X/Open Snapshot

Interworking API Style Guide

X/Open Company, Ltd.

X/Open Snapshot

Interworking API Style Guide

X/Open Document Number: XO/SNAP/90/030

Set in Palatino by X/Open Company Ltd., U.K.
Printed by Maple Press, U.K.
Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

INTERWORKING API STYLE GUIDE

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring greater value to users through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable system environment, called the *Common Applications Environment (CAE)*. This environment covers all the standards, above the hardware level, that are needed to support open systems. It ensures portability and connectivity of applications, and allows users to move between systems without retraining.

The interfaces identified as components of the Common Applications Environment are defined in the *X/Open Portability Guide*. This guide contains an evolving portfolio of practical applications programming interface standards (APIs), which significantly enhance portability of application programs at the source code level. The interfaces defined in the X/Open Portability Guide are supported by an extensive set of conformance tests and a distinct trademark - the X/Open brand - that is carried only on products that comply with the X/Open definitions.

X/Open is thus primarily concerned with standards selection and adoption. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organizations to encourage the creation of formal standards covering the needed functionalities, and to make its own work freely available to such organizations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

The X/Open Product Family - XPG

There is a single family of X/Open products, which has the generic name ''XPG''.

XPG Versions

There are different numbered versions of XPG within the XPG family (XPG1, XPG2, XPG3). Each XPG version is an integrated set of elements supporting the development, procurement and implementation of open systems products, and each comprises its own:

- XPG Specifications
- XPG Verification Suite
- XPG descriptive guides

- XPG trademark licensing materials

The XPG trademark (or ‘‘brand’’) licensed by X/Open always contains a particular XPG version number (e.g., ‘‘XPG3’’) and, when associated with a vendor’s system, communicates clearly and unambiguously to a procurer that the software bearing the trademark correctly implements the corresponding XPG specifications. Users specifying particular XPG versions in their procurements are therefore certain as to the XPG specifications to which vendors’ systems conform.

XPG Specifications

There are four types of XPG specification:

- **XPG*n* Formal Specifications**

  These are the long-life XPG specifications that form the basis for conformant/branded X/Open systems, and are the only type of XPG specification released with an XPG version number (e.g., ‘‘XPG3’’). They are intended to be used widely within the industry for product development and procurement purposes. Currently, all XPG Formal Specifications are included in Issue 3 of the X/Open Portability Guide.

  Individual XPG specifications are released as Formal Specifications only as part of the formal release of the complete XPG version to which they belong. However, prior to the launch of that XPG version, they may be made available as:

- **XPG Developers’ Specifications**

  These are specifically designed to allow developers to create X/Open-compliant products and applications in advance of the formal launch of a future version of the XPG.

  Developers’ Specifications may be relied on by product developers as the final, base specification that will appear in a future XPG. They are made available beforehand in order to meet the need of product developers for advance notification of the contents of XPG Formal Specifications, to assist in their product planning and development activities.

  By providing such advance notification, X/Open makes it possible for products conforming to future XPG Formal Specifications to be developed as soon as practicable, enhancing the value of XPG itself as a procurement aid to users.

- **XPG Preliminary Specifications**

  These are XPG specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for validation purposes. A Preliminary Specification is not a ‘‘draft’’ specification. Indeed, it is as stable as X/Open can make it, and on publication will have gone through the same rigorous X/Open development and review procedures as XPG Formal and Developers’ Specifications.

  Preliminary Specifications are analogous with the ‘‘trial-use’’ standards issued by formal standards organizations, and product development teams are intended to develop product on the basis of them. Because of the nature of the technology they are addressing, they are untried in practice, and they may therefore change before being published as an XPG Formal or Developers’ Specification.

- **Snapshot Specifications**

  These are ‘‘draft’’ documents, that provide a mechanism for X/Open to disseminate information on its current direction and thinking to a limited audience, in advance of formal publication, with a view to soliciting feedback and comment.

This Document

This document is a Snapshot (see above). It provides guidelines for the style of Interworking Application Program Interfaces (APIs) defined by X/Open. The present version does not cover some aspects needed for a common Interworking API style, in particular it omits:

- a semantic description of *open*/*close*, *bind*/*unbind*, *data_transfer*;
- security;
- Quality Of Service (QOS), and
- a requirement for use with Remote Procedure Call (RPC).

Some interworking APIs have already been defined, or are being finalised within X/Open (i.e., XTI, XDS, X.400); differences exist between these APIs.

It is intended in a future edition of this guide to define all of what is missing today, in order to ease harmonisation of Interworking API definitions.

Disclaimer:

This document represents the interim results of an X/Open technical activity. While X/Open currently intends to progress this activity towards publication of an X/Open Guide, X/Open is a consensus organisation, and makes no commitment regarding publication.

Similarly, this document does not represent any commitment on behalf of any X/Open member to make any specific products available now or in the future.

Do not specify or claim conformance to this document.

Trademarks

X/Open and the 'X' device are trademarks of X/Open Company Limited in the U.K. and other countries.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Palatino is a trademark of Linotype AG and/or its subsidiaries.

Acknowledgements

X/Open is grateful to the following organisations for permission to reproduce extracts from their respective publications:

- the IEEE TCOS POSIX 1003.1 Committee,
- the British Standards Institute,
- the X.400 API Association, and
- the MAP/TOP User Group and the TOP Vendor Group.

Referenced Documents

The following documents are referenced in this guide:

- ISO TC97/SC22 document N754, Proposed DTR 10182, Guidelines for Language Bindings. (The final draft of a proposed ISO Draft Technical Report (DTR).)
- MAP 3.0 Interface Model and Specification Requirements, Manufacturing Automation Protocol Specification Version 3.0., Appendix 7, Application Interface to Network Services, Attachment 1, Application Interface Model and Specification Requirements, Section 4, Interface Specification Requirements, Pages A7A1-4.1 to A7A1-4.28. (Published by General Motors Corporation.)
- X/Open Portability Guide, Issue 3:

  — Volume 1 - XSI: System Commands & Utilities

  — Volume 2 - XSI: System Interfaces & Headers

  — Volume 3 - XSI: Supplementary Definitions

  — Volume 4 - Programming Languages

  — Volume 5 - Data Management

  — Volume 6 - Window Management

  — Volume 7 - Networking Services
- IEEE Std 1003.1-1988, POSIX, IEEE Standard Portable Operating System Interface for Computer Environments.
- Converting POSIX into a Language-Independent Specification (Draft Version 0.03, August 89) by K.K. Barker, M.F. Connors Jr. and J.S. Kimmel.

  This proposal describes selected steps in a model-based process used to convert the C-based function interfaces described in the IEEE POSIX Standard (1003.1) into abstract procedure interfaces that are independent of any specific programming language. It is incomplete and will be revised as additional knowledge and review is gained.
- ANSI C Standard, American National Standard for Information Systems, Programming Language C (X3.159-1989).

  There is also a closely related Draft International Standard (DIS 9899: Programming Language C).
- Realtime Extension for Portable Operating Systems (IEEE Std 1003.4, Draft 9), Section 9, Asynchronous Event Notification. (A draft extension to the POSIX standard.)
- OSI Object Management API Specification, Draft 4.0 (April 1990). (To be published by X/Open and the X.400 API Association.)

Chapter 1

Introduction

1.1 OVERVIEW

X/Open is in the process of defining Application Program Interfaces (APIs) for Interworking, which will be documented in API specifications and published in the **X/Open Portability Guide**. It is the purpose of this Style Guide to assist in the design of the APIs and the preparation of their specifications. The benefits expected to accrue from the use of a Style Guide include:

- reduced effort in designing and documenting individual APIs;
- greater consistency among APIs, making them easier for users (e.g., application programmers) to learn, and
- reduced effort to implement APIs by sharing mechanisms between them.

Topics covered in this Style Guide in order to achieve these goals include:

- avoidance of conflict with other APIs, particularly over naming conventions;
- consistent choice of some essentially arbitrary decisions, such as the layout of specification documents;
- reminders of requirements imposed by the **X/Open Portability Guide** and other sources;
- the general style of an interface, and
- programming language-specific issues.

1.2 LANGUAGE-INDEPENDENT SPECIFICATION

Many standards-setting bodies are moving towards language-independent specifications, where the functions are described in abstract and the C syntax, for example, is given in a separate binding.

It is not yet appropriate to require X/Open specifications to be written in this way, and this Style Guide recommends combining the functional specification with the C binding.

Writing language-independent functional specifications and separate bindings to specific languages is likely to be introduced in the future.

1.3 CONTENTS

Chapter 2, **Language Bindings** provides general guidelines derived from an ISO Technical Report (see **Referenced Documents**). and is not specific to either the application services defined by X/Open or to particular programming languages, although some guidelines have been made more specific.

Chapter 3, **API Content** gives guidelines on the functional content of APIs. Topics covered include functional interface, event management, support for national languages (internationalisation), error handling and several other topics.

Chapter 4, **Bindings to APIs** gives guidance for the preparation of language bindings to X/Open Interworking APIs. It is largely concerned with naming conventions and the design of parameter lists.

Chapter 5, **Layout of Specifications** presents guidelines for the contents and layout of specifications for APIs. Considerable flexibility is provided to ease the process of adopting APIs from different sources and covering different service areas.

Chapter 6, **Rules for C Bindings** sets out rules for the preparation of bindings to the C language.

Appendix A, **Name Prefixes and Abbreviations** is an initial list of abbreviations to be used when forming function names according to the method given in **Chapter 4**, **Bindings to APIs**.

Appendix B, **Rationale** is a rationale for the Style Guide.

Appendix C, **Example Specifications** is part of the rationale and consists of example specifications from reference documents.

