GUIDELINES FOR BINDINGS TO FORTRAN 90
Version of July 1993

1 Introduction

1.1 Purpose
This document has been prepared by the ISO/IEC JTC1/SC22/WG5 (Fortran) committee to aid those designing a binding for a functional standard to the ISO Fortran programming language, informally known as Fortran 90 (ref 9). The status of the document is a WG5 paper offering recommendations and advice; it is not intended to become a formal standard or a mandatory requirement.

Fortran 90 incorporates many new and more powerful features relative to its predecessor, informally known as Fortran 77 (ref 11). For example new facilities allow the programmer to extend the language with new data types and operations and allow the specification of much simpler user interfaces to external procedures. The purpose of this note is to draw these features to the attention of designers of bindings and to recommend ways in which they might be used in keeping with the aims of the language designers; this is not intended as a comprehensive overview of features new in Fortran 90.

At the same time it is emphasized that, apart from the minor points listed in Annex A, the Fortran 90 language contains Fortran 77 as a subset. Therefore in general any binding designed for use by a Fortran 77 program may be accessed also by a Fortran 90 program.

1.2 Scope
It is assumed that the reader has access to the ISO Technical Report on Guidelines for Language Bindings (ref 7) and to the ISO Fortran 90 standard, abbreviated below as GLB and F90 respectively.

While this document draws attention to a few salient features of the Fortran 90 language, the reader is referred to the Fortran standard for the full definition. A number of text books specifically on Fortran 90 have been published which may be of help; those known to WG5 at the time of writing are shown in part two of the bibliography (1.4.2).

The document concentrates on the user interface and makes few recommendations on the detailed programming of the functional code of any binding since this need not be coded wholly or even partly in Fortran.

Note should be taken of development work in SC22/WG11 on language-independent arithmetic, data types and procedure-calling mechanisms (refs 4-6) which may affect bindings to Fortran 90 in future. In view of GLB guideline 5 they are not further considered here.

1.3 Levels of Standard
Fortran 90 is a unitary standard; there are no options or optional levels of conformance to

the standard (cf GLB guideline 25). There is an optional auxiliary standard in preparation (ref 12); this is a definition of a set of functional extensions to Fortran 90 which includes a non-normative binding using Fortran 90. It does not affect the definition of the Fortran 90 language.

1.4 Bibliography

1.4.1 Standards, including work in development

1. Information Processing Systems - Computer Graphics - Graphics Kernel System (GKS) language bindings - Part 1: Fortran, ISO/IEC 8651/1.
2. Information Processing Systems - Computer Graphics - Three-Dimensional Extensions to GKS language bindings - Part 1: Fortran, ISO/IEC 8806/1.
3. Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 1: Fortran, ISO/IEC 9593/1.
4. Information Technology - Programming Languages - Common Language-Independent Arithmetic, CD 10967 (1992).
5. Information Technology - Programming Languages - Common Language-Independent Datatypes, CD 11404 (1992).
6. Information Technology - Programming Languages - Common Language-Independent Procedure Calling Mechanism, Working Draft (ISO/SC22/WG11 N295, Dec 13, 1991).
7. Information Technology - Programming Languages, their Environments and System Software Interfaces - Guidelines for Language Bindings, TR 10182 (1993).
8. Posix Language Bindings/Fortran Language (in development).
9. Programming Languages - Fortran (Second Edition) ISO/IEC 1539:1991.
10. Programming Languages - Fortran (Second Edition) Technical Corrigendum (in preparation).
11. Programming Languages - Fortran ISO/IEC 1539:1980.
12. Varying Length Character Strings in Fortran, ISO/IEC CD 1539-2 (1992).
13. Terms of Reference for a SC22 Ad Hoc Group on Cross-language coordination, ISO/IEC JTC1/SC22 N1235 (September 1992).
14. Information Technology - 8-bit Single-byte Coded Graphic Character Sets ISO 8859, Parts 1 to 4: Latin alphabets 1 to 4; Part 5: Latin/Cyrillic; Part 6: Latin/Arabic; Part 7: Latin/Greek; Part 8: Latin/Hebrew; Part 9: Latin alphabet 5.
15. Computer Graphics Interface/Fortran Binding, ANSI X3H34/86-7.
16. Problems of Language Bindings, B.L.Meek, (ISO/SC22/N1278).
17. High Performance Fortran Language Specification, Version 1.0, May 3, 1993, High Performance Fortran Forum; reproduced in Scientific Programming, vol 2, no 1, June 1993.
18. Parallel Extensions for Fortran, ANSI X3H5/93-SD2-Revision A (April 2, 1993)

