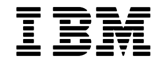


AIX® Linking and Loading Mechanisms

By

Bradford Cobb, Gary Hook, Christopher Strauss,
Ashok Ambati, Anita Govindjee, Wayne Huang, Vandana Kumar

May 2001

Table of Contents

Overview ____ *3*
A Brief History of Application Linking Mechanisms ____ *3*
Architectural View of Linking and Loading in AIX ____ *4*
Differences between AIX and System V Implementations ____ *5*
Developing Applications on AIX ____ *6*
Dynamic Loading ____ *9*
Rebinding Symbols ____ *11*
Re-linking Executables with Modified Object Files ____ *12*
Lazy Loading ____ *13*
Linking Considerations for AIX Application ____ *16*
Special Linking Options ____ *16*
Linking C++ Files ____ *18*
Linking Java™ programs with native C/C++ libraries using JNI ____ *21*
Linking with Threaded Libraries ____ *23*
Linking With Fortran Libraries ____ *24*
Linking/Loading libraries over NFS ____ *26*
LDR_CNTRL Environment variable ____ *27*
Common Problems ____ *29*
Tips and Tricks ____ *30*
Examples ____ *35*
Appendix A ____ *57*
Appendix B ____ *59*
Appendix C ____ *60*
Appendix D ____ *62*
Glossary ____ *65*

Overview

This document provides an overview of the various aspects of the linking and loading mechanisms that are important in developing AIX applications on the PowerPC architecture. It includes examples with source code, makefiles, command line options, and other details. This document should be used as a guide to understanding the key functional features of the linker and loader, with an emphasis on application development or migration. For specific details about the behavior of *ld* or compiler command line flags, please refer to respective software documentation. AIX documentation is available in a searchable format at: http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixgen/
The compiler documents are available on the compiler CDs.

A Brief History of Application Linking Mechanisms

The traditional UNIX compilation technique historically involved "Static Linking." With this technique, multiple object files defining global symbols, and containing code and data, were combined and written to an executable file with all the references resolved. The executable was a single self-contained file, which often became quite large. However, since the name and location of all symbols were resolved at link-time, a program could be executed by simply reading it into memory and transferring control to its entry point. Static linking, which is still used in certain circumstances, has the following drawbacks:

- If any of the libraries used by the program are updated, the program needs to be re-linked to take advantage of the updated libraries.
- Disk space is wasted because every program on the system contains private copies of every library function that it needs.
- System memory is wasted because every running process loads into memory its own private copy of the same library functions.
- Performance could suffer due to the fact that the binaries are larger than they need to be, causing more paging to occur.

To allow the libraries to be shared among different programs, the concept of *shared libraries* evolved. When a program is linked with a shared library, the shared library code is not included in the generated program. Instead, information is saved in the program that allows the library to be found and loaded when the program is executed. Shared library code is loaded into global system memory by the first program that needs it and is shared by all programs that use it. AIX provides facilities for creating and using shared libraries. There are four main advantages of shared libraries:

1. Less disk space is used because the shared library code is not included in executable programs.
2. Less real memory is used because the shared library code is loaded only once.
3. Load-time is reduced if the shared library has already been loaded by another program.

4. Customers obtain faster bug fixes, as they can download updated libraries from the Web and restart their application, instead of waiting for the re-linked executables from the vendor.

When a program is linked with shared libraries, the resulting executable contains a list of shared libraries needed by the program and a list of symbols imported by it. The actual code is not included. Therefore, if one of the shared libraries is updated, programs using that library do not need to be re-linked, automatically picking up the current version of the library on the next execution. This makes application service and support easier since new versions of individual libraries containing patches and fixes, can be shipped to customers without having to rebuild and ship a whole new version of the application.

This approach works well if the location of the symbol definition is known when the program is initially linked. The location of symbol definition is specified via "import files", which contain a list of symbols needed by the program, and the shared libraries that define them. During program execution, the symbols do not have to be searched, but can be found by simply looking up their addresses in the specified shared library.

In many new applications, the location of some symbol definitions may not be known at link-time. In addition, applications often provide alternate definitions for some library functions with the expectation that all modules calling those functions would use the alternate definitions. To support this style of development, AIX provides a "run-time linker." AIX also allows a program to dynamically load a shared object into the running process via a *dlopen()* function call within the code.

Architectural View of Linking and Loading in AIX

The linker and loader on AIX are designed such that libraries are self-contained entities with well-defined interfaces consisting of sets of imported and exported symbols. Symbol resolution is performed at link-time, simplifying the work of the system loader when they are loaded. The system loader looks up symbols to relocate references, but does not perform symbol resolution. As a result, a shared library and its dependents can be "pre-relocated" in global memory. Once a set of modules have been pre-relocated, a program using the modules can be loaded more efficiently since symbol lookup and relocation in the pre-relocated modules does not have to be performed.

Additionally, new features such as run-time linking (RTL) and dynamic loading, provide flexibility in how symbol resolution is performed. Please see the next section below for more information on RTL and dynamic loading. For the definitions of any terms found in this document, please refer to the Glossary.

To get a pictographic view of the timeline involved in program development and execution on AIX, see Figure 1 below.

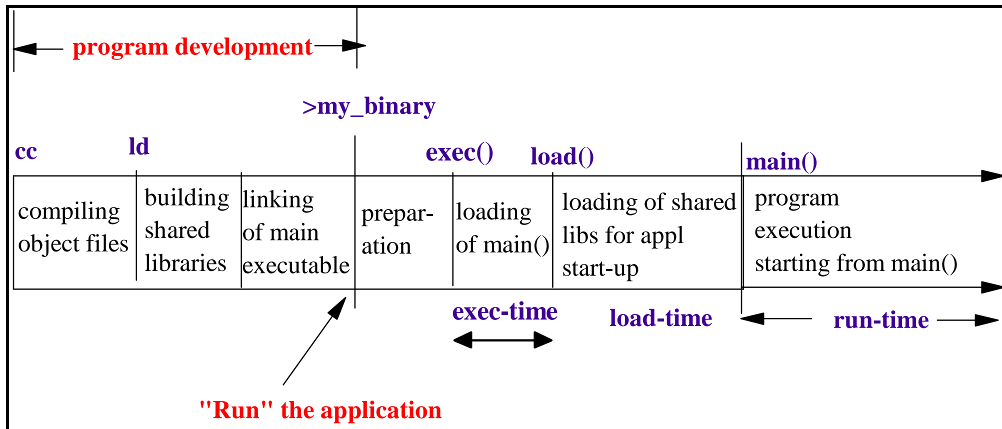


Figure 1.0 Program development, link, load, & execution timeline

Differences between AIX and System V Implementations

UNIX application developers typically work with several different UNIX implementations on various platforms. While all UNIX implementations provide the same general linking and loading functionality, there are some differences in the way shared libraries are created and symbols resolved. This section describes some of those differences between AIX and other System V based UNIX kernels.

AIX and System V have different views of shared objects. AIX sees shared objects as fully linked and resolved entities, where symbol references are resolved at link-time and cannot be rebound at load-time. System V sees them as being very similar to ordinary object files where resolution of all global symbols is performed at run-time by the linker. Thus, conventional shared modules in AIX cannot have undefined symbols, while System V can. All symbols must be "available" either as symbols defined in the CSECTs of the module itself, or as imported symbols in the import list of the module. However, with the introduction of run-time linking (RTL) in AIX, the shared object implementation has been made more flexible. Under RTL, symbols may be left undefined until load-time, at which time a search is performed among all loaded modules to find the necessary definitions.

A shared module in AIX can be an object file or an archive file member. A single shared archive library can contain multiple members, some of which are shared modules and some ordinary object files. In System V, shared libraries are always ordinary files, which are built from source files compiled with special options and linked in a special way. Typically in AIX system shared libraries are archive files. This makes it easier to provide updates to the system libraries with minimum impact to the applications. As an example, consider the Motif library libXm.a. In AIX, it is shipped as an archive of members, shr4.o and shr_32.o. The member shr4.o is a Motif 1.2 object and shr_32.o is a Motif 2.1 object. Applications that were linked with Motif 1.2 on previous operating systems load shr4.o, while new applications compiled on AIX 4.3.3 load shr_32.o. All this happens transparent to the application programmer. Other System V implementations would have

to provide two versions of the Motif library, and the application would have to be re-linked with the correct one.

In AIX, all of the linker's input shared objects are listed as dependents of the output file only if there is a reference to their symbols. In System V the names of all shared libraries listed on the command line are saved in the output file for possible use at load-time. However, in AIX 4.2 and later, the *-brtl* option causes all shared objects (except archive members) on the command line to be listed as dependent objects in the output file. If *-berok* is specified in addition to *-brtl*, undefined symbols are marked so that the system loader will ignore those symbols allowing them to be resolved by the run-time linker. (The undefined symbols are marked as being imported from ".." in the loader section of the symbol table.)

In System V when a shared object is loaded, all references are resolved by the run-time linker, which searches all the modules currently loaded in the process. In many cases, resolution of function symbols is deferred until the function is first called (references to variables must be resolved at load-time). This allows the definition for a function to be loaded after the module referring to that symbol is loaded. In AIX, even when the run-time linker is used, all symbols (except the "deferred" symbols) have to be resolved at load-time. The run-time linker allows certain symbols marked as "deferred", to pass through at load-time in an unresolved state. These symbols then become the responsibility of the application programmer to resolve. In any case, a definition *must* exist in the process at the time the referencing module is loaded.

While this implementation of the linker in AIX puts more pressure on the application programmer for maintaining a well-defined interface, this "up-front" work consistently yields better run-time results.

Developing Applications on AIX

We begin with the technique to create applications with simple shared libraries, and then moves on to dynamic loading, run-time linking, and rebinding. At the end of this section is a detailed section on symbol resolution for all the different kinds of linking mechanisms.

Here is the procedure to create simple shared libraries, put them in an archive file (if desired), and build the executable. This example illustrates the use of simple shared libraries created with a well-defined export/import interface.

Simple shared libraries
To create simple shared libraries:

1. Create one or more source files that are to be compiled and linked to create a shared library. These files contain the exported symbols that are referenced in other source files.

For this example, two source files, share1.c and share2.c, are used. The share1.c file contains the following code:

```
/************
* share1.c: shared library source.
*************/

#include <stdio.h>

void func1 (){
        printf("func1 called\n");
}

void func2 (){
        printf("func2 called\n");
}
```

The share2.c file contains the following code:

```
/************
* share2.c: shared library source.
*************/

void func3 (){
       printf("func3 called\n");
}
```

The exported symbols in these files are func1, func2, and func3.

2. Create a main source file that references the exported symbols that will be contained in the shared library.

For this example, the main source file is named main.c. The main.c file contains the following code:

```
/************
* main.c: contains references to symbols defined
* in share1.c and share2.c
*************/

#include <stdio.h>

extern void func1 (),
            func2 (),
            func3 ();
main (){
        func1 ();
        func2 ();
        func3 ();
}
```

3. Create the exports file necessary to explicitly export the symbols in the shared library that are referenced by other object modules.

Here, an exports file named shrsub.exp (with the following code) is used.

```
#! /home/sharedlib/shrsub.o
*
* Above, we have the full pathname to the shared library object file
* Below, we have the list of symbols to be exported:
*
func1
func2
func3
```

The #! line is meaningful only when the file is being used as an import file. In this case, the #! line identifies the name of the shared library file to be used at run time.

4. Compile and link the two source code files to be shared. (This procedure assumes you are in the /home/sharedlib directory.) To compile and link the source files, enter the following commands:

```
xlc -c share1.c
xlc -c share2.c
xlc -o shrsub.o share1.o share2.o -bE:shrsub.exp -bM:SRE -bnoentry
```

This creates a shared library name shrsub.o in the /home/sharedlib directory. The -bM:SRE flag marks the resultant object file shrsub.o as a re-entrant shared library.

(The -bnoentry flag tells the linkage editor that the shared library does not have an entry point.)

Each process that uses the shared code gets a private copy of the data in its private process area. A shared library may have an entry point, but the system loader does not make use of an entry point when a shared library is loaded.

5. Use the following command to put the shared library in an archive file:

```
ar qv libsub.a shrsub.o
```

(This step is optional. Putting the shared library in an archive makes it easier to update the libraries with minimum effect on the application later on.)

6. Compile and link the main source code with the shared library to create the executable file. (This step assumes your current working directory contains the main.c file.) Use the following command:

```
xlc -o main main.c -L/home/sharedlib -lsub
```

If the shared library is not in an archive, use the command:

```
xlc -o main main.c /home/sharedlib/shrsub.o -L/home/sharedlib
```

The program *main* is now an executable. The func1, func2, and func3 symbols have been marked for load-time resolution. At run-time, the system loader loads the module into the shared library (unless the module is already loaded) and dynamically resolves the references.

Dynamic Loading

Dynamic loading allows an application to load a new module into its address space for either adding new functionality or implementing an existing interface, without having to link the required libraries.

