Linking and Loading in AIX 4.3

By Gary R. Hook, Vandana Kumar, and Steve Peckham

This article provides an overview of various aspects of the AIX linker and system loader important in developing new applications on AIX and in porting existing UNIX applications from other platforms to AIX. It shows several examples, including source code, makefiles, suggested command-line options, and other details.

In the past, program compilation was always limited to static linking. This technique combined together multiple libraries and multiple object files along with their code and data, and created an executable file. This single and often very large file contained all the symbol definitions within it. Although this made the linking and execution process fast and simple, every running process had to load into memory its own private copy of every library function it used.

The need to share libraries that were already loaded in memory by other programs led to the concept of *shared libraries*. When a program is linked with a shared library, the code within the shared library is not included in the generated executable. Instead, enough information is saved in the program to find and load the library when needed.

Shared library code is loaded into global system memory by the first program, then it is shared by all subsequent programs. This technique reduces memory usage and loading time. Additionally, enhancements to the shared libraries can be added without relinking the executable.

This approach works well if the location of symbol definitions is known when the program is initially linked. When the location of some symbol definitions is not known at link time or if alternate definitions are to be provided for some functions, a more flexible linker is needed. AIX® 4.3 provides this flexibility via its *runtime linker*, which allows programs to dynamically load shared objects into the running process via the `libdl.a` routines.

This article describes the new linking and loading features available in AIX 4.3, including code examples.

Gary R. Hook

Vandana Kumar

Steve Peckham

Symbol Resolution in AIX 4.3

Without the runtime linker in AIX, all symbols must be accounted for when the module is linked. The undefined symbols must be listed in the module's import list or be deferred. Symbols are deferred if they are listed as being defined by #! in the import list, which provides the name of the defining module for each listed symbol. These symbols are resolved during load-time.

Deferred Symbols

The linker and loader effectively ignore deferred symbols because they are resolved later. This is usually done by dynamic loading of the defining module into the running process by the `load()` or `dlopen()` calls. All modules that are opened in the address space of the executing process cause the outstanding deferred imports to be resolved. This happens according to the scope of the flags specified in the `dlopen()` and `dlsym()` function calls. Note that the defining module has to export the referenced symbols for the referencing module to find them.

When using the runtime linker, however, symbols can be left undefined at link-time. These symbols are marked as being imported from ".." in the module's symbol table, which can be listed by using the `dump -HTv <module name>` command. The runtime linker resolves these symbols later. Import lists are not needed in this case.

The functioning of the deferred symbols does not change under runtime linking. All deferred symbols must be listed as defined by `#!` in the import file. The runtime linker ignores these symbols when they are designated as deferred. If no deferred symbols are identified, the linker treats all undefined symbols as ".." imports, and the runtime linker searches and resolves these symbols at load-time. If the runtime linker is unsuccessful in locating the definition of any of the referenced symbols, the module fails to load and the program exits.

Deferred imports (also called anonymous imports) are listed as being imported from `[noIMPid]` in the symbol table section of the `dump -HTv <module >` output.

Third-Party Plug-in Software

Programs often provide stubs for third-party plug-in software so that some function symbols referenced by the program are executed only if the plug-in software is available. If this software is unavailable, the symbols are referenced but never get defined or used.

The runtime linker cannot handle these symbols because the load-time search among all exported symbol definitions yields nothing. Therefore, they should be marked as deferred to get past the loading phase. Once the module is loaded, the deferred symbols must be defined before the functions in question are ever called, otherwise they can be left undefined. If a call is ever made to an unresolved deferred import, the application will terminate with a segmentation fault in the glink code (glue code for branching between modules).

Dealing with undefined symbols that do not have a definition at module load-time requires some extra work. However, it ensures symbol availability in advance for better execution performance. Note that for the best performance on any platform, the symbols under runtime linker control should be kept to a minimum.

In AIX, libraries are often archives of multiple object files, which could reference one another or have dependencies of their own.