1.4 TERMS AND ABBREVIATIONS

ANSI	American National Standards Institute
API	Application Program Interface
APIA	[X.400] Application Program Interface Association
BSD	Berkeley System Distribution
CAE	[X/Open] Common Applications Environment
GKS	Graphical Kernel System
GW	Gateway
IEEE	Institution of Electrical and Electronic Engineers
ISO	International Standards Organisation
MAP	Manufacturing Automation Protocol
MD	Management Domain
OM	Object Management
OSI	Open Systems Interconnection
POSIX	Portable Operating System Interface for computer environments
UTC	Co-ordinated Universal Time [French abbreviation]
XPG	X/Open Portability Guide
XSI	X/Open System Interface
XTI	X/Open Transport Interface

Chapter 2

Language Bindings

2.1 INTRODUCTION

This Chapter presents a number of language binding guidelines which should be followed by anyone who is binding the functional specification of an application service to a particular programming language.

These guidelines are based on an ISO document (see **Referenced Documents**) and are listed with the original reference numbers from that document. Guidelines which have been modified appear in brackets (i.e., [modified like this], and deleted material thus [ ]).

Where the guidelines refer to a system facility, they should be read as referring to an application service.

2.2 ORGANISATIONAL GUIDELINES

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

2 [The system facility committee] should have primary responsibility for the language binding. [ ]

3 [The language] committee needs to be consulted as early as possible. The two committees have complementary responsibilities and concerns.

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

[This Style Guide contains a set of guidelines for producing language bindings for interworking system facilities defined by X / Open.]

5 [This guideline is not applicable, since it refers to the OSI standards development cycle.]

2.3 GENERAL TECHNICAL GUIDELINES

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

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.

8 All system functions should appear atomic to the application program.

2.4 RECOMMENDATIONS FOR FUNCTIONAL SPECIFICATIONS

9 [A functional specification may use an abstract description, or may be described in terms of its C language binding.]

10 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.

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

12 The system facility should recover from errors wherever 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.

2.5 PROCEDURAL INTERFACE GUIDELINES

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

14 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.

15 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.

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

17 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.

2.6 SUGGESTED ACTIONS FOR STANDARDS COMMITTEES

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

(See also **Chapter 4**, **Bindings to APIs** on naming.)

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

20 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.

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

2.7 RECOMMENDATIONS FOR PROGRAMMING LANGUAGE COMMITTEES

22-24 [Guidelines 21 to 24 are not applicable, since they are design rules for future language standards.]

2.8 PROCEDURAL LANGUAGE BINDING GENERIC ISSUES

25 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.

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

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

28 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.

29-30 [Guidelines 29 and 30 are ignored.]

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

32 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.

33 If an error numbering scheme is used, the language-specific messages should be clearly separate from the common messages. In addition, the messages should be allocated separate ranges for each language.

34 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 recognised 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.

35 Parameters should be bound rigidly in the order they appear in the functional interface standard description. Added parameters should follow the rule that input parameters must precede output parameters.

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

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

38 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.

39 There are some instances when a single data type from the functional interface standard may be bound to more than one data type in the programming language, if the language allows the definition of data types equivalent to, or subsets of, the basic type.

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

41 In general, the standardised function calls must be distinguishable from user-defined functions.

42 Abbreviations should be generated from the functional interface standard names, with unnecessary words eliminated in a consistent manner. This provides a relatively straightforward method of mapping binding identifiers back to the functional interface standard.

[Refer to **Chapter 4**, **Bindings to APIs** for the specific rules to be followed.]

43 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.

44 [A single approved abbreviation list should be used for all languages.]

45 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.

46 Data types should be abbreviated in a consistent manner. However, 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.

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

48 [This Guideline refers to the contents and layout of specifications and is deferred to **Chapter 5**, **Layout of Specifications**.]

49-55 [Guidelines 49-55 are for user-defined interfaces and extending programming languages, and are not relevant.]

Chapter 3

API Content

3.1 INTRODUCTION

This section gives guidelines on the functional content of X/Open Interworking APIs. Topics covered include functional interface, event management, error handling, object management, support for national languages (internationalisation), resource constraints, the level of an API and extensions to an API.

3.2 FUNCTIONAL INTERFACE

A functional interface is preferred to a definition of data structures, since this permits different implementations of the API, including vendor-specific implementations with extensions.

As an example, consider that an *error* could be specified as an aggregate [structure] containing an *error_number*, an *error_class* and a *reference* to the operation which caused the error. Instead, it is preferable to specify functions to return the *error_number*, *error_class* and *reference* when given a particular *event* as parameter.

This frees implementors to choose the best underlying representation, and enables individual vendors to add extra information (e.g., a message) if required without changing the standard interface. Additional functionality simply results in additional functions, rather than changed aggregate definitions.

It also makes it easier to integrate errors arising in different parts of the implementation (e.g., local system errors versus remote operation errors).

3.3 EVENT MANAGEMENT

3.3.1 Introduction

The basic requirement for event management in application programs which use network services stems from the need to use remote services asynchronously, and to invoke multiple concurrent operations. It must also be possible to combine these with other sources of events such as file system pipes and window systems.

Event Management as used here is at a higher level than, for example, the Asynchronous Event Notification proposal of POSIX 1003.4, and is concerned with management of the data associated with the event as well as just signalling the occurrence of an event.

For the foreseeable future it is expected that applications will have to combine all event sources themselves. They will need to use whatever underlying system facilities exist for event notification and then separate the streams of events and route them to the appropriate interface.

This section presents a model event management interface to be used by all APIs which need one. A simple programming example is presented at the end of the section.

3.3.2 Background

This section contains some background material, intended to explain the particular meaning given to terms in this proposal.

Some form of management is required to deal with events which occur asynchronously, not in a process's main flow of control. Suitable mechanisms are not presently available in a system-independent way, so interim guidelines are presented together with an overview of what is likely in the future.

Events are asynchronous significant changes in a process which may require a change in the flow of control. They usually have some data associated with them and the term *event* is used here to describe both the occurrence and the object which encompasses the data. Examples of events include:

- asynchronous I/O completion (e.g., data arrives over communications link, user presses a key, on-screen window is revealed);
- timer expiration;
- resource availability (e.g., memory, device), and
- program exception (e.g., arithmetic overflow).

The complexity and need for management of events derives from the diverse nature of the events, which can arise from many disjoint sources in a single process. Event management is viewed here as proceeding in three stages: notification, synchronisation and manipulation.

Notification The means by which a process becomes aware that an event has occurred.

Synchronisation To ensure that all process data is consistent before and after event handling.

Manipulation The mechanism used to recover data associated with the event and process it.

These three stages can be combined and implemented in several ways, depending on the combination of event sources, their frequency and application requirements. Availability of the necessary system infrastructure is also a consideration. The three main paradigms are polling, out-of-line and event-driven (though there are many other possibilities):

Polling The simplest technique which requires least support from the system. The program calls in turn a function supplied by each API which may give rise to events. These functions return either ‘*no event has occurred on this API*’ or they return an event (an ID and the associated data). The program usually processes each event as it is notified.

With polling, synchronisation is automatic since the program has a single flow of control. System call overhead can be very low, making it advantageous when event frequency is high. Its disadvantages are that processor time is wasted in the polling loop when no events are occurring, and that high priority events may be significantly delayed. It is useful for simple applications or in combination with other techniques.

Out-of-line Combines notification and manipulation and is exemplified by interrupt or signal handlers. The main flow of the program is stopped and a transfer is made to a specific event handling routine; this then performs whatever manipulation of the event data is required and finally relinquishes control, resuming the main flow.

The major disadvantage of out-of-line event handling is that synchronisation must be explicitly provided by each event handling routine. Such provision requires careful programming and in general may require some means of converting the asynchronous event to a synchronous one which is dealt with later by polling in the main flow of control. Out-of-line event handling is appropriate for high-priority events where the synchronisation requirements are simple and well-defined.

Event-driven Programs which suspend on a system call and are resumed when any event occurs. After processing this event they return to the central suspension point. Thus they depend on the system mechanism for notification and synchronisation, which become automatic from the program’s viewpoint.

Event-driven programs are suitable where there are many sources of events which must be integrated. The major problem is the reliance on a system facility to suspend on a wide variety of event sources, because this is not yet available in a portable way.

3.3.3 System Facilities for Event Notification

The system facilities to suspend and await notification of an event are presently system-dependent although work is in progress to provide a portable mechanism.

Many vendors provide either the System V *poll*( ) or BSD *select()* system calls which each give the ability to suspend until there is activity on a member of a set of file descriptors or a timeout. For this reason APIs must provide access to their underlying file descriptors, where possible.

POSIX are working on a proposal for Asynchronous Event Notification, which provides a much enhanced signal-like mechanism with the required functionality. It is not yet known whether this proposal will be adopted and it could also be significantly changed. They are also considering a threads facility.

The X/Open Kernel/Real Time Working Group are also looking at the requirements and possible solutions in this area.

Implementations of APIs will require modification when this facility is standardised and available; note in particular that the new event notification facility makes use of an associated asynchronous I/O facility. These modifications will change the API slightly since applications will suspend by using new event classes rather than file descriptors. It will then be necessary for APIs to provide access to the event classes instead of the file descriptor, using a function similar to *file_number*() explained below. Application programs will also require changes to reflect the different underlying event mechanism.

3.3.4 Overview of Mechanism

The mechanism centres around an *event* object class. *Events* are associated with a particular *access_point* to a service and are, or appear to be, held in a queue for that *access_point*. The next two sections describe *access_points* and *events* in turn. See **Section 3.4**, **Error Handling** for full details of the *status* type which occurs as the result of the operations.

3.3.5 Access Points

An *access_point* identifies a particular association between the application program and a system facility or service provider. An application may have multiple *access_points* to the same service, if this is supported by the particular service provider. The notion of an access point corresponds to that of a file descriptor in the XSI file system.