Here is a brief description of the functions that implement dynamic loading:

<u>dlopen()</u>
This function loads the specified module into the address space of the calling process, and returns a handle to the loaded module. Dependents of the module are automatically loaded as well. If any of its dependents cannot load due to say, an unresolved symbol, the whole module will fail to load. If the module is already loaded, it is not loaded again, but a new unique handle is returned. The handle is used in subsequent calls to *dlsym()* and *dlclose().*

If the main application or previously loaded modules in the process address space have any deferred symbols (undefined symbols whose definition lookup has been deferred until run-time), an attempt is made to resolve them using the exported symbols from the loaded modules, unless forbidden by the application.

If the module or any of its dependents is being loaded for the first time, initialization routines for these newly loaded modules are called before the function returns. Initialization routines are the functions specified with the *–binitfini* linker option when the module was built. Also, if the module contains C++ code, constructors for the static objects are called before the function returns.

Note that *dlopen()* on AIX also allows loading of archive members of shared libraries via the RTLD_MEMBER flag.

<u>dlsym()</u>
This function looks up the specified symbol in the loaded module (and its dependents) and returns its address.

Note that where the module has been coded in C++, the symbol name given must match the name as mangled by the C++ compiler. To obtain the mangled names for C++ symbols in an object file a.o, use the c++filt utility as follows:

```
dump –tv a.o | c++filt
```

dlclose()
This function is used to close access to a module loaded with the *dlopen()* subroutine. In addition, access to dependent modules of the module being unloaded is removed as well. Modules being unloaded will however, not be removed from the process' address space if they are still required by other modules. Nevertheless, subsequent uses of that handle are invalid and further uses of symbols that were exported by the module being unloaded result in undefined behavior.

A *dlclose()* call results in several routines, which are termination routines specified with the *–binitfini* option to the linker. Also, if the module contains C++ code, destructors for the static C++ objects in the module are called before the function returns. Note that the termination/destructors are called when there are no more references to the module from within an application or other modules within the application and the module has been unloaded.

dlerror()
This function is used to obtain information on the last error that occurred in an immediately preceding call to: *dlopen(), dlclose(),* or *dlsym()*. Once a call is made to this function, subsequent calls, without any intervening dynamic loading errors, will return NULL. Hence it may be a better idea for multi-threaded applications to make error determination using the status held in *errno.*

Refer to the ***Examples*** section for several test cases involving dynamic loading.

Run-time Linking
Run-time linking is the ability to resolve undefined and undeferred symbols in shared modules after the program execution has already begun. It is a mechanism for providing run-time definitions and symbol rebinding capabilities. For example, if main() calls foo() in libfoo.so, which then calls bar() in libbar.so, and if libfoo.so and libbar.so were built to enable run-time linking, then the main application could provide an alternate definition of bar() which would override the one originally found in libbar.so. (Check in the "Symbol Resolution" section for more information on deferred symbols and the exact resolution process used by the linker.)

Note that it is the main application that has to be built to enable run-time linking. Simply linking a module with the run-time linker is not enough. This structure allows a module to be built to support run-time linking (say, third party modules), yet continue to function in an application which has not been so enabled.

Shared objects in run-time linking are created by using the shorthand notation -G. This option is a combination of the following options:

-berok	Produces the object file even if there are unresolved references.
-brtl	Enables run-time linking.
-bnosymbolic	Assigns this attribute to most symbols exported without an explicit attribute.

-bnortllib	Removes the reference to the run-time linker library.
-bnoautoexp	Prevents automatic exportation of any symbol.
-bM:SRE	Build this module to be shared and reusable.

The module built with -G option (hence containing the -bnortllib option) will be enabled for run-time linking, but a reference to the run-time linker library itself will be removed. Only the main program needs to have a reference to the run-time linker libraries. Therefore, linking the shared libraries with –G and the main module with -brtl enables run-time linking for the application.

For a simple introduction to run-time linking, please refer to ***Example 4*** in the ***Examples*** section. In this example the undefined symbols are searched for and located by the run-time linker during program execution. The programmer does not have to tracks which modules call and define which symbols, or 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. Run-time linking should be used only when necessary. Code ported from other platforms often does not have this sort of organization and can therefore require extra effort to enable it on AIX. It is important to emphasize the fact that the performance and efficiency of AIX 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 their definition may be picked up from a different location (*e.g.* from the main program) than the definition the module was originally built with. In this case the main program must be linked with the run-time linker and the module providing the alternate definition has to export the symbol.

Depending 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 gets bound at link-time and cannot be rebound.

When using run-time linking, a reference to a symbol in the same module can only be rebound if the symbol is exported with the proper attribute. 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, while direct function calls cannot.

Imported symbols have no export attribute. If a symbol is imported from another module, all references to the symbol can be rebound. Whenever 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 possible to rebind the definitions of system library symbols as well. For example, an application can define its own *malloc()* to override or extend the functionality of the system defined *malloc()* in libc.a. To do 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, run-time linker is able to override all references to *malloc().* Without rebuilding libc.a via rtl_enable, the internal workings of libc.a remain "tight" and cannot be rebound.

Several C++ applications overload system functions such as *new* and *delete*. This can be accomplished using the rebinding features of the linker. Please refer to ***Example 10***.

Note: stripped executables cannot be rebound.

Re-linking Executables with Modified Object Files

In AIX the binder can accept *both* object files and executables as input. This means that the user has the ability to rebind an existing executable to replace a single modified object file instead of building a new executable from scratch. *No other major UNIX implementation offers this feature*.

In other UNIX systems, the input to the *ld* command is a set of files containing object code, either from individual object files or from libraries. The *ld* command then resolves the external references among these files and writes an executable with the default name of *a.out*. If a bug is found in one of the modules that was included in the *a.out* file, the defective source code is modified and recompiled, and then the entire *ld* process must be repeated, starting from the full set of object files and libraries.

Since the binding process consumes storage and processor cycles according to the number of files being accessed and references resolved, rebinding an executable with the new version of a module is quicker than binding it from scratch.

To pre-bind a library, the following command can be used on the archive file:

```
ld -r libbar.a -o libbar.o
```

The compile and bind of a FORTRAN file file1.f would be:

```
xlf file1.f libbar.o
```

(Note the use of libbar.o as an ordinary object file, and not with the usual library identification syntax -lbar.)

To recompile the module and rebind the executable after fixing a bug:

```
xlf file1.f a.out
```

(Note that if the bug fix resulted in a call to a different subroutine in the library, the bind would fail.)

Another example:

```
xlc -c file1.c
```

The executable mynewapp can be quickly built in this way:

```
xlc -o mynewapp file1.o  myapp -llibraries
```

The old file1 is still in mynewapp, but the new file1 will be referenced from it, because the linker will find the new file1 definition first.

Lazy Loading

Lazy loading is a mechanism for deferring the loading of modules until one of its functions is executed. At this time this capability has limited use for AIX applications developed natively, but could be useful in some cases where the application is ported from other platforms.

As an example, if main() calls foo() and foo() is in the shared module libfoo.so, then libfoo.so is a candidate for lazy loading. If foo() is never called then libfoo.so need not be loaded at all. If foo() is called, the lazy loading code executes the load() function to load libfoo.so, patches a function descriptor to make sure subsequent calls simply go directly to the function itself, then invokes the called function. If for some reason the module cannot be loaded, the lazy-loader's error-handler is invoked.

By default, the system loader automatically loads all of the module's dependents at the same time. This occurs because a list of the dependent modules is saved in the loader section of the module at link-time (the *dump -H <filename>* command can be used to view this information). By linking a module with the *-blazy* option, it is possible to enable the loading of only the needed dependents. Please note that if the *-brtl* option is specified, the *-blazy* option is ignored and lazy loading is not enabled. This means that any module linked with the *-brtl* option to enable run-time linking cannot have its dependents lazy-loaded. Also, only those modules will be lazy-loaded which are being referenced for their functions only. If variables in the module are being referenced, the module is loaded in the normal way. Since lazy loading does not work for run-time enabled modules, it has significance and used only for the modules that are built using import lists to identify the undefined symbols. Refer to ***Example 8*** for an illustration.

By using lazy loading, the loading performance of a program can be improved if most of a module's dependents are never actually used. Every function call to a lazy-loaded module has an additional overhead of about seven instructions and the first call to a

function requires loading the defining module and modifying the function call. Therefore, if a module calls the functions in most of its dependents, the appropriateness of enabling lazy loading should be carefully considered.

Please note that a program that compares function pointers might not work correctly when lazy loading is used because a single function can have multiple addresses. In particular, if module A calls function *f* in module B, and if lazy loading of module B was specified when module A was linked, then the address of *f* computed in module A differs from the address of *f* computed in other modules. Thus, when using lazy loading, two function pointers might not be equal even if they point to the same function. Also, if any modules are loaded with relative path names and if the program changes working directories, the dependent module might not be found when it needs to be loaded. When using lazy loading, only absolute path names should be used to refer to dependent modules at link-time.

Since lazy loading works only with conventional style linking on AIX, individual modules that are: mostly self-contained, referred for function symbols only, and have a low probability of being loaded anyway should only be considered for lazy loading. The decision to enable lazy loading is made at link-time on a module-by-module basis. In a single program it is possible to mix modules that use lazy loading with modules that do not.

Symbol Resolution on AIX

Now that dynamic loading, run-time linking, and a general understanding of the shared library architecture on AIX have been introduced, it will be helpful to delve deeper into the exact mechanism for symbol resolution on AIX. In general, symbol resolution happens at link-time on AIX. While the actual binding does not happen till load-time, the export/import interface offers a "promise" to the loader that the imported symbols will be available from the specified library *when* they are needed. Run-time linking merely delays resolution of all undefined symbols that the system loader cannot ignore, until load-time of the shared modules.

At load-time these undefined symbols are resolved by searching among the exported symbols of all linked-in shared modules and are marked as being imported from ".." in the module's symbol table (the symbol table can be listed by using the `dump -HTv <file name>` command). Symbols could also be marked as deferred by listing them as being defined by "#!", followed by the name of the symbol in the import list. This list is then used to build the shared object that refers to this symbol.

References to deferred symbols are ignored by the run-time linker, so resolution of these symbols must be performed by the system loader either via a call to *loadbind()* or by loading a new module explicitly with *load()* or *dlopen()* calls. Deferred imports (also called anonymous imports) are listed as having "[noIMPid]" as their import ID in the symbol table section of the `dump -HTv <file name>` output. Refer to ***Examples 2 and 3***.

If no deferred symbols are identified, the linker treats all undefined symbols as ".." imports, and these symbols are searched for and resolved by the run-time linker at load-time. If the run-time linker is unsuccessful in locating the definition of any of the referenced symbols, the module fails to load and the program exits. On the other hand, deferred symbol resolution can be delayed until just before use, with the definition being provided later by dynamic loading of the defining module(s).

All modules that are opened into the address space of the executing process via *dlopen()* cause resolution of the outstanding deferred imports to happen according to the scope of the flags specified in the *dlopen()* and the *dlsym()* function calls. The defining module has to export the referenced symbols in order for the referencing module to find them.

Programs often provide stubs for third party plug-in software so that some of the function symbols that the program references get executed only when the plug-in software is available. If this software is unavailable, the symbols are referenced but never get defined or used. These symbols cannot be handled by the run-time linker because load-time search among all exported symbol definitions yields nothing. They should therefore be marked as deferred to get past the loading phase. Once the module is loaded though, the deferred symbols must be defined before being called. If a call is 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, which do not have a definition at module load-time can be painful. However, the linker implementation in AIX tries to ensure symbol definition availability in advance for better execution performance by identifying and accounting for all undefined symbols. For the best performance on any platform, the symbols under run-time linker control should be kept to a minimum.

Ordinarily, a shared object used as input is only listed in the loader section of the output file if a symbol in the shared object is actually referenced. When the run-time linker is used however, it might be necessary to have the shared objects listed in the loader section even if there are no symbols referenced. By using run-time linking (via the *–brtl* option), all shared objects listed on the command line (that are not archive members) are listed as dependent modules preserving the command line order. The system loader loads all such shared objects when the program runs, and the symbols exported by these shared objects may be used by the run-time linker. The shared objects that are members of archives are listed as dependents only if they are referenced. The linker does not assume that archive members are needed because it can lead to trouble with system libraries, some of whose members may have requirements that won't be satisfied at program execution-time and hence cause loading failure.

On other platforms, the simple addition of an option such as -lX11 is enough to create a reference to a dependent and make that dependent available at run-time for such activities as symbol look-ups (via the *dlsym()* function). On AIX, since 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 run-time), it is not enough to simply add a reference to them in the main executable. All the needed system libraries *have* to be linked in to build each shared module. By linking with system libraries, all local definitions of symbols get resolved within the shared module.

A point to note about AIX is that a lot of symbol resolution work is done at the initial load-time of the shared libraries. For a module to be successfully loaded, all its dependent modules also have to be successfully loaded. Therefore if a shared module references symbols defined by other shared modules, those modules also have to be loaded to resolve symbols referenced by the first module. This continues until no more modules need to be loaded. Only the deferred imports do not cause a search and load of the defining module at load-time. This up-front work consistently yields better run-time performance of AIX applications.

