Writing Runtime Libraries with SAS/C 6.x

by John Wiederhirn

The benefits of the Amiga's runtime libraries go beyond their ability to share code and save memory. For a single application, runtime libraries offer application developers a great deal of flexibility. Some potential benefits include:

1. Patching applications by releasing new libraries.

If an application's runtime libraries house most of the its functionality, a developer can easily release bug fixes or patches by sending registered users the new versions of runtime libraries. SAS/C is a good example of this, as most of the compiler's functionality is in the runtime libraries. Since the libraries aren't particularly useful without the executeable ‘‘front end’’ of the program (the program that opens and uses the libraries), developers can readily distribute patches electronically without supplying the entire program.

2. Makes program ‘‘leveling’’ easier.

A developer can make an application upgradable from a ‘‘consumer’’ to a ‘‘professional’’ edition using a runtime library. The user can upgrade by adding a library. When the software starts up, it tries to open the enhanced library with the additional functions. If the software can't open the library, it configures itself as the ‘‘consumer’’ version.

3. Allows greater extensibility.

With careful design, you can expose an interface to your program which allows ‘‘scripts’’ to be written in C or other compiled languages. This can greatly enhance your program's extensibility beyond simple ARexx scripting. Done correctly, others can even use such an Application Programmer Interface (API) to create large scale turnkey applications by tying together smaller applications using C bindings.

4. Provides simple overlaying.

If an application opens and closes libraries only as it needs them, Exec's library handling system can act as a simple overlay manager. For example, consider a desktop publishing application that keeps all its printing routines in one library and all its screen rendering routines in another library. If the application closes its rendering library when it needs to print, the OS can expunge the rendering library if the system is low on memory while loading and using the printing library. When the application finishes printing, it can close the printing library and reopen the rendering library, so the OS can expunge the printing library while the application isn't using it.

The benefits sound useful but, unfortunately, designing and implementing runtime libraries on the Amiga has always been rather arcane. Developing a runtime library typically requires assembly programming and an intimate knowledge of obscure system structures with names like ‘‘ROMTag’’ and ‘‘MatchWords’’. As a result, only a small number of programmers are willing to tackle writing a runtime library.

To help make the task of developing Amiga runtime libraries easy, SAS added library building features to their compiler which eliminates a lot of the confusion surrounding runtime library development.

This article assumes that the reader has a basic knowledge of the structure of an Exec runtime library (also called a shared library) and the terminology surrounding them. For more information on the basics of runtime libraries, see the ‘‘Introduction to Exec’’ chapter of the *Amiga ROM Kernel Reference Manual: Libraries* (3rd edition).

Creating a runtime library using the SAS tools in Amiga C 6.x breaks down into a few distinct steps:

1. Writing the Functions

When designing the functions, you need to explicitly declare the register usage to the compiler. SAS does this with the ‘‘__asm’’ directive and the ‘‘register’’ keyword:

```
ULONG  __saveds __asm mylibfunc( register __d0 ULONG somevar,
                                 register __a0 APTR someptr );
```

The ‘‘__asm’’ portion of this prototype tells the compiler that mylibfunc uses a specific CPU register for each function parameter. The ‘‘register’’ keyword tells the compiler which CPU register to use for a specific parameter. In the sample above, the ‘‘mylibfunc’’ function requires two parameters. When the compiler compiles a program that uses ‘‘mylibfunc’’, it will place the first parameter, somevar, in CPU register D0 and the other parameter, someptr, in CPU register A0.

The SAS/C 6.x tools do impose some rules on the functions that will go into an Exec runtime library. The most important of these is that each and every routine in the library callable from outside the library must be declared using the SAS/C ‘‘__saveds’’ directive. For example:

```
struct Armadillo * __saveds __asm CreateArmadillo( register __a0 struct NewArmadillo * );
```

A programmer might not want all of a library’s functions accesible from outside the library. Several functions may use an internal subroutine that has no legitimate purpose outside of the library. These internal functions should be declared as ‘‘static’’ routines. A sample prototype might look like this:

```
        static BOOL ValidateArmadillo( struct Armadillo * );
```

Don’t get internal functions confused with functions with private entry points. An internal function is only accessible from other functions within the same library as it does not have a entry in the Library Vector Offset (LVO) table. The static functions are not accessible outside of the library, and do not need the ‘‘__saveds’’ and ‘‘__asm’’ directives.