In general, the operations provided on *access_points* will depend on the individual APIs. *Access-points* are generally constructed as a result of operations such as *bind* or *open* and destroyed by operations such as *unbind* or *close*. They are then passed as a parameter to most other API functions.

We are only concerned here with one selector which returns a *file_descriptor*. This is required to support the present system-dependent event notification facilities, and similar selectors will be required in the future for any new event notification facilities which are introduced.

Where system facilities permit, vendors may provide extensions to allow use of additional notification mechanisms, such as semaphores.

NAME

file_number - obtain the file descriptor associated with an access point **(OPTIONAL)**

SYNOPSIS

int file_number(
Access_Point_Type access_point
);

DESCRIPTION

file_number() obtains the file descriptor associated with an *access_point*. The file descriptor may be used in subsequent calls to vendor-specific system facilities to suspend the process (e.g., System V *poll*( ) or BSD *select*( )). It should not be used for any other purpose.

ARGUMENTS

access_point (access-point-type)
An established access point to the service.

RESULTS

file_descriptor (integer)
The file descriptor associated with the given access point.

ERRORS

Applications are not permitted to call *file_number*( ) with an *access_point* argument which is invalid, and the result of doing so is undefined.

No errors are reported by *file_number*( ).

NOTE

This operation is optional and will only be provided if suitable underlying system facilities such as *poll*( ) or *select*( ) are available.

3.3.6 Events

Events are distinct occurrences which are usually the result of some preceding asynchronous operations on the API.

Events are placed in a queue associated with a particular *access_point* when they occur, and are held there until retrieved by the application. The event queue ordering discipline must be defined by each API. Additional facilities provided by an API may include the ability to examine the queue of pending events and select particular ones.

There is almost always some data which forms part of the event, often the results of some asynchronous operation. Access to this data is by means of selectors, and two are presented here which give access to the identity of the preceding asynchronous operation and to the results of that operation as a whole. Individual APIs may wish to specify selectors which give direct access to individual parts of the data, or to other attributes such as the time at which the event occurred. Such matters should be resolved by each individual API.

Every API must provide several standard operations on *events* and may provide additional operations as desired. The standard operations include a constructor *event_wait*, a destructor *event_delete*, and two selectors *event_data* and *event_identity*.

The application retains access to each event, once it has been returned as a result of *event_wait*, until access is explicitly relinquished by calling *event_delete*.

There is a special constant *no_event* which can be returned by *event_wait* when the queue is empty. The application can use this constant in equality tests after such a call, but it is not permitted to use it as an argument to the other operations.

NAME

event_data - retrieve data associated with an event

SYNOPSIS

```
API_dependent_type event_data(
        Event_Type    event
);
```

DESCRIPTION

event_data( ) retrieves the data associated with a given event.

ARGUMENTS

event (event-type)

A valid event (not *no_event*).

RESULTS

associated_data (API-dependent-type)

Definition of the data type depends on the particular API, which may choose to expose a data structure or provide further functions to access components of the data.

ERRORS

Applications are not permitted to call *event_data*( ) with an *event* argument which is invalid or which has the value *no_event*, and the result of doing so is undefined.

No errors are reported by *event_data*( ).

event_delete( )

NAME

event_delete - release resources associated with an event

SYNOPSIS

```
Status event_delete(
        Event_Type event
);
```

DESCRIPTION

event_delete( ) releases storage and other resources associated with an event. The application *must* call this operation for each event when it needs no further reference to the event, and must *not* make any further reference to the given event after calling this operation.

ARGUMENTS

event (event-type)
The event to which access is no longer required.

RESULTS

status (status-type)
Normally *success*, and otherwise an error.

ERRORS

Applications are not permitted to make any further reference to the given event after calling this operation, and the result of doing so is not defined.

Possible errors returned in *status* are:

error_bad_event
The specified *event* does not refer to an outstanding event.

NAME

event_identity - identify the operation which was the source of an event

SYNOPSIS

API_Dependent_Type event_identity(
    Event_Type   event
);

DESCRIPTION

event_identity( ) identifies the preceding operation which was the source of the event. Typically, the event will contain the results of asynchronous activity which was initiated by the preceding operation.

ARGUMENTS

event (event-type)
A valid event (not *no_event*).

RESULTS

identity (API-dependent-type)
The identity of the operation which caused the given event.

Definition of the identity type depends on the particular API. It might be the *invoke_ID* of a directory operation, for example. In any case it must serve as a unique identifier of the particular event instance.

ERRORS

Applications are not permitted to call *event_identity*( ) with an *event* argument which is invalid or which has the value *no_event*, and the result of doing so is undefined.

No errors are reported by *event_identity*( ).

NAME

event_wait - wait until an event occurs

SYNOPSIS

```
Status event_wait(
        Access_Point_Type   access_point,
        Uint32              timeout,
        Event_Type *        event_return
);
```

DESCRIPTION

event_wait( ) returns when an event has occurred on the given access point or after the given time.

ARGUMENTS

access_point (access-point-type)

An established access point to the service.

timeout (unsigned-32)

The length of time in milliseconds to wait for an event before returning.

If millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. Two special values can also be used, *no_wait* and *wait_forever*. *no_wait* has the numerical value 0 and causes the operation to return immediately. *wait_forever* has the numerical value 2^{32} -1 and causes the operation to block indefinitely, returning only when an event is available or when an error occurs.

RESULTS

event (event-type)

The first event in the queue.

If there is at least one event in the queue, then the next event is returned immediately. Otherwise, if an event occurs before the timeout expires, that is returned. In either case, the returned event is removed from the queue.

If an error occurs or if the queue remains empty until the timeout expires, the constant *no_event* is returned.

status (status-type)

Normally *success* even if there is no event in the queue, and otherwise reports an error.

ERRORS

Possible errors returned in *status* are:

bad_access_point

The specified *access_point* is not active.

NOTE

This operation is designed to be easily implemented using either *poll*( ) or *select*( ) or mechanisms such as the proposed POSIX event notification mechanism.

3.3.7 Programming Examples

This section presents two examples of how the proposed event mechanism might be used in the C language.

The first example is a simple application which processes a sequence of events from a single source (a hypothetical OSI Directory). Details of the preceding operations which give rise to the events are omitted. Errors are defaulted to terminate the process, no checks are needed.

```
main()
{
        DS_access_point dsap;          /* Directory Service access point */

        .
        ds_bind("a-directory", a_context, &dsap);
        .
        /*
         * do whatever setup is required
         * and then do asynchronous operations
         */
        ds_some_operation(dsap, ....);
        ds_another_operation(dsap, ....);
        .
        .
        /* process resulting directory events */
        while (1)
        {
                DS_event event;

                ds_event_wait(dsap, DS_WAIT_FOREVER, &event);
                .
                results    = ds_event_data(event);
                results_ID = ds_event_identity(event);
                .
                .
                ds_event_delete(event);
        }
        .
        .
        /* close down and exit */
}
```

Finally, below is the skeleton of an application program which uses the proposed mechanism alongside the equivalent mechanisms of other system facilities. A hypothetical interface to the OSI Directory has been used as an example of a networking system facility. The other facilities illustrated are the file system and X Windows. Error handling is ignored for simplicity; in reality the *status* which is returned as the C result of *ds_bind*( ) and *ds_event_wait*( ) would need to be checked. The System V *poll*( ) call has been used as an illustrative scheduling primitive.

```
main()
{
        /* declare access points */
        DS_access_point dsap;           /* Directory Service access point */
        int file_descriptor;            /* file system access point (pipe) */
        Display * display;              /* X Windows access point */

        DS_status status;
        .
        ds_error_fatal(FALSE);          /* ensure errors are reported */
        .
        /*
         * get access to facilities
         */
        status = ds_bind("a-directory", a_context, &dsap);
        file_descriptor = open("a-pipe", O_RDONLY+O_NONBLOCK);
        display = XOpenDisplay("a-display");
        .
        /* do whatever setup is required */
        .
        /*
         * and prepare to wait for an event
         * in this case using SysV poll()
         */
        pollfds[0].fd = ds_file_number(dsap);
        pollfds[0].events = POLLIN;
        pollfds[1].fd = file_descriptor;
        pollfds[1].events = POLLIN;
        pollfds[2].fd = ConnectionNumber(display);
        pollfds[2].events = POLLIN;

        while (1)       /* MAIN EVENT LOOP */
        {
                /*
                 * wait indefinitely for an event. When one occurs
                 * check each source in turn to see which.
                 */
                poll(pollfds, 3L, -1);

                if (pollfds[0].revents != 0)
                {
                        /* process directory events */
                        while (1)
                        {
                                DS_event event;

                                status = ds_event_wait( dsap,
                                                        DS_NO_WAIT,
                                                        &event);
                if (event == DS_NO_EVENT) break;
                .
                results  = ds_event_data(event);
                invokeID = ds_event_identity(event);
                .
                .
                ds_event_delete(event);
            }
        }

        if (pollfds[1].revents != 0)
        {
            /* process pipe input */
            .
            do
            {
                bytes_read = read(file_descriptor,
                                  buffer,
                                  NUM_BYTES);
                .
                .
            }
            while (bytes_read > 0);
            .
        }

        if (pollfds[2].revents != 0)
        {
            /* process display events */
            .
            while (XPending(display) > 0)
            {
                XEvent event;

                XNextEvent(display, &event);
                .
                .
            }
        }
    }
    .
    .
    /* close down and exit */
}
```

3.4 ERROR HANDLING

3.4.1 Introduction

Errors as referred to in this document should be regarded as something which is genuinely wrong, and not just a special case result. The definition of an Ada exception is appropriate:

> '*an exception is an event which causes suspension of normal program execution*'.

It is desirable to present a common method of error handling whilst not preventing APIs from exploiting individual opportunities to improve the facility. The API implementation is at liberty to combine all error data in the standard model when they are detected, or to convert them when the application program requires the data.

