SUN/202.2

Starlink Project
Starlink User Note 202.2

M. J. Bly

19 February 1998

Starlink Subroutine Libraries
A Guide for Program Development and Linking

Abstract

This note gives a general overview of the methods available for using Starlink Infrastructure subroutine libraries with applications. There is an outline of how to use the include files for a subroutine library, and a guide to the methods available for linking with the subroutine libraries.

Contents

1 Introduction

This note gives a general overview of the methods available for using Starlink Infrastructure subroutine libraries with applications. There is an outline of how to use the include files for a subroutine library, and a guide to the methods available for linking with the subroutine libraries.

All the Starlink Infrastructure libraries are organised in the same way, so it is possible to give a general guide to the principles involved. However, some libraries do differ, and for precise details of how to use a particular subroutine library, you should consult the Starlink document for that library.

2 Organisation

Suppose you wanted to use an Infrastructure library BLY (there isn't one – this is an example!). In the Starlink installation, the BLY subroutine library has several components:

(1) a library file `libbly.a`

(2) a shareable library `libbly.so`

(3) a development script `bly_dev`

(4) Fortran `INCLUDE` files `bly_err` and `bly_par`

(5) ADAM[1] versions of the library and shareable library `libbly_adam.a` and `libbly_adam.so`

(6) link scripts `bly_link` and `bly_link_adam`

Most of the Infrastructure libraries have all these components, but some have more `INCLUDE` files, and some do not have shareable libraries. Those that do not have any `INCLUDE` files will lack a development script. A few libraries do not need separate ADAM versions, so will not have libraries for use with ADAM.

It is best to consult the documentation for a particular library to see what `INCLUDE` files and libraries are available.

The components may all be used as part of the development and linking of programs that use the Starlink Infrastructure subroutine libraries.

3 Program Development – `INCLUDE` files

If the Infrastructure library you want to use has `INCLUDE` files, you need to be able to reference them from your source code.

[1] See SG/4 '*ADAM – The Starlink Software Environment*'

You might wish to use the full PATH name for the INCLUDE file, *e.g.*, /star/include/bly_par. This is fine, and works, but is not portable, and could lead to problems if you want to use a development version of the library.

The recommended way to use the INCLUDE files for a particular library is to create links in your development directory to the INCLUDE files, and reference the links in your source code.

A library 'dev' script will create the links for you in your working directory. The links are **UPPER-CASE**, and it is these upper-case links you reference in the source code, thus:

```
      PROGRAM MYPROG
*
*  demonstrate use of include file
*
      INCLUDE 'BLY_PAR'
*
      ...
*
      END
```

To generate the links, issue the development command:

```
% bly_dev
```

This will create links to ALL the INCLUDE files for the BLY library:

```
BLY_ERR ->  /star/include/bly_err
BLY_PAR ->  /star/include/bly_par
```

To remove the links, issue the command again with the remove option:

```
% bly_dev remove
```

You should keep the links in place while developing your program. If you want to move development directories, simply remove the links and create new ones in the new directory.

The Starlink software building system uses the soft link strategy in its makefiles, though the makefiles generate the links themselves. The links may easily be changed to pick up a development version of a library, without having to edit source code.

4 Library Link Scripts

4.1 Background

The Starlink Infrastructure libraries depend upon one another in a hierarchy of dependencies that is quite complicated – dependencies that mean one library may need several others at link time to get a full resolution of all the subroutine calls.

So that users do not have to remember the dependencies of a particular library, each library has a link script that contains references to its own libraries, and all the other libraries that it depends upon. Most of the other references will be to the link scripts for the other libraries.

This in itself presents a problem – the nested links scripts can generate a long list of libraries, often with each library occurring more than once.

To avoid this, each link script has an internal mechanism that trims unnecessary occurrences of a library out of the list.

The link script writes a list of libraries to its standard output, so to get the list into your compile or link command, you need to run the script as part of the compile or link command. To do this you just back-quote the link script name thus: `‘bly_link‘`.

The result is that when using the link scripts, you do not have to worry about remembering which libraries that the one you need depends upon, and the linker is provided with a simple list of dependant libraries in the correct order to resolve all external references in your source code.

4.2 Compiling and Linking on Linux

This section applies to Linux systems only.

The Starlink libraries on Linux systems are compiled so that they are compatible with the GNU `gcc/g77` system and the `f2c` compiler. To do this, the `‘-fno-second-underscore’` compiler flag is used.

This means that to compile and link code with the Starlink libraries on Linux systems, you must use the `‘-fno-second-underscore’` flag for the `g77` compiler/linker, thus:

```
% g77 -O -fno-second-underscore myprog.f -o myprog .....
```

4.3 Static Linking

Your application depends upon the BLY library, so when you link (or compile and link – it does not matter), you need to tell it to link with the `libbly.a` library.

In a one-stage compile and link, you would use the following:

```
% f77 -O myprog.f -o myprog -L/star/lib ‘bly_link‘
```

In a two stage compile and link, you would use the following:

```
% f77 -O -c myprog.f
% f77 -O myprog.o -o myprog -L/star/lib ‘bly_link‘
```

To use the ADAM versions of the library, if you are developing an ADAM application, use the following:

```
% alink myprog.f -o myprog -L/star/lib ‘bly_link_adam‘
```

or:

```
% f77 -O -c myprog.f
% alink myprog.o -o myprog -L/star/lib `bly_link_adam`
```

You should also include in the link phase any other library link scripts for those libraries that your application calls directly, and any libraries of your own *e.g.*:

```
% f77 -O myprog.f -o myprog ./libmine.a -L/star/lib \
         `bly_link` `other_lib_link` -lmine2
```

where `` `other_lib_link` `` causes linking with another library using the link script system, and `-lmine2` causes a link with a library `libmine2.a`.

4.4 Dynamic Linking

If you use the link scripts and the `-L/star/lib` tag in your link chain, your executable will be created using the ordinary libraries available in `/star/lib` (unless the linker discovers a shareable version of a library in `/star/lib` which it will use by default).

A statically linked binary includes all the necessary object code. This can create very large binaries, but at at load time, startup is quite fast, because the runtime linker only has to resolve the system library references.

In contrast, dynamically linked executables are much smaller – the linker just notes in the binary which shareable libraries were used to resolve which references, and leaves it at that.

When the binary is loaded for execution, the runtime linker looks for the shared libraries for which it finds references in the binary, and then does a 'fixup' to resolve all the external references using the libraries. For executables with a large list of shared libraries, this process can take a considerable time. What you save in link time during development and in disk space, you may pay for when waiting for the binary to load.

Since your binary will need the shared libraries at install time, your binary will only be portable to systems containing the Starlink shared library set.

Starlink builds its applications statically linked against the Infrastructure libraries, and dynamically linked with the system libraries. The `alink` command for ADAM applications triggers static linking by default.

If you want to take advantage of the speed of dynamic linking and the disk space savings, you can use the shared libraries in `/star/share` to link with:

```
% f77 -O myprog.f -L/star/share `bly_link`
```

There are some caveats (apart from portability already mentioned):

(1) This facility is only available on Intel Linux and SPARC Solaris 2 machines – shareable Infrastructure libraries are not provided for DEC Alpha Digital Unix machines.

(2) At run time, the loader needs to find the shared libraries. On a Starlink system your `LD_LIBRARY_PATH` will have been set to enable this to occur (if you use the Starlink login files). If not, you should add `/star/share` to your `LD_LIBRARY_PATH`.

(3) Some Infrastructure libraries do not have shared versions because it is not possible to generate them, even on Intel Linux and SPARC Solaris 2 systems.