ISO / IEC JTC1 / SC22
Programming languages, their environments and system software interfaces
Secretariat: CANADA (SCC)

ISO/IEC JTC1/SC22
N 1365

MAY 1993

TITLE: Final text of TR10182: Information Technology - Programming languages, their environments and system software interfaces -Guidelines for Language Bindings

SOURCE: Secretariat ISO/IEC JTC1/SC22

WORK ITEM: JTC1.22.14

STATUS: New

CROSS REFERENCE: N1194, N1364

DOCUMENT TYPE: Final text of DTR 10182

ACTION: For information to SC22 Member Bodies.
This document has been forwarded to ISO/IEC ITTF for publication as an ISO/IEC standard.

Address reply to: ISO/IEC JTC1/SC22 Secretariat
J.L. Côté
Treasury Board Secretariat
140 O'Connor St., 10th Floor, Ottawa, Ontario, Canada, K1A 0R5
Tel.: (613)957-2496 Telex: 053-3336 Fax: (613)957-8700

GUIDELINES FOR LANGUAGE BINDINGS ©ISO/IEC

ISO/IEC JTC1/ TR 10182

April 16, 1993

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Status of the Document

This document is a compilation of the experience and knowledge gained by the members of ISO/IEC JTC1/SC22/WG11 (Techniques for Bindings) from the generation of programmers' interfaces to FUNCTIONAL INTERFACE STANDARDS. Although current experience was derived from the fields of computer graphics and database management, the problems discussed are thought to be generally applicable for mappings of other functional interface standards to programming languages. This document is intended

a) to identify the problems and conflicts which must be resolved;
b) to suggest guidelines for future use;
c) to provide scope and direction to required additional work, such as common procedural calling mechanisms and data types; and
d) as a historical record of past experiences and decisions.

This document is incomplete; the authors have concentrated on those areas where experience and expertise was readily available. The ideas and issues brought forward here emerged from more than ten years of work, and are represented in International Standards.

Section 2 of this document contains the results of a survey of current methods used for language binding development. Characteristics of each method are given, followed by reasons for the selection of the method.

Application of the methods has suggested some guidelines that are presented in Section 3. Sections 2 and 3 contain documentation of the current state of language binding efforts; Section 4 addresses future directions for language bindings.

Circulation of this document is necessary at this stage, as input and discussion from representatives of ISO/IEC JTC1/SC21 (functional specification standards developers), ISO/IEC JTC1/SC24 (computer graphics standards developers), and ISO/IEC JTC1/SC22 (language standards developers) is urgently sought. The document in its current form may be useful for those about to embark on language binding developments.

1.2 Scope

This document is based on experience gained in the standardization of two major areas in information processing. One area covers programming languages. The other area is composed of the services necessary to an application program to achieve its goal. The services are divided into coherent groups, each referred to as a SYSTEM FACILITY, that are accessed through a FUNCTIONAL INTERFACE. The specification of a system facility, referred to as a FUNCTIONAL SPECIFICATION, defines a collection of SYSTEM FUNCTIONS, each of which carries out some well-defined service.

Since in principle there is no reason why a particular system facility should not be used by a program, regardless of the language in which it is written, it is the practice of system facility specifiers to define an 'abstract' functional interface that is language independent. In this way, the concepts in a particular system facility may be refined by experts in that area without regard for language peculiarities. An internally coherent view of a particular system facility is defined, relating the system functions to each other in a consistent way and relating the system functions to other layers within the system facility, including protocols for communication with other objects in the total system.

However, if these two areas are standardized independently, it is not possible to guarantee that programs from one operating environment can be moved to another, even if the programs are written in a standard

programming language and use only standard system facilities. A language binding of a system facility to a programming language provides language syntax that maps the system facility's functional interface. This allows a program written in the language to access the system functions constituting the system facility in a standard way. The purpose of a language binding is to achieve portability of a program that uses particular facilities in a particular language. Examples of system facilities that have had language bindings developed for them are GKS, NDL, and SQL (see Section 1.3, References and Bibliography). It is anticipated that further language binding development will be required. Some system facilities currently being standardized have no language bindings and additional system facilities will be standardized. There is a possibility of n x m language bindings, where n is the number of languages and m the number of system facilities.

The scope of this document is to classify language binding methods, reporting on particular instances in detail, and to produce suggested guidelines for future language binding standards.

Note that the language bindings and the abstract facility interfaces must have a compatible run time representation, but the abstract facility does not necessarily have to be written in the host language. For example, if the application program is using a Pascal language binding and the corresponding facility is written in FORTRAN, there must be a compatible run time representation in that operating environment. How this compatibility is achieved is outside the scope of these guidelines. This is generally a property of the operating environment defined by the implementor, and is reviewed briefly in this document.

1.3 References and Bibliography

Information Processing Systems - Computer Graphics - Graphical Kernel System (GKS) functional description, ISO/IEC 7942.

Information Processing Systems - Computer Graphics - Graphical Kernel System (GKS) language bindings - Part 1: FORTRAN, ISO/IEC 8651/1.

Information Processing Systems - Computer Graphics - Graphical Kernel System (GKS) language bindings - Part 2: Pascal, ISO/IEC 8651/2.

Information Processing Systems - Computer Graphics - Graphical Kernel System (GKS) language bindings - Part 3: Ada, ISO/IEC 8651/3.

Information Processing Systems - Computer Graphics - Graphical Kernel System (GKS) language bindings - Part 4: C, ISO/IEC 8651/4.

Information Processing Systems - Computer Graphics - Three-Dimensional Extensions to GKS, ISO/I 8805.

Information Processing Systems - Computer Graphics - Three-Dimensional Extensions to GKS - Part 1: FORTRAN, ISO/IEC 8806/1.

Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS), ISO/IEC 9592.

Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 1: FORTRAN, ISO/IEC 9593/1.

Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 2: Extended Pascal, ISO/IEC 9593/2.

Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 3: Ada, ISO/IEC 9593/3.

Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 4: C, ISO/DP 9593/4.

Computer Graphics Interface, draft, ISO/IEC JTC1/SC24

CGI/FORTRAN Binding, X3H34/86-7

Programming languages - Minimal BASIC, ISO/IEC 6373.

Programming languages - Pascal, ISO/IEC 7185.

Programming languages - 'C', ISO/IEC 9899

Programming languages - Extended Pascal, ISO/IEC 10206

Programming languages - FORTRAN, ISO/IEC 1539.

Programming Languages - COBOL, ISO/IEC 1989-1985

Programming Languages - COBOL - Addendum 1, ISO/IEC 1989-1985

Programming Languages - Ada, ISO/IEC 8652

Information Processing Systems - Database Language - NDL, ISO 8907

Information Processing Systems - Database Language - SQL, ISO 9075

Graphics Language Bindings Abbreviations List, ISO/IEC JTC1/SC24/WG4

D. Rosenthal, P.ten Hagen, GKS in C, Eurographics 1982 Proceedings, North-Holland.

C. Osland, Case Study of GKS Development, Eurographics Tutorials 1983, Springer.

J. R. Gallop, C. Osland, Experience with Implementing GKS on a Perq and Other Computers, Computer Graphics Forum 9(7), North-Holland 1985.

F. R. A. Hopgood, D. Duce,J. R. Gallop, D. Sutcliffe, Introduction to the Graphical Kernel System (GKS), Academic Press, 1983.

M. Sparks, J. R. Gallop, Language Bindings for Computer Graphics Standards, IEEE Computer Graphics and Applications, Vol 6, Number 8, August 1986.

M. Sparks, Graphics Language Bindings, Computer Graphics Forum, Journal of the European Association for Computer Graphics, vol. 4(4) 1985.

M. Sparks, Graphics Standards Bindings - What Are They and When Can We Use Them?, Computer Graphics/November 1985.

Language Binding Generic Issues (document within ISO/IEC JTC1 SC24/WG4 and ISO/IEC JTC1 SC22/WG11)

1.4 Terms and Abbreviations

ABSTRACT SERVICE INTERFACE: An interface having an abstract definition that defines the format and the semantics of the function invoked independently of the concrete syntax (actual representation) of the values and the invocation mechanism.

ALIEN SYNTAX: Syntax of a language other than the host language.

CGI: Computer Graphics Interface standard (ISO DP) -a functional specification of the computer graphics programming system facility.

EMBEDDED ALIEN SYNTAX: Statements in a special language for access to a system facility, included in a source program written in a standard programming language.

EXTERNAL IDENTIFIER: An identifier that is visible outside of a program.

FUNCTIONAL INTERFACE: The abstract definition of the interface to a system facility by which system functions are provided.

FUNCTIONAL SPECIFICATION: The specification of a system facility. In the context of this document, the functional specification is normally a potential or actual standard. For each system function the specification defines the parameters for invocation and their effects.

GKS: Graphical Kernel System standard (ISO/IEC 7942) - a functional specification of the computer graphics programming system facility.

HOST LANGUAGE: The programming language for which a standard language binding is produced; the language in which a program is written.

IDENTIFIER: Name of an object in an application program that uses a system facility.

IMPLEMENTATION-DEFINED: Possibly differing between different processors for the same language, but required by the language standard to be defined and documented by the implementor.

IMPLEMENTATION-DEPENDENT: Possibly differing between different processors for the same language, and not necessarily defined for any particular processor.