Note that some of the operations defined below are 'partial' in the sense that they are not defined for all values of their arguments. Such operations are also not required to return a *status*. This permits more efficient implementations of the operations. For example, consider the case of a C implementation where an error is represented by a pointer to an error structure, whilst a successful result is indicated by a NULL pointer. Then the restriction of *error_number* (see below) to arguments which actually represent an error means that it can be implemented as:

```
#define error_number(status)  (((status_type)(status))->number)
```

with no concern for the dereferencing of a NULL pointer, since that is an application programming mistake. Note that where specifications say '*application programs may not ... and the results of doing so are undefined*', the undefined behaviour may include immediate termination (by bus error, illegal address error, etc.).

3.4.2 Mechanism

The mechanism centres around a *status* object which is present as a result (output parameter) of all API operations, and which would typically be the function result in the C binding of an API. A *status* has several standard operations defined and each API may provide more as appropriate. It is not possible to access the value of a *status* object except by means of these operations; the representation of the object is private to the API and may differ from one vendor's implementation to another.

There is a special value which represents success and other values representing all possible errors.

The standard operations include a predicate *failed*, and three selectors *error_class, error_number, error_message*. *status* objects are constructed implicitly by the API. In addition the next section provides an explicit constructor *error_copy* and a destructor *error_delete*.

3.4.3 Status Lifetime

The lifetime of a status object needs consideration during the design of an API language binding and must be stated in its specification. The chief consideration is management of the storage used to hold the representation of the status. Possible options include:

- A fixed lifetime, typically from the point where the status is returned by one API call until the next call to the API (except for error handling operations, of course). This can be implemented simply by global storage but is not easily extensible to multithreading or multiple concurrent operations.
- An indefinite lifetime with unique storage allocated for each status works well for multithreading and multiple concurrent operations, but requires explicit storage release unless the programming language provides garbage disposal. If provided, the storage release operation should be called *error_delete*.

The following interface is proposed for the C language and other languages with similar memory management characteristics. Appropriate wording for multithreaded languages is included between brackets [thus] :

A *status* object returned by an API function is accessible until the next (non-error handling) call to the API [in that thread]. References after that point will produce undefined results. Where access to a *status* object is required after that point [or in a different thread] it can be obtained by making a copy using the *error_copy* function, but such copies must be explicitly disposed of by the *error_delete* function.

NAME

error_class - determine the class of an error

SYNOPSIS

```
int error_class(
        Status  error
);
```

DESCRIPTION

error_class( ) returns an integer which may be compared against a list of error classes defined by the API. The list must include three standard classes, common across all APIs, but may be extended. The standard classes are:

system_error
: An XSI defined error was returned by a system call internal to the API. The *error_number* below will be the same as returned by the macro *errno* defined in **<errno.h>**.

language_binding_error
: A language-dependent error occurred. For example, buffer overflow in languages which cannot dynamically allocate buffers.

API_error
: An error occurred within the API, such as invalid or inconsistent arguments to an interface function.

OSI networking APIs will all extend the list, typically by a single additional error class:

abstract_service_error
: An error was reported by the service to which the API provides an interface.

ARGUMENTS

error (status-type)
: Erroneous result of some operation.

RESULTS

class (integer)
: The class of error to which the given *error* belongs.

ERRORS

Applications are not permitted to call *error_class*( ) with an *error* argument which represents success, and the result of doing so is undefined.

No errors are reported by *error_class*( ).

NAME

error_copy - make a copy of an error object

SYNOPSIS

```
Status error_copy(
        Status  original,
        Status *        copy_return
);
```

DESCRIPTION

error_copy( ) takes a status object as an argument and returns an identical one with indefinite lifetime.

ARGUMENTS

original (status-type)

Result of some operation.

RESULTS

copy (status-type)

A copy of the original status.

The copy is identical to the original in the sense that the selector operations *error_class* and *error_number* produce the same results when applied to the copy, and *error_message* produces a string which is character-by-character the same.

status (status-type)

Normally success, and otherwise an error which occurred whilst copying.

If an error occurs, then both the *copy* and the *original* are undefined.

ERRORS

Possible errors returned in *status* are:

error_no_memory

There is not enough memory to complete the operation.

NOTE

error_copy( ) can be a null operation if the API implementation always provides unique *status* objects and does not reuse global storage.

NAME

error_delete - release resources associated with an error object

SYNOPSIS

```
Status error_delete(
        Status  error
);
```

DESCRIPTION

error_delete( ) releases storage and other resources associated with an error object. The application must *not* make any further reference to the original status after calling this operation.

ARGUMENTS

error (status-type)
The error object to which access is no longer required.

RESULTS

status (status-type)
Normally *success*, and otherwise an error.

ERRORS

Applications are not permitted to make any further reference to the original status after calling this operation, and the result of doing so is not defined.

Possible errors returned in *status* are:

error_bad_status
The specified *status* is not valid.

NAME

error_identity- identify the operation in which an error arose

SYNOPSIS

```
API_Dependent_Type error_identity(
        Status  error
);
```

DESCRIPTION

error_identity( ) identifies the preceding operation which was the source of the error. Typically, the error will contain the results of asynchronous activity which was initiated by the preceding operation.

ARGUMENTS

error (status-type)
Erroneous result of some operation.

RESULTS

identity (API-dependent-type)
The identity of the operation which caused the error.

Definition of the *identity* type depends on the particular API. It might be the *Invoke_ID* of a directory operation, for example. In any case it must serve as a unique identifier of the particular error instance.

ERRORS

Applications are not permitted to call *error_identity*( ) with an argument which represents success, and the result of doing so is undefined.

No errors are reported by *error_identity*( ).

NAME

error_fatal - determine program behaviour when errors occur

SYNOPSIS

```
void error_fatal(
        bool    terminate
);
```

DESCRIPTION

Any errors which arise after *error_fatal*( ) has been called with a *true* argument result in a message being sent to the standard error output followed by process termination. The message includes that which would be generated by *error_message*.

Any errors which arise after *error_fatal*( ) has been called with a *false* argument are reported in the *status* parameter of the operation.

The API initially behaves as though there has been a call to *error_fatal*( ) with a *true* argument.

ARGUMENTS

terminate (boolean)
Whether the application should be terminated by an error.

RESULTS

No results.

ERRORS

No errors are reported by *error_fatal*( ).

NAME

error_message - produce a message describing an error

SYNOPSIS

```
char * error_message(
        Status  error
);
```

DESCRIPTION

error_message( ) produces a message describing the error represented by the given *status*. The string is in the appropriate national language as discussed under Internationalisation in the **X/Open Portability Guide**, **Issue 3**, **Volume 3**, **XSI Supplementary Definitions**, and may contain details of the particular error as well as a general message (e.g., a particular name which was not found).

ARGUMENTS

error (status-type)

Erroneous result of some operation.

RESULTS

message (string)

A message describing the given error.

The message may contain as much additional information about the particular error as the API wishes to provide. The message returned for a *system_error* will include that returned by the XSI function *strerror*( ).

ERRORS

Applications are not permitted to call *error_message*( ) with an argument which represents success, and the result of doing so is undefined.

No errors are reported by *error_message*( ).

NAME

error_number - determine the number of a particular error

SYNOPSIS

```
Sint error_number(
        Status  error
);
```

DESCRIPTION

error_number( ) takes an error as argument and returns an integer which may be compared against a list of errors defined by the API.

ARGUMENTS

error (status-type)
Erroneous result of some operation.

RESULTS

number (integer)
The error number of the argument.

The error number returned for a *system_error* is the same as that returned in *errno* by the system.

ERRORS

Applications are not permitted to call *error_number*( ) with an argument which represents success, and the result of doing so is undefined.

No errors are reported by *error_number*( ).

NAME

failed - determine whether a status is *success* or an error

SYNOPSIS

```
bool failed(
        Status  status
);
```

DESCRIPTION

failed( ) returns *false* if the given status represents success and returns *true* otherwise. This operation is the only method of testing for success.

ARGUMENTS

status (status-type)

Result of some operation.

RESULTS

result (boolean)

Whether the given status represents an error (result = true) or not (result = false).

ERRORS

No errors are reported by *failed*( ).

3.4.4 Programming Examples

The examples are presented in the C programming language and use a prefix of **ZZ** on identifiers to denote that they belong to the hypothetical API. There are skeletons of two possible versions of the API header file, using different implementations of the status object.

