Using Shared Libraries on AIX 4.2

Stephen B. Peckham

Session 5362

Definitions

- Object File: A generic term for a file containing executable code, data, relocation information, a symbol table, etc. Object files on AIX are XCOFF (e**X**tended **C**ommon **O**bject **F**ile **F**ormat) files.

- Csect: the atomic unit of relocation.

- Module: The smallest, separately loadable and relocatable unit. A module must contain a loader section. Modules are loaded explicitly with exec() or load() calls. 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.

- Dependent Module: A module loaded automatically as part of the process of loading another module.

- Executable: A module with an entry point. An executable may have exported symbols.

Definitions...

- Shared Object: A module with the F_SHROBJ flag turned on.
- Import File: An ASCII file that may be used in place of a corresponding shared object as an input file to the ’ld’ command.
- Main Program: The initial module loaded by exec().
- Static Linking: Executing the ’ld’ command so that shared objects are treated as (non–module) object files. A shared object that has been stripped cannot be linked statically.

Any object file that has not been stripped can be used as an input file to the ’ld’ command. Modules, except for executables, may be added to an archive and loaded directly.

Conventional Static Linking

- Object files and archives are listed on command line.
- Entire object files are copied to the output file.
- Entire archive members are copied to the output file if they resolve undefined references at the time the archive is read.
- Duplicate symbols are not allowed.
- Example:

```
cc –o main main.o sub.o
```

causes the command

```
ld –o main /lib/crt0.o main.o sub.o –lc
```

to be generated. All of *main.o* and *sub.o* will be part of the output file. Members of */usr/lib/libc.a* will be part of the output file if they are referenced.

Conventional Shared Objects

- A shared object (or shared library) is a loadable file, distinct from its corresponding static archive, used instead of the static archive by the linker.
- A shared object defines same set of names as static archive, but the concept of distinct archive members is lost.
- A shared object (like other object files) may contain references to undefined symbols.
- A shared object may contain a list of other shared objects that can be loaded to resolve undefined symbols.
- When a shared object is loaded, its names are added to a flat namespace that can be used to resolve undefined references from other shared objects and the main program.

Runtime Characteristics Of Shared Objects On Other Platforms

- Position–independent code is required.
- Shared object can be loaded at any address.
- Code (.text section) is mapped into a process's address space using copy–on–write semantics. Runtime linking may require a private copy of some .text pages.
- By default, calls to out–of–module functions are made with indirect branches.

Building Conventional Shared Objects

- Source files must be compiled to position–independent code.
- Special linker option is used
  - –G
- Undefined symbols are allowed.
- Duplicate symbols from shared objects used as input files are allowed.
- Names of shared objects listed on command line are saved in output file for possible use at load time.
- All global names in input files are exported.
- Symbols resolved by definitions in shared objects are left unresolved at link time. They must be resolved at load time.

Building Conventional Executables

- Position–independent code is not needed.
- Programs are linked to absolute runtime addresses and cannot be relocated at load time.
- All global names are made visible, for possible use at load time.
- Names of shared objects listed on command line are saved in output file for possible use at load time.
- Duplicate symbols from shared objects used as input files are allowed.
- Undefined symbols are not allowed.

Execution Of Conventional Programs Using Shared Objects

- Runtime linker is used to resolve undefined references.
- Undefined references are satisfied by first definition found in main program or any shared object.
- Shared objects are searched in breadth–first search order.

AIX Static Linking

- Object files and archives are listed on command line.
- Csects containing referenced symbols are copied to the output file. The entry point, exported symbols, and symbols specified with –u flag define initial set of referenced symbols
- Garbage Collection: Unreferenced csects are discarded
- Prelinked archives can be used

```
ld –r *.o
```

- Csects of archive members used if they provide first definition for a symbol, even if first reference follows archive on command line.
- Warnings printed for duplicate symbols, unless duplicate comes from archive member.

AIX Shared Objects

- A module that defines a set of names (in its loader–section symbol table).
- Some global names in constituent object files may not be exported.
- No references to undefined symbols exist. Symbols defined outside the shared object must be imported from another shared object or defined as deferred imports.
- List of dependent modules is recorded in shared object. At load time, all dependent modules must be loaded successfully for loading of shared object to succeed.

