f90gl and C Interoperability

A report to J3 on a user's perspective of proposed C interoperability in the current draft of the next Fortran standard

J3/00-121

William F. Mitchell
Mathematical and Computational Sciences Division
National Institute of Standards and Technology
Gaithersburg, MD 20899-8910

February 24, 2000

1 Introduction

The purpose of this report is to provide user feedback on the C interoperability parts of the current draft of the next Fortran standard (J3/00-007 January 13, 2000, henceforth the *proposed standard*). It is hoped that this information will be useful to J3 as they continue the development of this standard.

There are three primary reasons for needing a C interoperability standard in Fortran:

1. writing mixed language programs,
2. calling routines in a Fortran library from a C program, and
3. calling routines in a C library from a Fortran program.

In this report I only consider the third situation. In particular, it is primarily based on an examination of the implementability of the Fortran 90 bindings for the OpenGL library (henceforth, the *bindings*) under the proposed standard. Currently the bindings have been implemented in Fortran 90 in the program f90gl. Such a program is by definition non-standard (since it uses mixed language programming) and relies heavily on preprocessing the Fortran source code, wrapper functions on both sides of the Fortran/C interface, and additional C code. The goals of a new implementation using standard C interoperability are:

1. the bindings are implemented as specified,
2. the interface handles all C interoperability issues for the user,
3. existing programs that use f90gl require no changes,
4. preprocessing of the interface code is not required,
5. no C code is used in the implementation,
6. a minimum number of Fortran wrappers are used, with most of the interface consisting of INTERFACE blocks for the C routines, and
7. the implementation of the interface is portable.

It is not possible to achieve all of these goals under the current proposed standard, nor do I perceive the possibility of achieving all of the goals under any modification of the standard. But it is hoped that some of the suggestions in this report will effect the final version of the standard in such a way that it minimizes the deviation from these goals, and improves the usability of C interoperability for all Fortran/C users.

In Sections 2 and 3 I present all of the interface issues I encountered while designing an implementation of the bindings using C interoperability in the proposed standard. Each subsection addresses one issue. Code snippets are presented for clarification. Usually these consist of a C prototype for the function to which the interface is being designed, Fortran code that would appear in the specification part of the opengl module, and Fortran code that would appear in the subroutine part of that module, the distinction of which should be clear from context. It should also be understood that the code surrounding these snippets will contain any required USE statements. In many cases, this code is believed to represent a correct solution to the stated issue, and is presented to verify that I have not misinterpreted some aspect of the proposed standard. In other cases, the proposed standard is not sufficient to address the issue, and the code snippets either illustrate the problem or present possible solutions under a modification of the proposed standard.

Finally, Section 4 contains a summary of the recommended modifications to the proposed standard.

2 Constants

2.1 Most kind type parameters

The bindings provide named constants to be used as kind type parameters to match Fortran types with C types. Most of these kind type parameters can be defined directly from the ISO_C_BINDING kind type parameters, for example

```
INTEGER, PARAMETER :: GLFLOAT = C_FLOAT
```

The short integers are more problematic as illustrated by GLSHORT. The bindings specify that INTEGER(GLSHORT) corresponds to the C type GLshort (assumed to be a C short) if the processor supports that kind of integer, and GLint otherwise. This can be determined as follows:

```
! short_exists equals 1 if C_SHORT is non-negative and 0 if negative
INTEGER, PARAMETER :: short_exists = (SIGN(1,C_SHORT)+1)/2, &
                      GLSHORT = short_exists*C_SHORT + (1-short_exists)*C_INT
```

The f90gl kind type parameters are not fully portable. Their definition depends on the definition of the corresponding type in OpenGL, which may vary among implementations. It is also remotely possible that the OpenGL library may use a type that is not supported by ISO_C_BINDING (besides the unsigned types). The following entities must be valid kind type parameters: C_INT, C_FLOAT, C_DOUBLE, C_CHAR. Also, C_SIGNED_CHAR must be valid to avoid using C wrappers. However, in practice these issues are not expected to be a problem.

The kind type parameter GLBOOLEAN will still be set to KIND(.TRUE.), which is portable, with the option of changing it "by hand" to the processor's kind type parameter for a one byte logical, because there is no SELECTED_LOGICAL_KIND function.