First is a possible implementation for a very simple API where the status consists simply of an error number:

```
/* file: zzheader.h */

#define ZZ_SYSTEM_ERROR              1
#define ZZ_LANGUAGE_BINDING_ERROR    2
#define ZZ_API_ERROR                 3

typedef int ZZStatus;

#define ZZfailed(status)      ((status) != 0)

#define ZZerror_class(status)   \
        ((status) < SOME_NUMBER ? ZZ_SYSTEM_ERROR : \
        ((status) < A_BIGGER_NUMBER ? ZZ_LANGUAGE_BINDING_ERROR : \
        ZZ_API_ERROR))

#define ZZerror_number(status)  (status)

extern char *ZZerror_message(ZZstatus status);
```

Second is a more complicated, and commonplace implementation where the status consists of an opaque pointer to a structure defined privately by the API:

```
/* file: zzheader.h */

#define ZZ_SYSTEM_ERROR                1
#define ZZ_LANGUAGE_BINDING_ERROR      2
#define ZZ_API_ERROR                   3

/* private structure */
typedef struct
{
        int     class;
        int     number;
        .. other structure members as required ..
}
_ZZStatus;

/* public definition - opaque pointer */
typedef void * ZZStatus;

/* standard operations */
#define ZZfailed(status)        ((status) != (ZZStatus)NULL)

#define ZZerror_class(status)   (((_ZZStatus *)(status))->class)

#define ZZerror_number(status)  (((_ZZStatus *)(status))->number)

extern char *ZZerror_message(ZZstatus status);
```

The next fragment is of a simple application program which relies on the API to automatically detect and report any errors and terminate the process.

```
#include <zzheader.h>

main()
{
        ZZaccess_point zzap;

        ZZbind(..., &zzap);
        ZZsome_operation(zzap, ...);
        /* and so on with no visible error handling */
        ZZunbind(zzap);
}
```

Finally there is a fragment of a more sophisticated application which takes control of the error handling in order to provide tailored recovery at each API call:

```
#include <zzheader.h>

extern void send_to_error_log(char * message);
extern void take_recovery_action(int class, int number);

main()
{
        ZZStatus err;
        ZZaccess_point zzap;

        .
        .
        /* prevent simple-minded termination */
        ZZerror_fatal(FALSE);
        .
        .
        if (ZZfailed(err = ZZbind(..., &zzap)))
        {
                send_to_error_log(ZZerror_message(err));
                exit(1); /* don't know how to recover */
        }
        .
        .
        if (ZZfailed(err = ZZsome_operation(zzap, ...)))
        {
                send_to_error_log(ZZerror_message(err));
                take_recovery_action(ZZerror_class(err),
                                ZZerror_number(err));
        }
        .
        .

        ZZunbind(zzap);
}
```

3.5 OBJECT MANAGEMENT

APIs should use the Object Management facilities adopted in collaboration between X/Open and the X.400 APIA, where this is appropriate. These facilities provide a method of describing and structuring arbitrarily complex data in order to provide a functional interface with data-hiding, and are defined in the OSI Object Management API specification (see **Referenced Documents**).

3.6 INTERNATIONALISATION

APIs should be internationalised as described in the **X/Open Portability Guide**, **Issue 3**, **Volume 3**, **XSI Supplementary Definitions**. In particular, all natural language text should be separated from the program into message catalogues and manipulated using international function extensions (see *strcoll*( ), *printf*( ) in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**, for example).

It should be possible for knowledgeable users to generate translations into other languages (using the *gencat* utility). This means the original message text source file must be documented and/or present in a documented location, as well as the generated catalogues.

3.7 RESOURCE CONSTRAINTS

Functions must guarantee to return all available status information. Specifically, they should ensure that memory is reserved or allocated for any return aggregate parameters before starting an operation which may return status information in the aggregate; they should not wait until the operation completes. It is not acceptable to return a resource error at completion time.

For the reason above, status information should not be returned in the same dynamically allocated aggregate as variable sized data.

Implementations of APIs should specify upper bounds on resource usage.

3.8 LEVEL OF API

There are no guidelines concerning the level of the API, though it is possible to make some related recommendations.

Whether an API should be high or low level in some sense must be judged individually for each case, and will depend on the base documents.

It is wise to provide simple access to the basic features, with defaults for all options.

API specifications should not contain details of anything below the level of the API itself, since this will constrain implementations.

3.9 EXTENSIONS

It is likely that there will be extensions to many APIs. The guidelines for extensions follow from classifying the functionality:

mandatory Specified in the API specification and present in all implementations of the API.

optional Specified in the API specification but only present in some implementations of the API.

vendor-private Specified by a particular vendor and only present in their implementation of the API.

future Specified in a revision of the API specification with consequent multiple implementations of the API (during the transition period).

reserved Present as a place-holder in the API specification to allow forward compatibility.

In order to allow portable applications to exploit the API in the presence of these variations, each API must provide means to allow the application to discover what functionality is present.

The means to discover *optional* functionality is given in the rule quoted below from the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**. This rule should be followed, except that the error should be returned using the API's own error handling policy rather than *errno*.

"*The interfaces to optional functionality exist on all implementations; however, on implementations that do not support the functionality, the interface will merely return an error, with errno set to* **[ENOSYS]**."

Check also that the application will be able to discover the [lack of] functionality in such a way as to enable it to recover, and take alternative action.

Reserved functionality can be treated exactly as unsupported *optional*functionality.

Vendor-private and *future* functionality can be discovered by the use of a version number, and the recommended interface is presented below. The version number is a zero-terminated string, informally defined by the following syntax and examples:

version ::= *revision* ';' *vendor* ';' *release* ';' *private*

where *revision* is the issue number of the API specification *vendor* is an agreed string identifying the vendor, *release* is the release number of the implementation from this vendor and *private* is an arbitrary string. The content and purpose of *private* are defined by the vendor and typically used to indicate additional features; it can be empty.

Below are two examples of version numbers, both for an API assumed to have revision number 4. One is the first release of an implementation by IBM, while the other is release 2 of Hitachi's implementation and contains some private extensions defined in their documentation.

"4;IBM;1;"
"4;Hitachi;2;Kanji"

NAME

version - discover the API version number

SYNOPSIS

```
Status version(
        char ** version_return
);
```

DESCRIPTION

This function returns the version number of the API.

ARGUMENTS

version_number (string)

The API version in the format given above.

status (status-type)

Normally success, otherwise an error.

RESULTS

None.

ERRORS

error_no_memory

Dynamic memory allocation failed.

NOTE

The function should return a pointer to a string present in the API library (or copied from it) since this shows correct linking as well as compilation. This also allows APIs whose functionality is provided by a separate process to report correctly what facilities are actually available.

Chapter 4

Bindings to APIs

4.1 INTRODUCTION

This section provides guidance for the preparation of language bindings to X/Open Interworking APIs and is largely concerned with naming conventions. After a general discussion of names, specific rules are presented for each type of name which occurs in an API: functions, variables, types, constants and parameters. There is also additional material on the design of parameter lists.

4.2 NAMING CONVENTIONS

Naming conventions are used to overcome differences between programming languages, and to produce names which can easily be interpreted and remembered by programmers. The overall technique is to choose an abstract name for each entity in an API which needs a name. This is then bound, or converted, to a name in a particular programming language by means of a set of derivation rules. The rules for each language are designed to produce unique legal names in that language which are similar to the abstract name and to those derived in other languages.

In general the choice of rules for binding to a particular programming language will depend on the syntax and semantics of that language, but X/Open supports particular dialects of C and COBOL in the **X/Open Portability Guide**, **Issue 3** and this permits a single set of rules to be used for both languages.

Some languages may need to adopt more restrictive rules because of their syntax, whilst others may be able to be more liberal, for example, Ada scope rules allow prefix letters to be dispensed with.

The common guidelines for C and COBOL presented here are based on two derivation rules: the 16-character rule and the 30-character rule. These, and a 6-character rule, are explained after a discussion of internal, external and abstract names.

4.3 INTERNAL AND EXTERNAL NAMES

Names can be divided into two classes which require different naming conventions:

- *External names* are those visible at link time and the rules for their construction are determined to some extent by the system linker as well as by the programming language and compiler. The naming convention must be applied to external names which are private to the implementation of an application service as well as to those which form part of the API.

- *Internal names* are those visible only at compile time, and which are resolved and eliminated before linking. The naming convention is concerned with those internal names which appear in the API and need not be applied to private internal names within the implementation of an application service. They include the names of constants, types, local variables and procedures which are typically defined in headers provided by the API and which are included in programs using it.

  Note that C and COBOL parameter names are private to the implementation, and so do not form part of the external interface and are not subject to these restrictions.

4.4 ABSTRACT NAMES

Abstract names are used in order to keep the names of functions and other entities similar and understandable in all supported languages. A set of rules are defined which, when applied to these standard name phrases, yields the identifier to be used in the language-specific version of the library. Thus, in all programming language-specific interface libraries, the name of any given function will be similar because of its common origin in the standard function name phrase assigned to that function.

The following rules should be followed when forming abstract name phrases:

- The phrase should meaningfully describe what the function actually does. Every effort should be made to avoid using terms which will have different meanings to different users. Avoid jargon.
- No more than three words should be used (except ‘noise’ words).
- Check existing APIs and use identical names for corresponding entities.
- Where the API is being derived from an existing, abstract specification (e.g., an International Standard) retain the names used there.
- As new functions or other names are added to a service, each identifier should be generated for all existing language bindings.
- A check should then be made that the generated identifiers are not identical to any others. If a collision does occur, a different abstract name phrase should be selected which does yield a unique identifier in all languages.

4.5 GENERAL NAMING RULES

Identifiers are formed using a prefix denoting the application service name followed by the abstract name. The prefix ensures that the names in one API are different to those in all other APIs, and the prefix used for each application service should be recorded to avoid possible conflicts. Prefixes should be two, three or four characters long, but when the six-character convention is used they must be exactly two characters long.

It is sometimes necessary to abbreviate words in order to ensure unique identifiers, and the abbreviations are standardised so as to increase familiarity for the programmer. Three-character abbreviations are used for all words.

Initial lists of prefixes and abbreviations are given in **Appendix A**, **Name Prefixes and Abbreviations**.

The abstract name is modified as prescribed by the particular naming convention and then added to the application service prefix to produce an identifier. These identifiers must be unique across the whole of the Common Applications Environment, and each must be checked for conflict with:

- existing functions and variables whose names do not follow a prefix scheme, such as most system calls and library functions;
- other functions and external variables in the new library, and
- reserved words in the programming language.

Any conflicts should be resolved by changing the abstract name. The **X/Open Portability Guide** should be used to check for conflicts with system facilities, and for C keywords and library names. The **X/Open Portability Guide**, **Issue 3** should be used together with the ANS X3.23-1985 standard for COBOL reserved words.

4.6 SIX-CHARACTER NAMING CONVENTION

The six-character convention is designed for use with names in environments with very small limits on identifier length. In this convention, the prefix denoting the application service name must consist of two characters (*t_* is used as an example). The abstract name is added to the prefix in a manner which depends on the number of words in the abstract name:

1. use *prefix word* where *word* is the entire abstract name. For example, *listen* becomes *t_listen*.