Building Aix Modules

- No special compiler flag is needed—all code is position–independent.
- Export files required to define set of exported symbols.
- Undefined symbols are not allowed—all symbols should be defined, perhaps by being imported from some other shared object.
- Referenced shared objects from the command line are included in the list of dependent objects in the output file. Other shared objects from the command line are not listed.

Characteristics Of Aix Modules

- Position–independent code is always used.
- Modules are always relocated. The .text and .data sections are relocated independently.
- Code (.text section) is always read–only.
- Calls to out–of–module functions must be made with indirect branches.

Building AIX Shared Objects

- Shared objects are built in the same manner as other modules, except that the linker option –bM:SRE is required to cause the F_SHROBJ flag to be set.

Building AIX Executables

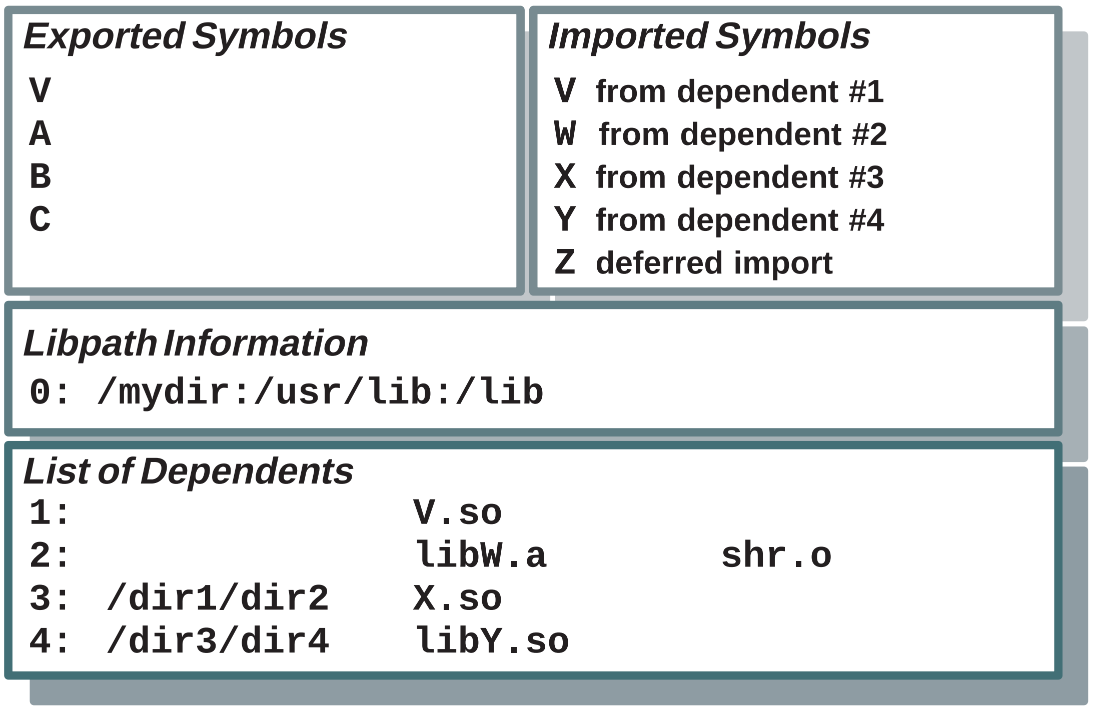
- Executables are built in the same manner as other modules, except that an entry point must be defined.
- Load–time origins are arbitrary. All modules are relocated at load time.

Execution Of AIX Programs Using Shared Objects

- No runtime linking. System loader performs symbol lookup and relocation.
- Two–level namespace associates imported symbols with a particular dependent module. Searching of all modules to resolve undefined symbol is not done. Therefore, load order is not defined.
- All dependents of the module must load successfully.
- Imported symbols must be exported from their defining dependent module, as recorded in the loader section.
- Since imports are by module/symbol basis, multiple symbols with the same name may exist.

AIX Modules

- Interface View
  - List of exported symbols
  - List of dependent modules
  - Table of imported symbols
    - each symbol is associated with a dependent.



Exported Symbols

```
V
A
B
C
```

Imported Symbols

V	from dependent #1
W	from dependent #2
X	from dependent #3
Y	from dependent #4
Z	deferred import

Libpath Information

```
0: /mydir:/usr/lib:/lib
```

List of Dependents

```
1:                V.so
2:                libW.a       shr.o
3:  /dir1/dir2    X.so
4:  /dir3/dir4    libY.so
```

- Display this information with the dump command

`dump -HTv` *module*

AIX Modules...