2.2 GLCPTR

The type GLCPTR is used for storing, but not dereferencing, C pointers. It seems that this could just be a type alias for C_PTR in ISO_C_BINDINGS:

```
TYPEALIAS :: GLCPTR => C_PTR
```

The bindings state that "the derived type TYPE(GLCPTR) is provided for storing C pointers", but I don't see where it makes any difference if it is a type alias instead of a derived type. If there is some difference, then GLCPTR can be

```
TYPE GLCPTR
  TYPE(C_PTR) :: ptr
END TYPE
```

It is common in C libraries for a NULL pointer to be used for special purposes in either input or output of a function. This means that a Fortran user must be able to assign NULL to a variable of TYPE(C_PTR), or at least pass NULL as an actual argument, and must also be able to compare a variable of TYPE(C_PTR) to NULL to check output results. These capabilities could be added to ISO_C_BINDING. The module could contain a symbolic constant C_NULL_POINTER of TYPE(C_PTR) which contains the C NULL value. It could also provide a function of type default logical, C_PTR_IS_NULL, which takes one argument of TYPE(C_PTR) and returns .TRUE. if and only if the argument equals C_NULL_POINTER. (The spelling of these two new entities is, of course, arbitrary.) Or to be more general, the module could overload the == operator to compare any two variables of TYPE(C_PTR) for equality. I assume that the usual intrinsic assignment for variables of derived type apply to TYPE(C_PTR). Thus, while there is no provision for dereferencing a C pointer, they can be copied, compared for equality, set to NULL, and tested for NULL.

The bindings specify that GLNULLPTR is a named constant of TYPE(GLCPTR), the operator == is extended to compare two variables of TYPE(GLCPTR), and the operator == will return .TRUE. if GLNULLPTR is compared to a variable of TYPE(GLCPTR) that has been assigned the C NULL pointer. The additions to ISO_C_BINDING suggested in the previous paragraph would enable this. Otherwise, an implementation of the bindings would either need to make assumptions on what a C pointer is, as is currently done in f90gl, which is not portable, or possibly use some C code to define GLNULLPTR and the function that compares for equality.

2.3 GLUT fonts

GLUT contains several large data structures with the data to draw characters. It is necessary to pass the address of one of these structures to the character drawing function. This should be possible by binding a Fortran variable to the C variable containing the data, and using C_LOC to get the address to be passed. I assume that when a Fortran variable, X, is bound to a C variable with external linkage, Y, then C_LOC(X) is the same as &Y, but the current wording of the proposed standard does not make this clear.

Using the stroke roman font as an example, GLUT contains the code

```
typedef struct {
  const char *name;
  int num_chars;
  const StrokeCharRec *ch;
  float top;
  float bottom;
} StrokeFontRec;

StrokeFontRec glutStrokeRoman = { "Roman", 128, chars, 119.048, -33.3333 };
```

where chars is an array of data containing the strokes for each character.

The symbol GLUT_STROKE_ROMAN, which the user passes to the character drawing function, is defined in glut.h (which the C user #includes) as

```
extern void *glutStrokeRoman;
#define GLUT_STROKE_ROMAN               (&glutStrokeRoman)
```

The interface of the character drawing function is

```
void glutStrokeCharacter(void *font, int character);
```

The Fortran interface is:

```
TYPE, BIND(C) :: strokefontrec
   TYPE(C_PTR) :: name
   INTEGER(C_INT) :: num_chars
   TYPE(C_PTR) :: ch
   REAL(C_FLOAT) :: top, bottom
END TYPE

TYPE(strokefontrec), BIND(C,NAME="glutStrokeRoman") :: GLUT_STROKE_ROMAN

INTERFACE
   BIND(C,NAME="glutStrokeCharacter") SUBROUTINE c_glutstrokecharacter(font,ch)
   TYPE(C_PTR), VALUE :: font
   INTEGER(GLCINT), VALUE :: ch
   END SUBROUTINE c_glutstrokecharacter
END INTERFACE

INTERFACE glutstrokecharacter
   MODULE PROCEDURE f90glutstrokecharacter
END INTERFACE

SUBROUTINE f90glutstrokecharacter(font,ch)
TYPE(strokefontrec), INTENT(IN), TARGET :: font
INTEGER(GLCINT), INTENT(IN) :: ch
CALL c_glutstrokecharacter(C_LOC(font), ch)
END SUBROUTINE f90glutstrokecharacter
```

An example of the use of this function is

```
USE opengl_glut
CALL glutStrokeCharacter(GLUT_STROKE_ROMAN, INT(ICHAR("a"),GLCINT))
```

2.4 Enumerators

OpenGL provides many constants of type GLenum to be passed as actual arguments to the library functions. It would be tempting to define these as a Fortran enumeration, bound to the C enum to insure that the values agree. However, this cannot be done for several reasons. First and foremost is that the OpenGL implementation is not required to use an enum for these constants. Indeed, I know of at least one implementation that uses preprocessor macros. Second, the bindings state that these constants are of type INTEGER(GLENUM), and the use of an enumeration would produce the type TYPE(GLENUM). But in considering the use of an enumeration, two issues were found.

First, there is no binding between a Fortran enumeration and C enum to help avoid errors in the assignment of values to the enumerators. The addition of such a binding, in which the Fortran enumerators get their values from the C enum, would be very useful.