1.4.2 Fortran 90 texts

1. Fortran 90 - Counihan, Pitman, 1991, ISBN 0-273-03073-6.
2. Fortran 90; Approche par la Pratique - Lignelet, Série Informatique Éditions, Menton, 1993, ISBN 2-090615-01-4.
3. Fortran 90: eine informelle Einführung - Heisterkamp, BI-Wissenschaftsverlag, 1991, ISBN 3-411153-21-0.
4. Fortran 90 Explained - Metcalf and Reid, Oxford University Press, 1992,

ISBN 0-19-853772-7 (also available in French, Japanese and Russian translation).
5. Fortran 90 Handbook - Adams, Brainerd, Martin, Smith and Wagener, McGraw-Hill, 1992, ISBN 0-07-000406-4.
6. Fortran 90; Initiation à partir du Fortran 77 - Aberti, Série Informatique Éditions, Menton, 1992, ISBN 2-090615-00-6.
7. Fortran 90 Referenz-Handbuch: der neue Fortran-Standard - Gehrke, Carl Hansen Verlag, 1991, ISBN 3-446163-21-2.
8. Programmer's Guide to Fortran 90 - Brainerd, Goldberg and Adams, McGraw-Hill, 1990, ISBN 0-07-000248-7.
9. Programmieren in Fortran - Langer, Springer Verlag, 1993, ISBN 0-387-82446-4.
10. Programmierung in Fortran 90 - Schobert, Oldenburg, 1991.
11. Programming in Fortran 90 - Morgan and Schonfelder, Blackwell Scientific Publications, 1993, ISBN 0-632028-38-6.
12. Software Entwicklung in Fortran 90 - Überhuber and Meditz, Springer Verlag, 1993, ISBN 0-387-82450-2.

1.5 Acknowledgement

The cooperation of Don Nelson, project editor of GLB, in providing a machine-readable copy of a WG11 working draft of GLB, is gratefully acknowledged.

2 Binding Methods

2.1 Definition of Binding Methods

The five methods of binding discussed in GLB are:

1. Provide a completely defined procedural interface (the System Facility Standard Procedural Interface).
2. Provide a procedural interface definition language (User-Defined Procedural Interface).
3. Provide extensions to the programming language, using native syntax.
4. Allow alien syntax to be embedded in the programming language.
5. Bind pre-existing language elements.

More recent work on cross-language coordination (ref 13) has suggested two additional methods:

6. Direct normative reference from the language standard to the cross-language facility standard.
7. Direct inclusion of (part of) the text of the cross-language facility standard in the language standard.

2.2 Advantages and Disadvantages of Binding Methods in General

This section reproduces parts of section 2 of GLB to demonstrate the advantages and disadvantages of binding methods in general terms. GLB section 2 contains fuller discussion of the various methods.

[From GLB 2.1 - Introduction] *A pre-processor is necessary for Method 4 above, and optional for Method 3. Method 1 does not require a pre-processor but it may be useful to provide a utility that checks the syntax of all the procedure calls. The function of a pre-processor is to scan a program source text, to identify alien syntax or syntax associated with a given facility, and to replace this text by host language constructs (for example, calls to system functions) that can be compiled by the standard compiler.*

The advantages of a pre-processor are:

- *A pre-processor can often carry out semantic checking not provided by the language compilers.*
- *A pre-processor can be independent of the particular language compiler.*
- *A pre-processor approach avoids problems that result from tampering with an existing language standard or with certified compilers.*
- *If the system facility is enhanced, it is easier to modify a pre-processor than a full compiler.*

The disadvantages are:

- *A pre-processor requires an extra pass through the source.*
- *There may be a problem with multiple pre-processors for different system facilities existing in the same environment.*
- *A pre-processor may produce code unfamiliar to the programmer and make debugging more difficult - for example, it may change statement numbers.*
- *Depending on the language extensions, it may be necessary to analyze the syntax of most of the language to detect the code to be replaced.*

There is often more than one way to implement a given method. In addition, it may be necessary to implement more than one method for any given facility.