Linking Considerations for AIX Application

This section explains various commonly encountered issues related to linking libraries on AIX. These issues involve special linking options, compiler versions, language considerations, memory considerations and development environments.

Special Linking Options

-binitfini:

Init/fini routines provide a generic method for module initialization and/or termination for both program exec and exit and dynamic loading and unloading. The *-binitfini* linker option is used to specify the names of the routines along with a priority number. The priority is used to indicate the order that the routines should be called in when multiple shared objects with initialization or termination routines are used.

When the *dlopen* function call is used to open a shared object within the code, any initialization routines specified with the *-binitfini* option will be called before the *dlopen* returns. Similarly, any termination routines will be called by the *dlclose* subroutine. See Appendix B for more information on using the *-binitfini* linking option.

Loading of archive members:

This 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 dlopen()'s case. As an example:

```
load("lib1.a(shr.o)", L_LOADMEMBER, NULL)
```

This function loads member shr.o from archive lib1.a.

Visibility Attributes:

This allows visibility specification of all symbols via the *-bsymbolic, -bnosymbolic,* and *-nosymbolic* flags. This feature can be used to control which references can be rebound by the run-time linker. Visibility keywords can also be used in an import file to control the visibility of individual symbols.

-bdynamic/-bstatic:

These flags control whether shared objects used as input files should be treated as regular files. These options are toggles and can be used repeatedly. 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. For example:

```
xlc -o main main.o -bstatic -lx -Lnewpath -bdynamic
```

- libx.a is treated as a regular object file
- libc.a is processed as a shared library

Additionally, when *-brtl* is specified and *-bdynamic* is in effect, the -l flag will search for files ending in .so as well as ending in .a, though .so is given precedence over .a.

```
xlc -o main main.o -brtl -lx -Lpath1 -Lpath2
```

- search path is path1/libx.so:path1/libx.a
- search path is path2/libx.so:path2/libx.a

-bautoexp:

This flag means that a module automatically exports a symbol if any command line shared object 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:

This option is used to save/discard the full path name of the specific shared object on the command line. The full path is saved in the loader section of the module. The *-bipath* option is the default. If *-bnoipath* is used, then only the base names will be saved in the output files loader section. For example:

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

This will cause the fully 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, even if the libraries are specified with a full path. To then find the libraries, there are three options: (1) use -L during the link to set up the library search path to be embedded in the module; (2) use *-blibpath* to fully specify the library search path to be embedded in the module; or (3) use LIBPATH at runtime.
If run-time linking is not needed, one can use *-brtl* and *-bnortllib* together. Lastly, a combination of *-bnoipath* and *-blibpath* should result in the most flexibility for the generated module, its dependents, and the exec-time library search path.

-brtllib/-bnortllib:

These options enable or disable run-time linking. The default is *-bnortllib*. In general it is recommended that run-time linking be enabled by linking in the *-brtl* option, and not by the *-brtllib* option directly. Run-time linking is a topic important enough to

require its own section. It is explained in more detail in the Run-time Linking section.

Linking C++ Files

The VisualAge V5.0 (VAC 5) compiler provides a new flag: *-qmkshrobj* to link C++ libraries. The VAC5 compiler for AIX also supplies the *makeC++SharedLib* shell script, as do the previous IBM C++ compiler products for AIX. VAC5, however, also allows the use of the new *-qmkshrob*j option to the compiler. The *-qmkshrobj* option is used to instruct the compiler to create a shared object from previously created object files and archive libraries. It provides similar functionality to the *makeC++SharedLib* command and, in addition, makes it much easier to create shared objects that use template functions. For more information on template functions and the *-qmkshrobj* compiler option, see the Redbook: *C and C++ Application Development on AIX*, at: http://www.redbooks.ibm.com and search for: SG24-5674-00.

Construction of C++-based modules requires more than just linking: munching of the C++ object code is required, as is template instantiation and other steps. Another facet to both the *makeC++SharedLib* tool and the new *-qmkshrobj* option -- which is considered a convenience -- is that an export list can be automatically constructed and saved. For the *-qmkshrobj* option, as long as the -bexpall or -bE: options are *not* specified on the command line, the exports file can be saved with the *-qexpfile=filename* option. The export list will include every symbol defined within the specified object code. As many symbols are generated ad hoc by the compiler, it is considered safest to take advantage of the ability to use the automatically generated exports file. Also, the *-bexpall* linker option is designed to export every defined symbol *except* those symbols beginning with an underscore ("_"). As the VisualAge product relies heavily upon mangled and compiler-created symbols that do happen to begin with an underscore, it is far more likely that using an automatically generated export list will result in a correct and usable module. Finally, the *makeC++SharedLib* script and the new *-qmkshrobj* option can be used to combine object code generated from any language: additional steps related to name-mangling, are only performed for C++ code.

C++-based applications that take advantage of the run-time linking function must be carefully considered. It is important to note that run-time linking on AIX is done before the static constructors are called. If a module's init() function depends upon the existence of a configured C++ system, it will fail. load() and friends can be called, but C++ will not be available. This problem can be fixed by using the C++ constructors to handle initialization.

When building C++-based modules, an additional argument is required to indicate priorities. When using the *makeC++SharedLib* script, priority is indicated with the argument: *-p#,* where # is the priority number. When using the *-qmkshrobj* compiler option, the priority is indicated as such: *-qmkshrobj=#* where # is the priority number. This priority value is used at runtime to ascertain the module initialization order. That is,

when an application that depends upon two or more C++-based modules begins execution, the C++ runtime locates the modules in use by the process, determines which contain a priority specification, sorts them according to their priority, and invokes the initialization code for each in turn. This initialization phase is the method by which static constructors are invoked at exec time or dynamic load time.

The priority value is an integer, where lower values indicate higher priority. Refer to the VisualAge C++ documentation (*not* the AIX Base Operating System documentation) for more information on the use of this option.

Just as object code that is intended to support threads must be compiled with the appropriate flavor of the compiler (e.g. XlC_r), it is also important to construct modules with the correct tool. A dynamically loadable module that is thread-safe, or *thread-enabled* should be constructed with the *makeC++SharedLib_r* or *xlC_r -qmkshrobj*. Either of these techniques will ensure that the module's dependencies are properly constructed, and that the use of this module in an application will result in proper behavior with respect to threads.

On a final note, in the same way that a C++-aware tool must be used to build loadable modules containing C++ code, it is required that any application that will take advantage of C++-based modules should be built with the C++ compiler. Thus, final program construction should be accomplished with either the xlC or xlC_r command, depending upon whether thread support is necessary.

Linking back level shared libraries
Applications often link with third party shared libraries or dynamically load them at run-time. If one or more of the shared libraries used by the application was built using an older compiler (such as CSet 3.6.6), while the application itself was built using the later version VisualAge C/C++ V 5.0, there wouldn't be any problems with C or Fortran programs. However, if both applications as well as the third party library are written in C++, the name mangling differences between the two compiler version would make it difficult to reference symbols. In that case it is recommended that every library that the program uses be built with the same version of the compiler as the main program.

There is a compiler option called *–qcompat* that could be used to build the main program, in case it is not possible to recompile third party libraries. This option ensures proper symbol references between third party shared libraries and the main program. However, this option is not useful if advanced C++ features specific to VAC 5.0 are used by the main program.

Linking third-party shared libraries
Applications are commonly built with free development tools such as the GNU tools. If the application is written in C, then libraries built using the native compiler and non-native compilers can be freely used. For C++ applications, due to name mangling differences between native and non-native compilers, it is not a good idea to mix object

code built with both compilers. As long as the entire C++ application uses object code built with one or the other, but not both compilers, symbols will be resolved properly.

If the third party libraries, or some of the application libraries were built using the GNU compiler, the following recommendations should be followed when building applications on AIX:

- Use of special GNU compile options, such as "Code Generation Options" or other switches which affect object creation should be kept to a minimum. Many of these remain untested with the native AIX linker and are better suited for use with the GNU linker. Compiler flags, which alter how system header files are implemented, can be safely used though.

- GNU compiler command g++ should be used for C++ code, and gcc for C (non C++ code).

- The native linker should be invoked through GNU using gcc. Some internal GNU flags can be passed to the linker by way of the "collect2" GNU interface.

- AIX linker flags such as "maxdata" and "bigtoc" can be passed to the linker using the GNU –Xlinker parameter. For example, to pass the –bbigtoc specification to the AIX linker, the following would be added to the gcc command line:
  ```
  -Xlinker -bbigtoc
  ```

- Name mangling between GNU and the VAC 5.0 or the xlC 3.6.6 compilers, is incompatible. Therefore any C++ interfaces created using both the GNU and the IBM compilers, will result in unresolved symbols.

- Certain alignment rules may differ between the IBM compilers and GNU. Specifically, variables at the beginning of a structure, which are smaller in word size than variables at the end of a structure, will generate inconsistent alignment between the two compilers. The IBM compilers default to the "power" alignment rules, whereas GNU uses the "natural" alignment rules. This difference can be resolved by setting the "align" flag to "natural" on the IBM compiler command line such as:
  ```
     xlc -c -qalign=natural mysource.c
  ```

- The GNU compiler generates special codes to identify template class name definitions, which are referenced by objects using the template classes. Normally, these templates must be explicitly instantiated within the source scope where they are being used. The GNU compiler provides the "-fexternal-templates" directive, as well as documentation regarding the use of "#pragma interface" and "#pragma implementation" to provide this capability. The IBM native compiler does not recognize these specifications passed to the compiled object, and thus generates unresolved errors even though the template class definitions exist.

The work-around is to explicitly instantiate the templates, wherever they are used.

Linking Java programs with native C/C++ libraries using JNI

Java has become a very popular language and for good reason – it's extremely portable. However, Java doesn't provide all of the capabilities and (sometimes) performance that a native language like C or C++ provides. Java addressed this by allowing programmers to either access native code from within a Java program, or have a native program access Java code.

The Java Native Interface (JNI) provides the mechanism by which native shared libraries can be accessed from Java programs. The native shared libraries can have either .a or .so extensions.

For linking JNI libraries to JVM 1.2.2 or higher, it is required that the linker option "-bM:UR" be used. Hence any JNI executable program, including third party applications built for previous versions of the JVM will need to be re-linked with this option in order to work with JVM 1.2.2 or higher.

For JNI programs, the native libraries must have all their undefined symbols accounted for prior to loading via export/import lists. In other words, there must not be any ".." imports in the dump output of the native libraries. Therefore no dependence on the run-time linker to resolve undefined symbols in the native libraries during Java run-time is supported.

To access Java code from a C/C++ application, the application must instantiate a Java Virtual Machine (JVM). Instantiating a JVM requires certain arguments that control the amount of heap space JVM will create at invocation, and the maximum heap space the JVM is permitted to use. By invoking a JVM, the application now consumes additional resource, which means that additional data segments might be required.

For example, consider an application that currently links with 2 data segments (via -bmaxdata flag) and invokes a JVM from within the application. One of the arguments to the JVM defines how much virtual memory (heap) the JVM should start with and how much virtual memory (heap) the JVM can use in total. The memory used by the JVM is in addition to the memory used by the application. If prior to invoking the JVM, the application consumed all of the 2 data segments, then adding the invocation of a JVM from within the application will push the application's memory requirements beyond the application's capabilities. To resolve this problem, calculate the application's memory requirements and add to it the JVM's requirements. As a general rule, an application that invokes the JVM should be linked with at least the same number of data segments as the java command. The Java command is linked with 5 data segments.

In general, if a JNI interface is being used, then there should not be any need to reference any symbol directly from the Java run-time library libjava.a. This is the preferred method since linking and installing the application software is easier, and it remains more portable across platforms.

Any updates to the JNI interface and linking mechanisms are distributed in the README file that installs in the root directory of the Java installation on AIX. Some examples of JNI applications are also generally distributed in the same way.

Linking with duplicate symbol definitions

The AIX linker differs from other systems when dealing with duplicate symbols.
On other systems, symbol definitions are not used until they are referenced, therefore the command line ordering of object code, archives, and other system libraries is critical. Suppose two distinct object files are passed to the linker, and both object files contain a definition for the symbol *foo*. Since object files are the most granular unit handled by the link-editor, the linker is incapable of determining a sensible solution to the duplicated symbol. The second object file encountered can't be dismissed either (as it may contain other useful/required symbols). Thus, the linker has an unresolvable symbol conflict and will emit an error message and exit. Hence the dreaded "duplicate symbol" problem.

The AIX linker, on the other hand, will collect all symbols from all input files specified on the command line, organized in any manner. The linker does consider the order in which it discovers symbols; the first definition found is the one that will be used when resolving all references within the module being constructed. Thus, for the most part, duplicate symbol definitions are harmless.