Some of the Amiga’s ROM libraries have private entry point functions. These functions have an LVO and parameters like public functions. The only difference is that the public functions are documented (via prototypes and pragma files, which are discussed later) so all programmers can use them. For private entry point functions, only the library designer knows about the functions, so only the library designer can use them.

2. Selecting Data for the Library Base

Many libraries need some amount of global data (data that is external to any library function). For example, if the functions in mylib.library use functions from the graphics.library and intuition.library, mylib.library would probably keep the library bases of graphics.library and intuition.library in its global data area.

When an application opens a library, the OpenLibrary() function returns a pointer to a Library structure (defined <*exec/libraries.h*>). This structure contains information that Exec uses to manage the library system. Typically, runtime libraries keep their global data directly after their Library structure. An example of an Amiga ROM library followed by its global data is GfxBase. There is a GfxBase structure defined in <*graphics/gfxbase.h*> which starts off with an Exec Library structure and is followed by global data for the graphics.library.

The SAS/C library tools also use this scheme for global data. When building a library, the SAS/C linker, *Slink*, gathers data declared as global (data outside of any function) and appends that data after the Library structure. The linker handles this automatically, so the programmer doesn’t have to do anything special to make this happen.

Because the linker creates the library base automatically, there is no library base structure like the GfxBase structure. When an application opens this library with OpenLibrary(), OpenLibrary() returns a pointer to a Library structure which is followed by the library's global data. Unfortunately, the application (which we'll call the library's client) does not know how the linker arranged this global data. As a result, it is impossible for a library client to access any part of the library's global data using only the pointer returned by OpenLibrary(). This does not include the Library structure, only the global data that follows it.

At first, this ‘‘feature’’ may seem quite annoying, but it actually serves a purpose. The global data is global to the library, not the whole world. If the information were made directly accessible via a C structure, the format of the library's global data would have to remain fixed if the library were to maintain backwards compatibility. If a library needs to make a global data item accessible outside the library, the library can supply a function that returns a pointer to that global data.
As an example, if your library has a variable in its base declared as:

```
ULONG ActiveArmadillos; /* Number of armadillos the client's using */
```

and you decide that library clients need to access this data, then you can provide a function such as:

```
ULONG __saveds CountArmadillos( VOID );
```

which would simply return the value of ActiveArmadillos to the caller. This also means that if the format of ‘‘ActiveArmadillos’’ should change (say from a simple counter to a linked list of names), then the function can transparently adapt to the new design.

The 32K Barrier

A potentially important limitation of the SAS/C library generation is that Slink restricts the size of a library base. They way in which Slink builds the library causes the OS to create a ‘‘near data’’ section and ‘‘far data’’ section when loading the library into memory. Slink places the library base into the near section. Because of the way programs access data in the near section, the near section is limited to 32K.

If 32K is too limiting, there are alternatives. One is to force read-only data into the far section. To force a variable into the far section, use the ‘‘__far’’ keyword when declaring the variable. For example, when SAS/C encounters the following line in library source code:

```
APTR __far mylibtable;
```

it knows to place mylibtable in the far data section. This will generate a warning at link-time, but in this case the warning is ignorable. These items get placed in the far data section, which is shared among *all* clients of the library. Since all clients have to share the far data, the data has to either be read-only or the clients must use a semaphore to gain access to the data.

If a library is low on near data space and strings occupy a significant amount of that space, try the STRMERGE compile option. This tells the compiler to put the strings in with the library code instead of in the near data section.

3. Choosing the Type of Library Base

The SAS runtime library tools offer two ways to handle library bases. There is the conventional method where there is only one copy of a library's LVO tables, Library structure, and global data. In this case, OpenLibrary() returns the same library base pointer for every program that opens the same library. There is also a second method where each library client receives its own copy of the library's LVO tables, Library structure, and global data. In this case, OpenLibrary() returns a different library base pointer for every program that opens the same library. The *socket.library* from Commodore's AS225 TCP/IP networking software is an example of a library that returns a unique library base for each client.

If a library returns a different library base to each client, each instance of the library has its own near data space. This can be useful if the library needs to allocate non-sharable resources (for example, file handles) for each client. It can also be used to store parsing state information, callback hooks, or anything else you need to design as a part of the library base. Note that, although such a library can have many near data spaces, there is still only one far data space area.