Second, because the enumeration is defined in a single statement, the number of enumerators is limited by the number of continuation lines. Granted, with the increase in number of continuation lines to 99 it is possible to define quite large enumerations, but they are still limited. In the MESA implementation of OpenGL, GLenum contains 644 enumerators, many of which have lengthy names (up to 31 characters), and requires 16679 non-blank characters to define the enumeration. This requires 128 continuation lines of dense text.

One solution for larger enumerations would be to allow the same *type-alias-name* to be used in more than one *enum-def*, with the interpretation that the second instance is a continuation of the previously defined enumeration. However, it may be difficult to uniquely define the enumerator values when they are not provided, especially if the definition spans multiple modules.

Another solution would be to use an ENUM construct, similar in form to the derived type definitions, instead of an ENUM statement. Then continuation lines are not needed. Without providing all the proper BNF, the form is

```
enum-def           is ENUM enum-kind-selector type-alias-name
                        enumerator-list...
                      END [ENUM [type-alias-name]]

enum-kind-selector is ,BIND(C) ::
                   or [kind-selector][::]

enumerator         is named-constant [= scalar-int-initialization-expr]
```

The problem here is that it may not be possible to detect the END, since that could be another enumerator.

3 Procedures

3.1 The easy case

Most OpenGL functions just require an interface block. These interface blocks are contained in the specification part of the opengl module. Note that the interface block contains a generic name because the binding states that all procedure names corresponding to OpenGL procedures are generic names. For example, the interface block for

```
GLint glRenderMode(GLenum mode)
```

is

```
INTERFACE glrendermode
  BIND(C,NAME="glRenderMode") INTEGER(GLINT) FUNCTION glrendermode(mode)
  USE opengl_kinds
  INTEGER(GLENUM), VALUE :: mode
  END FUNCTION glrendermode
END INTERFACE
```

3.2 Short integers

When the short integers are not supported by the processor, f90gl handles the discrepancy by calling the int version of the OpenGL routine instead of the short version, or by converting the type in a C wrapper. Since we wish to avoid C wrappers, consider the first approach. It would be desirable to handle this with a generic interface

```
void glColor3s(GLshort red, GLshort green, GLshort blue)
void glColor3i(GLint   red, GLint   green, GLint   blue)

INTERFACE glcolor3s

   BIND(C,NAME="glColor3s") SUBROUTINE glcolor3s(red,green,blue)
   USE opengl_kinds
   INTEGER(GLSHORT), VALUE :: red, green, blue ! or INTEGER(C_SHORT)
   END SUBROUTINE

   BIND(C,NAME="glColor3i") SUBROUTINE glcolor3i(red,green,blue)
   USE opengl_kinds
   INTEGER(GLINT),VALUE :: red, green, blue
   END SUBROUTINE

END INTERFACE
```

But, if C_SHORT is -1, this would result in either identical interfaces (using GLSHORT) or an illegal kind specification (using C_SHORT). So the discrepancy must be resolved in a Fortran wrapper, unless some other new feature of f2k applies. Using short_exists from Section 2.1, the interface blocks (in the specification part of the module) and wrapper functions (module procedures) look like:

```
PRIVATE
PUBLIC :: glcolor3s, glcolor3i

! generic interface block for module procedure

INTERFACE glcolor3s
   MODULE PROCEDURE f90glcolor3s
END INTERFACE

! generic interface block for the GLINT form, which doesn't need a wrapper

INTERFACE glcolor3i
   BIND(C,NAME="glColor3i") SUBROUTINE glcolor3i(red,green,blue)
   USE opengl_kinds
   INTEGER(GLINT), VALUE :: red, green, blue
   END SUBROUTINE c_glcolor3i
END INTERFACE

! interface block for C procedures

INTERFACE
   BIND(C,NAME="glColor3s") SUBROUTINE c_glcolor3s(red,green,blue)
   USE opengl_kinds
   INTEGER(GLSHORT), VALUE :: red, green, blue
   END SUBROUTINE c_glcolor3s

   BIND(C,NAME="glColor3i") SUBROUTINE c_glcolor3i(red,green,blue)
   USE opengl_kinds
   INTEGER(GLINT), VALUE :: red, green, blue
   END SUBROUTINE c_glcolor3i
END INTERFACE

! wrapper subroutine (after CONTAINS)

SUBROUTINE f90glcolor3s(red,green,blue)
INTEGER(GLSHORT), INTENT(IN) :: red, green, blue
IF (short_exists == 1) THEN
   CALL c_glcolor3s(red,green,blue)
ELSE
   CALL c_glcolor3i(red,green,blue)
ENDIF
END SUBROUTINE
```

Some OpenGL routines that take a short integer do not have an equivalent routine that takes a GLint. For example, glPolygonStipple takes an array of 128 one-byte integers. For these routines, I see no alternative but to use a C wrapper in which the GLint type integer from Fortran is copied to a short integer in C. Recall that this is only necessary when the processor does not support short integers.