IMPLEMENTOR: The individual or organization that realizes a system facility through software, providing access to the system functions by means of the standard language bindings.

LANGUAGE BINDING OF *f* TO *l* or *l* LANGUAGE BINDING OF *f*: A specification of the standard interface to facility *f* for programs written in language *l*.

LANGUAGE COMMITTEE: The ISO technical Subcommittee or Working Group responsible for the definition of a programming language standard.

MDL: A language for the specification of an interface to a generic system facility, the MDL (module definition language) is used to generate a module to support the specific system facility access needs of an application program.

NDL: Database Language NDL may be used to define the structure of a database using the network model of data.NDL is defined in IS 8907. (See Section 1.3, References). The standard also includes the data manipulation functions and their language bindings.

PHIGS: Programmers Hierarchical Interactive Graphics System standard (ISO/IEC 9593) - a functional specification of the 3-D computer graphics programming system facility.

PROCEDURAL BINDING (system facility standard procedural interface): The definition of the interface to a system facility available to users of a standard programming language through procedure calls.

PROCEDURAL INTERFACE DEFINITION LANGUAGE: A language for defining specific procedures for interfacing to a system facility as used, for example, in IS 8907 Database Language NDL.

PROCEDURE: A general term used in this document to cover a programming language concept which has different names in different programming languages - subroutine and function in FORTRAN, procedure and function in Pascal, etc. A procedure is a programming language dependent method for accessing one or more system functions from a program. A procedure has a name and a set of formal parameters with defined data types. Invoking a procedure transfers control to that procedure.

PROCESSOR: A system or mechanism that accepts a program as input, prepares it for execution, and executes the process so defined with data to produce results.

PROGRAMMING LANGUAGE EXTENSIONS WITH NATIVE SYNTAX or native syntax binding: The functionality of the system facilities is incorporated into the host programming language so that the system functions appear as natural parts of the language. The compiler processes the language extensions and generates the appropriate calls to the system facility functions.

SQL: Database Language SQL (Structured Query Language) defines the structure of a database using the Relational model of data. Database Language SQL is defined in IS 9075. (See Section 1.3, References). The standard also includes the data manipulation functions and their language bindings.

SYSTEM FACILITY: A coherent collection of services to be made available in some way to an application program. The system facility may be defined as a set of discrete system functions with an abstract service interface.

SYSTEM FACILITY COMMITTEE: The ISO technical subcommittee or Working Group responsible for the development of the functional specification of a system facility.

SYSTEM FUNCTION: An individual component of a system facility, which normally has an identifying title and possibly some parameters. A system function's actions are defined by its relationships to other system functions in the same system facility.

2 OVERVIEW OF FUNCTIONAL BINDING METHODS

2.1 Introduction to Methods

This section discusses the binding development problem in general by documenting a number of different approaches to bindings. Each approach has its own characteristics from the points of view of the user, the implementor, and the specifiers of standards.

The first task in specifying a binding of a system facility is to determine the usability, stability, and implementation goals of both the binding and the system facility, and to use these to help select the best method.

The functional binding methods are:

Method 1. Provide a completely defined procedural interface (the System Facility Standard Procedural Interface)

Method 2. Provide a procedural interface definition language (User-Defined Procedural Interface)

Method 3. Provide extensions to the programming language, using native syntax

Method 4. Allow alien syntax to be embedded in the programming language

Method 5. Binding pre-existing language elements.

Before addressing the individual methods, a discussion of a general issue that affects programming language implementations is indicated. This issue is whether to increase the capability of a given compiler to encompass the system facility, or to provide a pre-processor. Though this is of no direct concern to language binding developers, they may wish to consider the feasibility of each option when choosing a method.

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.

In the following sections, each functional binding method is discussed, circumstances that suggest a method be used or avoided are given, and relevant advantages and disadvantages are defined.

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.

2.2 System Facility Standard Procedural Interface (Method 1)

With functional binding Method 1, the system facility is designed to support a fixed number of procedures. Each procedure has formal parameters of defined data types and each procedure invocation passes actual parameter values which match the data types.

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.

It is often useful to define subsets of the facility to suit different modes of use. For example, where the functions are largely independent and a program only requires a few of them, it may be possible to reduce the size of the run-time system by omitting portions of the system facility. These subsets are reflected by levels of conformance to the functional interface standard.

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).

2.3 User-Defined Procedural Interfaces (Method 2)

With functional binding Method 2, the run-time procedural interface is defined by the user, and the system functions invoked by the procedures are defined in a language appropriate to the system facility.

This method is appropriate when the interfaces to the system functions provided by the system facility are too complex to be defined by a few parameters, and when they cannot be easily contained in an exhaustive list.

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.

Database facilities use this method, where a Procedural Interface Definition Language (in the database standards this is referred to as a Module Definition Language), containing both declarations and procedural statements, is provided. A module may declare the data to be accessed as a view of the database (as it may reference a predefined view) and it defines both the form and the execution of database procedures.

2.4 Programming Language Extensions with Native Syntax (Method 3)

With functional binding Method 3, the functionality of the system facilities is incorporated into the host programming language so that the system functions appear as natural parts of the language. The compiler processes the language extensions and generates the appropriate calls to the system facility functions.

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.

Method 3 has been tried with the COBOL and FORTRAN database facilities (Codasyl and ANSI) and with the graphics chapter for Basic.

2.5 Programming Languages with Embedded Alien Syntax (Method 4)

With functional binding Method 4, the system facilities are considered to be 'driven' by statements written in a 'system facility language' rather than in the host programming language. The embedded alien syntax must be clearly distinguishable from the host language so that it can be processed by a pre-processor.

Method 4 is suitable when the system facilities are too complex to be invoked by simple procedures (as for Method 2, User-Defined Procedural Interfaces). The method could be implemented by having the pre-processor generate Module Definitions as in Method 2.

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.

Method 4 is one of the options in the ISO database standards.

2.6 Binding Pre-Existing Language Elements (Method 5)

In some cases, the host language may contain language elements that can be directly identified with corresponding elements of the abstract system facility. For example, in a binding to a system facility that opened and closed files, the host language may already contain constructs for opening and closing files.

The advantage 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.

2.7 Conclusions

The subsections above have described five different methods for developing functional bindings, and the circumstances in which they can be used. None of the methods is appropriate in all circumstances, or for all languages. In practice, a combination of methods may be appropriate. In some languages it is necessary to combine Method 4 with Method 5.

It is possible, and often desirable, for a system facility to provide more than one method of binding, to give the implementor and user a choice. However, if an implementor provides only one of the standard methods, the user has no choice, and, unless there is a recognized way of converting between methods, portability problems result.

The objective of a standard language binding is to enable a program to be portable when it is written in a standard programming language and accesses a standard system facility. Often the system facility is written in a different language from the application program and requires a certain compatibility between the implementations of the two source language compilers. Of course, similar compatibility is necessary for different compiler implementations of the same source language. In particular:

a) the procedure calling mechanisms should be compatible, and
b) corresponding data types should have compatible machine representations.

Often, but not always, the hardware and operating system will determine appropriate standards or conventions for the representation of primitive data types and inter-program calls. Where there are mismatches, it is necessary for the implementor to create a layer of software to perform conversions between alternative data type representations or procedure calling mechanisms. There are now ISO/IEC JTC1 work items addressing a) and b) above. These are: work item 22.16 - Specification for a Model for Common Language-Independent Procedure Calling Mechanisms, and work item 22.17 - Specification for a Set of Common Language-Independent Data Types.

The methods described have been used in current ISO standards for database and graphics. Some papers defining bindings for communications facilities have also been reviewed, but the strategy to be adopted for ISO OSI bindings is yet to be determined.

3 GUIDELINES

3.1 Organizational Guidelines for Preparation of Language Bindings

This section describes some organizational guidelines that should be followed in order to facilitate the generation of binding standards. A general statement of each guideline is given, followed by some discussion. The guidelines appear in no particular order.

GUIDELINE 1

Standard bindings of some form should be developed for all standard system facilities that may be accessed from a standard programming language.

Here, "standard" means that an ISO standard or draft standard exists for both the system facility and the language.

There are standards describing system facilities which do not have standard language bindings associated with them. Lack of a standard may lead to implementor-defined interfaces, causing loss of portability.

GUIDELINE 2

Either the language committee or the system facility committee should have primary responsibility for the language binding.

In this area, current practice is 'whichever committee perceives the need for a binding' or 'if the language has an external procedure call mechanism, then the system facility, otherwise the language committee'. Unfortunately, sometimes a binding is required, yet no-one takes the initiative to start binding work; some method for resolving this impasse is required.

It is expected that it is the primary responsibility of the system facility committee to establish a reference binding to an arbitrary language and a generic binding. Subsequent bindings should be the responsibility of the appropriate language committees. In practice it is expected that the system facility committee will seek support from the applicable language committee in the creation of an arbitrary binding. This would increase the likelihood of getting a full range of language bindings to make a system facility useable. The system facility committee is more likely to have an interest in making the facilities accessible; furthermore, language committees might not have the necessary expertise to develop bindings to specialized facilities. Part of the primary responsibility is to respond to public comments.

GUIDELINE 3

Whichever committee is responsible for a particular binding, the other committee needs o be consulted as early as possible. The two committees have complementary responsibilities and concerns.