These warnings can be annoying though, and in the case of C++ development with its template instantiations, constructors and destructors, etc, it is possible that the compiler itself may generate duplicate instances of certain symbols. The actual program linkage will therefore produce warning messages about symbols over which the developer has no control. In this case it is best if the developer can simply ignore the symbols, once he is satisfied that the duplicates do not reflect any problem with the construction of the application.

There is a command line option though that can be used to silence certain levels of warning messages from the linker. Note that *all* duplicate symbol warnings are emitted at a "halt" level of 4. Since the linker by default emits messages at this level but does not stop working, a command line option *–bh:4* can be used to prevent the linker from emitting these messages. This option should be used only after the developer has confirmed that no real symbol conflict issue was hiding behind any of the warnings.

Linking with the –L Flag

The LIBPATH environment variable is a colon-separated list of directory paths, with the same syntax as the PATH environment variable and indicates the search path for libraries. It has the same function as the LD_LIBRARY_PATH environment variable on SystemV. If LIBPATH is defined when running the linker, its value is used as the default library path information. Otherwise, the default library path (/usr/lib:/lib) is used. If no -L flags are specified, nor is the *-blibpath* option, the default library path information is written in the loader section of the output file. The -L flag in the *xlc* and *ld* command line adds the specified directory to the library search path, which is then saved in the loader

section of the program. If LIBPATH is defined when a program is executed, the directories listed in the LIBPATH are used to search for dependent modules. Next, directories listed in the library path information in the loader section of the program are searched. The loader looks for the filename (given the library search path) and checks the dependents of it, if found. If there is more than one candidate, then the first instance found in the search path is used as long as it and its dependents satisfy the search criteria. Once a match is found, the loader stops looking. The LIBPATH variable can be used to specify a set of dynamic search paths that *dlopen()* searches for the named module. The running application also contains a set of library search paths that were specified when the application was linked. These paths are searched after any paths found in LIBPATH. The *setenv()* function can also be used to modify the value of LIBPATH during execution.

The binder option *-blibpath* forces the path name into the loader section, and overrides the -L path setting which goes in the loader section by default if the -l library path is not specified. This is used in build environments with NFS mounts. The *-bnolibpath* overrides both -L and *-blibpath* options.

For more information on LIBPATH, the -L and -l flags, see ***Appendix D***.

Linking with Threaded Libraries

With the introduction of AIX 4.3, all system libraries are now inherently thread-enabled. libc contains a mechanism whereby it can detect whether the pthreads environment must be enabled, and will perform all initialization required to support pthreads at program start-up, *as well as at the time that libpthreads is introduced into a running process via dynamic loading*. This means that a program can be constructed in a manner that does not take advantage of threads, but may become a threaded program due to a module dependency hierarchy that incorporates the pthreads library. It is advantageous to consider support for threads when an application is built.

As thread support is now integrated into the system libraries, there is really no distinction between building a threaded versus single-threaded application. Note, though, that there is a distinction between the flavors of the VisualAge compilers: when compiling code that is intended to support threads, the "_r" version of the compiler should be used. This is a convenience for the user, in that the compiler will be invoked in a manner that handles many of the details necessary for correctly building thread-enabled code. However, if an application is not designed to be threaded, it is important to keep in mind that the introduction of a dependent module via dynamic loading may result in the process "going multi-threaded" at any time.

As an example, suppose a non-threaded application dynamically loads and unloads a third-party module. If it changes its implementation and becomes threaded, it may suddenly begin depending upon libpthreads. If libpthreads is then dynamically unloaded from the running process, the system libraries will not "reset" themselves to a non-

threaded state, thus causing problems. Therefore, any application that must support the possibility of becoming multi-threaded during execution, should be built as a threaded application, and incorporate a dependency on libpthreads directly into the application. This technique ensures that the program will never undergo a failure due to a dynamic load and unload of libpthreads module.

Another benefit to the combined thread-safe libraries in AIX 4.3 is the elimination of a separate library search directory: /usr/lib/threads. It is no longer necessary to specify this path when building applications or modules (although the compiler, which supports multiple versions of AIX, continues to incorporate this path into its configuration file). All references to libc.a and libc_r.a, through /lib, /usr/lib, or /usr/lib/threads, refer to a single instance of libc.a on the system.

For building shared libraries written in C++, there is a thread-safe flavor of makeC++SharedLib, namely makeC++SharedLib_r that can be used to ensure proper threading support. If using the compiler option *-qmkshrobj* to link C++ files, the compiler will depend on the compiler binary (xlC or xlC_r) to determine whether or not to link in threads libraries. The dump command can be used to look at the loader header (the -Hv option) to ensure that the library search path and dependencies are correct.

Linking With Fortran Libraries

To create a shared object containing an unnamed Fortran common block and to link it with a main program that uses the same common block, the common block must be exported from the shared object and the program linked with the run-time linker. The common block can be exported by listing #BLNK_COM along with the other functions. (#BLNK_COM is the name internally used by the xlf 3.2 compiler for an unnamed common block.) For example, to link the shared object with the command:

```
ld -o shr.o xxx.o -bM:SRE -lxlf -lm -lc -bE:xxx.exp
```

The shared object can be put in an archive with the command:

```
ar cr libxxx.a shr.o
```

Then, when the main program is linked, use the *-brtl* option:

```
xlf -o program prog.o -lxxx -brtl
```

The unnamed common block will be exported automatically from the main program. When the program runs, the run-time linker will rebind all references to the unnamed common block in shr.o to the unnamed common block in the main program.

Linking Programs To Support Large Memory Usage

Some programs need larger data areas than allowed by the default address-space model. The linker itself can also run out of memory when trying to link very large files. When this happens, linking fails with the following error:

```
ld: 0711-101 fatal error, allocation of bytes fail in routine
initsymtab_info. There is not enough memory.
```

Application programs which require a larger data area can use the large address-space model. First try increasing the paging space, and setting limits to -1 in /etc/security/limits (the user has to log out and log back in to enable the ulimit changes). If that does not help, make the application into a "big-data" program so that it has access to more than one memory segment.

On 32-bit machines, the system hardware divides the 32-bit virtual address space into 16, 256M independent segments, each addressed by a separate segment register. Segment 0, 1, 2, 13, and 15 are reserved for kernel text, user text, process private data, shared library text, and shared library data respectively. Thus, a total of 10 segments are available for the process data.

On 32-bit machines, the system places user data and the user stack within a single segment (segment 2) by default. The combined maximum amount of stack and data is slightly less than 256M. This size is adequate for most applications. However, certain applications require large initialized or uninitialized data areas in the data section of a program. Large data areas can be created dynamically with the *malloc()*, *brk()*, or the *sbrk()* subroutines. To allow a program to use the large address-space model the *-bmaxdata* flag must be specified in the link line of *xlc* or *ld*. For example, to link a program that will have a maximum of 8 segments reserved for it, the following command line can be used:

```
xlc sample.o -bmaxdata:0x80000000
```

Here 0x80000000 refers to 8 segments (8 x 256M bytes) that will be reserved for the program's data and heap.

Depending on the number of memory segments specified with the *-bmaxdata* flag, the machine should have a total virtual memory = 256 x (x = # of segments)M available for use by the program. For information on another method to change the number of data segments - that does not require rebuilding the application - see the section below on the LDR_CNTRL environment variable and its MAXDATA option.

With the advent of 64-bit process support in AIX 4.3 and AIX 5L, the available address space of a 64-bit process is now much larger. This means that a single process has the potential for being much larger, utilizing more data, and dynamically loading more modules. Behaviorally, an application should not notice any differences with respect to program launch or dynamic loading, but this is predicated upon the idea that the application makes no assumptions about where modules exist within the address space.

Please see Appendix C for more information on the differences between 32-bit and 64-bit address spaces.

Linking/Loading libraries over NFS

Linking applications across NFS mounted file systems is very time consuming. The best and quickest method is to link your application on a local file system (*i.e.* the server). The performance issue with linking across NFS is that each library must be read completely prior to the linker/loader resolving all of the symbols. Writing out the executable across NFS is also time consuming. If executables must be stored on an NFS mounted file system, it is often better to link the application locally and move the file over to the NFS file system.

Often, the application runs on the client while the executables and shared modules are stored on the server. Since it is not possible to lock a module on the remote file system, AIX tries to maintain the reliability of the running process by copying modules to the client's paging space. While the likelihood of a module changing from under a running process is minimal, this behavior avoids the potential of a process abnormally terminating due to an NFS client and server losing synchronization with respect to a given file. If an application module is updated on the server, and the inode does not change, then the kernel believes that the existing copy is current, and therefore will not pick up the new file. In this case a *slibclean* (as root) should be performed on the client machine to pick up the new module on the server. This is only effective if the module in question is no longer in use by any running process on the client.

Loading Shared Libraries for Faster Development Performance
Shared libraries are loaded into segment 13 by default. Once loaded, they stay in memory until a system reboot or the *slibclean* command is executed. *slibclean* cleans out those shared libraries which are currently not being used by any process. If any currently loaded shared library changes on disk, that modified library does not load into memory to replace the old one just because it has changed on disk. It will only replace the previously loaded one if its inode changes, or if the previous one is cleaned out after a *slibclean* command or a reboot. However, a *slibclean* or a reboot cleans all the loaded shared libraries from the memory and hence all shared libraries have to be reloaded when the application is started up again.

In the customer's environment the shared libraries do not change frequently. When they do change on disk (due to application version updates) a *slibclean* command before the next application start-up is acceptable. In the development environment each developer works on his or her part of the code and modifies the associated shared libraries only while the rest of the application's shared libraries remain unchanged. Executing the *slibclean* command before testing every change becomes inefficient because it purges all of the application shared libraries from memory each time. In other words, a more efficient mechanism would be to somehow remove the developer's libraries while leaving all the other shared libraries in memory.

This problem has an easy fix. Simply change the permissions of the shared library under development to 550 on the developer's local disk until the file is final, after which it can be set back to 555 (or the default) and placed in the main source tree as usual. The file permission of 550 limits the read/write capability to the "group" and therefore loads the library into the process' private segment (volatile private space) instead of the shared memory segment. On program exit, while the contents of the shared memory segment remain intact, the contents of the process private segment are purged from memory and loaded fresh every time. The modified files are loaded fresh, but the other files in the process private segment might still remain cached in memory even after the program exits. In this way each program load will be able to reuse the contents of the shared memory segment while reloading the modified files in the process private segment.

To further illustrate this point, consider a developer who works with two shared libraries: libmath1.so and libmath2.so. All of her other application shared libraries are installed on an NFS server. The developer makes some changes to the two libraries. To try out the new code, do a *slibclean* to remove the previous versions of libmath1.so and libmath2.so from memory. This also cleans out all 200M of the application libraries from the memory. When re-launching the application, it has to pull in new versions of libmath1.so and libmath2.so, along with all of the remaining 200M of application shared libraries over NFS, causing considerable delay in program start-up. This delay can be avoided if the permissions of the libraries libmath1.so and libmath2.so are made such that read-other permission is removed (*i.e.* 550). This causes the two development libraries to be removed from memory after every program exit, without executing *slibclean*. This trick greatly improves application development performance.

LDR_CNTRL Environment variable

This environment variable was created to allow end-users and application developers to modify the behavior of the loader on a per-application basis. This variable can be set and exported when invoking an application, without affecting overall system behavior or producing unexpected and undesirable results from other applications. To display the value of the environment variable, type:

```
echo $LDR_CNTRL
```

To change the value of the LDR_CNTRL environment variable:

```
LDR_CNTRL={option1 | option2 / …}
 export LDR_CNTRL
```

Change takes effect immediately in the shell, and the change is effective until logging out of this shell. LDR_CNTRL is evaluated once at exec-time, and never again during the life of the process. Thus it is possible to set it within the shell (e.g. as part of a shell script) where it will not affect the currently executing shell, but only processes created by that shell.

Possible options are:

PREREAD_SHLIB	Specifying the PREREAD_SHLIB option will cause entire libraries to be read as soon as they are accessed. With VMM readahead tuned, a library can be read in from disk and be cached in memory by the time the program starts to access its pages. While this method can use more memory (modules are usually memory-mapped and accessed on a per-page basis, as necessary), it can enhance performance of programs that load many shared libraries at once.
LOADPUBLIC	Specifying the LOADPUBLIC option directs the system loader to load all modules (with read-other permission enabled) into the global shared library segment. If a module cannot be loaded publicly into the global shared library segment then it is loaded privately for the application.
IGNOREUNLOAD	The loader ignores the unload() system call for the modules. As a side effect of this option, you can end up with two different data instances of the same module.
USERREGS	Causes all general purpose registers to be saved on the stack when a system call is made, providing the user process the ability to "scan" memory and locate all references within that process to specific memory locations. An example of this would be a garbage collection scheme.
MAXDATA	This variable implements the same mechanism as the -bmaxdata linker option, but without the need to rebuild an application. Setting MAXDATA=0 disables this option.
DSA	AIX 5L introduces a new feature called "Dynamic Segment Allocation". The function provides a maxdata program the ability to acquire additional memory segments for the process heap as they are required. The maxdata value therefore specifies the maximum amount of memory the process will ever be able to acquire. The DSA option is valid only for 32-bit applications.