[From GLB 2.2 - Method 1] *Method 1 is appropriate when the syntax of the interface provided for each system function is fairly simple and can be fully defined by a few parameters. The method can become unwieldy when the functions that can be invoked use a large number of data types whose structure may be unknown until the time of invocation, and require parameters or data types that are unknown in structure until the time of invocation.*

Use of Method 1 requires that the procedural interface be redefined for each programming language, in terms of the syntax and data types of that language. Thus, separate language binding standards to the same functional interface standard are created.

Method 1 has been used by GKS and other graphics draft standards, where the syntax of the parameters is fairly simple.

It should be noted that, if languages used a common procedure calling mechanism and equivalent sets of data types (ISO/IEC JTC1 has assigned work items on these topics to SC22/WG11), then it would be possible to derive system facility standard procedural interfaces from the abstract definitions. It would also be possible to derive system facility standard procedural interfaces from abstract definitions under other conditions, particularly for languages of sufficient abstraction (like Pascal and Ada).

[From GLB 2.3 - Method 2] *Method 2 allows the binding document to be easily adapted to different programming languages, since the binding only deals with data types. The naming of procedures and parameters is done by the user and not the binding specifiers. The procedural interface definitions are compiled and the resulting object module must be linked both to the application program and to the system facility.*

Advantages of Method 2 are:

- *It may provide early diagnosis of errors.*
- *It is processed once and may allow specific optimization (for example, optimization of query*

searches) leading to run-time economies.

- *Modules may be shared among application programs, since they exist independently.*
- *The task of creating modules may be specialized and managed outside of the user's program.*

Disadvantages of Method 2 are:

- *The definition of modules is an extra design step and risks poor usability when the programmer has to define his own modules.*
- *The procedural interface definition language is another language to learn unless the procedural interface language is part of the host language already.*
- *There is generally an administrative overhead for managing modules to ensure that they get recompiled and relinked when necessary.*
- *Porting an application involves porting the program and all the referenced procedural interface definition language modules.*
- *An additional compiler has to be provided for the procedural interface language unless the procedural interface language is part of the host language already.*

[From GLB 2.4 - Method 3] *This method is viable only when the system facility is stable and when the application requirements are well understood, since the cost of changes to programming language standards is high.*

The main advantage is usability. The users of the language have little extra to learn except the new facilities. It also allows the language developers, when defining new versions of the language, to choose a conforming subset of the facilities or to change the appearance of existing language facilities if they believe this is helpful to their users. Another advantage is that new data types appropriate to the system facility can be constructed.

The disadvantages are that Method 3 ties a compiler to a particular system facility definition. It also ties the language specification to that of the system facility, making it highly desirable to process the standardization of both specifications together if enhancements are needed. It may also be more difficult to use this method in a mixed-language environment, since the same facilities may have confusingly different appearances in different host languages.

[From GLB 2.5 - Method 4] *The advantage of Method 4 over Method 2 is that simple programs, particularly those that may have a short life, may be easier to create. The advantage of Method 4 over Method 3 is that the independence of host language specifications from system facility specifications is maintained, so development of each can progress more quickly.*

The disadvantage of Method 4 over Method 2 is that this method substantially complicates the relationships between applications and system facilities. Although the alien syntax should be very similar for all host languages, the pre-processor will need to 'know about' the conventions of each host language to be able to generate the correct interfacing code.

The disadvantage of Method 4 compared with Method 3 is that the programmer has to know two languages and may be confused by the differences between them.

[From GLB 2.6 - Method 5] *The advantage* [of method 5] *is that pre-existing constructs are used and no extra work in binding needs to be done. If that facility is already present in the language, then making use of that facility avoids unnecessary perturbations to the language.*

Care should be taken that the language construct fully meets the requirements of the system facility.

2.3 Applicability of Fortran 90 Facilities to Binding Methods

WG5 considers that the *MODULE* offers the Fortran user such powerful facilities that it will become an intrinsic part of Fortran 90 usage. Therefore it is not useful to discuss the binding methods of GLB in this document without incorporating the concepts of modules into each method as appropriate. Thus for example the simple procedural interface in the style of Fortran 77 usage is assumed to be enhanced to involve procedural interfaces used in conjunction with the language extension facilities of modules.