A library that uses separate bases has drawbacks. Since each client receives its own copy of the library base, this scheme uses more memory. This scheme also complicates the open and close routines for the library, although the SAS/C runtime library generation tools hide the complications from the programmer. Also, the Exec routine SetFunction() will only work on one instance of a library base. If a client calls SetFunction() on its instance of the library base, it will not effect the existing library bases of other clients.

The difference between generating a single base library versus generating a multiple base library is almost too simple. When linking the library, the programmer only needs to use the libinitr.o object file instead of libinit.o. SAS/C takes care of the extra work involved in creating multiple base library, so the programmer doesn't have to worry about it.

Note that if a library opens a multiple base library, that library also has to be a multiple base library.

4. Handling Initialization and Shutdown

For every runtime library, the OS reserves several LVOs for system use. According to the ‘‘Introduction to Exec’’ chapter of the *Amiga ROM Kernel Reference Manual: Libraries*, the system currently uses three vectors for an open function, a close function, and an expunge function. The OS calls a library’s open function whenever a program opens the library with OpenLibrary(). The library’s open routine takes care of the initialization a library needs to do for every library client. The OS calls a library’s close function when a program closes a library with CloseLibrary(). The close routine relinquishes any resources allocated by the library’s open routine. When the system is low on memory, the OS calls a library’s expunge vector to ask the library to relinquish its resources so the system can unload the library.

The ‘‘Introduction to Exec’’ chapter neglects to mention the existence of a fourth function known as the library init function. This function does not have a true LVO like the other system reserved LVOs, which is one of the reasons its not well documented. The OS calls a library’s init function when opening the library for the first time. This routine handles initialization that is global to the entire library. It differs from the library’s open routine in that the system calls the init routine only once whereas the system calls the open routine for every OpenLibrary() call. The expunge routine cleans up after the init routine.

SAS/C automatically takes care of most of the work involved in creating these functions, but it can’t do everything. For example, if a library needs to open other libraries, the compiler can’t take of that automatically, so the library programmer has to take care of it.

When SAS/C builds a library, it looks for two functions in the library code, __UserLibInit() and __UserLibCleanup(). If SAS/C finds a function called __UserLibInit(), it includes that function in either the library’s init function or the library’s open function, depending on the library’s base type. If the compiler finds the __UserLibCleanup() function, it includes that function in either the library’s close function or the library’s expunge function.

In a multiple base library (a library base for each client), SAS/C makes the __UserLibInit() function part of the library’s open function and the __UserLibCleanup() function part of the library’s close function. In this case, the OS ends up calling the __UserLibInit() and __UserLibCleanup() functions every time a library client calls OpenLibrary() and CloseLibrary().

Because there are some significant restrictions on what OS functions a library can call from its expunge function, the __UserLibCleanup() function of a single base library is severely limited. Specifically, *the expunge vector cannot break a Forbid() state!* Since there are only a handful of OS functions that are guaranteed not to break the forbid state, very few OS functions are usable in the expunge function. Any function that uses the Wait() function breaks a forbid, so functions like

WaitPort() are illegal. DOS I/O is illegal. In fact, the only functions that are legal in expunge are those which specifically mention in their autodoc that they do not break a forbid. These include (but are not limited to):

```
AttemptSemaphore()          Disable()          FindPort()
ReleaseSemaphore()          Enable()           FindTask()
AllocMem()                  Signal()           AddHead()
FreeMem()                   Cause()            AddTail()
AllocVec()                  GetMsg()           RemHead()
FreeVec()                   PutMsg()           RemTail()
FindSemaphore()             ReplyMsg()         FindName()
```

In a single base library (one library base for all clients), SAS/C makes the __UserLibInit() function part of the library's init function and the __UserLibCleanup() function part of the library's expunge function. When the system encounters this type of library, it calls the __UserLibInit() and __UserLibCleanup() functions only as the system loads and unloads the library.