Shared Modules

Generally in AIX, a shared module used as input is listed only if a symbol in the shared module is actually referenced. If so, it would be listed in the loader section of the symbol table of the output file. When the runtime linker is used, however, it might be necessary to have the shared modules listed in the loader section, even if there are no symbols referenced.

By using runtime linking (via the `-brtl` option), all shared modules (those that are not archive members) on the command line are listed as dependents (preserving the command-line order). The system loader loads all such shared modules when the program runs, and the symbols exported by these shared modules are used by the runtime linker. Shared modules that are members of archives are listed as dependents only if they are referenced. The linker does not assume that archive members will be needed. Some archive members may have requirements that cannot be satisfied at program load-time, thereby causing loading failure.

In AIX, system libraries are often archives of multiple object files, which could reference one another or have dependencies of their own (which may or may not be available at runtime). Also, all archive members of system libraries are not necessarily shared objects. For this reason, it is not enough to simply add a reference to them in the main executable. All required system libraries must be linked in to build each shared module. By linking with system libraries, all local definitions of symbols get resolved within the shared module.

To force a module in an archive to be loaded, an import must be used with the *autoload* keyword and the member name. For example, to enable autoloading for an archive member `foo.so`, create a file with the following lines, and add the file as a member to the archive:

```
# autoload
#! (foo.so)
```

For the module load to be successful, all its dependent modules must also load successfully. Therefore, if a shared module references symbols defined by other shared modules, those modules also must be loaded to resolve symbols referenced by the first module. This continues until no more modules need to be loaded. Only deferred imports do not cause a search and load of the defining module at load-time.

The next section explores the runtime linker in more detail, including code examples.

Runtime Linking

Runtime linking is the ability to resolve symbols in shared modules after the program has begun to execute. It is a mechanism that provides runtime definitions and symbol rebinding capabilities.

Note that the main application must be built to enable runtime linking. Simply linking a module with the runtime linking capability is not sufficient. This structure allows a module to be built to support runtime linking (such as third-party modules), yet continues to function in an application that has not been so enabled. See Figure 1.

The option `-G`, which is used to enable runtime linking in shared modules, is a combination of the options found in Figure 2.

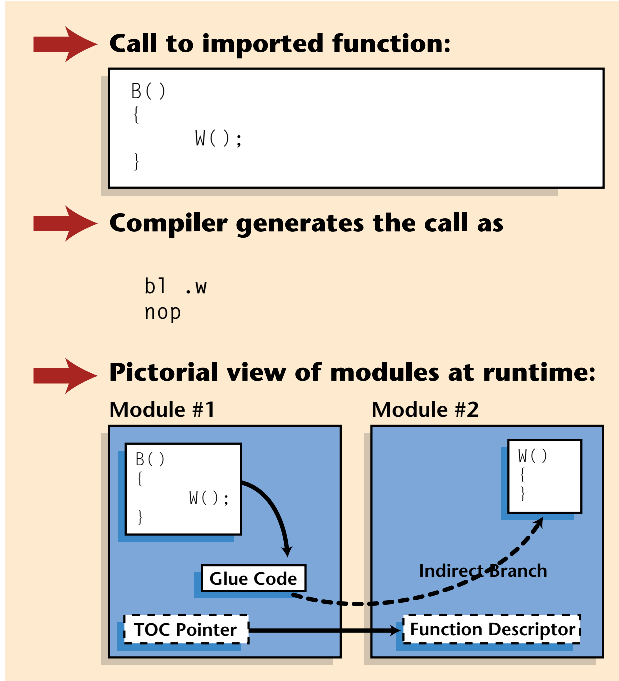

Figure 1. Generated code

`-berok`	Enables creation of the object file even if there are unresolved references.
`-brtl`	Enables runtime linking. All shared objects listed on the command line (those that are not part of an archive member) are listed in the output file. The system loader loads all such shared modules when the program runs, and the symbols exported by these shared objects may be used by the runtime linker.
`-bnosymbolic`	Assigns this attribute to most symbols exported without an explicit attribute.
`-bnortllib`	Removes a reference to the runtime linker libraries. This means that the module built with `-G` option (which contains the `-bnortllib` option) will be enabled for runtime linking, but the reference to the runtime linker libraries will be removed. Note that the runtime linker libraries should be referenced to link the main executable only.
`-bnoautoexp`	Prevents automatic exportation of any symbol.
`-bM:SRE`	Build this module to be shared and reusable.