Common Problems

1. Programs will suffer performance loss if they depend on the run-time linker to resolve all the undefined symbols in the shared libraries.
   Solution:
   For the best possible linking and loading performance, the shared libraries should be built with all needed symbols defined in it. For the remaining symbols, use AIX's capability to provide knowledge of dependencies between modules via the export and import lists.

2. On AIX, programs could suffer performance loss by having large shared libraries with lots of cross-references and unnecessary deferred imports.
   Solution:
   The architecture of the application should be such that the modules are built with as many symbols resolved at link-time as possible.

3. Programs can inadvertently cause loading of two copies of the same module, each due to a different LIBPATH.
   Solution:
   Check the linking process to ensure that no module is getting loaded twice due to different LIBPATHs. Use the *genkld* command for a listing of the currently loaded libraries to check for multiple loads.

4. The *nm* command does not provide the symbol table look-up information.
   Solution:
   On AIX, the dump command should be used to look at the loader-section symbol table information. Please see Example 11 in the Examples section for usage of the dump command.

5. Applications may be including the -H512 -T512 flags in the AIX 4.3.3 and AIX 5L link line for alignment, just like they did on earlier versions of AIX.
   Solution:
   There is no need to include the -H512 -T512 flags in any release of AIX version after 4.2. Alignment is taken care of by the linker.

6. Loading one library causes several other libraries to load also.
   Solution:
   Programs that do not have a well-defined interface may be sharing a lot of global variables among their libraries. This makes them dependents of one another with the result that when one is loaded, it causes loading of all its dependent libraries. For best performance, create self-contained shared libraries, with minimum dependencies between the libraries.

7. Modules will not execute properly if all the system libraries that it needs are not linked in.

Solution:
Unless the needed functions are part of shared members of the archive system library, all required system libraries should always be linked in while building the shared module.

8. Modules will not execute properly if the system libraries are linked in with their full path name on the command line for the *makeC++SharedLib* script.
   Solution:
   The command line for this script should always contain the system libraries with the *-l* flag. Full path names cause all of the system library's symbols to become part of the shared library being built, and will be exported from there instead of just being available to it for reference. The *-l* simply references the library during link-time, whereas specifying the library by name tells the tool to munch the archive. This can lead to unpredictable results.

9. Programs will not execute properly if the shared libraries are run-time link enabled while the executable is not.
   Solution:
   If an executable is linked in with any library that does run-time linking to resolve symbols, the executable must also be built to support run-time linking. This is done by specifying the *-brtl* option in the executable's link line.

10. Shared libraries that are linked with the *-brtl* option (to enable run-time linking) will not build properly.
    Solution:
    Shared libraries cannot be built with the *-brtl* option and should always be built with the *-G* option to enable run-time linking. The executable should be built with the *-brtl* option. The *-G* option enables run-time linking but removes the reference to the run-time linker libraries. The *-brtl* option enables run-time linking and allows the reference to the run-time libraries to exist. Only the main module can reference the run-time libraries directly. Thus, only the main module should be built with the *-brtl* option, while the shared modules/libraries should always be built with the *-G* option (or with *-brtl -nortllib* options together).

11. All instances of *dlopen()* in a library do not close with a single *dlclose()* of the handle from any one of the *dlopen()* calls.
    Solution:
    On AIX, the handle from *dlopen()* is specific to an instance of *dlopen().* The *dlclose()* call only closes the instance of the object to which the handle refers. This issue has been discussed at POSIX and the behavior on AIX is compliant with it. Success on other platforms with *dlclose()* of another instance of a shared object with a new handle is not necessarily standards-based behavior.

Tips and Tricks

1. Check the level of your installed filesets by using the lslpp command. For example, to determine the level of the binder software, type:
   ```
   lslpp -L |grep bind_cmds
   ```

   The output will tell the full name of the fileset and its installed version. Check for later versions of any fileset. The latest versions of any fileset on AIX can be downloaded from the site: http://aix.boulder.ibm.com . Then choose: Servers → IBM eServer pSeries → Fixes, drivers, updates, tools → then open 'Tools' under pSeries with AIX → select AIX FixDist Motif-based application → AIX FixDist Application. Once running the application, select the AIX version, and enter the search option. Then key in a search string such as bos.rte.bind_cmds for the binder, or the bos.rte.libc for the libc run-time environment fileset, select "fileset" to search for these filesets. Select the latest fileset for your AIX version, select the AIX version, then select "Get Fix Package." This will download the filesets in to the specified directory. To install these filesets, put them in a separate directory, then as root type smit or smitty, and select "Software Maintenance and Installation." Select "Custom Install" or "Install by fileset/package name" (the exact word in the selection menu may differ between different AIX levels, but the idea is to install fixes by name and not by pre-packaged bundle). Then follow the instructions for further menu selections. Check for the installed version again by using the lslpp command as shown above.

2. Use the following commands to get more information about the state of the system:
   To determine if a certain APAR is installed on the system:
   ```
   instfix -ik <IX----->
   ```
   To determine is a certain PTF is installed on the system:
   ```
   instfix -Bl <U------>
   ```

3. Use the "dump" command to look at the symbol table information.
   ```
   dump -HTv <module name>
   ```

4. Use the **map** subcommand of **dbx** to see what modules are actually loaded as follows:
   ```
   > dbx /bin/ls
   Type 'help' for help
   reading symbolic information ....warning:
   no source compiled with -g
   (dbx) map

   Entry 1:
   Object name:     ls
   Text origin:     0x10000000
   Text length:     0x46f8
   Text origin:    0x20000968
   Data length:     0x5e0
   File descriptor: 0x7

   Entry 2:
   Object name:     /usr/lib/libc.a
   member name:     meth.o
   ```

...

5. Diagnose problems in case of a linking failure by using the binder options as follows:
 -bloadmap:<file>
 -bmap:<file>

 The *-bmap:<file>* option is used to generate an address map. Unresolved symbols are listed at the top of the file, followed by imported symbols. The *-bloadmap: <file>* option is used to generate the linker log file. It includes information on all of the arguments passed to the linker along with the shared objects being read and the number of symbols being imported. If an unresolved symbol is found, the log file produced by the *-bloadmap* option lists the object file or shared object that references the symbol. In the case of using libraries supplied by a third party product, you can then search the other libraries supplied by the product in an effort to determine which one defines the unresolved symbol. This is particularly useful when dealing with database products that supply many tens of libraries for use in application development.

6. Clean the memory of shared libraries by using the *slibclean* system command. Once a shared object is loaded, it remains in memory even after the exit of the application that loaded it. The *slibclean* command flushes unused objects from the shared text segment. This is useful when repeatedly building and testing new shared objects.

7. To test whether or not a module was built "shared", run the following dump command:
 `dump -ov <library name>`
 If the "SHROBJ" is set in the FLAGS line, then the module was built as a shared object. If the library name specified in the above command is an archive library, then the loader information, along with the loader flags will be printed for each archive member.

8. If shared libraries are built with other shared libraries linked in, the size of the shared library will be larger, though loading/execution will be faster. On the other hand, if the same shared library is linked in with an import file, the size of the shared library will be smaller, though the loading/execution will be slower due to dynamic binding of the symbols.

9. Use -berok (or deferred imports) judiciously. Resolving deferred imports is slow. To see the dependent modules of a given module, use the command: `dump –Hv <module_name>`.

10. To see what modules are actually loaded by an executable you can use the *map* command in *dbx*, or directly execute the *genkld* system command for a list of all currently loaded shared libraries.

11. Put the symbols imported from the main program in a separate object, and link both the main program and other referencing objects with the new shared object.

Importing symbols from the main program affects performance.

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

13. Please note that the system libraries such as libc.a are shipped "pre-bound" and not as raw archives of object files. Therefore references among the library routines are already resolved. Thus it is strongly discouraged to create an object file from a system library.

14. To link several archive libraries into a shared object, use:
    `ld -r -o tmp.o  -lfoo -lbar …`
    then use tmp.o as a command line input to build the shared object:
    `ld  -o bigsharedobject.so  tmp.o  $(LD_FLAGS)`

15. At link-time for every shared object, all needed system libraries should be linked in, including (but not limited to) -lc –lC -lm -lX11 etc.

16. The option: –blibpath should be used when linking libraries to specify the library paths which must be searched for imports when the library is loaded. If application directory paths are to be added, they should be placed at the beginning, right after the "-blibpath:" string. One reason for this is that it ensures no reference to absolute paths is being made, which might not be present on another machine. The dump command: `dump –H <library name>` can be used to check the settings for the –blibpath string for any module.

17. If large parts of the shared libraries are paged in all at once due to C++ calls, or due to a lot of references between libraries, it might 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 in one big chunk. Please refer to the LDR_CNTRL section above for more information on using this environment variable.

18. If a call is made to an undefined symbol during run-time, and the application terminated with a segmentation fault, use the following method to find the unresolved function. Start the debugger from the same directory as the one in which the core file resides. The debugger will use this core file to debug the program. On the debugger's command prompt, type "registers" for a list of values in the registers. If the "IAR" register shows a value of 0x00000000, type "t" on the debugger's command line. This will list the traceback up until the failed routine. If at the very top of the traceback, there is a routine name "glink.funcname", where "funcname" is the name of some application function, then that is the function name that is unresolved. To find out the library that this unresolved symbol is exported from, use

the “symsearch” script included in Appendix A.

19. Reduce application build-time by re-linking stable object files into a single intermediate object file. Groups of files can be pre-linked in multiple steps. For example:
```
ld -r -o abc.o a.o b.o c.o
ld -r -o mnp.o m.o n.o p.o
xlc abc.o mnp.o
```

20. Some useful system commands:
lsattr -El sys0 | grep realmem - Prints installed REAL memory on the system
lsattr -El proc0 | awk '/type/{print $2}' - Prints processor type
lsattr -E -F value -l sys0 -a modelname - Prints the system model name
lsdev -C -c processor -S Available | wc -l | tr -d ' ' - Prints number of processors
lslpp -L |grep <fileset search string> - Prints the installed level of the fileset
For example:
```
> lslpp -L | grep libpthreads
bos.rte.libpthreads       4.3.3.27    C    pthreads Library
```
oslevel - Prints the oslevel of the machine
slibclean - Cleans the memory of currently unused shared libraries
genkld - Prints currently loaded shared libraries

21. If a *dlopen()* call is unsuccessful in loading any of the dynamically loaded module's dependents, the error message complains about a file or directory not being found. In this case, first make sure that LIBPATH is set properly for the shared modules to be found, then look at the dump command output to see which modules are dependent on the *dlopen()* module, then follow the tree to figure out which module did not load successfully.

22. If errors show up during linking or loading in a version controlled application development environment, then try linking outside the version control system. If errors happen only during linking/loading under version control, then check with your version control system vendor for known linker/loader problems.

23. The shared objects shipped with AIX are not enabled for run-time linking. They can be enabled by using the *rtl_enable* command. For example if you want to re-link the program to use your own version of malloc():
Create a new instance of libc.a:
```
rtl_enable -o /usr/local/lib/libc.a /lib/libc.a
```

Now link your program:
```
xlc … mymalloc.o -L /usr/local/lib -brtl -bE:myexports
```

Here mymalloc.o defines malloc() and myexports causes malloc() to be exported from your main program. Calls to malloc() from within libc.a will now go to your program.