The prototypes for these routines are:

```
int __saveds __UserLibInit(void);
void __saveds __UserLibCleanup(void);
```

The following code sample uses these function to open and close some libraries. Notice that the __UserLibInit() call returns 0 if it is successful otherwise it returns 1.

```
#include <exec/types.h>
#include <graphics/gfxbase.h>
#include <clib/exec_protos.h>
#include <pragmas/exec_pragmas.h>

struct Library *GfxBase = NULL;
struct Library *SysBase = NULL;

int __saveds __UserLibInit(void)
{
  int retval = 1;

  SysBase = (*((void **)4));

  if (GfxBase = OpenLibrary("graphics.library",39L))
       retval = 0;

  return(retval);
}

void __saveds __UserLibCleanup(void)
{
  CloseLibrary(GfxBase);
}
```

Note that SAS/C does not require the library programmer to supply these functions.

5. Constructing the Function Description File

The function descriptor or ‘‘.fd’’ file defines all the function entry points in a runtime library. SAS/C uses this file to help compile the library. The following is an excerpt from the fd file ‘‘armadillo_lib.fd’’, which is part of the example source at the end of this article:

```
##base _DilloBase
##bias 30
##public
CreateArmadillo()()
DeleteArmadillo(dillo)(A0)
NameArmadillo(dillo,name,len)(A0/A1,D0)
##private
. . .

##end
```

The ##base command names the global variable that points to this library’s base. The ##bias command is always going to be 30 for SAS/C generated libraries (the reasons are not relevant and are beyond the scope of this article). The ##public command tells SAS/C that the function descriptions that follow should be callable by all programs. This is the opposite of the ##private command, which tells SAS/C that the function descriptions that follow all have private entry points. The ##public and ##private commands don’t affect the library itself. These become important later when creating pragma files.

Each function description consists of the name of a library function, the name of each of the function’s parameters, and the corresponding CPU registers the function uses to pass its parameters. For example, from the NameArmadillo() function description, the ‘‘dillo’’ parameter is passed in register A0, the ‘‘name’’ parameter is passes in register A1, and the ‘‘len’’ parameter is passed in register D0. The slash tells the compiler it can use the assembler instruction MOVEM when handling these registers (although it does not indicate a range of registers). For more information on writing .fd files, please consult page 29 of the SAS/C 6.0 Library Reference Manual.

6. Compiling and Linking

When compiling library code modules, make sure that stack checking is turned off (with the NOSTKCHK option), and that all the modules are compiled with the LIBCODE parameter.

Once all the modules compile, the next step is to link them all together into the runtime library. The *Slink* syntax for doing this is:

```
SLINK TO <libname>.library \
FROM LIB:libent.o <lib init module> <lib code modules> \
LIBFD <fdfile>         \
[LIB <link libraries>] \
[LIBPREFIX <prefix>]   \
[LIBID <id_string>]    \
[LIBVERSION <version>] \
[LIBREVISION <revision>]
```

`<libname>`	= Library name, for example ‘‘exec’’ or ‘‘dos’’.
`<lib init module>`	= LIB:libinit.o for a single base library or LIB:libinitr.o for a multiple base library.
`<lib code modules>`	= Your code modules for the library
`<fdfile>`	= The .fd file for your library
`<link libraries>`	= [Optional] Any link libraries used by your code.
`<prefix>`	= [Optional] An in-library prefix on your function names.
`<id_string>`	= [Optional] Used to set the __LibID value.
`<version>`	= [Optional] Version value to be embedded in the library.
`<revision>`	= [Optional] Revision value to be embedded in the library.

If you were designing ‘‘armadillo.library’’ which was comprised of code modules called dillo_open.c, dillo_close.c and dillo_read.c, an .fd file called armadillo_lib.fd, the SLINK command line might look like this:

```
SLINK TO armadillo.library \
FROM LIB:libent.o LIB:libinitr.o dillo_open. dillo_close.o dillo_read.o \
LIBFD armadillo_lib.fd \
LIBPREFIX "LIB" \        <-- See ‘‘Using LIBPREFIX to Distinguish Library Functions’’
from the ‘‘Additional Hints and Tips’’ section below.
LIBID "Armadillo Library" \
LIBVERSION 1 \
LIBREVISION 0
```

The ’\’ character is used to indicate a line wrapping onto the next line. In reality, you’d enter the entire command above as a single typed command. More likely, though, all of this would be combined into a makefile.

If all goes well, Slink will create the runtime library. The next step is to create prototypes and pragmas so applications can call the new library.