- Implementation View:
  - List of symbols
  - List of undefined symbols

Global Symbols	*Local Symbols*	*Undefined Symbols*
`A`	`a`	`V`
`B`	`b`	`W`
`C`	`c`	`X`
`D`	`a`	`Y`
`E`	`b`	`Z`
`F`	`c`	

- Display this information with

`dump -tv` *object_file*

Runtime Context

- IAR (instruction address register) gives addressability to code.
- Because text and data are relocated independently and text is read–only, function calling conventions require a pointer to the data of a module.
- The pointer is the TOC pointer in R2. All data references are computed with respect to TOC pointer.
- Each module has a separate TOC pointer.
- Intra–module calls do not need to modify TOC pointer.
- Inter–module calls must save current value of R2, load new value, and restore old value when function returns.
- Function descriptors are tuples <code, data>.

Generated Code

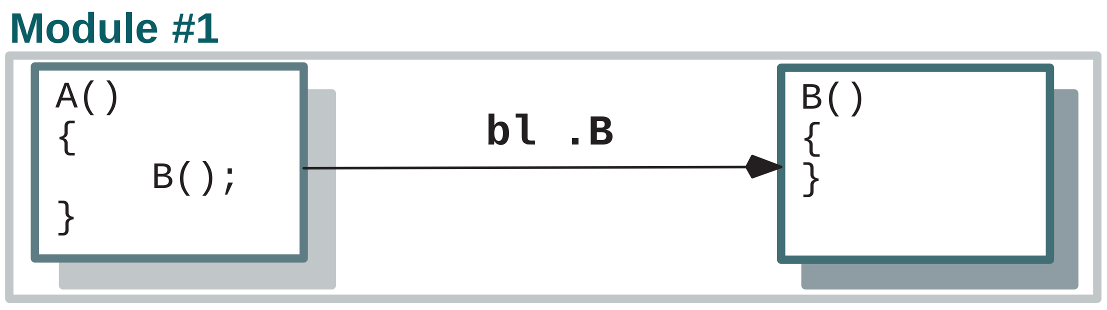
- Call to function defined in same module:

```
A()
{
     B();
}
```

- Compiler generates the call as

```
bl  .B
nop
```

- Pictorial view of module after linking



- Nop is not modified.
  - Call cannot be rebound at exec time.

Generated Code

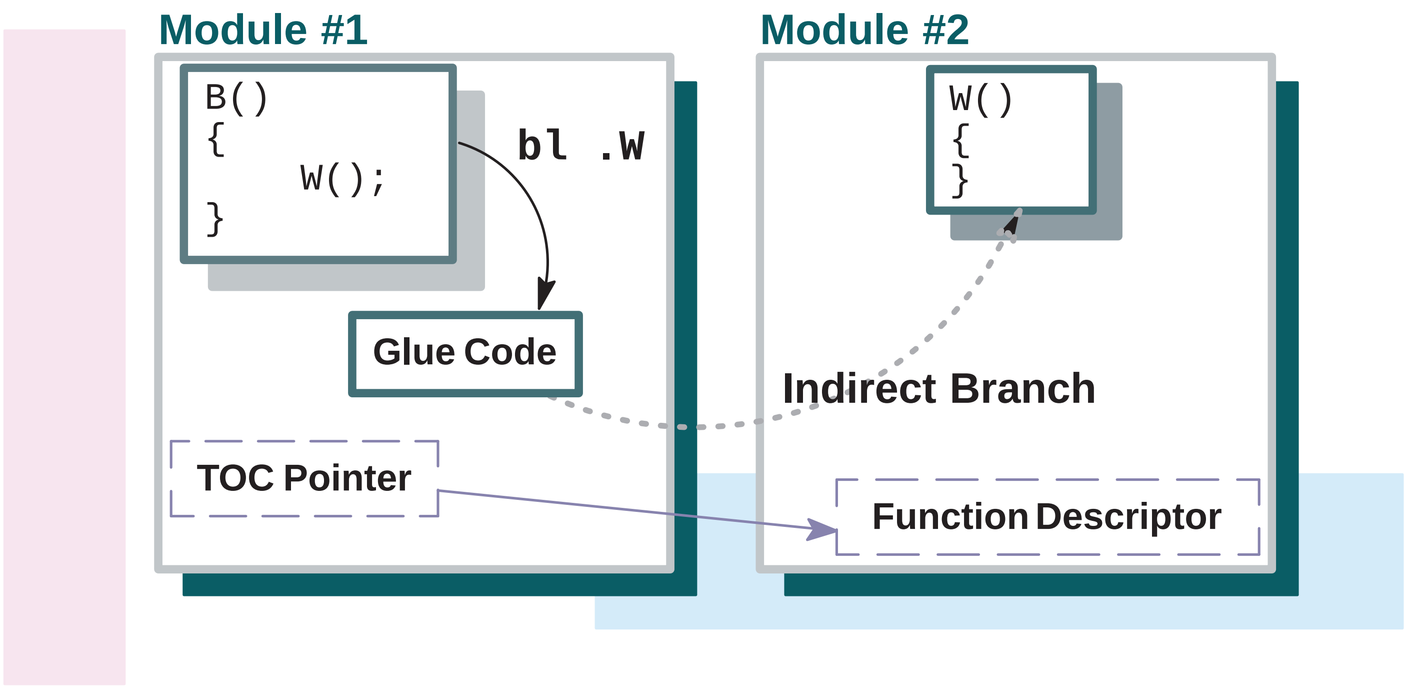
- Call to imported function:

```
B()
{
    W();
}
```