3.3 Rank one arrays

Array arguments of rank one can be handled in the interface block using an assumed size array. For example, the interface block for

```
void glVertex2dv(GLdouble *v)
```

is

```
INTERFACE glvertex2dv
   BIND(C,NAME="glVertex2dv") SUBROUTINE glvertex2dv(v)
   USE opengl_kinds
   REAL(GLDOUBLE), DIMENSION(*), INTENT(IN) :: v
   END SUBROUTINE glvertex2dv
END INTERFACE
```

3.4 GLboolean

The OpenGL type GLboolean is defined to be unsigned char, and the corresponding type in the binding is LOGICAL(GLBOOLEAN), where GLBOOLEAN is a named constant with a valid kind type parameter for LOGICAL. This requires a Fortran wrapper to convert the logical value to INTEGER(C_SIGNED_CHAR).

```
void glDepthMask(GLboolean flag)

INTERFACE
   BIND(C,NAME="glDepthMask") SUBROUTINE c_gldepthmask(flag)
   USE ISO_C_BINDING
   INTEGER(C_SIGNED_CHAR), VALUE :: flag
   END SUBROUTINE c_gldepthmask
END INTERFACE

INTERFACE gldepthmask
   MODULE PROCEDURE f90gldepthmask
END INTERFACE

SUBROUTINE f90gldepthmask(flag)
LOGICAL(GLBOOLEAN), INTENT(IN) :: flag
INTEGER(C_SIGNED_CHAR) :: cflag
IF (flag) THEN
   cflag = 1
ELSE
   cflag = 0
ENDIF
CALL c_gldepthmask(cflag)
END SUBROUTINE f90gldepthmask
```

Although the OpenGL Reference Manual says GLboolean is an unsigned char, and every OpenGL implementation I have seen uses this type, the OpenGL Specification states that it can be any type that contains at least one bit. If an OpenGL implementation uses a different type, the above implementation will fail.

If INTEGER(C_SIGNED_CHAR) is not supported by the processor, then I don't see any way of avoiding the use of a C wrapper to convert the argument to GLboolean. One would probably use INTEGER(C_INT) for passing the argument between the Fortran and C wrappers.

3.5 Character string arguments

INTENT(IN) character arguments require a Fortran wrapper to append the null character.

```
void glutAddMenuEntry(char *name, int value)

INTERFACE
   BIND(C,NAME="glutAddMenuEntry") SUBROUTINE c_glutaddmenuentry(name,value)
   CHARACTER(LEN=*,KIND=C_CHAR), INTENT(IN) :: name
   INTEGER(GLCINT), VALUE :: value
   END SUBROUTINE c_glutaddmenuentry
END INTERFACE

SUBROUTINE f90glutaddmenuentry(name,value)
CHARACTER(LEN=*,KIND=C_CHAR), INTENT(IN) :: name
INTEGER(GLCINT), INTENT(IN) :: value
CALL c_glutaddmenuentry(name//C_NULL_CHAR, value)
END SUBROUTINE f90glutaddmenuentry
```

3.6 Character string function result

Some OpenGL functions return a pointer to a null-terminated (sometimes static) string. The Fortran bindings for these functions return a pointer to an array of type CHARACTER(LEN=1), with the stipulation that the pointer is allocated in the interface and can be deallocated by the user. The last statement requires that a Fortran wrapper be used, but even without the stipulation it does not appear that these functions interoperate. However, I find the wording in section 16.2.1 of the proposed standard to be confusing, and am not sure when one can interoperate with a C string. For a C function

```
char *c_func(void)
```

(assume it returns a string of 10 characters) it is not clear to me that either of the following forms are legal.

```
CHARACTER(LEN=10) :: str1
CHARACTER(LEN=1), DIMENSION(10) :: str2
str1 = c_func()
str2 = c_func()
```

Also, there is the problem of not knowing the length of the C string until after the function call, so one does not know how large to allocate the array (or what to use as the length parameter of a character variable). These same problems exist with an INTENT(OUT) character string argument. I just don't see how character strings of unknown length can be returned from C.