7. Protos and Pragmas

To make it possible for applications to use the new library, it needs prototypes and pragmas for its functions. The functions should already have protoypes from the library’s source code, but you’ll have to collect them into a single file and remove private functions.

The SAS/C tool fd2pragma generates a pragma file from the function description (‘‘fd’’) file. It should not generate pragmas for any functions that the fd file has marked as ‘‘##private’’.

When making <libname>_protos.h and <libname>_pragmas.h files (and matching .i files if you intend them to be accessible from assembly language), if your library has externally-accessible private routines, remove these from the pragma files you plan to distribute.

Additional Hints and Tips

(provided by the nice folks at SAS)

C Standard Library and Runtime Libraries

Many standard library functions rely upon some initialization and clean up code that is part of the standard startup code. Since a library does not use such start up code, a library can't call many of these functions. The stdio functions are an example.

In some cases with multibase libraries only (*not single base libraries!*), there are ways around this limitation. You can use malloc() if you make an explicit call to _MemCleanup() in a _UserLibCleanup() function. You can use level 2 file I/O functions (fopen(), fclose(), fread(), fwrite(), fprintf(), etc.) if you explicitly close all files that you open (of course Amiga programmers should be used to reliquishing any resources they allocate). Note that for level 2 file I/O to work, you must have been called by a process, not a task, and if you are a general-purpose library, there is no good way to guarantee this. *Remember that for a single base library, the expunge routine cannot break a forbid, so a single base library is very limited as to what it can call in its __UserLibCleanup()!*

Don't Use Stack Extension or Checking

Don't use stack extension or stack checking in a runtime library. Also, the __stack variable will have no effect.

Using __aligned on Library Functions and Variables

If you want the __aligned keyword to work correctly on automatic variables in your library, you must put the __aligned keyword on each function listed in your .fd file (along with __saveds and __asm.) This is because your library might get called from some other task or process whose stack is not aligned. This use of __aligned will cost you a pointer register variable for the duration of the function so declared, however.

Function Pointers, Callback Hooks, and You

You can’t pass function pointers to CreateProc() or callback hooks, even if you correctly use __saveds. This is due to the fact that __saveds in a runtime library depends on register A6 being set up to contain the library base, and this probably won’t be the case when your callback is called or when CreateProc() branches to your entry point. A good workaround is to pass getreg(REG_A4) as the userdata item for your callback, then do putreg(REG_A4, userdata) in your callback routine.

Using LIBPREFIX to Distinguish Library Functions

A keen trick is to declare your library routine with an "Extra" parameter in register A6 that is the library base, and to use a LIBPREFIX value to differentiate between the outside-the-library version of the function and the inside-the-library version.
Example:

```
int myfunc(int x, int y);   // User’s version of the prototype
#pragma libcall MyLibBase myfunc <magic>  // Pragma for myfunc

....
/* Now, in the source code for the library: */
int __asm __saveds
LIBmyfunc(register __d0 int x,
register __d1 int y,
register __a6 struct Library *MyLibBase)
{
...
}
```

To tell Slink that LIBmyfunc is really the ’myfunc’ referred to in the .fd file, use the LIBPREFIX=_LIB option. If you do this, you may refer to simply ’myfunc’ in your code, both inside and outside the library, and all such references will go through the library base, giving the user a chance to SetFunction() your function correctly if desired. The fact that the library base is present as an explicit variable is a nice bonus.

No Auto-Open Library Code

You can’t rely on the auto-open library routines to work correctly in a runtime library. You must explicitly open each library base, and close it when you are done. This includes SysBase, DOSBase, etc. Be particularly careful when using UTILLIB or MATH=IEEE, since these routines silently try to open various library bases for you. You have to remember to open these bases yourself.

About the Example

The example code implements a do-nothing library, called armadillo.library. This library has five entry points, and they serve only to demonstrate some of the different library base access features. Pay particular attention to the prototypes and mapping of the function code to the prototypes and .fd file for the library. There are two library routines which are not externally-accessible and one private externally-accessible entry point which is not reflected in the prototypes or pragmas (but is noted in the .fd file).

The makefile is a ‘‘generic’’ framework for a runtime library’s makefile. With only slight modifications, this same file could be used for most runtime libraries you might want to create (and you are free to use it as desired).