- Compiler generates the call as

```
bl  .W
nop
```

- Pictorial view of modules at runtime.



Generated Code: Glue Code

- Glue code
  - saves R2 in stack
  - loads new value of R2 from function descriptor
  - loads code address from function descriptor into a register and branches indirectly
- When function returns, nop instruction is converted by linker to instruction to reload the old value of R2 from the stack.
- Call can be rebound, but not with standard AIX 4.1.

Shared Objects at Load Time

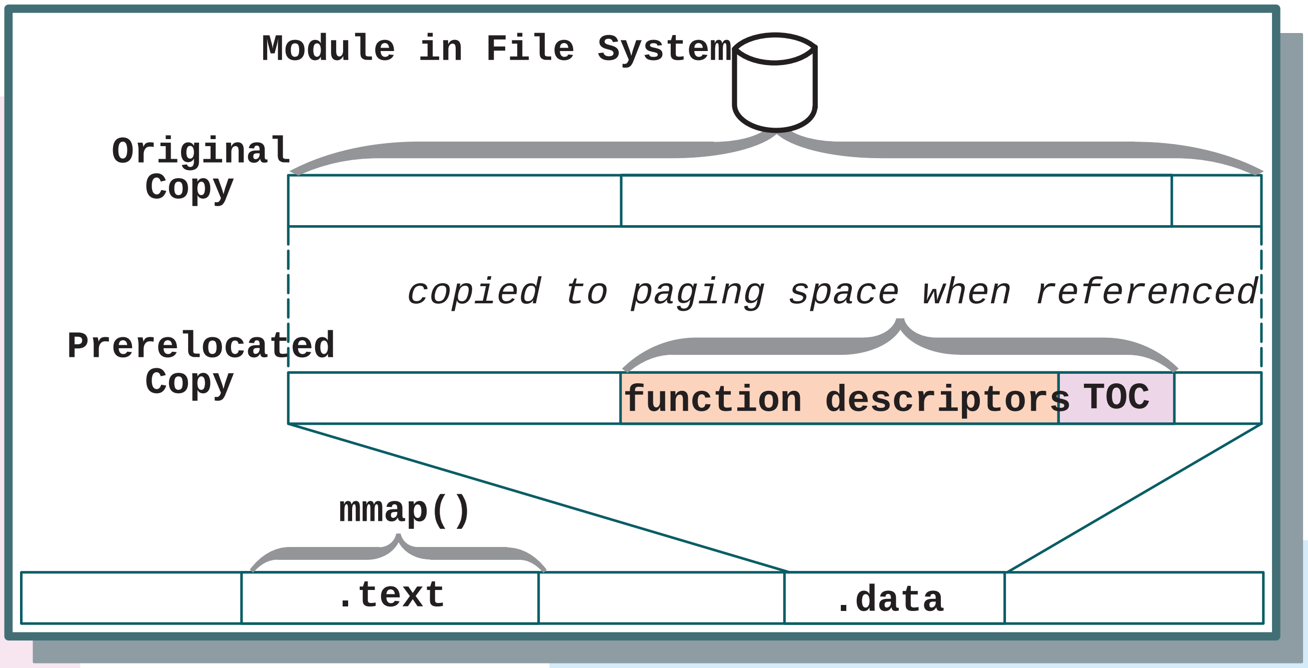
- Text loaded once at common system–wide address.
- Always position–independent
- Text section is backed to the file and requires no paging space
  - Exception: NFS files
- The linker collects TOC pointers and function descriptors to reduce the number of pages that need to be touched during relocation.
- Data section of a shared object is prerelocated.
- Data section is loaded at common system–wide address.
- Multiple prerelocations are possible if different dependent modules are needed.