3 Principal Features of Fortran 90 Applicable to Bindings

This section describes the principal features of Fortran 90 which are relevant to bindings, concentrating on those facilities that are different from Fortran 77.

3.1 Modules

The module is a concept new to Fortran in Fortran 90. It provides a number of new functionalities well suited to bindings and is described first to avoid continual forward references.

A *MODULE* is a program unit that may hold data, procedures, procedure interfaces and type definitions, all of which may be referenced by other program units. A module is not itself executable, but it may contain executable procedures. Other program units, including other modules, nominate the modules they reference by the *USE* statement (F90 section 11.3.2).

Modules permit the packaging of data (of both intrinsic and derived type), operations and procedures together with relevant information-hiding. They permit tighter control of name-spaces, removing the uncertainties and increasing the flexibility of naming procedures in a binding. They permit compile-time checks on the correctness of invocations of procedures. They therefore increase the reliability of access to the procedures and to the data forming a binding.

Readers unfamiliar with the concepts of modules are strongly urged to study the Varying Length Character Strings in Fortran module definition (ref 12), which provides a derived type and operations and procedures with which to manipulate objects of that type. Once the appropriate *USE* statement has been added to a program unit, the new type and its operations may be used exactly as if they were intrinsic to the language. Note that the varying length character string module definition describes an interface. A possible implementation of the module is shown in the document as an example but an arbitrary number of alternative implementations are possible.

The principal individual features of modules are shown in 3.1.1 to 3.1.5.

3.1.1 Global Data

Data to be referenced by more than one program unit may be placed in a module rather than in a common block. Logical groupings of global data may be obtained in the same way as having several named common blocks by using more than one module program unit. Additional features such as local renaming and subsetting are also available.

3.1.2 Global Constants

It is possible to define constants (parameters in F90 terminology) in a module and to use them, at compilation or execution time, in other program units. A noteworthy example of this is to determine, in a module, suitable parameters for intrinsic types by means of the numerical enquiry functions and to use them in type declarations in the remainder of the program. By this device it is possible to write a program that will for example map a real variable onto a representation of appropriate precision.

3.1.3 Derived type definitions

The definition of a derived type (section 3.5.2 below) must be available to each program unit which defines or manipulates objects of that type. This is best achieved by placing the definition in a module program unit and by referencing the module by a *USE* statement in each program unit where it is needed. Similarly, operations on objects of derived type are defined by means of ordinary subroutine and function programs, and associated with operators by interface blocks. These should be placed in the same module as the derived type definition, or in a related module.

Thus a binding may define derived types and operations both for its own internal purposes and to make them available to referencing programs by means of *USE* statements.

3.1.4 Global allocatable arrays

Allocatable arrays (section 3.6 below) may be local to the procedure in which they are instantiated and they may then be passed as arguments to other procedures in the same way as statically defined arrays. In addition, allocatable arrays may be local to a module, instantiated by a call from one procedure which defines the array limits and then referenced by means of *USE* statements in other program units. This offers a flexible method of manipulating arrays by a user program and by the procedures implementing a binding.

3.1.5 Procedure Groups

Modules may contain a group of procedures that amongst themselves share data that is of no interest outside the module. It is possible to protect such data from accidental corruption outside the module by declaring them to have the *PRIVATE* attribute (F90 section 5.2.3). This also allows changes to be made in the internal design of the module without affecting any use made of the module.

3.2 Characters

The Fortran character set is shown in F90 section 3.1. Relative to the previous standard it contains nine new characters, viz exclamation mark, quotation mark, per cent, ampersand, semicolon, less than, greater than, question mark and underscore. It does **not** contain lower case letters; use of lower case letters is permitted by the standard provided that upper and lower case forms are equivalent outside a `character context' (F90 section 3.1.1).

The character set used to compose statements is the only set guaranteed to be available for use in character data. Thus, formally, a binding should not assume support of characters outside the Fortran character set; in particular lower case characters should not be assumed. In practice this restriction may be relaxed to the assumption that all characters in the appropriate part of ISO 8859 (ref 14) will be available. Other rules will apply in those countries or cultures that use multi-byte characters.

However bindings should be designed so that processing of character arguments to

procedures is case-insensitive where lower case letters are allowed. Moreover the use of characters in the national character positions of ISO 8859 should be avoided wherever possible.

