ANL/MCS-TM-208

Users Manual for bfort: Producing Fortran Interfaces to C Source Code

by

William Gropp
Mathematics and Computer Science Division

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Users Manual for bfort: Producing Fortran Interfaces to C Source Code

by

William Gropp

Abstract

In many applications, the most natural computer language to write in may be different from the most natural language to provide a library in. For example, many scientific computing applications are written in Fortran, while many software libraries—particularly those dealing with complicated data structures or dynamic memory management—are written in C. Providing an interface so that Fortran programs can call routines written in C can be a tedious and error-prone process. We describe here a tool that automatically generates a Fortran-callable wrapper for routines written in C, using only a small, structured comment and the declaration of the routine in C. This tool has been used on two large software packages, PETSc and the MPICH implementation of MPI.

1 Introduction

The program described in this report is intended to help you create interfaces to routines written in C that are callable from Fortran. Specifically, the program `bfort` takes files containing C programs and generates a new file for each routine that contains a C routine, callable from Fortran, that will correctly call the original C routine. The interface, often called a wrapper routine or just wrapper, handles the issues of Fortran routine name, variables passed by value instead of reference, and values generated and returned by the C library that are pointers to opaque objects.

2 Getting Started

Using `bfort` is easy. For example, the command

```
bfort foo.c
```

will generate Fortran interfaces for all of the commented routines in the file '`foo.c`'. But before this will do you any good, you will need to add some structured comments to your file.

3 Structured Comments

The program `bfort` searches for C comments of the form `/*`*c* `...` *c*`*/`, where *c* is a single character indicating the type of documentation. The available types include `@` for routines and `M` for C macros. These structured comments can be used by `doctext` [2] to automatically generate manual pages and may containing additional information. In all cases, the structured comment has the form

```
/*@
    name - short description

    heading 1:

    heading 2:

    ...
@*/
```

The structured comment for a routine must immediately precede the declaration of the routine (currently only K&R-style declarations; ANSI-style prototypes will be supported in a later release). Figure 1 shows the structured comment and the routine being documented.

3.1 C Routines

C routines are indicated by the structured comment `/*@ ... @*/`. The program `bfort` uses the name in the first line (`Add` in this example) to generate the name of the routine for which a wrapper is being provided.

3.2 C Macros

C macros are indicated by the structured comment `/*M ... M*/`. Unlike the case of C routines, macro definitions do not provide any information on the types of the arguments. Thus, the body of a structured comment for a C macro should include a *synopsis* section, containing a declaration of the macro as if it were a C routine. For example, if the `Add` example were implemented as a macro, the structured comment for it would look like

```
/*M
   Add - Add two values
```

```
/*@
   Add - Add two values

   Parameters:
.  a1,a2 - Values to add
@*/
int Add( a1, a2 )
int  a1, a2;
{
return a1 + s2;
}
```

Figure 1: C code for an `Add` routine with `bfort`-style structured comment

```
   Parameters:
.  a1,a2 - Values to add

   Synopsis:
   int Add( a1, a2 )
   int a1, a2;
M*/
```

It is important that the word *Synopsis* be used; `bfort` and related programs (`doctext` and `doc2lt`) use this name to find the C-like declaration for the macro.

3.3 Indicating Special Limitations

Two modifiers to the structured comments indicate special behavior of the function. The modifiers must come after the character that indicates a routine, macro, or documentation. The modified `C` indicates that this routine is available only in C (and not from Fortran). If this modifier is present, `bfort` will not generate a wrapper for the routine. For example, a routine that returns a pointer to memory cannot be called from standard Fortran 77, so its structured comment should be

```
/*@C
  ....
@*/
```

The other modifier is `X`; this indicates that the routine requires the X11 Window System. This is used by `bfort` to provide an `#ifdef` around the call in case X11 include files are not available.

The modifiers `C` and `X` may be used together and may be specified in either order (i.e., `CX` or `XC`).

3.4 Indicating Include Files

It is often very important to indicate which include files need to be used with a particular routine. For this purpose, you may add a special structured comment of the form `/*I`*include-file-name*`I*/`. For example, to indicate that the routine requires that '`<sys/time.h>`' has been included, use

```
#include <sys/time.h>          /*I <sys/time.h> I*/
```

in the C file. A user-include can be specified as

```
#include "system/nreg.h"      /*I "system/nreg.h" I*/
```

This approach of putting the structured include comment on the same line as the include of the file ensures that if the source file is changed by removing the include, the documentation and Fortran wrappers will reflect that change.

4 Command Line Arguments

To use `bfort`, you need only to give it the name of the files to process. For example, to process every `.c` and `.h` file in the current directory, use

```
bfort *.[ch]
```

Command-line options to `bfort` allow you to change the details of how `bfort` generates output.

A complete list of the command line options follows. Some of these will be used often (e.g., `-anyname`); others are needed only in special cases (e.g., `-ferr`).

The following option controls the location of the output file:

-dir name Directory for output file

In all cases, the name of the output file is the name of the input file with an `f` suffix. For example, if the input file is '`foo.c`', the file containing the generated wrappers is '`foof.c`'.

The following options control the kind of messages that `bfort` produces about the generated interfaces.

-nomsgs Do not generate messages for routines that cannot be converted to Fortran.

-nofort Generate messages for all routines/macros without a Fortran counterpart.

The following options provide special control over the form of the interface and the generated file.

-anyname Generate a single wrapper that can handle the three most common cases: trailing underscore, no underscore, and all caps. The choice is based on whether one of the following macro names is defined.

FORTRANCAPS Names are upper case with no trailing underscore.

FORTRANUNDERSCORE Names are lower case with trailing underscore.