Figure 2. The -G option

The option -brtl alone should be used to enable runtime linking in main executables.

For a simple introduction to runtime linking, see Example 1 in this article. In this example, the runtime linker searches and locates the undefined symbols during program execution. This eliminates the need for programmers to track which module calls and defines which symbols, or to keep the export/import lists updated.

In general, to take advantage of the AIX architecture, the shared modules should be built to be as self-contained as possible. Note that AIX performance and efficiency is best exploited by a well-organized application structure with a well-defined interface between modules.

Rebinding Symbols

It is often necessary to rebind function symbols so that a new definition for them may be used instead of the definition from which the module was originally built. In this case, the main program must be linked with the runtime linker, and the module providing the alternate definition must export the symbol and must be listed before the original module on the command line.

Based on how the module was linked, some references to exported symbols are bound at link-time and cannot be rebound. For example, if a symbol is referenced in the same file that also defines it, the symbol will be bound at link-time and cannot be rebound.

When runtime linking is used, a reference to a symbol in the same module can only be rebound if the symbol is exported with the proper attributes. References to symbols with the "symbolic" attribute cannot be rebound. References to symbols with the "nosymbolic" attribute can be rebound if the symbols are variables. For function symbols, calls using a function pointer can be rebound, but direct function calls cannot. The "nosymbolic-" attribute, which is the default, is provided for compatibility with previous versions of AIX.

When a symbol is rebound, a dependency is added from the module using the symbol to the module defining the symbol. This dependency prevents modules from being removed from the address space prematurely. This is important when a module loaded by the dlopen() subroutine defines a symbol that is still being used when an attempt is made to unload the module with the dlclose() call.

It is also possible to rebind the definitions of system library symbols. For example, an application can define its own malloc() to override or extend the functionality of the system-defined malloc() in libc.a. For this, the defining shared module (in this case shr.o in libc.a) has to be rebuilt using rtl_enable so that the references internal to the module become hookable. Once this is done, the runtime linker can override all references to malloc(). Without rebuilding libc.a via rtl_enable, the internal workings of libc.a remain "tight" and cannot be rebound.

New Features

AIX 4.3 provides several new linker/loader options. These options, which give additional programmatic control over the behavior of the modules, are listed and described briefly in this section.

> ***When runtime linking is used, a reference to a symbol in the same module can only be rebound if the symbol is exported with the proper attributes.***

libdl.a routines: Provides the standard routines dlopen(), dlclose(), dlsym(), and dlerror() in libdl.a. These routines can be used with or without runtime linking. A new flag, RTLD_MEMBER can be passed to dlopen(), which allows an archive member to be loaded.

init/fini routines: Provides module initialization and termination routines via the -binitfini option.

Loading of archive members: Allows loading of archive members by passing the flag L_LOADMEMBER to load() or loadAndInit(). It may also be passed to dlopen(); the name of the flag is RTLD_MEMBER in the dlopen() case. This function loads member `shr.o` from archive `lib1.a`. For example:

```
dlopen("lib1.a(shr.o)", RTLD_NOW|RTLD_MEMBER);
```

-bexpall option: Exports most symbols. Exports all global symbols (except imported symbols, unreferenced symbols defined in archive members, and symbols beginning with an underscore). Additional symbols may be exported by listing them in an export list. Any symbol with a leading underscore will not be exported by this option. These symbols must be listed in an exports list to be exported. Linking the module with the `makeC++SharedLib` script will also export these symbols. Note that the `makeC++SharedLib` script generates a complete export list for C++ modules by default, without specifying the `-bexpall` option in its command line.