The workaround for this requires the use of two C wrappers, one to return the length of the string and a second to return the string. For the OpenGL function

```
const GLubyte * glGetString(GLenum name)
```

(GLubyte is unsigned char), the interface is

```
void f90glGetStringLen(GLenum name, GLint *length, const GLubyte *str)
{
   str = glGetString(name);
   length = strlen(str);
}
```

```
void f90glGetString(const GLubyte *str, GLubyte *fort_str, GLint length)
{
   int i;
   for (i=0; i<length; i++) fort_str[i] = str[i];
}

INTERFACE
   BIND(C,NAME="f90glGetStringLen") SUBROUTINE c_getstringlen(name,l,s)
   USE opengl_kinds
   USE ISO_C_BINDING
   INTEGER(GLENUM), VALUE :: name
   INTEGER(GLINT), INTENT(OUT) :: l
   TYPE(C_PTR), INTENT(OUT) :: s
   END SUBROUTINE c_getstringlen

   BIND(C,NAME="f90glGetString") SUBROUTINE c_getstring(c_str,f_str,len)
   USE opengl_kinds
   USE ISO_C_BINDING
   TYPE(C_PTR), INTENT(IN) :: c_str
   CHARACTER(KIND=C_CHAR,LEN=1), DIMENSION(*), INTENT(OUT) :: f_str
   INTEGER(GLINT), VALUE :: len
   END SUBROUTINE c_getstring
END INTERFACE

FUNCTION f90glgetstring(name) RESULT(res)
INTEGER(GLENUM), INTENT(IN) :: name
CHARACTER(LEN=1), DIMENSION(:) :: res
TYPE(C_PTR) :: c_str
INTEGER(GLINT) :: length
CALL c_getstringlen(name,length,c_str)
ALLOCATE(res(length))
CALL c_getstring(c_str,res,length)
END FUNCTION f90glgetstring
```

3.7 Argv

glutInit takes the standard C argc and argv. In the Fortran interface, they are optional arguments with argv being an array of character strings with one command line argument per array entry. This matches with argument_text from the main program and SIZE(argument_text), but a Fortran wrapper is required to convert this to a form that interoperates with the C form. The C form is an array of length argc of pointers to null terminated character strings, which can be realized as an array of TYPE(C_PTR) using C_LOC to get the addresses of the character strings. Because of the need to add the null terminator, the strings must be copied to character strings of length SIZE(argument_text)+1. This copy could be avoided if Fortran used null termination, i.e., appended the null character at position LEN(string)+1 for all strings. The copy could also be avoided by having the length of argument_text be one larger than the minimum required to hold the longest command line argument, in which case C_NULL_CHAR can be added in place.

```
void glutInit(int *argcp, char **argv)

INTERFACE
   BIND(C,NAME="glutInit) SUBROUTINE c_glutinit(argcp,argv)
   USE ISO_C_BINDING
   INTEGER(C_INT), INTENT(INOUT) :: argcp
   TYPE(C_PTR), INTENT(INOUT), DIMENSION(*) :: argv
   END SUBROUTINE c_glutinit
END INTERFACE

SUBROUTINE f90glutinit(argcp,argv)
! ignore the fact that the arguments are optional for this example
INTEGER(GLCINT), INTENT(IN) :: argcp
CHARACTER(LEN=*), INTENT(IN), DIMENSION(*) :: argv
INTEGER(GLCINT) :: local_argc, i
TYPE(C_PTR), DIMENSION(SIZE(argv)) :: c_argv
CHARACTER(LEN=LEN(argv)+1), DIMENSION(SIZE(argv)) :: long_argv
local_argc = argcp
long_argv = argv
DO i=1,SIZE(argv)
   long_argv(i)(LEN_TRIM(argv(i))+1:LEN_TRIM(argv(i))+1) = C_NULL_CHAR
END DO
c_argv = C_LOC(long_argv) ! is C_LOC elemental?
CALL c_glutinit(local_argc,c_argv)
END SUBROUTINE f90glutinit
```

3.8 Arrays allocated by C

Often a C library routine will take a pointer which is allocated as an array, the size of which is not known until the routine is executed, to return values to the calling procedure. This situation does not arise in OpenGL, but does in another library for which I have written a Fortran interface. For example,

```
void get_list(int *list, int size) {
/* perform calculations to determine and set size */
list = (int *) malloc(size*sizeof(int));
/* fill the list with values */
}
```

I do not see any way to interface with this under the proposed standard. The semantics do not seem unreasonable; the address of the array is returned by the C function. This may incur a performance penalty, but it should be doable provided there is syntax that indicates this is a different kind of array. Perhaps allowing both BIND(C) and ALLOCATABLE to be attributes of the same array could signal that this is an array that must be allocated by a means other than Fortran.

The approach I currently use for this situation is a hack that is not portable, not supported by the proposed standard, and possible only because I have the ability to modify the C library. `list` is declared as a Fortran pointer, and the malloc statement is replaced by a call to a Fortran routine that allocates the pointer. In general, this is unacceptable.

3.9 Void* arguments

Several OpenGL routines accept an argument of type void* which can be any of several types, and another argument of type GLenum which specifies what type is being passed. To allow different types to be passed, the void* argument can be declared TYPE(C_PTR) with C_LOC giving the address to pass. This will require Fortran wrapper functions with a generic interface so that the user passes the actual argument of whatever type.

