third parameter is the index relative the librarybase to the function that is used for ARexx communication.

6.2 ARexx calls a library function

When a function is used within an ARexx program which the ARexx interpreter does not recognize, ARexx will call all libraries, one at a time, to see if the library recognizes the function. The first library that recognizes the function runs it!

6.3 ARexx function

The function that gets called from ARexx will receive a RexxMsg(1) pointer in register A0. The ARG0 member of the RexxMsg holds the name of the function that ARexx wants to run (the comparison should be case independent), and the parameters to the function is put in ARG1-ARG15. If your library doesn't recognize the function, you should return with 1 set in register D0, otherwise you should run the function with the specified parameters and return error in D0 (0=OK, 5=WARN etc). When returning OK, you can return a result string by putting a pointer to an ArgString(2) in A1, otherwise set A1 to 0 (zero).

(1) = See proper ARexx header file for struct RexxMsg definition.
(2) = See documentation for rexxsyslib.library/CreateArgstring().

1.9 AmigaDOS Shared Libraries - Appendix

```
                                    Appendix
------------------------------------------------------------------------------
```

A. Version numbers and shared libraries

Commodore has introduced a general standard for shared library version numbering. The libversion number is the number of the library version. The librevision number is expected to be a counter from 0 and upwards, without any kind of preceding zero. This makes the first library version 1.0 and such as 1.9 is followed by 1.10, 1.11 and so on all the way to the maximum,

of the same version, 1.65535.

Failing in the version number check of a library opening leaves the library in memory. For example, when you want to open "myown.library", it's loaded into memory. If the version number check fails and you get a NULL in return, "myown.library" will still remain loaded. The 'FILE' command line option does tell 'version' to explicitly use the file specified.

There are utilities which automatically updates a source file with the version number, revision number and the IDstring on every invoke. 'bumprev' is one.

B. Further reading

* Amiga ROM Kernel Reference Manual: Libraries, 3rd edition.

* Amiga ROM Kernel Reference Manual: Includes & Autodocs, 3rd edition.

C. Library source examples

Here follows a few example source codes. These are put here as simple examples of how it can be done. It may not be the best or most suitable solution for your imaginary project, but gives you a hint about how things can be done. The library created with the sources below is called "myown.library"

The sources are:

* Makefile A - A makefile written as a SAS/C v6+ user would have written it when compiling a library.

* Makefile B - A more general makefile. Change it to fit your particular compiler and assembler.

* myownass.a - An library entry source code in assembler. This contains all important initial structures and the four required functions.

* sasanddice.h - A header file to include in the following C sources to enable compilings under both SAS/C and Dice.

* myowninit.c - A C source version of the four required functions. These are included for those not too familiar with assembler.

* myown.h - Header file for the myown.library functions.

* myown.c - myown.library function source.

* myown.fd - Function descriptor file for myown.library.

* myown_pragmas.h - SAS/C pragmas for the library functions.