Visibility attributes: Allows visibility specification of all symbols via the `-bsymbolic/-bnosymbolic/-bnosymbolic-` flags. This feature can be used to control which references can be rebound by the runtime linker. Visibility keywords also can be used in an import file to control the visibility of individual symbols.

-bdynamic/-bstatic: Controls whether shared objects used as input files should be treated as regular files. These options are toggles and can be used repeatedly in the same link line. When `-bdynamic` is in effect, shared objects are used in the usual way. When `-bstatic` is in effect, shared objects are treated as regular files. Additionally, when `-brtl` is specified and `-bdynamic` is in effect, the `-l` flag will search for files ending in `.so` as well as those ending in `.a`. Figure 3 shows an example.

-bautoexp option: Exports a symbol automatically if any command-line shared module imports the symbol from the special dot (.) file, or the module being linked contains a local definition of the symbol. The default is `-bautoexp`.

-bipath/-bnoipath options: Saves/discards the full path name of the shared module on the command line. The full path is saved in the loader section of the module. The `-bipath` option is the default.

```
cc -o main main.o -bstatic -lx -Lnewpath -bdynamic
-> libx.a treated as a regular object file
-> libc.a processed as a shared library

cc -o main main.o -brtl -lx -Lpath1 -Lpath2
-> Search path1/libx.so, then path1/libx.a
-> Search path2/libx.so, then path2/libx.a
```

Figure 3. -bdynamic/-bstatic

If `-bnoipath` is used, then only the base names will be saved in the output files loader section. For example:

```
cc -o main main.o dir/mylib.so /usr/lib/otherlib.a
```

This will cause the full specified path to be saved for `mylib.so` and `otherlib.a`. At load-time, the loader will always use these paths to find the shared objects. If the `-bnoipath` option was specified, only the base names would have been saved in the loader section.

-blazy option: Lazy loading is a mechanism for deferring the loading of modules until one of its functions is required to be executed. By default, the system loader automatically loads all of the module's dependents at the same time. By linking a module with the `-blazy` option, the module is loaded only when a function within it is called for the first time. Note that lazy-loading works only if runtime linking is not enabled. Also, only the modules referenced for their function definitions can be lazy-loaded.

Runtime linking: Consists of the `-G` and the `-brtl` options.

(`-G` option) Enables runtime linking. It is equivalent to the following: `-berok -brtl -bnortllib -bnosymbolic -bnoautoexp -bM:SRE`. These options can be overridden by subsequent options. The `-G` option is used to build runtime-enabled shared libraries, without actually linking in the runtime linker libraries.

(`-brtl` option) Enables runtime linking. It is equivalent to `-brtllib -bsymbolic`. The `-brtl` option is used to build runtime-enabled main executable and includes a reference to the runtime linker libraries in the link line.

Under the influence of runtime linking, all shared objects listed on the command line (those that are not archive members) are listed as dependent modules, which preserves the command-line order.

This option is also useful since it allows the `.so` files to be found with the `-l` flag. If the `.so` files are to be found but runtime linking is not needed, use `-brtl` and `-bnortllib` together.

".." imports: Marks the undefined imported symbols as ones that will be resolved by the runtime linker. If a `dump -HTv <shared module>` output contains `".."` as the IMPid for any symbol in the symbol table, the runtime linker will search for and resolve that particular symbol.

"." imports: Marks the undefined symbol as being defined in the main executable. For example:

```
#! .
foo
```

The runtime linker is not needed since the system loader resolves these `"."` symbols. However, the main program must still export these symbols.

rtl_enable command: Relinks modules to enable runtime linking. The shared libraries shipped with AIX are not enabled for runtime linking, but they can be enabled by using this command. For example, if a program uses its own version of `malloc()`, a new instance of `libc.a` must first be created as follows:

```
rtl_enable -o /usr/local/lib/libc.a
/lib/libc.a
```

Then the program must be relinked:

```
cc ... mymalloc.o -L /usr/local/lib
-brtl -bE:myexports
```

In this example, `mymalloc.o` defines `malloc()` and `myexports` causes `malloc()` to be exported from the main program. Calls to `malloc()` from within `libc.a` will now go to `malloc()` defined in `mymalloc.o`.