Examples

Example 1

This example demonstrates the use of *dynamic loading* in resolving deferred symbols. The application's main module (main.c) refers to a function defsym() that is deferred at link-time. This function is defined in the shared library, libdefsym.so that is built out of the source file defsym.c. The application dynamically loads the shared object and calls the function defsym().

main.c

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

extern void defsym(void);

int
main( int argc, char** argv )
{
	void* handle;
	void* symptr;

	/*** load the shared object containing the definition of the ***/
	/*** function defsym.                                        ***/

	if ( NULL == ( handle = dlopen( "./libdefsym.so", RTLD_NOW ) ) ){
		perror( dlerror() );
		exit( 1 );
	}

	/*** The deferred resolution of the func defsym should have ***/
	/*** been resolved by the abve load.                        ***/

	defsym();

	/*** Unload the previously loaded shared object.            ***/

	if ( 0 != ( dlclose( handle ) ) ) {
		perror( dlerror() );
		exit( 1 );
	}

	exit( 0 );
}
```

defsym.c

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

void
defsym(){
```

```
	printf( "defsym called.\n" );
}
```

defsym.imp

```
#!
defsym
```

Makefile

```
all:		main libdefsym.so
main:		main.o
		xlc -o main -bI:defsym.imp main.o

main.o:			main.c
		xlc -c main.c

libdefsym.so:		defsym.o
		ld -G -bnoentry -bexpall -o libdefsym.so defsym.o -lc

defsym.o:	defsym.c
		xlc -c defsym.c

.PHONY:			clean
clean:
		rm *.o *.so main
```

Example 2

This example is the same as above except that the main module of the application does not explicitly refer to the function defsym(). It dynamically loads the shared library libdefsym.so and then obtains the address of the function.

main.c

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

typedef void (*FP)(void);

int
main( int argc, char** argv )
{
	void* handle;
	void* symptr;
	FP    fp;

	/*** Load the shared object containing the definition of the ***/
	/*** function - defsym().                                     ***/

	if ( NULL == ( handle = dlopen( "./libdefsym.so", RTLD_NOW ) ) ){
		perror( dlerror() );
```

```
		exit( 1 );
	}

	/**** Obtain the address of the function defsym.		***/

	if ( NULL == ( symptr = dlsym( handle, "defsym"  ) ) ) {
		if ( 0 == errno ) {
			fprintf( stderr, "Symbol defsym not found. Exiting\n"
);
			exit( 1 );
		}
		perror( dlerror() );
		exit( 1 );
	}

	fp = ( FP )symptr;

	/*** Call the function via the obtained function address.	***/

	fp();

	/*** Unload the previously loaded shared object.		***/

	if ( 0 != ( dlclose( handle ) ) ) {
		perror( dlerror() );
		exit( 1 );
	}

	exit( 0 );
}
```

defsym.c

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

void
defsym()
{
	printf( "defsym called.\n" );
}
```

Makefile

```
all:		main libdefsym.so
main:		main.o
		xlc -o main main.o

main.o:			main.c
		xlc -c main.c

libdefsym.so:		defsym.o
		ld -G -bnoentry -bexpall -o libdefsym.so defsym.o -lc

defsym.o:	defsym.c
```

```
            xlc -c defsym.c

.PHONY:           clean
clean:
            rm *.o *.so main
```

Example 3
This example extends Example 1 by adding a second shared object, libcalldefsym.so containing the definition of a new function calldefsym() that calls the function defsym() in the shared object libdefsym.so. The application dynamically loads libdefsym.so and libcalldefsym.so in that order. When the second library is loaded, its reference to defsym is resolved by the definition in the previously loaded libdefsym.so. Note the need to build the main application so that it is enabled for runtime linking.

main.c

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


typedef void ( *FP )( void );

int
main( int argc, char** argv )
{
      void* handle;
      void* handle1;
      void* symptr;
      FP    fp;

      /*** Load the shared object libdefsym.so containing the      ***/
      /*** definition of the unction - defsym.                     ***/

      if ( NULL == ( handle = dlopen( "./libdefsym.so", RTLD_NOW |
RTLD_GLOBAL ) ) ) {
            perror( dlerror() );
            exit( 1 );
      }

      /*** Load the shared object libcalldefsym.so containing the  ***/
      /*** definition of the function - calldefsym.                ***/

      if ( NULL == ( handle1 = dlopen( "./libcalldefsym.so", RTLD_NOW )
) ) {
            perror( dlerror() );
            exit( 1 );
      }

      /*** Obtain the address to the function - calldefsym.        ***/

      if ( NULL == ( symptr = dlsym( handle1, "calldefsym"  ) ) ) {
            if ( 0 == errno ) {
```

```
                    fprintf( stderr, "Symbol calldefsym not found.
Exiting\n" );
                    exit( 1 );
              }
              perror( dlerror() );
              exit( 1 );
       }

       fp = ( FP )symptr;

       /*** Call calldefsym which in turn calls defsym  ***/
       /*** via the address obtained above.             ***/

       fp();

       /*** Unload the shared library libcalldefsym.so. ***/

       if ( 0 != ( dlclose( handle1 ) ) ) {
              perror( dlerror() );
              exit( 1 );
       }

       /*** Unload the shared library libdefsym.so.     ***/

       if ( 0 != ( dlclose( handle ) ) ) {
              perror( dlerror() );
              exit( 1 );
       }

       exit( 0 );
}
```

calldefsym.c

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

extern void defsym( void );

void
calldefsym( void )
{
       printf( "Calling defsym from module calldefsym\n" );
       defsym();
}
```

defsym.c

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

void
defsym()
{
       printf( "defsym called.\n" );
```

```
}
```

Makefile

```
all:        main libdefsym.so libcalldefsym.so

main:       main.o
            xlc -o main main.o -brtl

main.o:           main.c
            xlc -c main.c

libdefsym.so:     defsym.o
            ld -G -bnoentry -bexpall -o libdefsym.so defsym.o -lc

defsym.o:   defsym.c
            xlc -c defsym.c

libcalldefsym.so: calldefsym.o
            ld -G -bnoentry -bexpall -o libcalldefsym.so calldefsym.o -
lc

calldefsym.o:     calldefsym.c
            xlc -c calldefsym.c

.PHONY:           clean
clean:
            rm *.o *.so main
```

Example 4

This example shows how to build a shared library using the run-time linker. First, here are the source files for this example:

a.c

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

void a() {
    int c;
    printf( "Now in function a()\n" );
    b();
    c = 4 + 5;
}
```

b.c

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
#include <ctype.h>
#include <unistd.h>
```

```
extern void fpinit();
extern void c1();

int kk=1;

void b() {
    int c;
    printf( "Now in function b()\n" );
    c = 4 + 5;
    c1();
}
```

c1.c

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

void c1() {
    int *ptr1;
    printf( "Now in function c1()\n" );
}
```

hello.c

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

int main() {
    int *ptr1;
    int ipt = 0;

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

Now, let's build a shared library using the run-time linker:

```
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
```

This means that liba.so is: (1) being built enabled for the run-time linker (-G); (2) does not contain any entry points (-bnoentry); (3) exports all of its symbols for other shared objects to use (-bexpall); (4) uses C and C++ system libraries; and (5) has an error level set to 4 which is "warning" and is the default.

Symbol table information for liba.so can be obtained by the `dump -HTv liba.so` command:

```
                    ***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
```

The loader symbol table output above means that the symbol printf comes from the shared object shr.o, an archive member of the system library libc.a. Symbol "b" will be searched for by the run-time library at liba.so's load or exec-time. This is indicated by the ".." in the IMPid column for symbol b in the above output (recall that run-time symbol look-up is only performed if the main application is built with run-time linking enabled).

Here is how to build the final executable hello:

```
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()
```

The advantage of shared libraries is that they can be modified individually, without having to re-link the executable. That way, the next time the executable is run, it picks

up the modified shared library. With run-time linking, the export/import lists are no longer needed as long as the referenced symbol can be found in one of the linked libraries. To illustrate this point consider file b.c to have been modified to include a call to a new function c2(), and file c1.c to have been modified to include a definition for this new function:

b.c

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
#include <ctype.h>
#include <unistd.h>

extern void fpinit();
extern void c1();
int kk=1;

void b() {
    int c;
    printf( "Now in function b()\n" );
    c = 4 + 5;
    c1();
    c2(); /*** New function call! ***/
}
```

c1.c

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

void c1() {
    int *ptr1;
    printf( "Now in function c1()\n" );
}

void c2() {
    printf( "Now in function c2()\n" );
}
```

Rebuild *only* libb.so and libc1.so, and then run hello:

```
cc -c b.c -o b.o
cc -c c1.c -o c1.o
ld -o libb.so b.o -bnoentry -G -bexpall -lC -lc
ld -o libc1.so c1.o -bnoentry -G -bexpall -lC -lc

> hello


Hello World
Now in function a()
```

```
Now in function b()
Now in function c1()
Now in function c2()
```

Example 5

This example shows the use of deferred imports, and linking without the run-time linker.

Let's use files a.c and c1.c exactly as in the beginning of the previous example; and file b.c is now:

b.c

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.c>
#include <ctype.h>
#include <unistd.h>

extern void fpinit();
extern void c1();
extern void cc1();

int kk=1;

void b() {
    int c;
    printf( " Now in function b()\n" );
    c = 4 + 5;
    c1();

    if (kk == 0) {
        printf( "kk is 0\n" );
        cc1(); /*** Symbol is referenced but is never actually called
***/
        exit(1);
    }

}
```

a.imp

```
#!libb.so
b
```

b.imp

```
#!
* The following are deferred symbols
cc1
```

```
#!libc1.so
c1
```

Now if the shared libraries and the executable are built like this:

```
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 -bI:a.imp -bnoentry -bexpall -bM:SRE -lc
ld -o libb.so b.o -bI:b.imp -bnoentry -bexpall -bM:SRE -lc
ld -o libc1.so c1.o -bnoentry -bexpall -bM:SRE -lc

xlc -o hello42 liba.so libb.so libc1.so hello.o -L.
```

Note that the run-time linker was not needed in this case since the import/export lists provide for dynamic binding at run-time.

Example 6

Rebinding example:

f1.c

```
#include <stdio.h>

extern void func2();

void func1()
{
    printf( "\tinside of func1()/%s...\n", __FILE__ );
    printf( "Calling func2()...\n" );
    func2();
}
```

f2.c

```
#include <stdio.h>

extern void func3();

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

f3.c

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

```
extern void func4();

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

f4.c

```
#include <stdio.h>

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

main.c

```
#include <stdio.h>

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

main()
{
    void (*f1p)();
    extern void func1();

    f1p = func1;
    printf( "Calling func1()...\n" );
    func1();
}
```

When compiling and linking is done this way:

```
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 -bexpall
```

The following 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, and main.exp exports the symbol func4(), func4() defined by main is used.

The results of running main are:

```
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

If main.o is listed *after* libshr1.so and libsh2.so on the command line, func4() defined by f4.c is used. The results of running main built in this fashion are:

```
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()/f4.c...
```

This is an example of how functions' definitions can be rebound.

If not using run-time linking, compiling and linking is done this way:

```
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 -bE:libshr2.exp -lc -bnoentry
ld -o libshr1.so f1.o f2.o libshr2.so -L. -bE:libshr1.exp -lc -bnoentry
xlc -o main libshr1.so libshr2.so main.o -L.

ld: 0711-224 WARNING: Duplicate symbol: .func3
ld: 0711-224 WARNING: Duplicate symbol: .func4
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more
information.
```

In this case it doesn't matter if main.exp is specified in the command line to build main, the output of main is:

```
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()/f4.c...
```

Irrespective of where the main.o appears on the command line when building main.

Example 7
Run-time linking and dynamic loading of non-linked modules:

main.c

```
#include <stdio.h>

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

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

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 (*FP)(void);
```

```
void usr_routine()
{
    printf( "in usr_routine usr.c\n" );
    axs_routine();
}

void usr_preempt()
{
    void* routine;
    FP    fp;

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

axs.c

```
#include <stdio.h>

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

ext.c

```
#include <stdio.h>

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

Compile the source files and then link as follows (warnings will result):

```
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.
```

The output produced by this program is:

```
> 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
```

This example is especially interesting because several things are going on here. First, symbol usr_preempt() is defined in two modules: in libvmap.so and in libusr.so. Second, the 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, there is also an explicit dlopen() of module libext.so which is not linked to the main module.

Without the use of the run-time linker, additional code would have to be written to allow the program to find and access the desired symbol definitions.

Example 8

Lazy loading example:

Please refer to Example 4 for the source code, and Example 5 for the b.imp import list.

Lazy loading can be enabled by building the shared libraries and the executable like this:

```
ld -o libc1.so c1.o -bnoentry -bexpall -bM:SRE -lc
ld -o libb.so b.o -bnoentry -bexpall -bM:SRE -lc -bI:b.imp -blazy
ld -o liba.so a.o -bnoentry -bexpall -bM:SRE -lc -L. libb.so -blazy
xlc -o hello hello.o liba.so libb.so libc1.so -blazy -L.
```

Note the order in which the modules are built. While we don't need libc1.so to build libb.so, we can use libb.so to build liba.so and avoid the need for the a.imp import list, where a.imp lists symbol b defined by libb.so, and b.imp lists c1 defined by libc1.so, and symbol cc1 as deferred.

To confirm that the modules are really being lazy-loaded, use the map subcommand in dbx. After building the program, start hello under dbx and then type map. This will list all the modules currently loaded in memory. In the above example the modules liba.so, libb.so and libc1.so will not show up as loaded before the program execution. However, if the program was built without the *-blazy* flag all those modules would have shown up as loaded after program initialization.

Example 9

Autoloading of archive members:

main.c

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

bar.c

```
#include <stdio.h>

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

foo.c

```
#include <stdio.h>

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

barfoo.c

```
#include <stdio.h>

void 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
```

The goal in this example 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 run-time 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 run-time 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 run-time, the definition for foo() is used to resolve all references from any module. This will then override the definition provided by the module libbar.so.