A system facility committee is concerned with

- fidelity to the functional specification, including relevant level structures.
- similarity of program structure independent of the programming languages.
- suitability of the system facility for binding to the various languages.

A language committee is concerned with

- correctness of the binding definition, and adherence to good practice in the language, including the avoidance of obsolete and deprecated features.
- consistency in the binding of similar concepts in similar ways throughout different bindings (for example, 'is a 2x3 matrix bound as a (3,2) array or a (2,3) array or a (6) array?' is a question to be answered by a language committee).

- the needs of programmers accessing more than one system facility in one program.
- suitability of the language for various binding methods.

Both committees are concerned with ease of use and orthogonality of concepts.

GUIDELINE 4

Specific guidelines should be produced alongside standards for particular system facilities and particular programming languages.

- A set of guidelines for producing language bindings for a system facility should be associated with the standard functional specification of any system facility.
- A set of guidelines for producing language bindings of different system facilities to a language should be associated with the standard of any programming language.

Potentially, there is an n x m problem of agreeing and processing n language bindings to m system facilities. Ideally, the development of appropriate guidelines should reduce the problem to one of order n+m.

Binding guidelines could be published as appendices to standards, and in some cases a supplemental standard could be considered. Sample programs would also be helpful. Publication of such guidelines by language committees would help not only system facility committees, but also any producers of packages needing bindings to languages, and would thus help promote portability.

GUIDELINE 5

Draft proposals for bindings should not progress beyond the standardization stage of the system facility or the language.

In addition, no system facility should progress to DIS (Draft International Standard) or IS (International Standard) until there is at least one language binding at or above the level of DP (Draft Proposed International Standard) or DIS, respectively. Users cannot fully judge a semantic standard without seeing the specific syntax which is their only access to it. Also, there are some difficulties with abstract semantics which are only revealed by the production of a language binding. Reference ISSUE 1: What should be the criteria for determining if the functional interface standard should be bound to a particular programming language?

3.2 General Technical Guidelines

This section contains guidelines that are general over all binding methods.

GUIDELINE 6

Language bindings to the same system facility should be similar in those respects where the languages are similar.

A functional specification might have a system facility data type that is easily represented in some languages but not in others. For example, GKS uses points which can easily be represented as records in Pascal and Ada, but not in FORTRAN. It seems unreasonable for the Pascal and Ada bindings to be constrained by the FORTRAN binding. On the other hand, after language bindings for a few very dissimilar languages have been produced, bindings for additional languages might be produced by analogy.

GUIDELINE 7

Different language bindings to a system facility should not be the cause of substantial differences in program structure except where warranted by substantial language differences.

Some factors inherent in the language substantially influence the structure of a program. For example, a program written in a language with a WHILE construct is likely to possess a different structure than a language without such a construct. However, the different bindings of the system facilities should not be the cause of substantial differences in structure. An example of such a problem would be if a functional parameter is bound as an output parameter in one language binding and as an input parameter in another.

This principle may well break down when the languages have different processing paradigms or other deep concepts, such as the inherent differences between imperative and functional languages.

GUIDELINE 8

A binding must be reviewed and, if necessary, updated each time the system facility or the language standard is enhanced. (The forward compatibility requirements of the user must be identified and taken into consideration when any of the standards is enhanced).

Reference ISSUE 2: To what event(s) should new versions of a binding be tied?

GUIDELINE 9

Bindings should take account of likely language processor limits. (A standard-conforming processor may fail to process a standard-conforming program if it is caused to exceed its limits on size, complexity, etc.) Approaches to handling such limits may appear in guidelines to bindings implementors and working papers.

As an example, a language processor may fail if its limits on number and length of identifiers are exceeded.

Reference ISSUE 3: How are 'real world' limitations of the standard taken into consideration?

3.3 Recommendations for Functional Specifications

This section contains recommendations for the preparation of standard functional specifications for system facilities. These recommendations are intended to assist the subsequent preparation of language bindings to those functional specifications.

GUIDELINE 10

A functional specification should use an abstract description, and should avoid being influenced by a particular programming language.

This very general statement is amplified in some of the guidelines that follow. This guideline allows a language binding to be appropriate to its host language. There is a need for a formal notation for the definition of abstract interfaces. The notation would need to be independent of programming languages and of implementations.

GUIDELINE 11

A data type in a functional specification is not intended to correspond with a particular internal representation in the host computer, but rather is an abstract entity intended to be mapped to the most appropriate type in the host language.

Specifying and implementing language bindings to functional standards are made more difficult both by differences between superficially similar data types in different languages, and by differences between standard-conforming implementations of the same datatype of one language (which may be permitted through properties of the datatype being left as implementation-defined or implementation-dependent in the standard). Such differences make it difficult to be certain what properties a particular datatype will possess in a given case. Basing the functional standard on a particular implementation model for a particular language is likely to increase such problems, and Guideline 10 is aimed at discouraging this.

In general, the task of specifying and implementing bindings will be greatly eased by language standards specifying required properties of datatypes more precisely than has been customary in the past, and also by greater consistency and commonality in these respects between different languages (an issue addressed by ISO/IEC JTC1 work item 22.17). This will aid the task of implementors in ensuring compatible representations and consistent implementations of bindings, and enhance the portability of application programs which need to invoke such bindings.

GUIDELINE 12

Parameters that take values that have predefined meanings should be defined in abstract terms, and not necessarily be associated with numeric values. In certain situations, however, an ordering may be needed.

This guideline is intended to encourage the use of the enumerated data type, as in Pascal and Ada. Early drafts of GKS used numeric values unnecessarily. Note that enumerated types will need encodings for languages such as FORTRAN. For portability, these encodings should be defined in the system specification.

GUIDELINE 13

The system facility should recover from errors whenever possible. A report on the status of errors should be returned to the host program where that is possible. It should be possible for the host program to determine where the error arises in the system facility.

3.4 Method-Dependent Guidelines for Language Bindings

3.4.1 Introduction to Method-Dependent Guidelines

This section contains recommended guidelines for the methods described in Section 2.

3.4.2 Guidelines for Standard Procedural Interfaces

The task of a procedural language binding is to express the system functions and data types of a particular system facility in terms of the constructs available in the host language.

This section attempts to document the experiences and ideas resulting from actual development of procedural language bindings of the graphical functional interface standards to programming languages. These bindings use procedures (rather than language modifications or a separate language), whose names and argument lists are defined in the binding standard.

A group of generic issues has been discovered, discussed, and resolved for all language bindings of graphics. Most of this section is concerned with guidelines arising from these generic issues. In addition, it is recommended that each language committee keeps a separate list of language-specific issues.

First, some general guidelines for which no issues were generated. This group of guidelines includes statements about the relationship between the functional specification and the language binding, suggested action for both types of committees involved, and some recommendations for programming language committees resulting from the experiences of bindings developed to those languages. Following these recommendations, a set of recommendations for procedural bindings, based on issues, is provided.

3.4.2.1 Relationship of the Functional Interface Standard to the Binding

The following guidelines have been used in the GKS functional interface standard and may be applicable to other functional specifications.

GUIDELINE 14

The language binding needs to specify, for each system function name, exactly one identifier acceptable to the language.

The names used for system functions in the standard are merely tools for describing the semantics of the standard; they should be replaced by actual identifiers conforming to the restrictions of the host language. A one-to-one mapping from language functions to system functions is preferred.

A method of avoiding name-clashes is needed for some languages; the name of a system function must not clash with a name in a binding to a different system facility.

Distinguishing between the names of system functions should not rely on case sensitivity alone.

GUIDELINE 15

The language binding needs to specify, for each of the system facility data types, a corresponding data type acceptable to the language. Where convenient for the host language, additional data types may be specified in terms of the system facility data types.

The data types used in the functional interface standard are merely tools for describing the semantics of the standard; they should be mapped to host language data types in the binding. In a language which allows aggregation of data into records, data may be grouped but group elements must be system facility data types.

GUIDELINE 16

The language binding needs to specify, for each system function, how the corresponding language function is to be invoked, and the means whereby each of the formal input parameters is transmitted to the language function and each of the formal output parameters is received from the language function.

Where the host language allows, the system functions are mapped to language functions or procedures. The parameters are typically transmitted via a parameter list. The items in such a list may either be, or be references to, items of the data types corresponding to the system data types, or aggregates of these types. Parameter passing attributes such as Ada's IN, OUT, and IN OUT or Pascal's VAR should be specified.

GUIDELINE 17

The language binding needs to specify a set of identifiers acceptable to the language, which may be used by an implementation for internal communication.

An implementation is normally unable to restrict its use of externally visible identifiers to those specified as a consequence of the preceding guidelines. Suggested identifiers for use by the implementation help circumvent clashes. Still, applications must avoid using identifiers from the sets required and suggested by the binding.

Because this set of identifiers is, in general, a superset of the names specified by the language binding, programs transported to an implementation from other implementations of the same binding might have used names that clash. Documentation is required to enable potential clashes to be detected.

GUIDELINE 18

The documentation of a system facility implementation needs to include a list of all identifiers for procedures, functions, global data aggregates, and files that are visible either to an application program or to the underlying operating system.