System Loader

- Loads a module and its dependents recursively.
- Resolves symbols—each symbol must be found in its appropriate defining module.
- Deferred imports are not resolved at exec time.
- Relocates references.
- Data sections for shared objects with prelocations need no additional relocation. They are map–copied.

Performance Benefits

- Text pages demand paged.
- Prerelocated data pages demand paged—no work is done at load time to fix up shared library data pages.
- Symbol resolution is done at load time by imported module.
- Searching all loaded modules for definitions not required. Only a single module can possibly define a symbol.



Inhibiting Use Of Prelocations

- Modules without “read other” permission are loaded privately in a process’s address space, not in the shared library, and so prerelocation is not computed.
- Modules that depend on privately loaded modules cannot be prerelocated.
- Modules that depend on the main program cannot be prerelocated.

Limitations Of Aix Shared Objects

- No run time linking.
- Intra–module references are fixed at link time.
- Inter–module references are associated with shared objects with specific names. Only LIBPATH setting can affect module that is used.
- Undefined symbols are not allowed, even if symbols are never used.

New Features of AIX 4.2

- Runtime linking
- *libdl.a* routines
- init/fini routines
  - –binitfini option
- –G flag
  - Shorthand notation for building shared objects when runtime linking is being used.
- –brtl option to enable runtime linking for a module
- Loading of archive members allowed
- . and .. imports
- –bexpall/–bnoexpall options

New Features...

- Visibility attributes
  - symbolic/non–symbolic
  - global or per symbol
- –bdynamic, –bstatic toggles
- –bautoexp
- rtl_enable command
- –bipath/–bnoipath options
- –brtllib/–bnortllib

Runtime Linking

- Implemented as part of startup code in user space, not part of system loader.
  - Behavior of existing programs unaffected.
- Runtime linker rebinds references to definitions before main() or init routines are called.
- Any imported symbol can be rebound.
- Intra–module references can only be rebound for modules that were linked appropriately.
- By default, function references cannot be rebound.
- By default, variable references can be rebound.
- Definition used is first in breadth–first search order.
- Behavior meets draft ASPEN spec.

libdl.a Routines

- Standard routines dlopen(), dlclose(), dlsym(), dlerror().
- May be used with or without runtime linking.
- Use dlopen(NULL, ...) to find symbols in entire process.
- Behavior meets draft ASPEN spec.

Init/Fini Routines

- Each module can have its own set of routines.
- Init routines called when module loaded, including at exec time. Main program must be linked on AIX 4.2 for init routines to be called.
- Modules are initialized in breadth–first search order.
- Init routines for a given module are called in priority order.
- Fini routines called when module physically unloaded, or when exit() is called. When exit() is called, fini routines called after atexit() routines.
- Note: Calling unload() may not remove the module if it is still being used by other modules in the process.
- Fini routines not called if _exit() called or process exits abnormally.
- Fini routines are called in reverse order.

New Flag: –G

- Shorthand notation for creating shared objects when runtime linking is being used.

- Equivalent to:

  `–berok –brtl –bnortllib –bnosymbolic`
  `–bnoautoexp –bM:SRE`

- If you choose to define all symbols, you can link with

  `–G –bernotok`

  This allow you to detect symbols that should have been defined in the shared object itself.

New Options: –brtl/–bnortl

- The –brtl option implies –brtllib, –bsymbolic
- In addition
  - All shared objects listed on command line (that are not archive members) are listed as dependent modules, preserving the command line order.
  - When used with –berok option, associates undefined symbols with the dummy import file name "..". These symbols must be resolved by the runtime linker.
  - Allows *.so files to be found with –l flag.
    - Hint: You can use –brtl –bnortllib if you want .so files but don't want runtime linking. Do not use –berok option in this case.
- When used with the –bautoexp option, causes certain symbols to be exported automatically.