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *lists)

INTERFACE
   BIND(C,NAME="glCallLists") SUBROUTINE c_glcalllists(n,type,lists)
   USE opengl_kinds
   USE ISO_C_BINDING
   INTEGER(GLSIZEI), VALUE :: n
   INTEGER(GLENUM), VALUE :: type
   TYPE(C_PTR), VALUE :: lists
   END SUBROUTINE glcalllists
END INTERFACE

INTERFACE glcalllists
   MODULE PROCEDURE shortglcalllists, &
                      intglcalllists, &
                    floatglcalllists
END INTERFACE

SUBROUTINE shortglcalllists(n,type,lists)
INTEGER(GLSIZEI), INTENT(IN) :: n
INTEGER(GLENUM), INTENT(IN) :: type
INTEGER(GLSHORT), DIMENSION(*), INTENT(IN) :: lists
CALL c_glcalllists(n,type,C_LOC(lists))
END SUBROUTINE shortglcalllists

SUBROUTINE intglcalllists(n,type,lists)
INTEGER(GLSIZEI), INTENT(IN) :: n
INTEGER(GLENUM), INTENT(IN) :: type
INTEGER(GLINT), DIMENSION(*), INTENT(IN) :: lists
CALL c_glcalllists(n,type,C_LOC(lists))
END SUBROUTINE intglcalllists

SUBROUTINE floatglcalllists(n,type,lists)
INTEGER(GLSIZEI), INTENT(IN) :: n
INTEGER(GLENUM), INTENT(IN) :: type
REAL(GLFLOAT), DIMENSION(*), INTENT(IN) :: lists
CALL c_glcalllists(n,type,C_LOC(lists))
END SUBROUTINE floatglcalllists
```

Like the discussion about short integers in Section 3.2, this will not work if short integers are not supported by the processor. But unlike that discussion, we do not have multiple subroutines to select from at run time. I do not see how to get around this one without using conditional compilation. It would be useful to have compilers be required to support Part 3 of the Fortran standard, CoCo.

Using conditional compilation, we only need to remove the short version from the generic interface block. It is not necessary to remove the short version of the subroutine, but may be desirable to reduce the size of the executable. The generic interface block with conditional compilation is:

```
INTERFACE glcalllists
   MODULE PROCEDURE &
?? if (short_exists) then
      shortglcalllists, &
?? endif
        intglcalllists, &
      floatglcalllists
END INTERFACE
```

Note, however, that here short_exists is a preprocessor variable. To define this within the program requires that the conditional compilation facility be extended to allow the use of named constants from intrinsic modules in a *coco-initialization-expr*. The alternative is to use the same process employed by the current version of f90gl: compile and execute a program that determines whether or not the short integers are supported, and writes appropriate lines into the preprocessor's initialization file (CoCo SET). This process can be built into a Unix makefile or DOS batch file, but may not be appropriate for other compilation schemes. Another alternative is to set these variables "by hand" in the initialization file.

3.10 Callback functions

Many C libraries, including GLU and GLUT, use callback functions. These are functions that are registered by sending a pointer to the function to a routine which saves the pointer so that the function can be called at a later time. For example, in GLUT the function is called when some event occurs, such as a mouse button being pressed. C_LOC should provide the capability to implement this. The bindings indicate that dummy procedure arguments are passed in the same manner as procedure arguments in Fortran, so a wrapper function for the registration will be required to convert the argument to TYPE(C_PTR). Moreover, the specification of C_LOC says that if the argument is a procedure then it must have the BIND(C) attribute. The bindings do not impose this restriction, but this can be handled by using a reverse-wrapper function that is registered instead of the user's function, and using a procedure pointer to invoke the user's function. For example:

For the C prototype

```
void glutMouseFunc(void (*func)(int button, int state, int x, int y))
```

the specification part of the module contains

```
INTERFACE PROCEDURE()
   SUBROUTINE spec_mousefunc(button,state,x,y)
   USE opengl_kinds
   INTEGER(GLCINT), INTENT(IN) :: button, state, x, y
   END SUBROUTINE func
END INTERFACE

PROCEDURE(spec_mousefunc), POINTER, SAVE :: user_mousefunc

INTERFACE
   BIND(C,NAME="glutMouseFunc") SUBROUTINE c_glutmousefunc(func)
   USE ISO_C_BINDING
   TYPE(C_PTR), VALUE :: func
   END SUBROUTINE c_glutmousefunc
END INTERFACE

INTERFACE glutmousefunc
   MODULE PROCEDURE f90glutmousefunc
END INTERFACE
```

and the module subprogram part contains

```
SUBROUTINE f90glutmousefunc(func)
PROCEDURE(spec_mousefunc) :: func
user_mousefunc => func
CALL c_glutmousefunc(C_LOC(wrapper_mousefunc))
END SUBROUTINE f90glutmousefunc