3.4.2.2 Suggested Actions for Standards Committees

As discussed earlier the potential exists for an n x m problem when developing n procedural language bindings to m system facilities. To begin to solve this problem, certain actions may be taken within the area of expertise of one committee or the other.

GUIDELINE 19

Lists of abbreviations for function names should be part of a guideline drawn up by the developers of the system facility.

See also guidelines 43 and 44.

GUIDELINE 20

Each language should have guidelines for selecting the abbreviation list to use.

See also guidelines 43 and 44.

GUIDELINE 21

Whether a procedural binding or a native syntax binding is developed depends on the host programming language, and is the decision of the language committee.

GUIDELINE 22

How compound data types are bound depends on the host programming language.

3.4.2.3 Recommendations for Programming Language Committees

This section contains some proposed recommendations to be considered in the preparation of future programming language standards.

GUIDELINE 23

Some agreement is needed on cross-calling of procedures between languages, with particular attention to the passing of data. Standardization in this area would allow system facilities to be accessible from more than one language, with a minimal portable layer between.

GUIDELINE 24

It should not be an error in a language if the dynamically specified length of an array is zero.

GUIDELINE 25

Language standards should not unduly restrict the number of characters in external identifiers. They should allow occurrences of at least one punctuation symbol in identifiers, and at least one of the punctuation symbols allowed should be significant in the spelling of the identifier.

3.4.2.4 Procedural Language Binding Generic Issues

The design objectives of language bindings include:

- Portability of application programs.
- Consistency with host language concepts, including ease of use.
- Shared use of system facility by programs written in different languages, e.g., consistency of implementation with other host language bindings for the same facility.

Facility language bindings should achieve all these goals. Metrics for measuring both objectives and achievement of these goals are under study elsewhere.

In practice, the binding developer is required to make sensible trade-offs between these goals. The guidelines are offered here to assist in making these trade-offs and to encourage some consistency within ISO on the way the trade-offs are made for different system facilities.

When developing a binding of a functional interface standard to a programming language, there are times when a trade-off must be made between using 'language-type' facilities (which would be expected by an expert in that language) and making the binding easy to use by an inexpert programmer. In addition, there are times when steps could be taken to make the bindings to similar languages look more alike (for portability of programmers' expertise across languages).

Some arguments which have been brought up are:

- Program portability across programming languages is of less importance, since compatible arguments and parameters are rarely achievable anyway, not to mention other major differences between languages.
- Automatic conversion between languages may be possible if proper consideration is made during language binding development.
- Different language cultures dictate different styles in name length, etc.
- Choosing "ease of use" over best programming practice in the host language may be unfair to both expert and inexpert programmers.
- Perhaps binding decisions may minimize problems for multi-language environments.

- An argument could be made that programmer and program portability should be the foremost goals of a programmer's interface standard.
- Programmer portability is important since programmers often use different languages either at the same time, or at different times in their careers.
- Program portability across languages is not important since old languages tend to hang around to support old programs written in them, so translation of programs between languages is not common.

The complexity of the problem has been acknowledged, and no definite conclusions have been drawn. The following guidelines reflect the consideration of all of the above goals, with preference given to different goals for different situations. The guidelines arise from some issues, a complete set of which may be found in Annex B.

GUIDELINE 26

All system functions should appear atomic to the application program.

This guideline avoids unintentional, inconsistent state changes and incomplete actions. A procedural binding should not map single system functions into sequences of language procedures called by the application program, except in the case where these procedures do not change the state of the system facility. (For example, in GKS, for "inquiry functions" in certain language bindings, the procedures may need to be called once for each element of a dynamic list; however, these functions do not change the state of the system facility.)

GUIDELINE 27

The binding of a system facility to a language should use appropriate facilities of the basic level of the language standard, to assist in usability, performance, etc. It should not attempt to simplify implementation by use of a restricted subset of the basic language level. Facilities in higher optional levels of the language standard, however, should be used only if necessary.

As an example, an Extended Pascal binding should use all appropriate facilities of "Level 0" of the standard, but should not use facilities of "Level 1" unless necessary.

Reference ISSUE 4: How much of the 'spirit' of the language is taken into consideration when developing a binding?

GUIDELINE 28

The reader of the binding specification should be assumed to be a programmer skilled both in the language and in the system facility.

Reference ISSUE 5: For what types of programmers should the bindings be developed?

GUIDELINE 29

Bindings for each programming language for which a need exists must be prepared. This does not preclude the development of a generic binding for less used languages.

Reference ISSUE 6: Should a generic binding (i.e., a single binding that can be used for many programming languages) be the only binding developed for the functional interface standard?

GUIDELINE 30

The development of language specific bindings must be supported for those functional interface standards that require such bindings; however, a single generic binding may be supported for rapid adoption and implementation of such functional interface standards as those in the database arena.

Reference ISSUE 7: Should the development of a generic binding be supported?

GUIDELINE 31

If a generic binding is required, only one should be developed.

Reference ISSUE 8: If the development of a generic binding is supported, should the development of more than one generic binding be supported?

GUIDELINE 32

Within the stated goals, the facilities used should be as consistent throughout the many language bindings to a functional interface standard as possible.

Reference ISSUE 9: What provisions/attention should be taken for the possibility of `shell' bindings built on top of a generic binding?

GUIDELINE 33

Care must be taken to provide clear and logical mappings from the bound functions to their source in the functional interface standard. In general, one to one mappings provide such clarity. However, it is recognized that certain constraints of the programming language (such as number of parameters, etc.) as well as implementation considerations may suggest alternative groupings/splittings of functions. These alternatives should be used infrequently.

Reference ISSUE 10: How should standard functions be bound to a language?

GUIDELINE 34

Parameters should be bound rigidly in the order they appear in the functional interface standard description. Extra parameters, required by a language should follow the rule that input parameters must precede output parameters.

Reference ISSUE 11: Must strict conformance be followed in ordering of parameters, or should it be possible to change the order of the parameters bound to a given language from those presented in the functional interface standard document? For example, some languages may require that an output array parameter be accompanied by its length as an input parameter.

GUIDELINE 35

Strict conformance to the functional interface standard must be maintained in the definition of input and output parameters.

Reference ISSUE 12: Must strict conformance be followed in the definition of parameters as input and output, as defined in the functional interface standard document, or should it be possible to change the type of a parameter from input to output and vice versa?

GUIDELINE 36

Related semantic parameters in the functional interface standard may be combined into single syntactic parameters (such as records) when such grouping is obvious. It is suggested that any grouping evident in the functional interface standard be followed.

Alternatively, it might be better to design the interface with the intent of grouping elements and passing individual elements when binding to a language that does not support any grouping. With this approach, the grouping would be defined within the base design of the functional interface standard rather than being considered separately for each binding. "Ungrouping" is a mechanical operation, but grouping requires more consideration.

Reference ISSUE 13: Should the combination of related SEMANTIC parameters into a single SYNTACTIC parameter be encouraged?

GUIDELINE 37

The functional interface standard should be followed strictly when binding enumerated types to enumerated types in a language. When binding enumerated types to integers in a language, a consistent numbering scheme should be used. It is suggested that consistent criteria be used in the development of the functional description, since inconsistencies imply a run-time conversion. Often there is a natural ordering implicit in the meaning of the enumerated values (for example, LEFT, MIDDLE, RIGHT). Other schemes are to put the default first or to always put values that map to 'null' (i.e., =0) in the first position.

Reference ISSUE 14: Should there be a consistent numbering (ordering) of enumerated type members across the various language bindings?

GUIDELINE 38

If a language allows the definition of data types equivalent to, or subset of, some basic type, then any data type of the functional interface standard may be bound in different occurrences to different types in the programming language.

As an example, in Pascal a single data type from a functional interface may be bound to more than one subset of type integer.

Reference ISSUE 15: How should functional interface standard data types be bound?

GUIDELINE 39

Implementation-defined language types may be used if portability of applications within the language is not affected.

As an example, in Pascal type 'char' may be used.

Reference ISSUE 16: May there be TYPES which are implementation dependent?

GUIDELINE 40

In general, the standardized functions calls should be distinguishable from user-defined functions as long as the language allows this.

As an example, in Ada the package name distinguishes calls to the system facility from calls to *user-defined functions*.

Reference ISSUE 17: Should functions be easily identifiable in the applications code ('lexical differentiability')?

GUIDELINE 41

Abbreviations should be generated from the functional interface standard names in a consistent manner.

This provides a relatively straightforward method of mapping binding identifiers back to the functional interface standard.

One way of achieving this guideline is to eliminate unnecessary words.

Reference ISSUE 18: From what roots should binding identifiers or abbreviations be derived?

GUIDELINE 42

Due to differences in the practices of programmers as well as the constraints/characteristics provided for different languages, the concatenation character, if any, must be chosen separately for each programming language.

Reference ISSUE 19: How should abbreviations of the function names be concatenated?

GUIDELINE 43

A single approved abbreviation list should be used for all languages that have unrestricted lengths for identifiers. The BASIC and FORTRAN bindings have special considerations for identifier syntax, these considerations should be used for any other bindings developed for these languages or other languages that have similar identifier restrictions.

Reference ISSUE 20: Should the BASIC/FORTRAN bindings be considered in the development of the abbreviations list?

GUIDELINE 44

Abbreviations of function names should be derived from an approved list that contains a single abbreviation for each word. Either the single abbreviation, or the word in full, may be used in the binding.