Loading of Archive Members Allowed

- A new flag, L_LOADMEMBER, may be passed to load() and loadAndInit().
- A new flag, DL_LOADMEMBER, may be passed to dlopen().
- Member name is surrounded by parentheses and concatenated with file name.
- Examples:

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

loads member shr.o from archive lib1.a.

```
load(”lib2.o()”, L_LOADMEMBER, NULL);
```

loads file lib2.o.

```
load(”lib3.o(shr.o)”, 0, NULL);
```

loads file ”lib3.o(shr.o)”.

New Import File Name

- . imports

- Using “.” as an import file name in an import file indicates that the symbol should be defined in the main executable:

```
#! .
foo
```

- The system loader resolves these symbols. The runtime linker is not needed. The main program must still export foo.

- Modules importing symbols from “.” cannot have prerelocations.

New Import File Name

- .. imports

  ```
  #! ..
  foo
  ```

- Using “..” as an import file name in an import file indicates that the symbol should be resolved by the runtime linker. At load time, ***foo*** must be defined by some module or load fails.

- The system loader ignores symbols imported from “..”

New Options: –bexpall/–bnoexpall

- Exports almost all symbols.
- Exports all global symbols except
  - imported symbols
  - unreferenced symbols defined in archive members
  - symbols beginning with an underscore
- You may export additional symbols by listing them in an export file.
- Some existing programs, such as *makeC++SharedLib*, generate export lists.
- If you have a well–defined interface, an explicit export list is preferable.

Visibility (Symbolic/Nonsymbolic)

- On a global or per–symbol basis, can control whether intra–module references can be rebound.
- Symbolic
  - References cannot be rebound
- Nosymbolic
  - References can be rebound
- Nosymbolic–
  - For variables, just like nosymbolic
  - For functions, direct calls cannot be rebound, but calls with function pointers can be rebound. Calls with function pointers are used if
    - compiler option –qinlglue is used
    - pointer–to–function variable is used.

Visibility (Symbolic/Nonsymbolic)

- Use –bsymbolic/–bnosymbolic/–bnosymbolic– to specify visibility for all symbols.
  - Exception: The default visibility for BSS symbols is “nosymbolic”, even if one of these options has been specified.
- Visiblity for specific symbols can be specified in an export file.

New Keywords in Import/Export Files

- To control the visibility of individual symbols, new keywords are allowed in export files.
- Existing keywords ’svc’ and ’syscall’ still exist and are used when linking kernel extensions.
- Keywords ’symbolic’, ’nosymbolic’, and ’nosymbolic–’ affect the visibility of individual symbols.
- Keyword ’list’ may be used to enter a symbol in the loader section symbol table without marking it as imported or exported. (No AIX program uses this capability).
- Keywords ’cm’ and ’bss’ are used in import file in conjunction with the –bautoexp option. When these keywords are used in export files, they mean the same as ’nosymbolic’.

New Options: –bdynamic/–bstatic

- Control whether shared objects used as input files should be treated as regular object files.
- These options are toggles that 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 object files.
- In addition, when –brtl is specified and –bdynamic is in effect, the –l flag will search for files ending in ”so” as well as in ”a”.

–bdynamic/–bstatic...

- Example:

  ```
  cc –o main main.o –bstatic –lmylib
  –Lmylibdir –bdynamic
  ```

  - File libmylib.a will be processed as a regular object file.
  - File libc.a (which is always specified by the ’cc’ command), will be processed as a shared object.

- Example:

  ```
  cc –o main main.o –brtl –llib1 –L/dir1
  –L/dir2
  ```

  - will search for
    - /dir1/liblib1.so
    - /dir1/liblib1.a
    - /dir2/liblib1.so
    - /dir2/liblib1.a

New Options: –bautoexp/–bnoautoexp

- Use to automatically export symbols in a few limited cases. The default is –bautoexp.

- Example

  - ❑ You are using a shared object that imports “foo” from “.”, that is, from the main executable. you define “foo” in your main program. To avoid having to write an export file when linking your main program, use the command line:

```
cc –o main main.o –lfoo
```

  - ❑ Symbol foo will be exported automatically. Runtime linker is not needed in this case.