The output will therefore be:

```
> main
in bar()
in foo() which is correct...
```

Example 10

This example shows how to overload the default new and delete operators for C++ by using runtime linking. The example consists of four files: main.cpp, memory.cpp, utility.cpp and Makefile. The Makefile creates two executables named: (1) works, and (2) fails. The works executable shows the proper method to overload the operators and the fails executable shows the typical mistake for overloading the operators.

utility.h

```
#ifndef _utility_h
#define _utility_h

//
// Typical boring headers
#include <iostream.h>
#include <stdio.h>

//
// Utility functions
void debugInfo( const char* string );
void debugInfo( const int num );
```

```
void debugInfo( const void* ptr );
void debugInfo( const char* string, const void* ptr );
void debugInfo( const char* string, const int num, const void* ptr );

//
// Dummy class
class dummyClass {
	public:
		dummyClass( );
		~dummyClass( );

	private:
		char *dummyInfo;
		int  *dummyPtr;
	};

#endif
```

main.cpp

```
#include "utility.h"

int appStarted = 0;
int main(int argc, char *argv[]) {

	fprintf(stderr, "Application started\n");
	appStarted = 1;

	dummyClass* dP;

	int* intArray;

	dP = new dummyClass;
	intArray = new int[4];

	delete dP;
	delete [] intArray;
}
```

memory.cpp

```
//
// Inclusion of the header <new> is recommended by
// the compiler whenever you overload new and delete
#include <new>

//
// Typical boring headers
#include <stdlib.h>
#include "utility.h"
extern int appStarted;

//
```

```
// New operator overload
void* operator new(size_t size) {
      void* ptr = malloc(size);

      if ( appStarted )
            debugInfo( "operator new", size, ptr );

      return ptr;
}

// New [] operator overload
 void *operator new[](size_t size) {
      void* ptr = malloc(size);

      if ( appStarted )
            debugInfo( "operator new[]", size, ptr );

      return ptr;
}

//
// Delete operator overload
void operator delete(void* ptr) {
      if ( appStarted )
            debugInfo( "operator delete", ptr );

      free(ptr);
}

//
// Delete [] operator overload
void operator delete[](void *ptr) {
      if ( appStarted )
            debugInfo( "operator delete[]", ptr );

      free(ptr);
}
```

utility.cpp

```
//
// Typical boring headers
#include "utility.h"

//
// Utility functions
void debugInfo( const char* string ) {
      fprintf(stderr, "%s ", string);
      }

void debugInfo( const int num ) {
      fprintf(stderr, "%d ", num);
      }

void debugInfo( const void* ptr ) {
```

```
	fprintf(stderr, "%x ", ptr);
	}

void debugInfo( ) {
	fprintf(stderr, "\n");
	}

void debugInfo( const char* string, const void* ptr ) {
	debugInfo( string );
	debugInfo( ptr );
	debugInfo( );
	}

void debugInfo( const char* string, const int num, const void* ptr ) {
	debugInfo( string );
	debugInfo( num );
	debugInfo( ptr );
	debugInfo( );
	}
//
// Dummy class
dummyClass::dummyClass() {
	fprintf(stderr,"Enter dummyClass\n");

	dummyInfo = new char [80];
	*dummyInfo = NULL;

	dummyPtr = new int;
	fprintf(stderr,"Exit dummyClass\n");
	}

dummyClass::~dummyClass() {
	fprintf(stderr,"Enter ~dummyClass\n");

	delete [] dummyInfo;
	delete dummyPtr;

	fprintf(stderr,"Exit ~dummyClass\n");
	}
```

<u>new_delete.exp</u>

```
__nw__FUl
__vn__FUl
__dl__FPv
__vd__FPv
```

<u>Makefile</u>

```
CXX = xlC
CXFLAGS = -g -c
CC  = cc
```

```
CFLAGS = -g -c
LDFLAGS = -brtl -bE:new_delete.exp
LIBFLAGS = -L./ -lutility

all: works fails

works: main.o memory.o libutility.a
	$(CXX) $(LDFLAGS) $(LIBFLAGS) -o works main.o memory.o

fails: main.o memory.o libutility.a
	$(CXX) $(LIBFLAGS) -o fails main.o memory.o

main.o: main.cpp
	$(CXX) $(CXFLAGS) -o main.o main.cpp

memory.o: memory.cpp
	$(CXX) $(CXFLAGS) -o memory.o memory.cpp

utility.o: utility.cpp
	$(CXX) $(CXFLAGS) -o utility.o utility.cpp

libutility.a: utility.o
	makeC++SharedLib -p 0 -o libutility.o utility.o
	ar rv libutility.a libutility.o

clean:
	rm -f works fails libutility.a libutility.o memory.o main.o
utility.o
```

Example 11

Examples of using the *dump* command:

`dump -HTv <library name> -` To view symbol table information
`dump -Hv <library name> -` To see the dependent modules of a given module
`dump -ov <library name> -` To test if a module was built "shared" or not. Look if "SHROBJ" is set in the 'Flags' line.

Appendix A

Listing of the "symsearch" script:

```
#!/bin/ksh
# Try and find all the references to $SYMBOL in any lib
#

usage()
{
printf "\nUsage: \n%s -s SYMBOL -i|-e [-v] [-o]\n" `basename $0`
printf "-i means symbol is being imported\n"
printf "-e means symbol is being exported\n"
printf "-v means verbose mode\n\n"
printf "-o means search object files not libs\n\n"
printf "%s will search the directories in \$DIR, if set, " `basename $0`
printf "otherwise it will search the current dir and those in \$LIBPATH\n"
exit 1
}


# Default the search symbol to one being IMPorted
IMPEXP=IMP
DUMPFLAGS="-Tv"
LIBPATHDIRS=`echo $LIBPATH | tr ":" " "`
PWD=`pwd`
VERBOSE="OFF"


while getopts :voies: value
do
      case $value in
      i) IMPEXP=IMP;;
      e) IMPEXP=EXP;;
      v) VERBOSE="ON";;
      s) SYMBOL="$OPTARG";;
      o) OBJECTS=1;;
      ?) usage
      esac
done

if [[ $SYMBOL = "" ]]; then
   usage
fi

# Make the necessary IA-64 conversions, if needed
#
if [[ `uname -m` = "ia64" ]]; then

   if [[ $VERBOSE = "ON" ]]; then
      printf " Switching to IA-64 values. \n"
   fi

   if [[ $IMPEXP = "IMP" ]]; then
      IMPEXP=GLOB
   else
```

```
        IMPEXP=LOCL
    fi

    DUMPFLAGS="-tv"
fi

# Set DIR to default value if it did not exist
if [[ $DIR = "" ]]; then
        if [[ $VERBOSE = "ON" ]]; then
                printf " Setting DIR to default value.\n"
        fi
        DIR="$PWD $PS_HOME/bin $LIBPATHDIRS"
fi

if [[ $VERBOSE = "ON" ]]; then
        printf  "Checking for $SYMBOL in the .a and .so files in these
directories:.\n"
        echo $DIR | tr " " "\n"
        printf "\n\n"
fi

if [[ $OBJECTS = 1 ]]; then

for i in `find $DIR \( -name "*.o" \) -prune -print 2>/dev/null`
do
        if [[ $VERBOSE = "ON" ]]; then
                printf "Now checking $i .\n"
        fi
        nm $i | grep $SYMBOL 2>/dev/null
        if [[ $? -eq 0 ]]; then
                printf " $SYMBOL was found in $i.\n"
        fi
done

else

for i in `find $DIR \( -name "lib*.a" -o -name "lib*.so" \) -prune -print
2>/dev/null`
do
        if [[ $VERBOSE = "ON" ]]; then
                printf "Now checking $i.\n"
        fi
        dump $DUMPFLAGS $i | grep $IMPEXP | grep $SYMBOL
        if [[ $? -eq 0 ]]; then
                printf " $SYMBOL was found in $i.\n"
        fi
done
fi
```

Appendix B

The -binitfini Linker Option:
Init/fini routines provide a generic method for module initialization and/or termination for both program exec and exit and dynamic loading and unloading. The facility allows for an arbitrary number of init or terminate functions per module, in both shared libraries and the executable. For each module, the -binitfini linker command line option specifies an optional initialization function, an optional termination function, and the priority of these functions. Multiple -binitfini options may be used. The priority value causes the linker to sort the specified functions while building the module. Lower numbers correspond to a higher priority, and certain ranges for the priority value are reserved; refer to the *ld* documentation for specifications of these reserved values.

At program exec time, the load order of the modules is determined. Initialization functions are invoked in *module load order*. This implies that init functions are called in the main executable first, and in the last loaded module last. As each module is processed, the init functions are invoked in the order specified at module construction time according to their priority values. At dynamic load time (during program execution) modules added to the address space are again processed in load order. The module loaded first will have its init functions processed first, and the last newly loaded module will be processed last.

At dynamic unload time, the set of modules to be unloaded from the process is determined. Then the module termination functions are invoked in the *last* module in this list, with the first module in the list being processed last. Note that this order is also dependent upon overall module load order, and in a casual manner can be construed as the reverse of the order in which initialization functions were processed at load time. At program exit time, since all modules will be terminated, the termination functions are processed in reverse load order, with the fini functions in the main executable being processed last.

Note that there is an interaction here with module dependencies. For example, it's possible to dynamically load several modules that all utilize a common dependent module. The loader manages dependencies between modules and will not unload a module that is still in use by the process. The termination functions within a module will not be invoked until the module is actually removed from the process address space. Furthermore, due to the behavior of deferred imports, it is possible to dynamically load a module that resolves a deferred import in some other module that is already part of the process, essentially disallowing that module to ever be unloaded until the depending module is removed from the process. Note that if the depending module was part of the process exec, then it can never be unloaded until program exit, which implies that any module upon which it depends cannot be unloaded either. Thus module dependencies and dynamic loading should be carefully considered when utilizing init/fini functions.

Appendix C

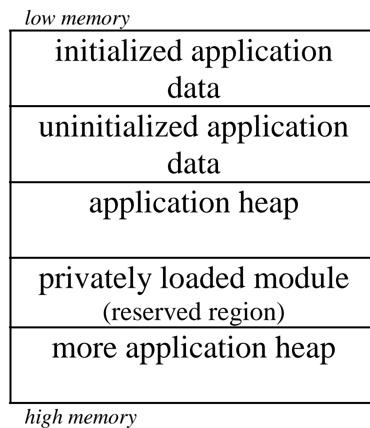
32-bit versus 64-bit address space:

To summarize the address space differences between 32-bit and 64-bit mode, consider the following table:

	32-bit Address Space	**64-bit Address Space**
Public module text	Seg 13	Seg 0x090000000
Public module data	Seg 15	Seg 0x09001000A
Private module text	Seg 2	Seg 0x080000000
Private module data	Seg 2	Seg 0x08001000A
Process private data	Seg 2 (or 3-10 for maxdata)	Seg 0x11

As can be seen, the 64-bit address space provides much more room in which to place components of the application. In particular, consider the differences for publicly loaded shared text. In the 32-bit address space, a single segment, 13 (or 0xD), is available to contain shared modules; in the 64-bit address space, the shared text region begins at address 0x0900000000000000, *and* extends for a total of 16 segments (as of this writing). This provides for 4 GB of shared text, contrasted with a maximum of 256 MB in the 32-bit address space. The data regions for publicly loaded modules enjoy similar breathing room: an additional 16 segments are available for data regions. Again, a single segment is available in the 32-bit address space for shared module data.

Now consider privately loaded modules. In 32-bit, the modules occupy portions of the process-private data segment. The loader maps the text and data of the module into a reserved section of the segment and keeps track of its use of that area. This behavior has the unpleasant side effects of: (a) possibly producing holes in the process heap, which also occupies segment 2, and (b) reserving that region for dynamic loading during the lifetime of the process.



This behavior results from the interaction of the loader and memory allocation in the process. If the loader moves the break value to dynamically load a module, and the process then requests more memory (thus moving the break value further) the hole created in the heap for the module will always be reserved for dynamic loads, even if the module is unloaded. This situation can be avoided if the application does not move the break value while the module is part of the process. If the module is unloaded and the break value is still immediately after the module, where the loader moved it at load time, then the loader will move the break value back to the end of the heap. The process heap may then grow into the region formerly occupied by the module. The result is some fragmentation of the heap, as well as a reduction in the total available data space available to the process.

In 64-bit mode, an additional 16 segments have been reserved for privately loaded module text, and 16 for privately loaded module data. This layout emulates the 64-bit shared library region, but modules may be loaded and unloaded into these segments without visibility to other processes. There is no impact upon the process heap because of its location in a completely different segment. There is less work for the loader to do during dynamic loading since it does not need to keep track of regions of process memory in use by the loader.

One additional point regarding the 32-bit address space: the 32-bit model allows for the process heap to occupy either segment 2, or to begin in segment 3. The latter occurs in "maxdata" programs, also called "big data" or "large address space model" programs, as explained earlier. In the case of maxdata programs, there is a slight variation on the behavior of the loader for module data layout at exec time. Shared libraries remain in segments 13/15; process private data and the heap begin in segment 3, extending as far as the maxdata value allows. However, modules that are privately loaded at *exec time* are mapped into segment 2, *not the heap segment*. Thus, maxdata programs may have privately loaded modules in either segment 2 or the private data segments (3-X). Private modules added to the address space during execution continue to occupy space in the private data segment(s), thus following the scenario outlined above.