Reference ISSUE 21: In languages which have no restriction on identifier lengths, how should abbreviations of the function names be derived?

GUIDELINE 45

Names of data types should be abbreviated in a consistent manner. However, names of data types are not constrained to the approved abbreviation list for abbreviations. Other identifying abbreviations and conventions may be used in a consistent way throughout the binding.

Reference ISSUE 22: Should bindings abbreviate the names of the data types in the same manner as functions (within a single language binding)?

GUIDELINE 46

When system facility data types are bound to different languages in a similar way, the identifiers to which they are bound should be similar.

Reference ISSUE 23: Should bindings abbreviate the names of the data types in the same manner as functions consistently across languages (excluding sentinel tags).

GUIDELINE 47

For readability, ease of maintenance, and to ensure completeness, all documents defining bindings to a functional standard should use a common schema (i.e., a common structure and a common set of contents) which is based on the functional standard itself, as the central point of reference for all bindings. This common schema should be such as to aid cross-referencing and to ensure that all relevant aspects of the functional standard are addressed by all bindings. The contents of a bindings document should address all language-related aspects of the binding, including mapping of functions, mapping of parameters, and binding-specific errors. To minimize revisions in binding documents following revisions of the functional standard. Also, they should not attempt to address issues (such as errors in performing the functionality) which are within the scope of the functional standard. Overdetailed references from the bindings documents to the functional standard should be avoided. The functional standards themselves, of course, should not attempt to address issues at the language binding level.

Reference ISSUE 24: What format (documentation structure) should the binding document follow?

3.4.3 Guidelines for User Defined Procedural Interfaces

The following are suggested as guidelines for specification of user-defined interfaces:

GUIDELINE 48

Put all the definitions for a single program's interaction into a single module definition. This simplifies compilation and facilitates semantic checking.

GUIDELINE 49

Use declarative statements to simplify the imperative definitions of procedures and avoid repetition.

GUIDELINE 50

Make the level of imperative statements match the functions supplied by the system facility.

GUIDELINE 51

Allow several imperative statements to be combined within a single procedure to simplify the calling programs. This may require the addition of test statements if there is a possibility of errors preventing the normal completion of a procedure.

This recommendation has added significance if the overhead of the procedure call is profound, e.g., if the application program and the supporting service are mutually remote. Even if this is not the case, procedure calls may involve data conversion overheads.

Exception handling might be more clumsy if successive actions are specified in a single procedure. If combining imperative statements is permitted, bindings should also be defined for the individual imperative statement.

3.4.4 Guidelines for Programming Language Extensions with Native Syntax

Extending a programming language to provide access to a system facility carries certain risks, the principal one being a creeping inconsistency between the explanations in the programming language specifications and those in the system facility specifications.

GUIDELINE 52

Keep the facilities provided in one to one correspondence and define the semantics of the system functions in the system facility specifications only.

GUIDELINE 53

If new concepts are introduced in the programming language definition, they must be explained in terms of a mapping to underlying system facility functions. However, the introduction of new concepts and of new state variables should be avoided.

GUIDELINE 54

Language specifiers should consider the problem of subset or superset implementations of the system facilities. Such implementations may not work effectively with the host language extensions and it may be difficult or impossible to mix access through language extensions and access through procedure calls in the same program.

3.4.5 Uses of Programming Languages with Embedded Alien Syntax

In general, the method of using embedded alien syntax within a programming language is not recommended. If it is required, the considerations for the design of the sublanguage are similar to those for the design of the module definition language (see Section 3.4.3), but the final definition of the escape mechanism is the responsibility of the language standards committee.

4 FUTURE DIRECTIONS

To date most programming languages have developed in splendid isolation, both from each other and from system facilities. The problems of interworking with system facilities or with other programming languages have not been taken into consideration. Indeed, there is little that any particular language committee can do on its own, since these issues require joint action by at least two committees.

The need for action is growing rapidly. At present, commercial application programs generally run in an environment determined by an operating system and including database systems and transaction processing monitors. Within workstations, system facilities may include complex man-machine interfaces, local databases and access to remote services. New applications involving parallel numerical computation or rule-based knowledge processing will require interworking with processes developed using different programming paradigms.

Sections 2 and 3 offer general and specific guidelines for overcoming the problems of incompatibilities, but their arbitrary selection within different system facility bindings could lead to unacceptable demands on users. For example, one could imagine having to process a program source text through several different pre-processors before being able to compile it.

One solution to the problem of overcoming incompatibilities is to build an intelligent system to handle it. The start of this process may be seen in Data Dictionary Systems (Information Resource Dictionaries) and in the development of Integrated Project Support Environments (IPSE's). Such systems may be able to offer more unified user interfaces, facilitate the management of system generation and control the compatibility of system components.

In general, the task of interfacing components would be much simplified if there were more precision within and more commonality between language specifications. In particular, there is a need for a common set of data types, common construction mechanisms (e.g., records, lists, arrays) and common interworking mechanisms (procedure calls, events, process synchronizations). The need for remote interworking between different hardware/software systems is an added imperative for action.

INFORMATIVE ANNEX A - GRAPHICS BINDING EXAMPLES

Since much of this section draws on the experience in the subject area of computer graphics, a summary of GKS (ISO 7942) is given, with particular emphasis on those aspects relevant to language bindings.

It should be noted that GKS will not be the only programming standard in the graphics area. GKS-3D is an extension of GKS to three dimensions. PHIGS is a work item which allows more sophisticated graphics workstations to be used. Each of those three systems is intended for application programmers. Another potential standard (the Computer Graphics Interface) will address interfaces supported by graphics devices and can exist in the form of a data stream or as a programming interface. This particular programming interface will be used by programmers building tools for application programmers.

'The Graphical Kernel System (GKS) provides a set of functions for computer graphics programming. GKS is a basic graphics system that can be used by the majority of applications that produce computer generated pictures'. (Quote from GKS Clause 0)

'The Graphical Kernel System (GKS) provides a functional interface between an application program and a configuration of graphical input and output devices. The functional interface contains all basic functions for interactive and non-interactive graphics on a wide variety of graphics equipment.' (Quote from GKS Clause 4.2).

GKS functions deliver/receive graphics data to/from a graphics device or set/inquire the internal GKS states which determine how the graphics data is to be treated. The definition of a GKS function includes:

- its name,
- the valid preconditions for the function
- the lowest GKS level at which the function may be used,
- the input and output parameters of the function: no GKS parameter is both an input and an output parameter.

An example of the definition of a GKS function is given in Figure 1.

REQUEST STROKE WSOP,WSAC,SGOP L0b

Parameters

In	workstation identifier		N
In	stroke device number	(1..n)	I
Out	status	(OK,NONE)	E
Out	normalization transformation number	(0..n)	I
Out	number of points	(0..n)	I
Out	points in stroke	WC	nxP

Effect: GKS performs a REQUEST on the specified STROKE device. If the break facility is invoked by the operator, the status NONE is returned; otherwise, OK is returned together with the logical input value which is the current measure of the STROKE device. This consists of a sequence of not more than 'input buffer size' (in the stroke data record) points in world coordinates, and the normalization transformation number, which was used in the conversion to world coordinates. The points in the stroke all lie within the window of the normalization transformation.

NOTE: If an operator enters more points than the stroke input buffer size (in the workstation state list) allows, the additional points are lost. The operator should be informed of this situation.

References:

4.6.5
4.8.1
4.8.2
4.8.3
4.8.4

Errors:

7 GKS not in proper state: GKS shall be in one of the states WSOP, WSAC, or SGOP.
20 Specified workstation identifier is invalid.
25 Specified workstation is not open.
38 Specified workstation is neither of category INPUT nor of category OUTIN
140 Specified input device is not in REQUEST mode.

Figure 1 - Example of the definition of a GKS function

In the definition of a parameter of a GKS function, the data type, coordinate system, and range of permitted values are defined in clause 6.1 of GKS - see Figures 2, 3, and 4 respectively.

The data type can be a simple type, which is one of the following:

I	integer	whole number
R	real	floating point number
S	string	number of characters and character sequence
P	point	2 real values specifying the x and y coordinates of a location in WC, NDC, or DC space.
N	name	identification (used for error file, workstation identifier, connection identifier, workstation type, specific escape function identification, GDP identifier, pick identifier, segment name, and identification of a GKS function). In a programming language, not all these instances of the name data type need be bound to the same data type in the language
E	enumeration	a data type comprising a set of values. The set is defined by enumerating the identifiers which denote the values. This type could be mapped, for example, onto scalar types in Pascal, or onto integers in FORTRAN.

Alternatively, the data type can be a combination of simple types, thus:

f) a vector of values, for example 2xR
g) a matrix of values, for example 2x3xR
h) a list of values of one type: the type can be a simple type or a vector, for example, nxI and nx4xR
i) an array of values of simple type, for example, nxnxI
j) an ordered pair of different types, for example, (I;E)

or it can be:

D	data record	a compound data type, the content and structure of which are not defined in this standard.

Figure 2 - The possible data types in GKS

For coordinate data, the relevant coordinate system is indicated:

k) WC : world coordinate system;
l) NDC : normalized device coordinate system;
m) DC : device coordinate systems.