BIND(C) SUBROUTINE wrapper_mousefunc(button,state,x,y)
INTEGER(GLCINT), VALUE :: button, state, x, y
CALL user_mousefunc(button, state, x, y)
END SUBROUTINE wrapper_mousefunc
```

Unresolved issue 151 asks if the BIND *prefix-spec* should be allowed for module subprograms (currently a constraint prohibits it). If this change is not made, then the reverse-wrappers (or any procedure argument to C) must be external subprograms, which is highly undesirable.

3.11 Multiple callback functions

The callback functions in GLUT are further complicated by the fact that GLUT supports multiple simultaneous graphics windows, and each window has its own set of callback functions. The current window is only known within GLUT, not in the Fortran interface, so GLUT provides a pair of set/get functions for the Fortran interface to use for specifying and retrieving the callback functions for the current window. The prototypes are

```
void __glutSetFCB(int which, void *func);
void* __glutGetFCB(int which);
```

The set function can be called from the Fortran wrapper of the routine that sets the callback function. But the get function, which should be called from the reverse wrapper to obtain a pointer to the user's callback function, cannot be called from Fortran under the proposed standard. The additional capability needed here is interoperability with a function whose return value is a pointer to a function (in the C sense) to get a procedure pointer (in the Fortran sense). The alternative is to use reverse-wrappers in C instead of Fortran.

3.12 GLUTNULLFUNC

The bindings specify that the symbol GLUTNULLFUNC will be provided to be passed as the actual argument in cases where a C program would pass NULL to a pointer to a function. Since Fortran does not use "procedureness" as a disambiguator, GLUTNULLFUNC must be a procedure. ASSOCIATED can be used to determine if the actual argument is GLUTNULLFUNC, however this means that the dummy argument must be a procedure with an implicit interface. Using the same example as section 3.10, and assuming the existence of C_NULL_POINTER recommended in section 2.2, this adds the (public) module subroutine

```
SUBROUTINE GLUTNULLFUNC()
END SUBROUTINE GLUTNULLFUNC
```

and changes the Fortran wrapper to

```
SUBROUTINE f90glutmousefunc(func)
PROCEDURE() :: func
PROCEDURE(), POINTER :: func_ptr
func_ptr => func
IF (ASSOCIATED(func_ptr,GLUTNULLFUNC)) THEN
   NULLIFY(user_mousefunc)
   CALL c_glutmousefunc(C_NULL_POINTER)
ELSE
   user_mousefunc => func
   CALL c_glutmousefunc(C_LOC(wrapper_mousefunc))
ENDIF
END SUBROUTINE f90glutmousefunc
```

The *abstract-interface-name* spec_mousefunc must also be removed from the declaration of user_mousefunc. This removes all uses of the *abstract-interface-name*.

3.13 GLU tesselator

This is an example of a struct in the GLU library with a corresponding structure in the Fortran bindings, however they do not interoperate. The Fortran structure stores a pointer to the C structure (which is returned by the creation routine) and procedure pointers for any registered callback functions, which are associated with a particular tesselator.

```
INTERFACE PROCEDURE()
   SUBROUTINE spec_tess_begin(type)
   USE opengl_kinds
   INTEGER(GLENUM), INTENT(IN) :: type
   END SUBROUTINE spec_tess_begin

   SUBROUTINE spec_tess_edge_flag(flag)
   USE opengl_kinds
   LOGICAL(GLBOOLEAN), INTENT(IN) :: flag
   END SUBROUTINE spec_tess_edge_flag

   ...

END INTERFACE

TYPE glutesselator
   TYPE(C_PTR) :: object
   PROCEDURE(spec_tess_begin), POINTER :: user_tess_begin
   PROCEDURE(spec_tess_edge_flag), POINTER :: user_tess_edge_flag
   ...
END TYPE
```

The bindings specify that the creation routine returns a Fortran pointer to the structure:

```
GLUtesselator* gluNewTess(void)

INTERFACE
   BIND(C,NAME="gluNewTess") FUNCTION c_glunewtess()
   USE ISO_C_BINDING
   TYPE(C_PTR) :: c_glunewtess
   END FUNCTION c_glunewtess
END INTERFACE

FUNCTION f90glunewtess()
TYPE(glutesselator), POINTER :: f90glunewtess
ALLOCATE(f90glunewtess)
f90glunewtess%object = c_glunewtess()
IF (f90glunewtess%object == C_NULL_POINTER) THEN ! or however that is checked
   DEALLOCATE(f90glunewtess)
ELSE
   NULLIFY(f90glunewtess%user_tess_begin,...)
ENDIF
END FUNCTION f90glunewtess
```

The callback registration for a tesselator is different from other callback registrations, in that the dummy procedure may have one of several interfaces, which requires that the interface be implicit.