New Command: rtl_enable

- Relink non–stripped modules to enable runtime linking.
- Intra–module references cannot be rebound by the runtime linker unless the module is built appropriately.
- Shared objects shipped with AIX are not enabled for runtime linking.

rtl_enable...

- Example:
  - You want to link programs that use your 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

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

    where 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.

- **Important**: If you omit –brtl, no runtime linking will take place. The modified libc.a will still work, but performance will be affected by the indirect branches.

New Options: –bipath/–bnoipath

- If you specify a full path for a shared object on the command line, the full path is saved in the loader section import file strings.

- For example

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

  will cause full paths to be saved for mylib.so and otherlib.a. At load time, the loader will always use these full paths to find the shared objects.

- If you use the –bnoipath option, only the base names will be saved.

New Options: –brtllib/–bnortllib

- Implied by –brtl, so this option almost never needs to be specified explicitly.
- Adds a reference to runtime linker.
- Implies –lrtl (librtl.a contains the runtime linker.)
- Main program must be linked with –brtllib if runtime linking is being used.
- Other modules do not require –brtllib option, but it is harmless.
- Causes runtime linker to be called at exec time and after load() calls.

Common Errors

- Using 'nm' to see if a symbol is defined in a main object.

- Example:

```
main() {
     PRINT();
}
```

$ cc main.c

results in

```
ld: 0711-317 ERROR: Undefined symbol: .PRINT
ld: 0711-345 Use the -bloadmap or -bnoquiet option
to obtain more information.
$ nm /usr/lib/libc.a | grep PRINT
.PRINT                    T     605948
```

- Only exported symbols are visible. You should use dump to see what symbols are exported.

```
$ dump -Tv /usr/lib/libc.a | grep PRINT
$
```

Common Errors

- Using –berok (or deferred imports) indiscriminantly
- Resolving deferred imports is slow.
- When you build shared objects, link with all avaiable shared objects that your shared object will need.
- If necessary, list symbols as deferred imports in an import file.

Diagnosing Problems

- To see the dependent modules of a given module, use the command

  ```
  dump -Hv
  ```

- To see what modules are actually loaded by an executable, you can use the ’map’ subcommand in dbx.

  ```
  $ 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
     Data origin:     0x20000968
     Data length:     0x5e0
     File descriptor: 0x7
  Entry 2:
     Object name: /usr/lib/libc.a
     Member name: meth.o
     ...
  ```

Diagnosing Problems...

- When a linking failure occurs, use the following options to assist in your diagnosis:
  - –bloadmap:<file>
  - –bmap:<file>

Common Problems

- Main program defines symbol that needs to be imported by shared object.

- 4.1 solutions:
  - If main program is called 'foo', link shared object with import file

    ```
    #! foo
    xx
    ```

  - Use deferred imports:

    ```
    #!
    xx
    ```

    - Use loadbind() to resolve deferred imports.
  - Put xx in a separate object. Link both main program and other object with new shared object.

Common Problems...

- 4.2 solution:
  - Use . import

    ```
    #! .
    xx
    ```

    - System loader handles import. Runtime linker is not needed.
    - **Important:** Importing symbols from the main program can affect performance. Prerelocations for modules importing from the main program cannot be created.

Common Problems...

- Defining module for a symbol is not known when building another module:

- 4.1 solution

  - Deferred imports and loadbind() calls.

    - hard to manage
    - Cannot be sure symbols are resolved.
    - Using with C++ static constructors hard to manage.

- 4.2 solution

  - .. imports

```
#! ..
xx
```

    - Runtime linker must be used. Imports from ".." imports are ignored by system loader. Prerelocations can still be used.

Common Problems...

- Using nlist().

- All modules are relocated when loaded. The symbol table value returned cannot by nlist() cannot be used directly.

- Text and data sections are relocated independently—the n_scnum field must be examined to compute relocation. (BSS relocation amount is the same as the data section relocation amount.)

- Use loadquery() to find load–time origins of text and data sections.
  - ldinfo_textorg is the beginning of the mapped module
  - ldinfo_dataorg is the actual beginning of the data

Common Problems...

- Data Relocation Amount:

  Load–time data origin

  from loadquery

  – link–time data origin

  from s_vaddr field of data–section header

- Link with –D0 flag (–Wl,–D0) to set link–time data origin to 0.