3.3 Names

Fortran 90 uses the term "name" where GLB uses "identifier".

Names in Fortran 90 follow the same rules whether they be for constants, variables, derived types, procedures or any other nameable entity (F90 section 2.5.1, 3.2.2). If a processor supports lower case characters, such characters used in names are considered to be equivalent to the corresponding upper case character (F90 section 3.1.1). Thus a programmer using a binding may use lower case letters for referencing external names in the binding but a binding designer should not require their existence.

Names must begin with a letter (that is one of the 26 letters in the Fortran character set) and may otherwise contain letters, digits and underscore up to a maximum of 31 characters. This contrasts with a limit of six characters in the previous standard. Contrary to GLB guideline 24 a punctuation mark, in the normal English sense of the term, is not allowed in a Fortran name. It is recommended to use the underscore character for this purpose.

3.4 Statements

Constraints on the length of a Fortran statement impose an upper limit on the length of a procedure call. Although it is unlikely to be a serious imposition, a single Fortran statement in free-form format is subject to an upper limit of 5241 characters, including a possible label (F90 sections 3.3.1, 3.3.1.4); in fixed-form format the upper limit is 1320 characters plus a possible label (F90 section 3.3.2). These limits are subject to the characters being of "default kind", that is in practice single-byte characters; for double-byte characters, used for example for ideographic languages, the limits are processor-dependent but are likely to be of the same order of size.

There is no limit in the Fortran Standard on the maximum number of arguments to a procedure.

3.5 Data Types

3.5.1 Intrinsic Data Types

The data types intrinsic to the language are integer, real, complex, character and logical (F90 section 4.3). The type "double precision" in the previous standard is available for compatibility but in Fortran 90 is considered to be a parameterized form of the intrinsic type "real". Each implementation of the language must provide at least the intrinsic types mentioned. In addition it may provide parameterized forms allowing for example short integers, double precision complex, etc. A binding may make use of such parameterized types but must not depend on their existence. It is possible for a program to determine the arithmetic environment in which it is running (F90 section 13.10.8) and for declarations to be dependent on the environment, so that for example a variable may be mapped onto a single word in a long word machine or onto a double word otherwise; this is best done by using a module to define global constants (section 3.1.2 above).

So far as the language itself is concerned no limits are imposed on the maximum and minimum values which a numeric variable may take although the values applying to a

particular program execution may be determined from within the program (F90 section 13.10.8). The limits which apply are typically those of the underlying hardware. Fortran has no intrinsic delimited or enumerated data types.

The Fortran Standard places no requirements on the internal representation of data values. Thus the representation of Fortran intrinsic data types in a particular implementation should not automatically be assumed to be that adopted by other languages in the same environment.

A character variable may contain any character representable on the processor (F90 section 4.3.2.1); this may include characters from other sets than the Fortran character set (F90 section 3.1). In practice it is safe to assume that at least the 128 character set based on one of the variants of ISO 8859 (ref 14), for example ASCII, will be available. The possibility exists that non-default character sets may occupy different storage space from default characters.

The standard imposes no restriction on the length of a character variable. It is safe to assume that all implementations allow variables of lengths up to at least 511 characters of default kind. Intrinsic Fortran character variables are always of fixed length. The usual implementation method is to allocate storage for exactly the storage implied by the variable definition, with no counts or terminating symbols employed. This is different from some other languages and it should not be assumed for a binding written in a language other than Fortran that Fortran character expressions or variables (including arrays) may be passed directly as arguments to character variables in other languages.

The maximum number of dimensions in an array, of any intrinsic type, remains at 7.

The intrinsic data types may apply to any entity which has a data type, that is constant, variable, expression, function or any sub-object thereof.

3.5.2 Derived Data Types

In Fortran 90 it is possible to define derived data types as aggregations of entities of intrinsic data types, or of other derived types (F90 section 4.4). Although objects of derived type may be used as simple aggregate records it is possible also to define operations on them and to place the definitions of both derived types and operations in a module so that they may be accessed and used either selectively or throughout an entire program.

This facility is one of the principal extensions in Fortran 90. It allows the programmer to extend the language by adding new data types and new operations (cf ref 12). It also allows considerable abbreviation of the argument lists to procedures, relative to bindings to the previous standard. Derived data types are most advantageously used in conjunction with modules (section 3.1.3 above).