* uselibrary.c - A small program that uses the functions in myown.library.

```
Makefile A
==========

# This makefile uses the standard way of making a
# shared library with SAS/C. Using the already
# created object files SAS supports us with.

CC      = sc
CHEADER = myown.h
CSOURCE = myown.c
OBJ     = myown.o
LIBRARY = myown.library
FLAGS   = STRINGMERGE NOSTKCHK NOSTANDARDIO\
       DATA=NEAR NOVERSION LIBCODE\
       OPTIMIZE

$(LIBRARY): $(OBJ)
      slink with <<
     LIBFD myown.fd
     to $(LIBRARY)
     FROM lib:libent.o lib:libinit.o $(OBJ)
     noicons
     SD SC
     libid "myown.library 2.1 (18.04.93)"
     libversion 2 librevision 1
<
copy $(LIBRARY) LIBS: CLONE # Copy library to LIBS:

$(OBJ): $(CSOURCE) $(CHEADER)
     $(CC) $(FLAGS) $*.c

Makefile B
==========
# This makefile compiles everything and uses no
# pre-compiled files.
# Easy changed to fit DICE, Aztec or other
# compilers and assemblers.

CC      = sc
CHEADER = myown.h
CSOURCE = myown.c
ASOURCE = myownass.a
OBJS    = myown.o myownass.o
LIBRARY = myown.library
FLAGS   = STRINGMERGE NOSTKCHK NOSTANDARDIO\
       DATA=NEAR NOVERSION LIBCODE\
       OPTIMIZE
ASM     = asm
ASMFLAGS= -iINCLUDE:

$(LIBRARY): $(OBJS)
      slink to $(LIBRARY) FROM $(OBJS) noicons SD SC
      copy $(LIBRARY) LIBS: CLONE # Copy library to LIBS:

myown.o: $(CSOURCE) $(CHEADER)
```

```
	$(CC) $(FLAGS) $*.c

myownass.o: $(ASOURCE)
	$(ASM) $(ASMFLAGS) $*.a


myownass.a
==========

************************************************
*
* myown.library assembler source code
*
************************************************
* Author: Daniel Stenberg (dast@sth.frontec.se)
************************************************

   SECTION     code

   NOLIST
   INCLUDE "exec/exec_lib.i"
   INCLUDE "exec/types.i"
   INCLUDE "exec/initializers.i"
   INCLUDE "exec/libraries.i"
   INCLUDE "exec/lists.i"
   INCLUDE "exec/alerts.i"
   INCLUDE "exec/resident.i"
   INCLUDE "libraries/dos.i"

   LIST

   XDEF InitTable
   XDEF Open
   XDEF Close
   XDEF Expunge
   XDEF LibName

   XREF _SysBase

   XREF _LVOOpenLibrary
   XREF _LVOCloseLibrary
   XREF _LVOAlert
   XREF _LVOFreeMem
   XREF _LVORemove

   XREF _Min
   XREF _Abs

   ; Prevent library execution:

Prevent:
   MoveQ #-1,d0
   rts

;-----------------------------------------------
; The romtag structure is next:
;-----------------------------------------------
```

```
MYPRI EQU 0     ; priority zero...

VERSION EQU 2   ; version 2

REVISION EQU 1  ; revision 1

RomTag:
          ;STRUCTURE RT,0
   dc.w RTC_MATCHWORD ; UWORD RT_MATCHWORD
   dc.l RomTag        ; APTR  RT_MATCHTAG
   dc.l EndCode       ; APTR  RT_ENDSKIP
   dc.b RTF_AUTOINIT  ; UBYTE RT_FLAGS
   dc.b VERSION       ; UBYTE RT_VERSION
   dc.b NT_LIBRARY    ; UBYTE RT_TYPE
   dc.b MYPRI         ; BYTE  RT_PRI
   dc.l LibName       ; APTR  RT_NAME
   dc.l IDString      ; APTR  RT_IDSTRING
   dc.l InitTable     ; APTR  RT_INIT

; the name of our library
LibName:
   dc.b 'myown.library',0

; standard name/version/date ID string
IDString:
   dc.b 'myown.library 2.1 (21.11.93)',13,10,0

; force word alignment
   ds.w 0

; The init table
InitTable:
   dc.l LIB_SIZEOF  ; size of library base data, sizeof(struct Library)
   dc.l funcTable   ; pointer function pointer table below
   dc.l dataTable   ; pointer to the library data initializer table
   dc.l initRoutine ; routine to run

funcTable:

   ;----- standard system routines
   dc.l Open
   dc.l Close
   dc.l Expunge
   dc.l Extfunc

   ;----- our library functions
   ;The function names get those '_' in the
   ;beginning when compiling in C.
   dc.l _Min
   dc.l _Abs

   ;----- function table end marker
   dc.l -1


; The data table initializers static data structs.
```

```
dataTable:
   INITBYTE     LN_TYPE,NT_LIBRARY
   INITLONG     LN_NAME,LibName
   INITBYTE     LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
   INITWORD     LIB_VERSION,VERSION
   INITWORD     LIB_REVISION,REVISION
   INITLONG     LIB_IDSTRING,IDString
   dc.l 0

; The init routine.
initRoutine:     ; (segment list:a0)
   move.l a5,-(sp)       ; save a5
   lea    seglist(pc),a5 ; get address of our seglist storage
   move.l a0,(a5)        ; store segment list pointer
   move.l (sp)+,a5       ; restore previous a5
   move.l #0,d0          ; return zero
   rts

seglist:
   dc.l 0

;--------------------------------------------
; The four required functions:
; ****Assembler source code version****
;--------------------------------------------

Open:     ; (libptr:A6, version:D0)

  ; Increase the library's open counter
  addq.w   #1,LIB_OPENCNT(a6)

  ; Clear delayed expunges (standard procedure)
  bclr     #LIBB_DELEXP,LIB_FLAGS(a6)

  ; Return library base
  move.l   a6,d0
  rts

Close:     ; (libptr:a6)

  ; set the return value
  moveq    #0,d0

  ; Decrease the library's open counter
  subq.w   #1,LIB_OPENCNT(a6)

  ; If there is anyone still open, return
  bne.s    retlabel

  ; Is there a delayed expunge waiting?
  btst     #LIBB_DELEXP,LIB_FLAGS(a6)
  beq.s    retlabel

  ; Do the expunge!
  bsr      Expunge ; returns the segment list

  retlabel:
```

```
  rts

Expunge:     ; (libptr:a6)

  ; Is the library still open?
  tst.w    LIB_OPENCNT(a6)
  beq      notopen

  ; It is still open. set the delayed expunge flag
  ; and return zero
  bset     #LIBB_DELEXP,LIB_FLAGS(a6)
  moveq    #0,d0
  rts      ; return

  notopen: ; Get rid of us!

  movem.l  d2/a5/a6,-(sp) ; save some registers
  move.l   a6,a5

  ; Store our segment list in d2
  lea      seglist(pc),a6
  move.l   (a6),d2

  move.l   4,a6     ; get SysBase

  ; Unlink from library list
  move.l   a5,a1
  jsr      _LVORemove(a6) ; This removes our node from the list

  ; Free our memory
  moveq    #0,d0
  move.l   a5,a1
  move.w   LIB_NEGSIZE(a5),d0

  sub.l    d0,a1
  add.w    LIB_POSSIZE(a5),d0

  jsr      _LVOFreeMem(a6) ; This frees the memory we occupied

  ; Return the segment list
  move.l   d2,d0

  movem.l  (sp)+,d2/a5/a6 ; Get back the registers
  rts

Extfunc:     ; should return zero
   moveq   #0,d0
   rts

   ; EndCode is a marker that show the end of our
   ; code.
EndCode:
   END
```