2. use *prefix abbreviation word* where *abbreviation* is the standard 3-character abbreviation of the first word in the abstract name and *word* is the entire second word. For example, *receive-connection* becomes *t_rcvconnection*.

3. use *prefix letter letter word* where *letter* is the first letter of the first and second words in the abstract name and *word* is the entire third word. For example, *receive-unit-data* becomes *t_rudata.*

4.7 SIXTEEN-CHARACTER NAMING CONVENTION

In cases where more characters are significant in identifiers, the need for abbreviation is not as great and so the following conventions can be used:

- The abstract name should be bound to a programming language identifier by prefacing it with the application service prefix.

- The words in the name should be differentiated by a suitable technique for the particular programming language. In COBOL, the hyphens in the abstract name should be retained. In C, the hyphens may be converted to underscores or they may be deleted and the first letter of the following word capitalised. The same technique should be applied consistently throughout the API.

- Identifiers must be unique with no regard to case-sensitivity and must be unique within sixteen characters.

- An identifier which is not unique may be disambiguated either by changing the abstract name or by abbreviating one or more words using the standard abbreviations. All occurrences of the word in all functions should be abbreviated in the interests of consistency.

4.8 THIRTY-CHARACTER NAMING CONVENTION

The thirty-character naming convention is identical to the sixteen-character one except that identifiers need only be unique in thirty characters rather than sixteen. Abbreviation of the abstract name will be very rare using this convention.

4.9 FUNCTION NAMES

The sixteen-character naming convention should be followed by all functions included in an application service interface, and the prefix should additionally be applied to any functions which are visible but not in the interface (e.g., private functions called across compilation units).

Some functions in the X/Open Transport Interface (XTI) are presented as examples of how the naming scheme works.

Abstract name	Derived name	Description
listen	t_listen	listen for a connect request
open	t_open	establish a transport endpoint
receive-connection	t_receive_connection	receive the confirmation from a connection request
receive-data	t_receive_data	receive data or expedited data sent over a connection
receive-unit-data	t_receive_unit_data	receive a data unit
receive-unit-error	t_receive_unit_error	receive a unit data error indication

4.10 EXTERNAL VARIABLE NAMES

The sixteen-character naming convention should be followed for all external variables (globals) included in an application service interface, and the prefix should additionally be applied to any variables which are visible but not in the interface (e.g., private variables passed between compilation units).

For example, an abstract *error_number* implemented as a global variable would be bound to the identifier *t_error_number* if the relevant prefix was *t_*.

4.11 INTERNAL NAMES

Constants, data types and any other internal names which appear in an API also have a naming convention based on abstract names. However, the need for abbreviation is not as great as for external names, and so the thirty-character convention should be followed.

4.12 ERROR NAMES

Error names should follow the convention for internal names, with the added constraint that the first word of the abstract name should be *error*. This should be abbreviated to upper-case **E** rather than lower-case **e** when a single letter abbreviation is required.

The scope of error names is deemed to be application service-wide and this is reflected in the specification layout described in **Chapter 5**, **Layout of Specifications**. Error names and their standard descriptions should be chosen carefully to match all of the functions which can raise them. Each error should have a single meaning, and the meanings of different errors should be clearly differentiated.

Error descriptions and abstract names should be the same as those in other application services where possible (programming language identifiers will be distinguished by the prefix).

Language-specific errors should be clearly separate from the common errors.

4.13 PARAMETER NAMES

There is no particular convention in parameter naming. However, within one application service, parameters with the same semantic and syntactic definitions should have exactly the same abstract name. Some of the very common parameter names will be the same across all application services.

Parameter names should be derived from the abstract name by the same mechanism as described above for internal names, without adding the prefix.

A further convention should be used for output parameters in C; this is presented in **Section 6.1**, **General Rules**.

4.14 PARAMETER PLACEMENT

Parameters should be bound rigidly in the same order they appear in the functional specification. Input parameters should precede output parameters. Additional parameters required by a language binding which are not part of the functional specification should also follow this rule (e.g., array or string lengths).

Where the same parameters are shared by several functions, they should occur in the same order each time. For example, a file descriptor should generally appear as the first parameter, and an *x* (coordinate) should appear before a *y*.

4.15 AGGREGATE PARAMETERS

In many programming languages it is possible to collect several variables into one aggregate (e.g., a C structure). This may be done either to express a relationship between the variables by collecting them into a single conceptual object, or because of implementation considerations in the programming languages such as a restriction on the number of parameters which can be passed. In the former case it is appropriate to collect the variables into a language-independent *abstract aggregate* type.

Choices must be made as to which input and output parameters are directly exposed to the user and which are aggregated. When the relationship between variables is not strong the trade-offs to be considered are:

Exposed	Aggregated
obvious to user	not obvious to user
easy to read / set	harder to read / set
lengthen parameter list	shorten parameter list
complicate parameter list	simplify parameter list

Parameters that are required, but not frequently modified by the user, can usefully be aggregated into a composite state parameter.

Parameters which are required but have no meaningful default value should be exposed.

Parameters which would usually be set by the user are better left exposed.

Parameters which would usually not be set by the user are better aggregated.

The total exposed parameter list should be kept short. Less than eight parameters excluding parameters common to all functions is appropriate.

References by the function to external visible data items (global variables) must be included in the abstract parameter list in the specification. They should generally be avoided and replaced by parameters in the programming language, perhaps aggregated with other parameters.

4.16 PARAMETER USAGE

Various details of the use which an application service may make of parameters are given here in the interests of simple and consistent interfaces. Some details depend on whether a function is *synchronous* or *asynchronous* in the following sense:

- A synchronous function is called, performs a job to completion, and then returns all its results; there is no further activity visible to the user. Functions should be synchronous wherever possible.
- An asynchronous function is called, performs some initial work and makes an *immediate return*, performs the rest of the job to completion, and makes a *final return*. Details of the return mechanisms are dependent on the particular application service and programming language.

Note that functions may be able to behave in a synchronous or asynchronous way dependent on a certain parameter (such parameters must appear in the function specification even if implemented as a global mode or some such), or separate functions may be provided instead.

Parameters identified as input only shall not be changed by the implementation of the application service, regardless of the parameter passing mechanism used in the programming language. This also applies to components such as array elements and indirections.

The calling program is free to change or destroy variables passed as input parameters to an interface function as soon as the function returns. That is, the implementation of an API must not reference any parameters after the interface function returns. A copy must be made of any data which must be referenced later.

An exception to the above paragraph is allowed for large parameters, such as buffers of data, when (and only when) the function is asynchronous. Such parameters shall be clearly identified in the parameter specification of the function, and all reference shall be complete by the time of the final return.

All functions will have a return-code as an output parameter which will return the constant *success* or one of the defined error codes.

Ideally, no changes to the contents of any output parameters except *return-code* shall be made unless *success* is returned, but in any case it must be possible to discover exactly which parameters have been changed by examining the function return-code and other parameters.

4.17 GUIDELINES FOR OTHER LANGUAGES

Name derivation rules for other languages should use these rules as models, making any desirable changes. They should also take into account any guidelines laid down by standards organisations for the particular language. If and when developed, the rules, together with any other advice and conventions, will be collected into an additional section at the end of this Style Guide.

Chapter 5

Layout of Specifications

5.1 BASIC PRINCIPLES

Guidelines for the contents and layout of specification documents for APIs are presented in this Chapter. Considerable flexibility is provided to ease the process of adopting APIs from different sources and covering different service areas.

No single overall structure is suggested for specifications, but details are given of a number of sections which should appear in every specification. The remainder of this Chapter discusses each of these specification sections in turn:

- introduction
- types
- constants
- error handling
- parameters
- C language names
- object specifications
- function specifications

The layout of the specification should follow ISO Guideline 48 as modified below. The Guideline should be applied as indicated to any additional language bindings and to the combined abstract functional specification and C language binding (which fulfils the purpose of the ISO functional interface standard). In the following text, brackets [like this] indicate a modification to the ISO text.

48 *For readability and ease of maintenance, a single set of contents should be used for every binding developed to a functional interface standard. Guidelines concerning the format are:*

- *[Each function description in the functional interface standard should start on a new page.] It is not necessary that there be one function description per page [for additional language bindings.]*
- *Descriptive information should not be copied from the functional interface standard [for additional language bindings.]*
- *The functional description schema should have titles for the sections relating to arguments, errors and references.*
- *[Additional language bindings] should not contain explicit references back to the functional interface standard document.*
- *A short description of the arguments, mapping back to the parameters from the functional interface standard, should be given within the schema.*

- *It is not necessary for there to be a page break, with corresponding page headers, for every major section in the functional interface standard.*
- *The structure definitions should not be split between two pages.*
- *Descriptions should not be broken across two pages.*
- *Error Messages, for binding errors only, should be a part of the function description schema.*
- *Along with the table of abbreviations, there should also be a table containing the function names alphabetically, by level.*

5.2 INTRODUCTION

The introduction to the specification, containing a statement of the objective of the API, a general overview, and a list of reference documents.

5.3 TYPES

Definitions and descriptions of all types used in the interface to the application service including any aggregates and enumerations, except those defined in object class definitions or function definitions. A complete list of *all* C language identifiers must appear, usually in the definition of headers. It may be appropriate to define types in the header definition where the usage and semantics are both simple and clear.

5.4 CONSTANTS

A list of all named constants which appear in the interface, together with descriptions, except those defined in object class definitions or function definitions. A complete list of *all* C language identifiers must appear, usually in the definition of headers. It may be appropriate to define types in the header definition where the usage and semantics are both simple and clear.

5.5 ERROR HANDLING

A description of the mechanism used to report errors, and a list of all the error codes which can be generated by this package, together with a standard description of their cause.