One significant conclusion that can be drawn from all of this is that an application should not make assumptions about the location of any program or module text or data, especially with respect to dynamically loadable modules. An application that has expectations about the location of modules may have significant problems when, for example, it is converted from 32-bit to 64-bit. Also, assumptions of this type may preclude converting a common 32-bit application into a 32-bit maxdata application (a procedure that any customer can follow, as it only requires a binary edit of an executable).

Finally, a comment about the coexistence of the 32-bit and 64-bit address spaces. While the 64-bit address space has been designed to inhabit segments 16 and up, a 64-bit process does *not* have addressibility to segments 13 and 15. That is, it can't see the 32-bit shared library segments. By the same token, a 32-bit process has no method for accessing the 64-bit address space. Aside from programmatically accessing shared memory segments (via *shmat()*, for example) the two address spaces are distinct. It is not possible to combine 32-bit and 64-bit modules in a single application, either, due to these address space issues. A program may function in one environment or the other, but not both simultaneously.

Appendix D

Details on the linker, loader search path and LIBPATH

The following lines are in a test script that is used in all the examples below:

```
cc -c america.c
cc -o shramerica.o -bE:america.exp -bM:SRE -bnoentry america.o
ar qv libamerica.a shramerica.o
cc -o main2 main.c -lamerica -L. -blibpath:"/myappslib:/usr/lib:/lib"
cc -o main3 main.c -lamerica -L. -blibpath:"/myappslib"
cc -o main4 main.c -bI:america.imp -L.
cc -o main5 main.c -bI:america.imp -L/fixapplib2
cc -o main6 main.c -bI:america.imp
```
(with LIBPATH set to /tmp/home)

Note, that the linker handles path information in the following ways:

`-L` the path used for searching the libraries specified with the –l flag. The -L paths are recorded in the loader section as the default search path for dynamically loadable modules.

`-blibpath` the path to be inserted into the default path (Index 0 path) field of the loader section. When this flag is presented, the -L paths will not be stored. More information on this in the examples below.

`-bnolibpath` will not store -L and -blibpath path names. It causes the LIBPATH information to be stored.

`#!path` the path associated with a shared library or an object in an import list. It is to be recorded as a fixed path in the loader section.

`LIBPATH` is not for searching libraries, but it may get recorded in the loader section if no -L or -blibpath is presented. More information on this in the examples below.

<u>Example 1, linking with an import file and -L:</u>

For this example, the following lines are within a test script:

```
cc -c america.c
cc -o shramerica.o -bE:america.exp -bM:SRE -bnoentry america.o
ar qv libamerica.a shramerica.o
cc -o main4 main.c -bI:america.imp -L.
```

Now, let's look at the program components: america.c contains three functions: usa, canada and mexico. This program is used as the source to create a shared library libamerica.a as shown in the script file above.

The main program, main.c, has calls to usa, canada and mexico. It is compiled and linked with: `cc -o main4 main.c -bI:america.imp -L.` to produce the executable, main4.

The import file is called america.imp. The first line inside america.imp is: "#! ./libamerica(shramerica.o)". This import library information is recorded as the "Index 2" line in the loader section. (Please run *dump –HTv main4* for the listing of the loader section). The path "." indicates that the libamerica.a must be found in the local subdirectory ".". The "Index

0” line lists a path of “.:/usr/lib:/lib”. The “.” is from the “-L.” in the cc command line. The /usr/lib and /lib are appended to the –L paths by the linker. The LIBPATH data is not used by the linker for this case.

What happens at exec time? The main4 program is loaded first. From the loader section of main4, the system loader knows it needs to load libamerica.a (and then shramerica.o) from the current directory, denoted by a period in Index 2 line. Since this is a fixed path associated with the library, no other path information (*ie,* Index 0 path and LIBPATH) is involved in searching the library.

Now, let’s take a look at another, similar example.
```
cc -o main5 main.c -bI:america.imp -L/fixapplib2
```

Note that the /fixapplib2, from the –L flag, is now the first part of the default search path labeled by Index 0 in the loader section.

<u>Example 2, Linking with –l, -L and -blibpath flags:</u>

Example 2 uses the following test script:
```
cc -c america.c
cc -o shramerica.o -bE:america.exp -bM:SRE -bnoentry america.o
ar qv libamerica.a shramerica.o
cc -o main2 main.c -lamerica -L. -blibpath:"/myappslib:/usr/lib:/lib"
```
Again, as in example 1, america.c contains three functions: usa, canada and mexico. This program is used as the source to create a shared library libamerica.a as shown in the script file above. The main program main.c has calls to usa, canada and mexico. It is compiled and linked with:
```
cc -o main2 main.c -lamerica -L. -blibpath:"/myappslib:/usr/lib:/lib"
```
to produce the executable, main2.

In this example, Index 2 path is blank. This is due to the use of the –l flag for the library instead of the import file as was the case in example 1. When a library denoted by –l flag is found (-L search), it is listed in the loader section without a fixed path. We intentionally put both –L and -blibpath in the command line:
```
cc -o main2 main.c -lamerica -L. -blibpath:"/myappslib:/usr/lib:/lib"
```

Note that the –blibpath causes the path associated with it to be recorded entirely, without change, in the loader section as Index 0 path data. The –L path on the same command line is not recorded this time, as opposed to what we saw in example 1. The rule says, if –blibpath is specified, the path data is the only one to be recorded, all others including –L and LIBPATH are ignored.

To run main2, it will require libamerica.a to be placed in /myappslib, /usr/lib or /lib first.
If libamerica.a is not placed in any of the locations above, we may use the LIBPATH variable to go before the Index 0 path. That is, if the libamerica.a is in /tmp, then the following will resolve a potential “shared library not found” problem:
```
> export LIBPATH=/tmp:$LIBPATH
```

This, if libamerica.a is in /tmp, and we add /tmp to LIBPATH, the shared library will be found.

Now, let's look at another example, and see what happens if we do the following command line to generate a main3:

```
cc -o main3 main.c -lamerica -L. -blibpath:"/myappslib"
```

Will main3 work? Let's display the loader section:

First, note that the libamerica.a must be placed in /myappslib for it to work. But, will it really work? If we try to run main3, we will discover that even the system library libc.a cannot be loaded. Why? We don't have /usr/lib:/lib in the Index 0 search path.

Let's take a look at using -blibpath now. Here, we demonstrate one design decision about using –blibpath: the path will simply be recorded as is. Not even the system search paths are appended to it. Whenever –blibpath is used, the path should be coded to include the system path information.

Example 3 - linking with an import file and no -L:

We use the following lines in a test script for this example:

```
cc -c america.c
cc -o shramerica.o -bE:america.exp -bM:SRE -bnoentry america.o
ar qv libamerica.a shramerica.o
cc -o main6 main.c -bI:america.imp
```
(with LIBPATH set to /tmp/home)

And run the following command:

```
cc -o main6 main.c -bI:america.imp
```

The resulting loader section, has the LIBPATH listed in Index 0 path. When –blibpath and –L paths are not specified in a link command line, the current value of LIBPATH environment variable is copied to the Index 0 path.

Summary

- In the link command line, -l library is searched through –L paths.
- The –L path is recorded in the Index 0 path line, appended with /usr/lib and /lib.
- If an import list is also presented: the path and library information on the #! line is recorded in the loader section, the path becomes fixed for the particular library, and this path cannot be changed during runtime.
- If –blibpath is on the command line, the libpath information gets recorded, the –L path is not recorded, but it is still used in searching for –l libraries.
- If both –L and –blibpath are not used in the command line, the LIBPATH value is then recorded in the loader section as the Index 0 path.

There are still other flags such –bnolibpath which are not normally used, see *ld* man page for details.

Glossary

The following definitions will be useful in understanding the technical details of the linking and loading mechanisms of AIX.

Archive Files are composite objects, which usually contain object files. On AIX, archives can contain shared objects, import files, or other kinds of members. By convention, the name of an archive usually ends with .a, but the magic number of a file (that is, the first few bytes of the file) is used to determine whether a file is an archive or not.

Control SECTion (CSECT) is the atomic unit of relocation as far as link-editing is concerned. A CSECT can contain code or data. One CSECT can contain references to other CSECTs. These references are described by relocation entries (RLDs) contained in a section of an object file.

Dependent Module is module loaded automatically as part of the process of loading another module. A module is loaded automatically if it is listed in the loader section (dependency list) of another module being loaded.

Dynamic Binding is the technique in which imported symbols are looked up or resolved at load-time or run-time.

Dynamic Loading is a technique, which allows addition of modules to a running (executing) process. Modules can be loaded with the load(), dlopen(), or loadAndInit() functions.

Exec-time is load-time for the main program. A new program is loaded when the exec() function is called.

Executable is a module with an entry point. An executable may have exported symbols.

Export files are ASCII files that list global symbols to be made available for another module to import. The file formats of the import and export file are the same.

Export/Import Lists are ASCII files that list symbols to be resolved at load-time, and often, their defining modules. This information is saved in the loader section of the generated module, and is used by the system loader when the module is loaded. The import file can be used in place of a corresponding shared object as an input file to the ld command.

Lazy Loading is the ability to defer the loading of a dependent module, until the process calls a function in the dependent module.

LIBPATH environment variable is a colon-separated list of directory paths that can also be used to specify a different library search path. At run time the library search path is used to tell the loader where to find shared libraries. Its format is identical to that of the PATH environment variable.

The directories in the list are searched to resolve references to shared objects. The /usr/lib and /lib directories contain shared libraries and should normally be included in the library search path.

Libraries are files containing the definitions of multiple symbols.

Linker is an application program that combines multiple object modules into an executable program. On AIX, the ld command is the system linker. The ld command processes command line arguments and generates a list of commands, which are piped to another program called the binder (/usr/ccs/bin/bind). For the purpose of this document, the terms linker and binder are used interchangeably.

load() This call is used to add a module into a running process, allowing a program to expand its own capabilities. The unload() system call can be used to remove object modules from an executing program, which were loaded with the load() routine.

loadbind() This call is used to allow "deferred" references to be resolved after a module has been loaded.

Load-time is the time period during which the system loads a module and its dependents. This includes the processing required to resolve symbols and relocate the modules. A module can be loaded with the exec(), dlopen(), or load() system calls. (The dlopen() and loadAndInit() functions are higher level calls that ultimately call load().)

Module is the smallest, separately loadable and relocatable unit. A module is an object file containing a loader section. Modules are loaded implicitly when they are dependents of another loaded module. A module may have an entry point and may export a set of symbols.

Object File is a generic term for a file containing executable code, data, relocation information, a symbol table, and other information. Object files on AIX are in XCOFF (eXtended Common Object File Format) format.

Rebinding is associating an alternate definition with an imported symbol at run-time. By default, the linker identifies a specific dependent module with each imported symbol. When the run-time linker is used, the imported symbol can be associated with a symbol in a different module.

Resolution is the act of associating a reference to a symbol, with the definition of that symbol. Symbol resolution is performed by the linker at link-time and by the system loader at load-time. Some special kinds of symbol resolution can occur during run-time.

Run-time is the time interval from when the control enters main() and ends when the program exits.

Run-time Linking is a technique where imported symbols are resolved at load-time. When run-time linking is used, symbols may resolve to alternate definitions that were not available at link-time.

Shared Object/Shared Library/Shared Module is a module with the F_SHROBJ flag turned on in its XCOFF header. This flag is turned on if the object is built using the -bM:SRE or -bM:SRO linker option. A shared object defines external symbols that are available to other referencing modules at run-time. When linking with a shared object, the linker does not include code from the shared object into the program. Only the name of the shared object and symbols imported from that shared object are saved in the program.

In AIX, a shared object can be an archive member of a shared library. Conversely, a shared library on AIX can be a single shared object module, or an archive of one or more objects, of which, at least one is shared. By industry convention though, any file ending in .a is usually an archive, while a file ending in .so is a shared object. Typically, system libraries in AIX are archives consisting of some shared and some non-shared objects. AIX accepts both .a and .so as shared library suffixes, though precedence is given to .so over .a libraries.

Static Linking is executing the ld command so that shared objects are treated as ordinary (non shared) object files. A shared object that has been stripped cannot be linked statically.

Strip is a system command that reduces the size of a XCOFF module by removing the linking and debugging information from it. Object files that have been stripped cannot be used as input to the linker. A module that has been stripped can still be loaded, since the module's loader section, contains the information needed to load the module is still intact.

System Loader is a kernel subsystem that creates a process image from an executable file. The loader loads the executable file into memory, loads (or finds) the program's dependent modules, looks up imported symbols and relocates references.

Copyright International Business Machines Corporation. 2001. All rights reserved.

AIX, pSeries, IBM RISC System6000, RS/6000 are trademarks of the IBM Corporation.
Java is a trademark of Sun Microsystems Inc.
All others are trademarks or registered trademarks of their respective companies.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Improvements and/or changes in the products and/or programs described in this publication may be made at any time.