Figure 3 - Possible coordinate systems in GKS

Permitted values can be specified by:

n) a condition, for example, >0 or [0,1]; the latter implies that the value lies between 0 and 1 inclusively;
o) a standard range of integer values, for example, (1..4);
p) a range of integer values in which the maximum is determined by implementation or other constraints, for example, (32..n). An occurrence of n does not necessarily imply any relationship with other occurrences of n: n merely denotes a variable integer in this context;
q) a list of values which constitute an enumeration type, for example, (SUPPRESSED, ALLOWED).
r) an ordered list of any of the above.

Figure 4 - Range of permitted values in GKS

In a language binding of GKS:

a) 'The abstract functions and data types of GKS need to be expressed in terms of the constructs available in the host language ... in a natural and efficient manner' (Quote from Annex C of GKS).
b) In one case (pick identifier), the default for a particular GKS state variable is language dependent and must be bound so.
c) Error conditions specific to a language binding may be defined (for example, 'array size too small' in FORTRAN). GKS specifies the range in which language-binding-dependent error messages must lie.

Following through with the example shown in Figure 1, specific instances from the Pascal, FORTRAN, Ada, and C GKS bindings are given in Figure 5.

```
Procedure GReqInput(
      InputClass          :          GEInputClass;
      Wsld                :          GTWsld;
      InputDeviceNum      :          GTInt1;
      VAR  Status         :          GEReqStatus;
      VAR  inputvalue     :          GRInput          );

(* The constants are defined elsewhere *)
Type
GEInputClass    =        (GVLocator..GVString);
GTWsld          =        INTEGER;
GTInt1          =        1..MAXINT;
GEReqStatus     =        (GVStatusOK, GVNoInput, GVStatusNone);
GRInput         =        record
                         case InputClass : GEInputClass of
                                  ...
                         GVStroke: (NormTranStroke:GTint0;
                         Num          :   GTint0;
                         Points       :   GAPointArray);
      only stroke record construct is shown
        ...
       end;
GTint0          =              0..MAXINT;
GAPointArray    =              array[GTMaxPoint1] of GRpoint;
GRPoint         =              record
                               x,y : REAL;
                               end;
GTMaxPoint1     =              1..GCMaxpoint;
```

The Pascal GKS binding contains the following alternative binding for this function:

```
Procedure GReqStroke(
          Wsld                 :      GTWsld;
          StrokeDeviceNum      :      GTInt1;
     VAR  Status               :      GEReqStatus;
     VAR  StrokeMeasure        :      GRStroke);
```

with the additional data type -

```
type GRStroke          =     record
                               NormTranStroke   :   GTInt0;
                               Num              :   GTInt0;
                               Points           :   GAPointArray;
                             end;
```

Figure 5 - Example in GKS Pascal binding

```
SUBROUTINE GRQSK (WKID, SKDNR, N, STAT, TNR, NP, PXA, PYA)
Input:    INTEGER   WKID
          INTEGER   SKDNR
          INTEGER   N
Output:   INTEGER   STAT
          INTEGER   TNR
          INTEGER   NP
          REAL      PXA (N), PYA (N)
```

Figure 6 - Example in GKS FORTRAN binding

```
Gint      -    integer
Gfloat    -    floating point number

typedef   struct {
          Gfloat      x;
          Gfloat      y;
} Gwpoint;

typedef   struct {
          Gint        transform;
          Gint        n_points;
          Gwpoint     *points;
} Gstroke;

typedef   enum {
          GOK,
          GNONE
} Gistat;

typedef   struct {
          Gistat      status;
          Gstroke     *stroke;
} Gqstroke;

greqstroke (ws, dev, response)
Gint      ws;
Gint      dev;
Gqstroke  *response;
```

There are alternative methods for binding this function in Appendix A of the C GKS binding for nonconforming C Compilers.

Figure 7 - Example in GKS C Binding

```
procedure REQUEST_STROKE
      WS                    :  in    WS_ID;
      DEVICE                :  in    DEVICE_NUMBER;
      STATUS                :  out   INPUT_STATUS;
      TRANSFORMATION        :  out   TRANSFORMATION_NUMBER;
      POSITION              :  out   WC.POINT);

type WS_ID is new POSITIVE;
type DEVICE_NUMBER is new POSITIVE;
type INPUT_STATUS is (OK,NONE);
type TRANSFORMATION_NUMBER is new NATURAL;

package WC is new GKS_COORDINATE_SYSTEM(WC_TYPE);
type WC_TYPE is digits PRECISION;
```

NOTE: GKS_COORDINATE_SYSTEM is a generic package which defines an assortment of types that support each of the GKS coordinate systems.

Figure 8 - Example in GKS Ada Binding

INFORMATIVE ANNEX B - GKS BINDINGS GENERIC ISSUES

The following issues do not have a solution or a resolution. They are intended to illustrate the pros and cons of various issues that occured during implementation of GKS bindings.

ISSUE #1: What should be the criteria for determining if the functional interface standard should be bound to a particular programming language? (Guideline 5)

Discussion: Some guideline must be imposed to determine when a binding may be done to a language. In general, only languages for which there are existing standards have been bound. However, there may be sufficient interest in binding the functional interface standard to a language for which a standard has not been currently adopted. The possibility of a 'Generic' Binding has been raised, which offers another solution to this problem.

Note: It is assumed that the appropriateness of a binding development effort to the language has been assured.

Alternatives:

1 - Only languages for which a national body or ISO standard has been previously developed.

2 - Only languages for which a national body or ISO standard has been developed or for which a standard is in process of being developed. The binding will not be adopted as a standard until the language is, even though the work might be completed before that time.

3 - Any language, with the choice being made on user need.

4 - Any language for which a capable and willing worker shows up.

5 - Bind to all languages with one generic binding.

Arguments:

a. Pro 1,2 - eases problems with bureaucratic overhead.

b. Con 5 - a generic binding probably won't work for weird languages like APL, teco, SNOBOL.

c. Pro 3 - speaks to the needs of the user community.

d. Pro 4,5 - benefits of system function standardization should be universally available even in non-standard languages.

e. Con 2,3,4,5 - no standard exists for the language, so what specifications are bound to?

f. Con 3,4 - possibility of overlapping standards for a language.

g. Pro 2, Con 1 - the intent is the same (to bind to standard languages) but the timing problems of binding to new languages are minimized. An effort to bind to a language may be started at any time, but can NOT be completed until the dependent standards are at ISO IS stage.

h. Con 3,4 - if the user community is diverse enough to want a standard binding, let them standardize the language as well.

ISSUE #2: To what event(s) should new versions of a binding be tied? (Guideline 8)

Discussion: Once a binding is accepted as a standard, the question arises as to when, if ever, consideration must be made to reviewing that binding and, perhaps, updating it.

Alternatives:
1 - Only when new versions of the language are developed.

2 - Only when new versions of the functional interface standard are developed.

3 - Only when both the language and the functional interface standard have changed.

4 - Neither 1 nor 2, but on its own five year cycle.

5 - When either the language or the functional interface standard have changed.

6 - Never.

Arguments:
a Pro 1,5 - New language means new functionality which can substantially improve the language binding or may remove functionality which the language binding relied upon.

b. Con 5 - Could have updates on two to three year cycle.

c. Con 1,2,3,4 - Does not properly reflect current practice (may be four years out of date.)

d. Pro 2,5 - Without this, the new functionality would not become available until after a delay.

ISSUE #3: How are 'real world' limitations to the standard taken into consideration? (Guideline 9)

Discussion: There are cases where many implementations of a language have chosen to be non-standard to reduce the resources consumed. As an example, many Pascal compilers have a limitation on the length of identifiers.

Alternatives:
1 - Bind to the standard only. Ignore actual implementation considerations (non-standard conforming).

2 - Provide two separate bindings.

3 - Bind to the standard only, but consider submissions for non-standard bindings for typical non-standard implementations of a language. These may be produced as 'working papers' or guidelines to implementors.

Arguments:
a. Pro 1, Con 2 - this encourages conformance to the standard language.

b. Pro 3 - provides a guideline for doing non-standard things in a standard way and makes the standard more accessible.

c. Con 2 - conflicting standards, two standards for a language.

d. Con 3 - difficult to determine what is 'typical'.

e. Pro 3 - implementors' interests have dictated the non-standard modifications found in some C and FORTRAN compilers. Implementors of a language binding and a system facility should have guidance in order to help 'standardize' their non-standardization.

ISSUE #4: How much of the 'spirit' of the language is taken into consideration when developing a binding? (Guideline 27)

Discussion: Multi-language shops and implementation peculiarities could require changes from or avoidance of certain standard capabilities of a language.

Alternatives:

1 - Use the 'full richness' of the language, as specified by the standard for the language, if applicable.

2 - Only bind to a minimal subset of the language.

Arguments:

a. Pro 1 - Bindings are being developed to standard languages and should use the features of the language.

b. Pro 2 - In the real world, there are limitations that the standard sometimes doesn't take into consideration.

c. Pro 1 - These problems are actually problems of the respective language committees.

d. Pro 2 - Provides more programmer and program portability.

e. Pro 2 - More effective in a multi-language environment.