5.6 PARAMETERS

A list of function parameters which occur frequently in the interface which allows the reader to become familiar with common parameters. It also reduces the size of individual function specifications while encouraging good description of the parameters. The list should be in the order in which parameters occur in function parameter lists, not in alphabetic order.

5.7 INDEX OF C LANGUAGE NAMES

In order that application programmers can write portable programs, and so that support staff can determine the source of name conflicts, it is necessary for each API to fully specify the names which implementations will use. An ordered list is needed of all the C function names, global variables, macro names, typedef and enum names, structure tags and any other symbols. This requirement can be met by the Headers chapter in **X/Open Portability Guide** practice.

The list should reserve some additional names for use by particular implementations of the application service. It is recommended that two particular subspaces are reserved:

- All names starting with the API prefix followed by the letter **P** for private (i.e., internal) use by implementations of the interface. Some compilation and linking environments may make it necessary to expose names which would ideally not be visible to the user, so this namespace provides a means of doing so which will not cause conflicts.
- All names starting with the API prefix followed by the letter **X** for vendor-specific extensions of the interface.

Note that this does not prohibit an API from using public abstract names which start with the letters P or X (e.g., an interface function named *Print*( ) which might bind to *t_Print()* in C). It simply warns application programmers who use the API that they should not use any identifiers (e.g., they may not define the identifier *t_Poll*).

5.8 OBJECT SPECIFICATIONS

API designs which make use of Object Management will include object class definitions (or just class definitions) in their specifications. These class definitions describe the major data entities which are supported and manipulated by the API. Object Management is described and specified in the OSI Object Management API (see **Referenced Documents**), and that material is not repeated here. This section is concerned with the layout of class definitions.

All names used in the class definition are abstract names, except when the C binding is discussed below. The scope of all names is package-wide, and they must be unique within the package.

5.8.1 Class Definitions

Each class is defined in a separate section of the API specification containing the following:

- The section header is the name of the class.
- The first paragraph is a very brief description of the purpose of the class.
- The second paragraph lists the class hierarchy from object down to the class. Abstract classes in the hierarchy are listed in italics, while concrete classes are listed in bold.
- The next item is a table of the class-specific attributes, with the following columns:

  Attribute The name of the attribute, which is spelled identically to the attribute-type.

  Value Syntax The kind of value(s) which may be assumed by the attribute. Possible value syntaxes are defined in the Object Management specification.

  Value Length For value syntaxes which are of variable length (e.g., strings), this column specifies the range of permitted lengths. For other value syntaxes it is blank.

Value Number This column specifies the range of number of values which an attribute may possess. An entry of 1 specifies a mandatory, single-valued attribute; an entry of 0-1 specifies an optional, single-valued attribute; an entry of 0-*n* specifies a multi-valued attribute with up to *n* values; and an entry of 0 or more specifies a multi-valued attribute with an unlimited number of values. This column is never blank.

Ordered For multi-valued attributes, this column indicates whether the values have a fixed ordering (so that one value is always first, and so on). The column entry is either Yes or No; and for single-valued attributes it is blank.

Value Initially The value of the attribute when an instance of the class is created. This is a value of the given syntax, or blank if the attribute has no default value.

- Following the table of attributes is a list of the attributes, describing the purpose of each.

5.8.2 C Binding of Classes

The C binding is derived as set out below, and documented in a table at the end of the section which defines the class. All C identifiers are prefixed with the package prefix characters, and all are **#define** object-like macro names except for the enumerations. They are all entirely in upper-case letters, except for any enumeration tags.

class name The package prefix is followed by *C_* and then the abstract name of the class. The numeric value is not part of the API specification.

attribute type The package prefix is followed by the abstract name of the attribute. The numeric value is not part of the API specification.

value syntax The appropriate name as defined by the Object Management specification.

value lengths The package prefix is followed by *VL_* and then the abstract name of the attribute. A C binding only exists for maximum value lengths greater than one, and the numeric value is as specified in the attribute table. There is no C binding to minimum value lengths.

value numbers The package prefix is followed by *VN_* and then the abstract name of the attribute. A C binding only exists for maximum value numbers greater than one, and the numeric value is as specified in the attribute table.

enumerations Where the value syntax is an enumeration (e.g., *Enum*(*Priority*)), the C binding is also an enumeration.

The enumeration tag identifier is composed of the package prefix followed by the abstract name of the value set (i.e., the name in parentheses in the value syntax). This name is spelled in the same case as the abstract name.

Each enumeration constant identifier is composed of the package prefix followed by the abstract name of the constant, spelled entirely in upper-case. The numeric value is not part of the API specification.

5.8.3 Example Class Specification - Message

An instance of class *Message* is a primary information object conveyed between users by the MTS. It conveys arbitrary binary data from one user, the originator, to one or more users, the recipients.

The class hierarchy is *Object*, *Communique*, *Message*. An instance of this class has the attributes of an instance of its immediate superclass, and additionally the attributes listed below.

Attribute	Value Syntax	Value Length	Value Number	Ordered	Value Initially
Content	Object(Content)	-	1	-	-
Content-Return-Requested	Boolean	-	1	-	false
Deferred-Delivery-Time	String(Time)	0-17	0-1	-	-
Disclosure-Allowed	Boolean	-	1	-	false
Priority	Enum(Priority)	-	1	-	normal

Attributes of a Message

1. **Content.** The arbitrary binary information the message is intended to convey to its recipients. The MTS modifies the value of this attribute only for purposes of conversion.
2. **Content-Return-Requested.** Whether the Content attribute is to be included, as the Message-Content attribute, in any reports of non-delivery the message provokes.
3. **Deferred-Delivery-Time.** The date and time, if any, before which the message shall not be delivered. Delivery deferral is normally the responsibility of the MD that originates the message. Thus messages whose Deferred-Delivery-Time attributes are present shall be transferred between MDs only by bilateral agreement between those MDs.
4. **Disclosure-Allowed.** Whether the O/R names of other recipients are to be indicated to each recipient at delivery.
5. **Priority.** The relative priority at which the message is to be transferred. Its value may be *low, normal* or *urgent*.

Attribute	Value Syntax	Value Length	Value Number
MT_CONTENT	OM_S_OBJECT		
MT_CONTENT_RETURN_REQUESTED	OM_S_BOOLEAN		
MT_DEFERRED_DELIVERY_TIME	OM_S_UTC_TIME		
MT_DISCLOSURE_ALLOWED	OM_S_BOOLEAN		
MT_PRIORITY	OM_S_ENUMERATION		

C Binding for Class *MT_C_MESSAGE*

Tag	Constants		
MT_Priority	*MT_LOW*	*MT_NORMAL*	*MT_URGENT*

Enumerations

5.9 FUNCTION SPECIFICATIONS

Each function in the API is described in an individual specification. These are ordered by abstract name and this name should appear in the page header.

An index of the C language names is also provided as described above. Each function is described using the section headings described below, and illustrated in the following example from the X/Open Transport Interface (XTI).

NAME

The abstract name of the function, which also appears in the header at the top of each page. This is followed by a very brief statement of the function's purpose.

SYNOPSIS

The C binding to the function expressed as a C code fragment including:

- any headers which are necessary for its use, and
- the ANSI C prototype of the function (with names derived by the method described in **Section 4.4**, **Abstract Names**). The list of parameters should contain parameter names as well as types, and must be in the same order as the abstract arguments and results below.

DESCRIPTION

The detailed specification of the function's behaviour. Note that much of this behaviour is better expressed in the description of the arguments and results.

ARGUMENTS

A list of the abstract input parameters of the function, which includes not only those in the C parameter list but also any external state the function reads (e.g., environment variables), and individual components of aggregated parameters.

Each parameter is characterised by:

name The abstract name of the parameter.

type The abstract type as defined earlier (see **Section 5.2**, **Types**).

description A description of the parameter, and its relationship to the function's behaviour.

RESULTS

A list of the abstract output parameters of the function, which includes not only the function return value, but also those in the C parameter list, any external state changes (e.g., *errno*), and individual components of aggregated parameters. The description of each result should state under what conditions it will or will not be valid, unless it will always be valid when the status is success, and invalid otherwise. The format of the entries is as for the input parameters. Any parameters which are input-output should appear in both lists.

The function status (which is bound to the C function result) should be the last result in the list. This causes the list of arguments and results to match the C binding precisely, and allows easy reference from the status result to the errors section below.

ERRORS

The abstract and C names of any errors which can be generated by the function, together with the standard description of the error. A more precise explanation of the circumstances which give rise to the error may be given in additional commentary, although they are normally explained in the DESCRIPTION, ARGUMENTS or RESULTS above.

SEE ALSO

The abstract names of any related functions, and the title and number of any other appropriate sections of the document. Use the actual C binding name for functions which do not (yet) have an abstract name, such as system functions like *fcntl*( ).

5.9.1 Example Function Specification - Receive-Data

This section comprises an example function specification using *Receive_Data*( ).

NAME

Receive-Data - receive data or expedited data sent over a connection

SYNOPSIS

```
#include <xti.h>

t_status t_Receive_Data(
        int     file_descriptor,
        int     buffer_length,
        char *  buffer_return,
        int *   transfer_length_return,
        bool *  more_data_return,
        bool *  expedited_data_return
);
```

DESCRIPTION

This function receives either normal or expedited data.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for *event_data_available* or *event_expedited_data_available* using the *get_event*( ) function. Additionally, the process can arrange to be notified via the Event Manager interface.

ARGUMENTS

file_descriptor (file-descriptor-type)

The local transport endpoint through which the data will arrive.

buffer_length (integer)

The size, in bytes, of the buffer available for data.

non_block_mode (boolean)

By default, *Receive_Data*( ) operates in synchronous mode and will wait for data to arrive if none is currently available. However, if the global variable *non_block_mode* is set (via *open*( ) or *fcntl*( )), it will execute in asynchronous mode and will fail if no data is available. (See *error_no_data* below).