FORTRANDOUBLEUNDERSCORE Names are lower case, with *two* trailing underscores. This is needed when some versions of `f2c` are used to generate C for Fortran routines. Note that `f2c` uses two underscores *only* when the name already contains an underscore (on at least one FreeBSD system that uses `f2c`). To handle this case, the generated code contains the second underscore only when the name already contains one.

In addition, if `-mpi` is used, the MPI profiling names are also generated, surrounded by `MPI_BUILD_PROFILING`.

-ferr Fortran versions return the value of the routine as the last argument (an integer). This is used in MPI and is a common approach for handling error returns.

-I name Give the name of a file that contains `#include` statements that are necessary to compile the wrapper.

-mapptr Generate special code to convert Fortran integers to and from pointers used by the C routines. The special code is used only if the macro `POINTER_64_BITS` is defined. It is also used to determine whether pointers are too long to fit in a 32-bit Fortran integer. (You have to insert a call to `__RmPointer( pointer )` into the routines that destroy the pointer.) The routines for managing the pointers are in '`ptrcvt.c`'.

-mnative Multiple indirects (`int ***` are native datatypes; that is, there is no coercion to the basic type).

-mpi Recognize special MPI [1] datatypes (some MPI datatypes are pointers by definition).

-ptrprefix name Change the prefix for names of functions to convert to/from pointers. The default value of `name` is `__`.

-voidisptr Consider `void *` as a pointer to a structure.

-ansiheader Generate ANSI-C style headers instead of Fortran interfaces. This is useful in creating ANSI prototypes without converting the code to ANSI prototype form. These use a trick to provide both ANSI and non-ANSI prototypes. The declarations are wrapped in `ANSI_ARGS`, the definition of which should be

```
#ifdef ANSI_ARG
#undef ANSI_ARG
#endif
#ifdef __STDC__
#define ANSI_ARGS(a) a
#else
#define ANSI_ARGS(a) ()
#endif
```

After the command-line arguments come the names of the files for which Fortran interfaces are to be constructed.

5 Examples

This section shows the code generated by `bfort` with various command-line switches.

The command `bfort add.c` produces

```
/* add.c */
/* Fortran interface file for sun4 */
 int  add_( a1, a2)
int*a1,*a2;
{
return Add(*a1,*a2);
}
```

The command `bfort -anyname add.c` produces

```
/* add.c */
/* Fortran interface file */
#ifdef FORTRANCAPS
#define add_ ADD
#elif !defined(FORTRANUNDERSCORE) && !defined(FORTRANDOUBLEUNDERSCORE)
#define add_ add
#endif
 int  add_( a1, a2)
int*a1,*a2;
{
return Add(*a1,*a2);
}
```

The command `bfort -ansiheader foo.c` produces

```
/* add.c */
extern int  Add ANSI_ARGS((int, int ));
```

For a more sophisticated example, here is the result of `bfort -ferr -mpi -mnative -mapptr -ptrprefix MPIR_ -anyname -I pubinc send.c` for the MPI routine `MPI_Send` (implemented in the file `send.c`, from the MPICH implementation). The file '`pubinc`' contains the single line `#include "mpiimpl.h"`.

```
/* send.c */
/* Fortran interface file */
#include "mpiimpl.h"

#ifdef POINTER_64_BITS
extern void *MPIR_ToPointer();
extern int MPIR_FromPointer();
extern void MPIR_RmPointer();
#else
#define MPIR_ToPointer(a) (a)
#define MPIR_FromPointer(a) (int)(a)
#define MPIR_RmPointer(a)
#endif

#ifdef MPI_BUILD_PROFILING
#ifdef FORTRANCAPS
#define mpi_send_ PMPI_SEND
#elif defined(FORTRANDOUBLEUNDERSCORE)
#define mpi_send_ pmpi_send__
#elif !defined(FORTRANUNDERSCORE)
#define mpi_send_ pmpi_send
#else
#define mpi_send_ pmpi_send_
#endif
#else
#ifdef FORTRANCAPS
#define mpi_send_ MPI_SEND
#elif defined(FORTRANDOUBLEUNDERSCORE)
#define mpi_send_ mpi_send__
#elif !defined(FORTRANUNDERSCORE)
#define mpi_send_ mpi_send
#endif
#endif

 void mpi_send_( buf, count, datatype, dest, tag, comm, __ierr )
void             *buf;
int*count,*dest,*tag;
MPI_Datatype     datatype;
MPI_Comm         comm;
int *__ierr;
{
*__ierr = MPI_Send(buf,*count,
        (MPI_Datatype)MPIR_ToPointer( *(int*)(datatype) ),*dest,*tag,
        (MPI_Comm)MPIR_ToPointer( *(int*)(comm) ));
}
```

Note that the `-mapptr` option has caused the generated code to call routines to convert the integers in Fortran to valid pointers. The option `-ptrprefix` changed the names of the routines to be `MPIR_ToPointer` and `MPIR_FromPointer`. The option `-mpi` informed `bfort` that `MPI_Datatype` and `MPI_Comm` were pointers rather than nonpointers. The option `-ferr` converted a routine `MPI_Send` that returns an error code to a Fortran subroutine that returns the error code in the last argument.

6 Installing bfort

The `bfort` program is part of the PETSc package of tools for scientific computing, but can be installed without installing all of PETSc. The program is available from '`info.mcs.anl.gov`' in '`pub/petsc/textpgm.tar.Z`'. Additional information is available through the World Wide Web at http://www.mcs.anl.gov/petsc.

Please send any comments to `gropp@mcs.anl.gov`.

Acknowledgment

The author thanks Lois Curfman McInnes and Barry Smith for their careful reading and vigorous use of the `bfort` manual and program.

References

[1] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[2] William Gropp. Users manual for `doctext`: Producing documentation from C source code. Technical Report ANL/MCS-TM-206, Argonne National Laboratory, March 1995.