3.6 Arrays

Fortran 90 contains several new features concerned with arrays, notably array operations and dynamic allocation of arrays. Array handling includes the ability to define array expressions and to specify numeric, logical and character operations and assignment on whole arrays or subarrays rather than on individual elements; it also allows for example the use of an array cross-section as a single entity. Dynamic allocation gives the ability to allocate arrays during the execution of a program, with array limits which are suitable for the problem in hand. These are known in Fortran 90 as allocatable arrays (F90 section 6.3).

Pointers (F90 section 5.1.2.7) are available in Fortran 90. Declaration of zero-length arrays, and of zero-length character variables, is permissible (cf GLB guideline 23).

For bindings, dynamic allocation obviates the need for arrays required for local work space either to be declared with fixed, typically too large, limits within a procedure or to be passed as arguments from the calling program.

Further, Fortran 90 contains assumed-shape arrays (F90 section 5.1.2.4.2) whereby the shape of an array used as a dummy argument is assumed from that of the corresponding actual argument; this again simplifies the argument lists of binding procedures.

The traditional Fortran order of elements within arrays, of the leftmost index varying most rapidly, is retained. It is recommended that, other than defining the sequence of operations to optimize execution time, programmers do not make use of this ordering by assuming the sequence ordering of storage. This is because the mapping of storage to variables is more fluid in Fortran 90 and such assumptions could lead to errors which are difficult to detect. Similarly, it is recommended that the *SEQUENCE* statement not be used within derived type definitions as in addition to possibly causing obscure errors this can severely inhibit the efficiency of the code produced by compilers on certain systems.

3.7 Optional arguments to procedures

Fortran 90 allows the ability for a program referencing a procedure in a module to specify only a subset of the arguments required by the procedure and for the procedure to be able to identify the arguments which were omitted and hence to supply default values or actions as required.

The called procedure must declare the dummy arguments that may be omitted to have the attribute *OPTIONAL* (F90 section 5.2.2) and the presence of corresponding actual arguments at execution time is determined by use of the logical intrinsic function *PRESENT* (F90 section 13.13.80). The calling procedure must have access to an interface block which defines the arguments of the called procedure. This may be included in the calling procedure or, preferably, may be in a module which is referenced from the calling procedure.

For bindings this facility, with suitably chosen defaults, typically allows much shorter actual argument lists than was possible with the former standard. It will come to be expected as normal by programmers using Fortran 90 so that where an external procedure in a binding is provided by means other than Fortran, a comparable facility should be provided.

There are certain other circumstances when a procedure interface block is required, for example for functions with array-valued results. In all such cases the *MODULE/USE* mechanism is a convenient and orderly way of specifying the interface.

3.8 User-defined Generic Procedures

A Fortran 90 procedure may be defined to be generic in the sense that a reference to a named (generic) procedure may be fulfilled by any one of a set of actual procedures according to the types of the actual arguments. This is also achieved by using the interface block (F90 section 12.3.2).

4 Application of Binding Methods to Fortran 90

4.1 System Facility Standard Procedural Binding (Method 1)

The procedural interface is a very portable method of binding in the sense that a user's program requires no processing software in addition to the standard Fortran processor and that a program written to conform to the language standard and to the binding interface will run in any environment which supports the two.

It is the method used for example by the Fortran 77 bindings to various graphics standards (refs 1-3 and 15) and to Posix (ref 8); it has come to be considered the norm for bindings to Fortran wherever it is possible to use it. The facilities afforded by Fortran 90 greatly enhance the possibilities of defining the interface to the system facility, for example by adding new data types and new operators, while staying within the capability of standard Fortran processors.

The procedural interface therefore, when taken together with the language extension facilities offered by the module, is the preferred method of binding to Fortran 90 (other than the exceptional case of method 5).

A further extension of this method is one not described in GLB and falls part way between methods 1 and 2. This is where it is not possible to provide complete modules for a binding, but where the binding consists of a partial module or modules which are completed by the programmer for a particular implementation and then used in a Fortran program. This is potentially a very powerful method and should be the first alternative considered when complete modules are not possible.

4.2 User-Defined Procedural Interfaces (Method 2)

This method is appropriate where the system functions are too complex to be defined by a few parameters and or may involve complex or variant data structures. The method assumes the existence of a Procedural Interface Definition Language, which acts as an interface between the Fortran program and the system function. The user writes procedures in the PIDL and then interfaces the Fortran program to the PIDL code.