ISSUE #5: For what types of programmers should the bindings be developed? (Guideline 28)

Discussion: This issue is similar to the preceding `full richness' issue. In addition, it impacts the amount and trueness of mapping of binding names and data types directly from the corresponding functional interface standard names and types. If knowledge of the functional interface standard is assumed, then its terminology should be used consistently. However, if only the language is known, perhaps more appropriate or easily recognized terms should be used. This issue actually is an attempt to determine the user community of the standard.

Alternatives:

1 - Assume knowledge of the functional interface standard.

2 - Assume expertise in the language, but not in the functional interface standard

3 - Assume neither.

4 - Assume both.

Arguments:

a. Con 3, Pro 1 - users of the functional interface standard will often be tool builders, and thus experts in that field of application.

b. Pro 2 - Most programmers know the language they write in.

c. Con 1,4 - The lower levels of the functional interface standard may be used by those who do not have any knowledge of the functional interface standard.

d. Pro 1,4 - Ignorance of the functional interface standard is not a reason for avoiding the terminology used in that standard.

e. Pro 1,4 - The language binding should not be a tutorial on the functional interface standard or the language.

f. Pro 1 - A programmer shouldn't be using the functional interface standard without knowledge.

g. Pro 4 - Programmers inexperienced in either the functional interface standard or the language can gain experience quickly and, in fact, seldom remain inexperienced.

h. Pro 4 - In order to do the job, the programmer must understand the problem, the goals, and the method to achieve the goals of the project. If he doesn't understand these, he must be willing to obtain whatever knowledge is required; the standards are not obliged to hand-feed him this knowledge.

i. Pro 2 - The role of the systems function in an application is rarely dominant; most application developers have many other things to know and might appreciate help in the functional area.

ISSUE #6: Should a generic binding (i.e., a single binding that can be used for many programming languages) be the only binding developed for the functional interface standard? (Guideline 28)

Discussion: A system facility committee might produce only a generic binding and require a preprocessor for each language to create appropriate statements for the target language.

Alternatives:

1 - Yes, only provide a generic binding, with no other bindings to other languages for the functional interface standard.

2 - No, other bindings are necessary. This does not preclude an additional generic binding for those who need one. It may not be economic to generate specific bindings for less used languages

Arguments:

a. Pro 1 - precedent has been set by abstract databases.

b. Pro 1 - this should be the simplest to validate.

c. Pro 2 - Other bindings have already been produced and accepted internationally.

d. Con 1 - will not work for interpretive languages.

e. Pro 1 - Maximum utility in multi-language environment (saves economy!)

f. Pro 2 - maximum flexibility for implementors who want to take advantage of either.

g. Con 2 - Possibility of two bindings to a language (but only if a generic binding is created as well).

h. Con 1 - the feasibility of a single run-time procedural interface depends on factors beyond the control of the graphics system,i.e., operating systems factors.

ISSUE #7: Should the development of a generic binding be supported? (Guideline 29)

Discussion: A generic binding can be presumed to be used in one of three ways:
1) to be translated by a preprocessor into some host language for further processing,
2) as a 'lowest common denominator' for the functional/data interface used by all supported languages,
3) as the common basis for a set of shell routines. The shell routines would presumably translate function references from the host language to the Kernel routines.

Alternatives:
1 - Oppose development of all but language specific bindings for the functional interface standard.

2 - Support the development of a Generic Binding for the functional interface standard.

Arguments:
a. Pro 1 - less confusion in the market place.

b. Pro 1 - produces only verifiable implementations.

c. Pro 1 - most economical approach in terms of resource commitment.

d. Pro 2 - there is a well documented desire to have one.

e. Pro 2 - could standardize lowest common denominator in the functional interface standard.

f. Pro 2 - would maximize flexibility for implementors who want to take advantage of either.

g. Con 1 - if the work is done to develop a generic binding, why not take advantage of it?

h. Con 2 - there is no generic language standard yet.

i. Con 2 - would create possibility of two bindings for one language.

j. Pro 2 - no reason not to document a well used interface.

k. Pro 2 - provides greater consumer choice by providing greater independence between compilers and system facilities.

l. Pro 1 - a general binding (if it does not fit any programming language) requires each implementor to define language specific bindings - thus failing to achieve portability of users' programs.

ISSUE #8: If the development of a generic binding is supported, should the development of more than one generic binding be supported? (Guideline 30)

Discussion: As shown in the discussion of Issue #4b above, there are three, and possibly more, methods for handling the multi-language environment.

Alternatives:
1 - No, a single generic binding should be enough.

2 - Yes, any number of generic bindings should be supported.

Arguments:
a. Pro 1 - there is the possibility that many request for generic bindings would develop otherwise.

b. Pro 1 - reduces confusion to have only one generic binding.

c. Pro 2 - no reason not to document a well used interface.

d. Pro 2 - If work has been done to develop a given generic binding, why not take advantage of it?

e. Con 2 - Could be difficult to verify program or package conformance.

ISSUE #9: What provisions/attention should be taken for the possibility of `shell' bindings built on top of a generic binding? (Guideline 31)

Discussion: When binding the functional interface standard to a specific language, there may be occasions when a choice must be made between two adequate methods of binding. Should the binders be careful to make the choice so as not to preclude the building of a `shell' for those in the multi-language environments?

Alternatives:
1 - None.

2 - The language bindings should not use facilities which cannot be layered with the generic binding.

Arguments:
a. Pro 2 - supports a known constituency.

b. Con 2 - capabilities of the generic binding are not known yet.

c. Pro 2 - Reduces the maintenance costs for implementors.

d. Con 2 - changes to the Kernel will have wide-spreading effects.

ISSUE #10: How should standard functions be bound to a language? (Guideline 33)

Discussion: For easy mapping back to the functional interface standard document, some consistent procedure must be defined for binding the standard functions. An investigation has been made into those standard functions which it is felt fall into a category which does not require one for one mapping into a binding.

Alternatives:
1 - bind one to one.

2 - bind one to many.

3 - bind many to one.

4 - bind many to many.

5 - bind one to one, with the exception of certain functions, which seem to fit nicely into multiple combinations.

ISSUE #11: Must strict conformance be followed in ordering of parameters, or should it be possible to change the order of the parameters bound to a given language from those presented in the functional interface standard document? For example, some languages may require that an array, which is an output parameter, be accompanied by its length as an input parameter. (Guideline 34)

Discussion: New parameters, not described in the functional interface standard, must sometimes be added. Sometimes these new parameters are input parameters directly associated with given output parameters. Grouping of input parameters followed by output parameters is perceived by some as good software engineering, though it is changing the ordering from the functional interface standard document.

Alternatives:

1 - Follow strict conformance in ordering. (All input parameters in the binding must precede output parameters in the binding).

2 - Change parameters depending upon the language constructs and philosophy. (An input parameter in the binding arising out of an output parameter in the functional interface standard (length of an array) is always associated with the array).

Arguments:

a. Pro 1 - programmer portability.

b. Pro 1 - consistency with the functional interface standard document and other bindings.

c. Pro 1 - all languages can handle this alternative (ease of use in multi-language environments.)

ISSUE #12:Must strict conformance be followed in the definition of parameters as input and output, as defined in the functional interface standard document, or should it be possible to change the type of a parameter from input to output and vice versa? (Guideline 35)

Discussion: Some languages can return pointers to data areas in an efficient manner. Others handle structures efficiently.

Note: the following arguments actually center around the changing of input parameters to output parameters in the case of the C binding, where workstation id and segment name could be pointers assigned by the system instead of being assigned directly by the programmer.

Alternatives:

1 - Follow strict conformance in input/output.

2 - Change parameters depending upon the language constructs and philosophy.

Arguments:

a. Con 1 - doesn't necessarily use the full language capabilities.

b. Pro 2 - efficiency of implementation.

c. Pro 1 - programmer portability.

d. Pro 1 - consistency with the functional interface standard document and other bindings.

e. Pro 1 - may actually change the meaning of the function. given to generate the 'next number'.

f. Con 2 - the programmer will be lulled into a fake sense of security.

g. Pro 1 - all languages can handle this alternative.

h. Con 2 - there may be genuine problems in the functional interface standard with this approach. For example, how would a metafile containing segment names be interpreted if the data stored was a pointer?

ISSUE #13: Should the combination of related SEMANTIC parameters into a single SYNTACTIC parameter be encouraged? (Guideline 36)

Discussion: In some languages, certain information which is spelled out in the GKS functional description as a parameter is implicitly supplied by the language (such as the number of characters in a string, for FORTRAN 77). Also, there are certain functional interface standard entities which naturally fall into groups, as they are used together. These entities may be defined as separate items in the functional interface standard, but could be grouped into record structures in many programming languages.

Alternatives:
1 - yes.

2 - no.

Arguments:
a. Con 2 - doesn't necessarily use the full language capabilities (why add a parameter for information that is already present?).

b. Pro 1 - Long lists of parameters make invoking the functionality more tedious to use, and prone to error. Both the functional standard itself, and the language bindings of that standard, should be designed to reduce lengths of parameter lists as far as possible by allowing all the features of the language, including aggregate type and default parameter values, to be exploited. (An example of where this was not fully done was the GKS functional standard, which seems to have been based in part on the facilities of FORTRAN 66).

c. Pro 1 - efficiency of implementation.

d. Con 2 - requires the programmer sometimes to provide unnecessary information.