-brtllib/-bnortllib: Enables or disables runtime linking. Default is `-bnortllib`. In general, it is recommended that runtime linking be enabled by linking in the `-brtl` option and not by the `-brtllib` option directly.

Tips and Techniques

The following tips and tricks can be used to help linking and loading on AIX 4.3.

- Shared modules that use any C++ code must always be linked using the `makeC++SharedLib` script. This script is a wrapper around the `ld` command and does some C++ related processing before linking the module. Check the compiler documentation for details on its use.

- The `dump -HTv <module name>` command provides information about the shared module's symbol table.

- The `genkld` command provides a listing of the currently loaded shared libraries. This command is part of the `perfagent.tools` fileset.

- The shared modules built in AIX 4.1.5 can be referenced by an AIX 4.3 binary if the symbols are not under the runtime linker control. The AIX 4.1.5 modules should be manually loaded into the executable's address space using `dlopen()` calls.

- Diagnose problems in case of a linking failure by using the binder options as follows:

  ```
  -bloadmap:<file>
  -bmap:<file>
  ```

- In AIX, libraries can be listed multiple times and in any order (except when symbols are rebound). The first occurrence of a symbol wins, although warnings appear on additional occurrences.

- At link-time for every shared module, all needed system libraries must be linked in, including (but not limited to)

  ```
  -lc -lC -lm -lX11
  ```

- If the `makeC++SharedLib` script is used to build shared libraries and export symbols, make sure that the system libraries are always specified with an `-l` option (for example, `-lc`) and never by name on the `script_s` command line. This allows the system libraries to be simply referenced and not go through special C++ related processing.

- If large parts of the shared libraries are paged in all at once because of C++ calls or many references between libraries, it may be faster to read the library rather than demand-page it into memory. Remove read-other permission from all shared libraries and see if the loading performance improves. If it does, then reset the original permissions and set the following environment variable:

```
LDR_CNTRL=PREREAD_SHLIB
```

  By using this environment variable, the libraries are read very fast into the shared memory segment.

- If a loader error occurs at program execution time in which a certain library cannot be found (even though the `LIBPATH` clearly points to it), check the following:
  - The library should have read permission.
  - The loader section of the calling module's `dump` output should point to the correct directory.
  - The `-bnoipath` option should not be used with the library to link the module; all dependent modules of the library should be available for loading.
  - All dependent modules of the loading module have to load successfully, so follow the module tree to see which leaf did not load (possibly due to wrong `LIBPATH` or disabled read permissions).

- If linking or loading difficulties arise, it is best to simplify the environment (including disabling the version control system) and debug the problem by taking one step at a time.

Examples

This section begins with a few basic C files (shown in Figure 4), which will be used to illustrate several examples later in this section.

a.c

```
#include <stdlib.h>
#include <stdio.h>

extern int b();

int a()
{
    printf( "Now in function a()\n" );
    b();
    return( 0 );
}
```

b.c

```
#include <stdlib.h>
#include <stdio.h>

extern void c1();

int b()
{
    printf( "Now in function b()\n" );
    c1();
    return( 0 );
}
```

c1.c

```
#include <stdlib.h>
#include <stdio.h>

int c1()
{
    printf( "Now in function c1()\n" );
    return( 0 );
}
```

hello.c

```
#include <stdlib.h>
#include <stdio.h>

extern  int     a();

int main()
{
    printf( "\nHello World\n" );
    a();
    return(0);
}
```

Figure 4. Basic C files

Example 1

The example in Figure 5 shows building a shared library in AIX 4.3 using the runtime linker.

Figure 6 shows symbol table information for `liba.so` obtained by the command `dump -HTv liba.so`.

The loader symbol table output in Figure 6 means that the symbol `printf` comes from the shared module `shr.o`, an archive member of the system library `libc.a`. The runtime library will search for the symbol `b` when `liba.so` is loaded. This is indicated by the " .. " in the `IMPid` column for symbol `b` in Figure 6 (recall that runtime symbol lookup is only performed if the main application is built with runtime linking enabled).