It appears that there are no existing examples of Method 2 bindings to Fortran. So far as the Fortran parts of the interface are concerned, the discussion above on Method 1 applies also to Method 2. GLB is somewhat lukewarm in its commendation of Method 2 and it must therefore take a relatively low place in the list of preferred methods.

4.3 Native Syntax Extension Bindings (Method 3)

This is a long-term binding method, involving the incorporation of the system facility into the Fortran language standard itself, in the same way as a graphics section was added to the BASIC language standard. It requires the system facility to be `stable and well understood' (GLB section 2.4) and could be suitable for facilities which are likely to be used by a significant proportion of Fortran users. For example it could be suitable for various aspects of parallel processing once they have become sufficiently stable.

This method requires that the Fortran standards committee assume formal responsibility for the maintenance and development of the standard Fortran language interface to the system facility and ties together the development cycles of the Fortran language and the system

facility standards; indeed the Fortran language subsumes the systems facility.

It was presumably to circumvent these potential disadvantages that binding methods 6 and 7 were introduced. WG5 therefore recommends that, while it is open to discussion on possible development of method 3 bindings, method 6 is generally to be considered preferable in the shorter term; method 7 however appears to offer the worst of all possibilities.

4.4 Embedded Alien Syntax Bindings (Method 4)

This method involves the addition of alien syntax to a program which is interpreted by a preprocessor or an extended language processor and is suitable when the system facilities are too complex to be implemented by methods 1 or 2. A major advantage over method 3 is that the language standard itself is not affected. A disadvantage is that the user has to know two languages. Relative to methods 1 and 2, method 4 can, with inappropriate design, confuse the relationship of language and system facilities.

Discussion in WG5 has indicated a preference for a variant of method 4 in which the alien syntax takes the form of comments in the underlying language (i.e. Fortran 90), so that in certain circumstances a program could be simultaneously meaningful to both a standard and an extended language processor. Such use of programming language comments as commands to a preprocessor or to an extended compiler has a very long history in Fortran and is the general philosophy followed by HPF (ref 17). This has the further advantage that the language standard itself is not affected by the additional functionality. However it is not in general possible to add significant functionality in this way and it can become necessary to add new statements in the style of the parallel processing extensions of ref 18.

When this is the case WG5 recommends the following guidelines:

- WG5 must be consulted at every stage of the design to ensure that there are no incompatibilities between developments in or plans for the fundamental language and developments in the binding syntax or semantics, or that there are no incompatibilities between two or more bindings being developed in parallel;
- added syntax should be kept to a minimum and the language extension features of Fortran 90 should be used as far as possible;
- the order of preference of added syntax, in decreasing order, is:
    - directives as comments;
    - new constructs of the form *CONS .... END CONS* which serve to isolate the binding syntax as much as possible;
    - new declaration or executable statements;
    - syntactic extension to existing standard statements;
- semantic extension without change of syntax must be avoided.

4.5 Pre-existing Language Element Bindings (Method 5)

This method may be used wherever it happens that the system facilities can be mapped onto the facilities in the language standard. Since this involves a binding with no change whatever to the language standard and no additional software, not even a procedure written in standard Fortran, needed to implement the binding, it is the optimal solution to the binding problem and is to be encouraged. However the occasions when it will be relevant appear to be limited in the case of Fortran.

One possible case that has been suggested as a potential candidate as a method 5 binding is one to relate the arbitrary values taken by kind parameters (F90 section 2.4.1.1) more closely to the characteristics of the variables with which they are associated.

4.6 Direct External Reference Bindings (Method 6)

As outlined in ref 13, this method is presumed to involve new Fortran language syntax implementing the systems facility which is defined in a standard that is referenced from the Fortran language standard. This reference is normative so that any standard conforming processor must implement the facilities of the referenced standard. It is thus a variant of Method 3 with decoupling of the standards revision cycles.

It would appear that the same criterion of the systems facility being stable and well understood should apply also to this method so its discussion in the context of ref 4-6 must be seen as a long-term aim.