sasanddice.h
============

```
/*********************************************
*
* Set some defines to enable either SAS or Dice
* compilings.
*
*********************************************/

#ifdef SAS
/*
 * Things to set for the SAS/C compiler:
 */
#define ASM __asm
#define DREG(x) register __d ## x
#define AREG(x) register __a ## x

#else
/*
 * Defines for the Dice compiler:
 */
#define ASM /* not used */
#define DREG(x) __D ## x
#define AREG(x) __D ## x
#endif
```

myowninit.c
==========

```
/*********************************************
*
* The four required library functions
* C source code version (SAS and Dice)
*
*********************************************/

#include "sasanddice.h"

long ASM Open(AREG(6) struct Library *MyBase)
{
  /* Increase the library's open counter */
  MyBase->lib_OpenCnt++;

  /* Clear delayed expunges (standard procedure) */
  MyBase->lib_Flags &= ~LIBF_DELEXP;

  return MyBase; /* return library base */
}

struct Library * ASM Close(AREG(6) struct Library *MyBase)
{
   struct Library *ret=NULL;

   /* Decrease the library's open counter */
   MyBase->lib_OpenCnt--;

   if(!MyBase->lib_OpenCnt) {
     /* not opened any more */
```

```
      if(MyBase->lib_Flags & LIBF_DELEXP)
        /* There is a delayed expunge waiting */
        ret = Expunge( MyBase );
    }
    return ret;
}

ULONG ASM Expunge( AREG(6) struct Library *MyBase )
{
    ULONG ret=NULL;
    long size;
    if(MyBase->lib_OpenCnt == 0) {
      /* we are not opened */

      ret = seglist; /* the 'seglist' we stored in the assembler init
                        routine! */
      /* remove us from the list */
      Remove ( (struct Node *) MyBase);

      /* get size to FreeMem() */
      size = MyBase->lib_NegSize + MyBase->lib_PosSize;

      /* FreeMem() */
      FreeMem( (char *) MyBase-MyBase->lib_NegSize, size );

    } else
      /* we are opened */
      MyBase->lib_Flags |= LIBF_DELEXP;
}

long ExtFunc()
{
  /* reserved for future use, should return 0 */
  return 0;
}
```

myown.h
=======

```
/*********************************************
*
* myown.library header file
*
*********************************************

/* Library function prototypes */

int Min(int, int); /* return minimum value */
int Abs(int);      /* return absolute value */
```

myown.c
=======

```
/*********************************************
*
```

```
* myown.library functions source code
*
*********************************************

#include "sasanddice.h"

int ASM Min(DREG(0) int a,
            DREG(1) int b)
{
  int c = a < b ? a : b;

  return (c);
}

int ASM Abs(REG(0) int a)
{
  int c = a < 0 ? -a : a;

  return (c);
}

myown.fd
========

##base _MyBase
##bias 30
##public
Min(num,num)(D0/D1)
Abs(num)(D0)
##end


myown_pragmas.h
===============

/* pragmas */

#if defined(SAS) || defined(DICE)

#pragma libcall MyBase Min 1E 1002 /* d0 and d1 */
#pragma libcall MyBase Abs 24 001  /* only d0 */

#elif defined(MAXON) || defined(AZTEC)

#pragma amicall(MyBase,0x1e,Min(d0,d1))
#pragma amicall(MyBase,0x24,Abs(d0))

#endif

uselibrary.c
============

#include "myown_pragmas.h" /* if using SAS/C or Aztec C */
#include "myown.h"
struct Library *MyBase=NULL;

void main(void)
```

```
{
  int min, abs;
  MyBase=OpenLibrary("myown.library", 2);

  if(MyBase) {
    min = Min( 3, 2 ); /* library Min() function */
    abs = Abs( -12 );  /* library Abs() function */

    CloseLibrary( MyBase );
  } else
    printf("Couldn't open myown.library!\n");
}
```