```
void gluTessCallback(GLUtesselator* tess, GLenum which, GLvoid(*CallBackFunc)())
```

```
INTERFACE
   BIND(C,NAME="gluTessCallback") SUBROUTINE c_glutesscallback(tess,which,func)
   TYPE(C_PTR), VALUE :: tess
   INTEGER(GLENUM), VALUE :: which
   TYPE(C_PTR), VALUE :: func
   END SUBROUTINE c_glutesscallback
END INTERFACE

SUBROUTINE f90glutesscallback(tess,which,func)
TYPE(glutesselator), POINTER :: tess
INTEGER(GLENUM), INTENT(IN) :: which
PROCEDURE() :: func
TYPE(C_PTR) :: func_loc
SELECT CASE(which)
   CASE(GLU_TESS_BEGIN)
      tess%user_tess_begin => func
      func_loc = C_LOC(wrapper_tess_begin)
   CASE(GLU_TESS_EDGE_FLAG)
   ...
END SELECT
CALL c_glutesscallback(tess%object,which,func_loc)
END SUBROUTINE f90glutesscallback
```

However, the tesselator object, in which the pointer to the callback function is stored, does not get passed to the callback function, so this does not provide access to the pointer to the user's callback function. This requires using a module variable

```
TYPE(glutesselator), POINTER, SAVE :: current_tess
```

which get assigned to the tesselator argument of every call to a GLU procedure that uses tesselators,

```
current_tess => tess
```

and then current_tess%user_tess_begin (for example) is called in the reverse-wrapper.

Note that the edge_flag callback function has a LOGICAL argument, which does not interoperate with any C type. The type transformation occurs in the reverse-wrapper.

```
BIND(C) SUBROUTINE wrapper_tess_edge_flag(flag)
INTEGER(C_SIGNED_CHAR), VALUE :: flag
IF (flag == 0) THEN
   CALL current_tess%user_tess_edge_flag(.FALSE._GLBOOLEAN)
ELSE
   CALL current_tess%user_tess_edge_flag( .TRUE._GLBOOLEAN)
ENDIF
END SUBROUTINE wrapper_tess_edge_flag
```

Some of the callback functions contain void* arguments. For these, the procedure pointers must have an implicit interface to avoid conflicts between the *proc-interface* and the interface of the actual argument provided by the user. The corresponding argument in the reverse-wrapper is more problematic. There is no way to determine the actual type of the void* argument. The only approach I can think of is to use a non-conformant implementation in which the argument is declared as (say) an assumed size rank one integer array, and hope that the compiler simply passes the address along.

```
void vertex(void *vertex_data)
```

```
TYPE glutesselator
   TYPE(C_PTR) :: object
   ...
   PROCEDURE(), POINTER :: user_tess_vertex
   ...
END TYPE

BIND(C) SUBROUTINE wrapper_tess_vertex(vertex_data)
! In the C routine that calls this, vertex_data is void*.
! In the user routine being called, vertex_data may be integer or real
! (or possibly another type) and may be a scalar or an assumed size array.
! Note that the interface to user_tess_vertex is implicit.
INTEGER(GLINT), INTENT(IN), DIMENSION(*) :: vertex_data
CALL current_tess%user_tess_vertex(vertex_data)
END SUBROUTINE wrapper_tess_vertex
```

4 Summary

This section contains a summary of the suggested changes to the proposed standard. It is realized that some of these may be technically impossible (e.g. binding a Fortran enumeration to a C enum) and some may be politically impossible (e.g. changes to Part 3 of the standard). Nevertheless, summarizing all of the issues encountered seems worthwhile.

- Add C_NULL_POINTER to ISO_C_BINDINGS and provide the ability to compare an entity of TYPE(C_PTR) for equality with C_NULL_POINTER.
- Change the wording to make it clear that if a Fortran variable, X, is bound to a C variable, Y, then C_LOC(X) is the same value as &Y.
- Allow a Fortran enumeration to be bound to a C enum to insure that the enumerators have the same values.
- Change the syntax or semantics of an enumeration declaration in such a way that the number of enumerators is not limited by the limit on the number of continuation lines.
- Clarify the wording about interoperating with C strings.
- Provide a means by which a character string of unknown length can be returned from a C function, both as an argument and as a function result.
- Have the length of argument_text be one larger than the minimum required to hold the longest command line argument.
- Provide a means by which an array allocated in a C function can be returned to a calling Fortran procedure.
- Require that processors support Part 3 of the standard.
- Allow the use of named constants from intrinsic modules in CoCo initialization expressions.
- Allow the BIND *prefix-spec* on module procedures.
- Provide a means by which a C pointer to a function can be returned to Fortran and assigned to a procedure pointer.