4.7 Direct Imported Text Bindings (Method 7)

As outlined in ref 13, this method is presumed to involve new Fortran language syntax implementing the systems facility which is defined in an external standard and is copied verbatim into the Fortran language standard. This method has the disadvantages of Method 3 and the further disadvantage that, in principle at least, part of the standard Fortran language syntax may be defined without reference to WG5. Other than removing the need to have more than one standards document, compared to method 6, it appears to have nothing to commend it and is the least preferred of all methods.

5 Recommendations

WG5 strongly recommends:

(a) that the facilities of Fortran 90 modules be used as extensively as possible in all bindings to Fortran 90, including the possibility of partial modules,

and

(b) that the preference order of methods of binding to Fortran 90 be:

Method 5 (Pre-existing language element binding)
Method 1 (System facility standard procedural binding extended to embrace use of modules and partial modules)
Method 4 (Embedded alien syntax)
Method 2 (User-defined procedural interface)
Method 6 (Direct external reference)
Method 3 (Native syntax extension)
Method 7 (Direct imported text).

6 Obsolescent Features

During the design of Fortran 90 a conscious attempt was made to include only safe, regular language features which accorded with modern recognized good programming practice. However for reasons of history and compatibility Fortran 90 still contains language features which may now be considered undesirable, and for various reasons it contains irregular, redundant features, some of them new in Fortran 90. It is however possible to write

programs in Fortran 90 using only what is generally accepted to be good practice.

In attempting to define a long-term evolution plan for Fortran, the designers specified some language facilities which it was considered could be phased out of the language in future revisions because better facilities existed. These are shown in Appendix B2 of F90. It is recommended that a binding does not make use of these features nor require their use in programs accessing the binding. The single most significant relevant language feature so far as bindings are concerned is the alternate return.

7 Responsibility for Production of Bindings

There are two main phases to designing a binding to Fortran 90, namely choosing the method and defining the detailed specification.

WG5 recommends (cf GLB guidelines 2 and 3 and ref 16) that in general the committee responsible for the functional specification is better placed to define a functional binding to Fortran 90 than is WG5. This also allows for bindings to different languages to be as consistent as possible. However WG5, in accordance with GLB guideline 3, should be involved with consultation on Fortran 90 bindings as early in the design process as possible and both the committee itself and the various national body Fortran committees are willing to help develop such bindings in any way possible. The degree of consultation is likely to vary according to the binding method chosen.

Produced by the BSI Fortran Panel, IST/5/-/5, on behalf of SC22/WG5
Approved at the WG5 meeting, Berchtesgaden, July 5-9, 1993.

Annex A. Use of Fortran 77 Bindings in a Fortran 90 Environment.

This annex notes the inconsistencies between Fortran 77 and Fortran 90, which may be relevant if a Fortran 77 binding is used in a Fortran 90 environment or if a Fortran 77 binding is developed to become a Fortran 90 binding.

The design intent of Fortran 90 was that Fortran 77 be included as a subset and that any program conforming to Fortran 77 should also conform to Fortran 90. Four minor differences are listed in section 1.4.1 of the ISO Fortran Standard. They are:

a. when a real constant is used to initialize a double precision variable, a matter left to be processor-dependent in Fortran 77 is explicitly defined in Fortran 90;

b. when a variable is in a DATA statement and not in a SAVE statement, a matter left to be processor-dependent in Fortran 77 is explicitly defined in Fortran 90;

c. when a formatted input record is shorter than is implied by the corresponding READ statement, Fortran 77 required that an error be generated; Fortran 90 pads such a record with blanks on the right unless required not to do so by a new user control (keyword PAD in the OPEN statement);

d. Fortran 90 has more intrinsic functions than Fortran 77 and introduces intrinsic subroutines to Fortran. Thus a Fortran 77 program may have a different interpretation under Fortran 90 if it invokes a procedure with a name which is the same as one of the new intrinsic procedures, unless that procedure is specified in an EXTERNAL statement (cf Appendix B15 of the Fortran 77 Standard).

The following additional item for this list is in the Fortran 90 maintenance document (ref 10):

e. there is a difference in the definition of operation of the G edit descriptor for output of real numbers; the Fortran 90 description corresponds more closely to what a user would expect and corrects what is now deemed to be an error in Fortran 77.

Analogously with point d. above, when accessing a Fortran 77 binding in a Fortran 90 environment, attention should be paid to possible conflicts between the Fortran 90 intrinsic procedure names and the procedure names in the Fortran 77 binding.