ISSUE #14: Should there be a consistent numbering (ordering) of enumerated type members across the various language bindings? (Guideline 37)

Discussion: The language bindings currently being reviewed show no particular consistency of numbering enumerated type members. One suggestion is to follow a guideline in numbering any 'null' value as zero, then following the functional interface standard document order (the null-value rule).

Alternatives:

1 - Follow the functional interface standard strictly.

2 - Develop a consistent numbering which includes the null-value rule.

3 - Don't bother.

4 - As in Alternative 1 when binding to enumerated types, as in Alternative 2 when binding to integers.

Arguments:

a. Pro 1,2 - Why be inconsistent if there is a way to make all the bindings similar?

b. Pro 2 - The null-value rule makes it easier for users of languages whose enumerated types map to integers.

c. Pro 3,4 - Not all languages have enumeration types and explicit ordering and mapping to integers is not always a requirement. For languages without enumeration types, representations other than integer codes might be more appropriate.

d. Pro 1,2 - Easier to implement all bindings in a multi-language environment.

e. Pro 1,4 - (more strongly 1 than 4) Transparent interface to the sequence in the functional interface standard; default values would appear first in the binding in all cases.

f. Con 2 - Some languages conflict with the zero-value rule; in Ada the notation is 'FIRST which implies 1 not zero.

ISSUE #15: How should functional interface standard data types be bound? (Guideline 38)

Discussion: For consistency, there may be a procedure for mapping functional interface standard data types to language data types. However, for languages with sophisticated data types, there may be a more flexible method.

Note: There is a recognized need for a consistent set of rules for matching data types between languages. This is an area under research by other ISO committees, and will be welcomed guidance for binding development.

Alternatives:

1 - each data type from the functional interface standard should be bound to a single data type in the language.

2 - a data type from the functional interface standard may be bound to more than one data type in the language.

Arguments:

a. Pro 1 - ease of conformance checking.

b. Pro 2 - utilizes the 'full richness' of the language.

c. Con 2 - possibility of changing the meaning of the data type, thereby causing portability problems.

d. Con 1 - unnecessarily restrictive.

e. Con 2 - if data typing problems exist in the functional interface standard document, that's where they should be changed, not in the binding.

f. Pro 1 - allows compile-time checking.

g. Pro 2 - can follow the semantics behind the parameters in the functional interface standard.

ISSUE #16: May there be TYPES which are implementation dependent? (Guideline 39)

Discussion: Some languages provide the facility to define data types which are implementation-dependent (e.g., 'generic' in Ada). For example, a binding may require 16 bits precision in integers; this does not preclude using a 32 bit integer. These types conform to the language standard, but their definitions in terms of the base types of the language may differ.

Alternatives:
1 - Yes, where portability of applications within the language is not affected.

2 - No.

3 - Yes, without worrying about portability.

Arguments:
a. Pro 1,3 - Uses the 'full richness' of the language.

b. Con 3 - The whole point of a standard language binding is to provide portability.

ISSUE #17: Should functions be easily identifiable in the applications code ('lexical differentiability')? (Guideline 40)

Discussion: There may be a need to distinguish the standard functions from applications functions.

Alternatives:
1 - Yes

2 - No

3 - Yes, unless language considerations make it unnecessary.

Arguments:
a. Pro 1 - Good programming practice.

b Pro 1 - maintainability.

c. Pro 2 - The selected sentinel characters may have been already used in the implementation.

d. Pro 1,3 - A language like Ada has a package name which may precede the Ada procedure name. If used, this makes a procedure reference easily identifiable.

ISSUE #18: From what roots should binding identifiers or abbreviations be derived? (Guideline 41)

Discussion: In order to be consistent in naming the functions of a binding, there must be a set of words from which to determine abbreviations. Some of the GKS function names have what may seem to be `unnecessary' words (TO, OF). Some of the functional interface names may seem to be unnecessarily complex.

Alternatives:

1 - None (take each language on an individual basis).

2 - Map abbreviations one-for-one from function names and not eliminate any words.

3 - As with 2, but eliminate unnecessary words, in a consistent manner.

Arguments:

a. Con 1 - Transportability problems.

b. Pro 1 - Readability.

c. Con 2 - Would produce a multitude of abbreviations.

d. Pro 2,3 - Transportability.

e. Pro 2 - Good reference back to the functional interface standard document.

f. Con 3 - Poor mapping to functional interface standard document (possible change of meaning).

g. Pro 3, Con 2 - eliminates words like `to', `of', etc.

h. Con 3, Pro 2 - Arbitrary selection of insignificant words. Can a distinction be made concerning which words are unnecessary?

i. Con 1 - Makes each language as hard to bind as all the others; no benefit from experience.

j. Con 2 - Doubtful that this can be done for all languages.

ISSUE #19: How should abbreviations of the function names be concatenated? (Guideline 42)

Discussion: It is possible to have identical names across similar languages. However, the `spirit' (or usual practice) of some languages is to use case distinctions or underscores to concatenate words within identifiers.

Alternatives:

1 - Use no special characters.

2 - Use special, language-dependent characters when available.

3 - Use case distinctions when available.

4 - Use special characters and case distinctions (2 and 3).

5 - Decide on a language by language basis.

Arguments:

a. Con 3 - Difficult to use on terminals with uppercase only.

b. Pro 5 - Each language can use special characters and/or case distinctions which are commonly used in that language and therefore well understood by that user community.

c. Pro 1 - Would provide for consistent naming across bindings.

d. Contra (a) - That is a language-dependent argument. If the normal practice in the language is to use case distinction, difficulties with certain terminals are assumed to be solvable.

ISSUE #20: Should the BASIC/FORTRAN bindings be considered in the development of the abbreviations list? (Guideline 43)

Discussion: Since these bindings have different restrictions than those to the other languages, the consideration of them and their abbreviations could be a determining factor when attempting to develop consistent abbreviations for other bindings.

Alternatives:

1 - yes

2 - no

3 - Only when binding to languages with naming restrictions similar to BASIC and FORTRAN.

Arguments:

a. Pro 1 - consistency across all languages.

b. Con 1; Pro 2,3 - BASIC and FORTRAN are similar in that they have similar naming restrictions. Their abbreviations should not apply to non-similar languages.

c. Pro 1,3 - Promotes program/programmer portability.

d. Con 3 - what similar languages?

ISSUE #21: In languages which have no restriction on identifier lengths, how should abbreviations of the function names be derived? (Guideline 44)

Discussion: It is hoped that each function from the functional interface standard may be abbreviated in some consistent manner across languages.

Alternatives:

1 - Systematically map same-number-of-character abbreviations identically (from a single list of previously defined abbreviations, grouped by number of characters). New entries will be made upon first occurrence of an abbreviation.

2 - Take each language on an individual basis.

3 - Same as #1 above but with the list predefined in advance.

4 - Only allow ONE abbreviation, of a fixed length, for each word of a function name.

5 - Same as #4, but with the choice of using either the single abbreviation or spelling the functional name out in full.

Arguments:

a. Pro 1,3; Con 2 - transportability.

b. Con 1 - need upkeep on list.

c. Pro 2 - uses the 'full richness' of the language.

d. Con 3,4 - the abbreviations would be unduly restrictive for Ada, Pascal, and C.

e. Pro 4,5 - consistency across languages.

f. Pro 1,2 - most flexible.

g. Con 1 - confusing.

ISSUE #22: Should bindings abbreviate the names of the data types in the same manner as functions (within a single language binding)? (Guideline 45)

Discussion: In some languages, it is allowed to have identical names for functions and for associated data types.

Alternatives:

1 - Use names of 8 characters or less.

2 - Use the full functional interface standard data type names.

3 - Use names identical to the functions.

4 - Be consistent within the language.

Arguments:

a. Con 3 - too much conflict, loss of lexical differentiability.

b. Con 1 - 8 characters are often too restrictive and less meaningful names would be forced on users.

c. Con 2 - The data type names specified with the functional interface standard may not be consistent with the design of the language.

d. Pro 4 - Any syntax that is to appear within a program of a given language type should be consistent with the design of that language.

ISSUE #23: Should bindings abbreviate the names of the data types in the same manner as functions consistently across languages (excluding sentinel tags). (Guideline 46)

Discussion: For similar languages, similar data types may have the same names.

Alternatives:
1 - yes, where practical.

2 - no.

Arguments:
a. Pro 1 - consistency, easier for programmer portability.

b. Con 1 - hard to determine whether data types are really similar.

ISSUE #24: What format (documentation structure) should the binding document follow? (Guideline 47)

Discussion: With multiple bindings being developed, the need may exist for similar formats for all.

Alternatives:
1 - Each one should be formatted individually.

2 - Follow a single format as closely as possible.

Arguments:
a. Pro 1 - some languages require more information (i.e., the data structures of Pascal; the packaging of Ada). This alternative allows the document to be in a style suitable for the language.

b. Con 1, Pro 2 - Programmer portability.

c. Pro 2 - Possibility of automating some of the document preparation.

d. Pro 2 - All bindings will eventually be packaged together in the same standard, and therefore should look similar.

e. Pro 2 - Can be reformatted for other uses besides the standard.

f. Pro 2 - Easier for users across languages.

g. Pro 2 - Ease of maintainability.

h. Pro 1 - Could eliminate superfluous information, making the binding less voluminous.