Figure 7 shows how to build the final executable.

```
cc -c a.c -o a.o
cc -c b.c -o b.o
cc -c c1.c -o c1.o
cc -c hello.c -o hello.o

ld -o liba.so a.o -bnoentry -G -bexpall -lc
```

Figure 5. Build shared library with runtime linker

Loader Section

```
Loader Header Information
   VERSION#       #SYMtableENT      #RELOCent        LENidSTR
  0x00000001       0x00000003       0x00000006       0x00000023

  #IMPfilID         OFFidSTR         LENstrTBL        OFFstrTBL
  0x00000003       0x000000b0       0x00000000       0x00000000
```

Import File Strings

```
    INDEX            PATH              BASE            MEMBER
      0         /usr/lib:/lib
      1                              libc.a            shr.o
      2                                ..
```

Loader Symbol Table Information

```
[Index]      Value        Scn    IMEX Sclass    Type       IMPid            Name

  [0]     0x00000000    undef       IMP      DS EXTref    libc.a(shr.o)    printf
  [1]     0x00000018    .data       EXP      DS SECdef    [noIMid]         a
  [2]     0x00000000    undef       IMP      DS EXTref    ..               b
```

Figure 6. Symbol table information for liba.so

```
ld -o libb.so b.o -bnoentry -G -bexpall -lc
ld -o libc1.so c1.o -bnoentry -G -bexpall -lc
xlc -o hello liba.so libb.so libc1.so hello.o -L. -brtl

On executing "hello":

>hello

Hello World
Now in function a()
Now in function b()
Now in function c1()
```

Figure 7. Building the final executable

Example 2

Figure 8 shows a rebinding example.

f1.c

```
#include <stdio.h>
int func1()
{
    printf( "\tinside of func1()/%s...\n", __FILE__ );
    printf( "Calling func2()...\n" );
    func2();
}
```

f2.c

```
#include <stdio.h>

int func2()
{
    printf( "\tinside of func2()/%s...\n", __FILE__ );
    printf( "Calling func3()...\n" );
    func3();
}
```

f3.c

```
#include <stdio.h>

int func3()
{
    printf( "\tinside of func3()/%s...\n", __FILE__ );
    printf( "Calling func4()...\n" );
    func4();
}
```

f4.c

```
#include <stdio.h>

int func4()
{
    printf( "\tinside of func4()/%s...\n", __FILE__ );
}
```

main.c:

```
#include <stdio.h>

int func4() /* func4 is being redefined here in main.c */
{
    printf( “\tinside of func4()/%s...\n”, __FILE__ );
}

extern int func1();

main()
{
    printf( “Calling func1()...\n” );
    func1();
}
```

(continued on following page)

Figure 8. Rebinding example

(continued from previous page)
Compiling and linking:

```
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c main.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f1.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f2.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f3.c
xlc -D_POSIX_SOURCE -D_ALL_SOURCE -g -qextchk -c f4.c
ld -o libshr2.so f3.o f4.o -G -bnoentry -bexpall
ld -o libshr1.so f1.o f2.o -lshr2 -G -bnoentry -L. -bexpall
xlc -o main main.o -lshr1 -lshr2 -L. -brtl -bE:main.exp
```

The following WARNING message will be emitted by the binder:

```
ld: 0711-224 WARNING: Duplicate symbol: .func4
ld: 0711-224 WARNING: Duplicate symbol: func4
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
```

Since `main.o` appears before `libshr1.so` and `libsh2.so`, the `func4()` that is defined by `main` is used. The results of running `main` are as follows:

```
Calling func1()...
        inside of func1()/f1.c...
Calling func2()...
        inside of func2()/f2.c...
Calling func3()...
        inside of func3()/f3.c...
Calling func4()...
        inside of func4()/main.c...
```

Note that symbol `func4` has been rebound to the one defined in `main.c`.