- Text Relocation Amount:

  Load–time data origin

  from loadquery

  – link–time text origin

  from s_vaddr field of text–section header

  + offset in the module of the beginning of text section

  from s_scnptr field of text–section header

- See Listing 2 for an example

Common Problems...

- Not specifying “extern”

```
int errno;
main(int argc, char *argv[])
{
    open(”no file”, 0);
    printf(”errno = %d\n”, errno);
}
```

```
cc –o main main.c
```

- This program will print

```
errno = 0
```

- main program has one instance of errno

  libc.a(shr.o) has another instance

Common Problems...

- Fortran common blocks
- Cannot say “extern” so definition in shared object must be first definition seen.
- Export “#BLNK_COM” from shared object.
- Link with command:

```
xlf -o main -lshr main.o
```

- See Listing 1 for an example

Searching for Modules

- Search order at exec time:
  - Directories in LIBPATH environment variable
  - Libpath information in main program
  - Libpath information in module whose dependents are being sought

- Search order at load time:
  - For module specified in load() call and its dependents
    - If L_LIBPATH_EXEC is set, use directories from the first two steps above
    - Use explicit argument passed to load()
    - If no explicit argument was passed to load(), use the current value of LIBPATH
  - For dependents of loaded module only, add
    - Libpath information in module whose dependents are being sought

Listing 1

main.f:

```
        program main
        common  i
        integer*4 i
        i = –20
        write(6,1) i
1       format( ’Data value i = ’, I8 )
        call fn1()
        write(6,2) i
2       format( ’Data value i = ’,I8 )
        end
```

sub.f:

```
        subroutine fn1()
        common  i
        integer*4 i
        write(6,1)
1       format( ’Changing i to 457’ )
        i = 457
        return
        end
```

sub.exp:

```
fn1
#BNLK_COM
```

Commands:

```
xlf –c main.f
xlf –c part1.f
ld –o shr.o part1.o –bE:part1.exp –bnoentry –lxlf90 –bM:SRE
rm –f libshr.a
ar cr libshr.a shr.o
xlf –o main main.o –lshr –L.
xlf –o main1 –lshr main.o –L.
```

Duplicate symbol warning is expected from the last command.

Output from main:

```
Data value i =      –20
Changing i to 457
Data value i =      –20
```

Output from main1:

```
Data value i =      –20
Changing i to 457
Data value i =      457
```

Listing 2

nlist.c:

```
#include <stdio.h>
#include <nlist.h>
#include <xcoff.h>
#include <sys/types.h>
#include <sys/ldr.h>
static int data_reloc;
static int text_reloc;
static int sntext;
static void
get_reloc(char *module) {
    char *h;
    char buffer[1024];
    struct ld_info *info;
    h = (char *)load(module, L_LIBPATH_EXEC, ””);
    loadquery(L_GETINFO, buffer, 1024);
    for (info = (struct ld_info *)(&buffer[0]);
      info;
      info = (info–>ldinfo_next == NULL ? NULL
           : (struct ld_info *)((char *)info + info–>ldinfo_next))) {
     if ((char *)(info–>ldinfo_dataorg) <= h
         && (char *)info–>ldinfo_dataorg + info–>ldinfo_datasize > h) {
         FILHDR  *filhdr = (FILHDR *)(info–>ldinfo_textorg);
         AOUTHDR *aouthdr = (AOUTHDR *)((char *)filhdr + sizeof(FILHDR));
         SCNHDR  *scnhdrs = (SCNHDR *)((char *)aouthdr
                          + filhdr–>f_opthdr);
         sntext = aouthdr–>o_sntext;
         text_reloc = (int)info–>ldinfo_textorg
          – scnhdrs[aouthdr–>o_sntext–1].s_vaddr
              + scnhdrs[aouthdr–>o_sntext–1].s_scnptr;
         data_reloc = (int)info–>ldinfo_dataorg
          – scnhdrs[aouthdr–>o_sndata–1].s_vaddr;
         return;
     }
    }
}
main(int argc, char *argv[]) {
    struct nlist mynl[2];
    get_reloc(argv[1]);
    mynl[0].n_name = argv[2];
    mynl[1].n_name = NULL;
    nlist(argv[1], mynl);
    printf(”Symbol table value for %s is %x, load–time address %x\n”,
        mynl[0].n_name,
        mynl[0].n_value,
        mynl[0].n_value + (mynl[0].n_scnum == sntext
                     ? text_reloc : data_reloc));
}
```