Figure 8. Rebinding example

Example 3

Figure 9 shows an example of runtime linking and dynamic loading of non-linked objects.

main.c

```
#include <stdio.h>

int main()
{
    vmap_routine();
    usr_preempt();
}

void main_routine()
{
    printf( "in main_routine in main.c\n" );
}
```

(continued on following page)

Figure 9. Runtime linking and dynamic loading of non-linked objects

(continued from the previous page)

vmap.c

```
#include <stdio.h>

void vmap_routine()
{
    printf("in vmap_routine in vmap.c\n");
    main_routine();
    usr_routine();
}

void vmap_axs_routine()
{
    printf( "in vmap_axs_routine in vmap.c\n" );
}

void usr_preempt()
{
    printf( "in standard usr_preempt routine in vmap.c\n" );
}
```

usr.c

```
#include <stdio.h>
#include <dlfcn.h>

typedef void (*void_fn)();

void usr_routine()
{
    printf( “in usr_routine usr.c\n” );
    axs_routine();
}

void usr_preempt()
{
    void_fn routine;

    printf( “in correct usr_preempt routine in usr.c\n” );
    routine = (void_fn) dlsym( dlopen( “./libvmap.so”, RTLD_NOW ), “usr_preempt”);
    routine();
    routine = (void_fn) dlsym( dlopen( “./libext.so”, RTLD_NOW ), “ext_routine” );
    routine();
}
```

axs.c

```
#include <stdio.h>

void axs_routine()
{
    printf( "in axs_routine in axs.c\n" );
    vmap_axs_routine();
}
```

(continued on following page)

Figure 9. Runtime linking and dynamic loading of non-linked objects

(continued from the previous page)

ext.c

```
#include <stdio.h>

void ext_routine()
{
    printf( "in ext_routine in ext.c\n" );
}
```

Compile and link:

```
xlc -c -g usr.c axs.c ext.c main.c vmap.c -D_ALL_SOURCE
ld -o libusr.so usr.o -bnoentry -G -bexpall -bM:SRE -lc -ldl
ld -o libvmap.so vmap.o -bnoentry -G -bexpall -bM:SRE -lc
ld -o libaxs.so axs.o -bnoentry -G -bexpall -bM:SRE -lc
ld -o libext.so ext.o -bnoentry -G -bexpall -bM:SRE -lc
cc -o main main.o libusr.so libvmap.so libaxs.so -brtl -L.
```

Ignore the `ld` warning regarding duplicate symbols in this instance; there *are* two instances of `usr_preempt`. The output produced by this program is, as expected:

```
> main
in vmap_routine in vmap.c
in main_routine in main.c
in usr_routine usr.c
in axs_routine in axs.c
in vmap_axs_routine in vmap.c
in correct usr_preempt routine in usr.c
in standard usr_preempt routine in vmap.c
in ext_routine in ext.c
```

Figure 9. Runtime linking and dynamic loading of non-linked objects

This example is particularly interesting. First, symbol `usr_preempt()` is defined in two modules: `libvmap.so` and `libusr.so`. Second, symbol `usr_preempt()` in `usr.c` (`libusr.so`) uses the same symbol as defined in `vmap.c` (in`libvmap.so`) via an explicit `dlopen()` of `libvmap.so`. Third, an explicit `dlopen()` of module `libext.so` is not linked to the main module.

Example 4

Figure 10 shows an example of calling an AIX 4.1.5 shared library from AIX 4.3 binary.

main.c

```
#include <stdio.h>
#include <dlfcn.h>

main()
{
    void *handle;
    void (*fct)();

    printf("hello from main\n");
```

(continued on following page)

Figure 10. Calling AIX 4.1.5 shared library from AIX 4.3 binary

(continued from the previous page)

```
    /* main() needs to run foo_() which is defined by libfoo.so.
     * libfoo.so has been previously built on AIX 4.1.5.
     */
    handle=dlopen("libfoo.so",RTLD_NOW);
    if (handle==NULL)
    {
        perror("dlopen");
        exit(1);
    }

    fct=(void (*)())dlsym(handle,"foo_");

    if (fct==NULL)
    {
        perror("dlsym");
        exit(1);
    }

    (*fct)();
    printf( "exit from main\n" );
}
```

foo.c

```
#include <stdio.h>
void foo_()
{
    printf( "hello from foo()\n" );
}
```

To build `main` on AIX 4.3:

```
xlc -brtl -g -o main main.o -L. -ldl
```

To build `libfoo.so` in AIX 4.1.5:

```
ld -o libfoo.so foo.o -bM:SRE -bE:exp.lst -bnoentry -lc
```

Where `exp.lst` contains the symbol `foo_`.

Running `main` yields the following output:

```
> main
hello from main
hello from foo()
exit from main
```

Figure 10. Calling AIX 4.1.5 shared library from AIX 4.3 binary

Example 5

Figure 11 shows autoloading of archive members.

main.c

```
main()
{
    bar();
}
```

bar.c

```
int bar()
{
    printf( "in bar()\n" );
    foo();
}
```

foo.c

```
#include <stdio.h>

int foo()
{
    printf( "in foo() which is correct...\n" );
}
```

barfoo.c

```
#include <stdio.h>

int foo()
{
    printf( "in barfoo() which is wrong...\n" );
}
```

foo.auto

```
#!
# autoload
#! (shr.o)
```

Makefile

```
all:        libbar.so libfoo.a main

main:       main.o
            cc -o main main.o -L. -lfoo -lbar -brtl

libbar.so:  bar.o barfoo.o
            ld -G -o libbar.so -lc -bexpall bar.o barfoo.o -bnoentry

libfoo.a:   foo.o foo.auto
            ld -G -o shr.o foo.o -bexpall -lc -bnoentry
            $(AR) $(ARFLAGS) $@ shr.o foo.auto

clean:      -rm -f *.o *.a *.so core main *.lst *.map
```

Figure 11. Autoloading archive members

The goal for the example in Figure 11 is to provide a module that will override a definition from another module. Looking at the rules for building `main`, we see that `libfoo.a` is listed before `libbar.so`. This latter module contains a definition for `bar()`, which calls `foo()`. The definition for `foo()` is in the same module, `libbar.so`.

When building `libfoo.a`, the shared module containing an alternate definition for `foo()` is in another shared module, which is then placed within an archive. The linker will not (even when using runtime linking) automatically create references to shared modules within archives. To force it to accept and reference a module within an archive, an autoload script is used (`foo.auto`).

This script tells the linker that the named module within the archive containing the script is to be referenced by the module being built. Therefore, when linking `main`, the linker examines `libfoo.a` first, sees the autoload script, and understands that a reference to the module `shr.o` within `libfoo.a` is to be placed in the loader section. The linker then examines `libbar.so`, sees the definition of the referenced symbol `bar()`, and creates a loader section reference to that file.

Since runtime linking is enabled in this example, the loader section will list modules in the order in which they were specified on the linker's command line. At runtime, the definition for `foo()` resolves all references from any module. This will then override the definition provided by the module `libbar.so`. Therefore, the correct output will be as follows:

```
> main
in bar()
in foo()
```

References

Online documentation is available at:
http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixgen/

Gary R. Hook, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Hook joined IBM in 1990 and is currently an advisory software engineer in the IBM Server Group. He is the author of the Shared Libraries Hookable Symbols/6000 product and is currently a member of the AIX kernel development team, working on object file tools and the system loader. Mr. Hook has a BS in Electrical Engineering from the University of New Mexico.*

Vandana Kumar, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Ms. Kumar is an advisory engineer in the IBM Server Group. She has worked with several AIX application vendors in porting, enabling, and tuning their software on the RS/6000 platform. Ms. Kumar has a MS in Electrical Engineering from Wayne State University.*

Stephen Peckham, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Dr. Peckham is an advisory software engineer in the IBM Server Group. He developed the AIX Version 4 link editor and, for the past several years, he has been a member of the AIX kernel development team, working on the system loader and runtime linker. Dr. Peckham has a Ph.D. in Computer Science from Cornell University.*