RESULTS

buffer (byte-array)

A receive buffer of *buffer_length* bytes where user data will be placed.

transfer_length (integer)

The number of bytes of data received.

more_data (boolean)

If *more_data* has the value *true*, this indicates that the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple calls to *Receive_Data*( ). Each call with *more_data* set indicates that a further call must follow immediately to get more data for the current TSDU. The end of the TSDU is identified by the return of a call with *more_data* taking the value *false*. If the transport provider does not support the concept of a TSDU as indicated in the *information* returned from *open*( ) or *get_information*( ), then *more_data* should be ignored.

expedited_data (boolean)

The data returned is expedited data if this result is *true*. If the number of bytes of expedited data exceeds *buffer_length*, both *expedited_data* and *more_data* will be *true* on return. On subsequent calls to retrieve the remaining ETSDU *expedited_data* will also take the value *true* on return. The return of a call with *more_data* taking the value *false* identifies the end of the ETSDU.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (*more_data* is *false*), will the remainder of the TSDU be available to the user.

status (boolean)

true if the function is successful and *false* if an error occurs.

error_number (integer-type)

Indicates which particular error has occurred. It is a global variable (*errno* - strictly in C it is a modifiable lvalue) which is set only when the status indicates an error has occurred. It is not changed if the function was successful.

ERRORS

On failure, *status* [the function result] is set to *false* and *error_number* [the global *t_errnumber*] is set to one of:

Error [C Name]	**Description**
error_bad_file [TBADF]	The specified *file_descriptor* does not refer to a transport endpoint.
error_no_data [TNODATA]	*non_block_mode* was set, but no data is currently available from the transport provider.
error_asynchronous_event [TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
error_not_supported [TNOTSUPPORT]	This function is not supported by the underlying transport provider.
error_wrong_state [TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by the *file_descriptor*.
error_system_error [TSYSERR]	A system error has occurred during execution of this function.

SEE ALSO

fcntl( ), *get_information*( ), *get_event*( ), *open*( ), *send_data*( ).

Chapter 6

Rules for C Bindings

6.1 INTRODUCTION

This Chapter presents specific advice on the preparation of C bindings to X/Open Interworking APIs. Some of the advice is repeated from **Chapter 5**, **Layout of Specifications** in order to provide a complete checklist in this section.

6.2 GENERAL RULES

The following rules should be followed by all C bindings:

1. ANSI C requires that wherever a library function is implemented by a macro with parameters (a function macro), a genuine function with an identical name is also provided. This can be accessed by using **#undef** on the macro or by enclosing the name in parentheses (e.g., *library_func*(*parameters*)) and permits users to perform operations not possible with macros, specifically to take the address of the function.

2. All functions in an API should return the *return-code* as their result. A value of zero indicates success (i.e., the abstract constant *success* is bound to **0**) and other values match one of the errors which the function can raise.

3. If a function result must be used for something other than the *return-code*, the return value shall at least encode success/fail. If it returns a pointer, **NULL** indicates failure. If it returns an integer, at least one value shall be reserved to indicate failure and that value should be *((int)-1)*. If it would otherwise return nothing, it should have the *return-code* as its result.

4. A further naming convention should be used for output parameters because all parameters are passed by value in C. If the derived parameter name is *parameter_name* and the returned value is of type *parameter_type* then the C parameter should have *_return* appended and be declared as:

   ```
   parameter_type * parameter_name_return;
   ```

5. Abstract types should be bound in the API binding to underlying intermediate types having an explicit size, and each implementation should then define the intermediate types in terms of *short*, *int*, *long* and so on. The intermediate types should be the same from API to API:

   Sint32 Signed 32-bit integer. Used as the representation of all signed integral types requiring more than 16-bit range.

   Sint Signed integer guaranteed to provide at least 16 bits. Used as the representation of general signed integral types requiring not more than 16-bit range.

   Sint16 Signed 16-bit integer. Used only as the representation of signed integral types requiring exactly 16-bit range. Typically used in large arrays to conserve space, or for transfer to machines with different architecture.

Uint32 Unsigned 32-bit integer.

Uint Unsigned integer guaranteed to provide at least 16 bits.

Uint16 Unsigned 16-bit integer.

Intermediate	16-bit Definition	32-bit Definition
Sint32	long	int
Sint	int	int
Sint16	int	short
Uint32	unsigned long	unsigned
Uint	unsigned	unsigned
Uint16	unsigned	unsigned long

Definition of Intermediate Types on Different Architectures

This technique allows application programs using the API to be portable across different word lengths without requiring explicit casts of every parameter. An API is able to specify the numeric ranges it requires without causing inefficiencies. For example, and ignoring naming conventions for clarity:

```
typedef Uint32  File_length;
typedef Uint    Packet_length;
typedef Uint16  Historical_packet_length;
typedef Historical_packet_length Transmission_statistics[30000];

/*
 * Files can be very long, so need 32 bits.
 * Packets are quite small, and 16 bits is sufficient but 32 bits
 * is also fine (whichever is more efficient on each machine).
 * A large collection of packet lengths means we must be
 * explicit to avoid wasting space and time.
 */
```

6.3 IMPLEMENTATION CONSIDERATIONS

Where the C binding is the fundamental interface to the implementation (e.g., because the implementation is written in C) it may be used as the basis to implement other bindings. The following additional rules should be followed in order to ease this task:

1. Large blocks of data should not be passed as zero-terminated strings, but should have a separate count of the length. This avoids the necessity for a block copy in other languages in order to obtain the extra byte for the terminating zero. An additional function in the C binding can easily permit zero-terminated strings, as perhaps:

   ```
   result_type  user_friendly_function(char * string)
   {
   return basic_function(string, strlen(string));
   }
   ```

2. Any functions which perform dynamic memory management which is visible to the user must provide a means to inhibit this. An example would be a *read* function

which is used *malloc*( ) to allocate a buffer for the return data. The function may provide explicit means of control (e.g., a Boolean parameter *use_dynamic_memory*), or implicit means (e.g., passing in a non-NULL output buffer), or a completely separate function. This eases the preparation of bindings to languages not possessing equivalent dynamic memory mechanisms.

Appendix A

Name Prefixes and Abbreviations

This Appendix is provided as a starting point which should be maintained and expanded as necessary.

A.1 APPLICATION SERVICE PREFIXES

The prefixes for various system facilities are shown below together with the **X/Open Portability Guide**, **Issue 3** volume number if appropriate. In most cases it can be assumed that the same prefix in other letter-cases is also used (e.g., SQL is also used); refer to the appropriate manual for details. The importance of this kind of record is illustrated by the already existing clash for the 'is' prefix.

Prefix	**XPG**	**Service Name and Comments**
E	4	ANSI C standard error macros
ft		FTAM (proposed by MAP)
is	4	ANSI C standard functions
is	5	X/Open ISAM (also uses some other names)
LC_	4	ANSI C standard macros
mem	4	ANSI C standard functions
rt_		POSIX realtime extensions (proposed)
SIG_	4	ANSI C standard macros
sql	5	X/Open SQL
str	4	ANSI C standard functions
to	4	ANSI C standard functions
t_	7	X/Open Transport Interface
wcs	4	ANSI C standard functions
X	6	X Window System

A.2 ABBREVIATIONS

The words in abstract function names should be abbreviated as shown below when forming the equivalent name in the programming language.

Word	Abbreviation
abort	abt
acknowledge	ack
block	blk
confirm	cnf
connection	con
directory	dir
error	err
event	evt
get	get
indication	ind
in	in_
initiate	ini
input	in_
message	msg
modify	mod
option	opt
out	out
output	out
receive	rcv
remove	rem
request	req
response	rsp
select	sel
send	snd

Appendix B

Rationale

B.1 INTRODUCTION

The rationale for the API Style Guide presents additional information supporting some of the decisions made about the content. It uses the same section numbering as the main document and should be read in conjunction with the corresponding text.

B.1.1 Overview

Provision of guidance for functional specification depends on the architecture to be chosen, and particularly on the choice of programming model.

B.1.2 Language-Independent Specification

Language-independent techniques are becoming more common, and are used in the MAP specification, for example. The ISO Guidelines were developed partly as a result of the effort to produce the language-independent GKS (Graphical Kernel System). Work is also underway towards converting POSIX to a language-independent specification. However, such efforts still stand as disparate individual efforts.

It is premature for X/Open to take this route at this time, partly because of the several current efforts. It will become an appropriate route when a standard emerges and receives widespread support, and particularly if the POSIX effort succeeds.

In the meantime it is better to avoid the additional effort of rewriting specifications to be language-independent, knowing that the format is likely to be revised for compatibility in the future. For X/Open, using C-specific specifications also has the advantage of retaining a similar style to the rest of the **X/Open Portability Guide**.

X/Open specifications are presently expressed almost entirely and exclusively in the C language. COBOL is the only other language supported by the **X/Open Portability Guide**, **Issue 3**, though FORTRAN and Pascal were mentioned in **Issue 2**. X/Open COBOL programs are able to call C functions, so an interface specification expressed in C suffices for the Common Applications Environment.

There are some steps which can be taken to ease the production of bindings to other languages and the later transition to language-independence, particularly by the identification of all inputs to and outputs from functions.

It is not suggested that any existing language-independent standards are rewritten when adopted or adapted by X/Open. Indeed it is necessary that this Style Guide does offer guidelines rather than rules, because X/Open adopts standards rather than setting them.

B.1.3 Contents

It will be appropriate to separate all the programming language-specific aspects of **Chapter 5**, **Layout of Specifications** into a separate section when X/Open moves to language-independent specifications.

B.1.4 Terms and Abbreviations

B.2 LANGUAGE BINDINGS

B.2.1 Introduction

The ISO SC22/WG11/N466 document used in this report is the Draft Technical Report (DTR) dated February 1990.

The original text of modified or deleted ISO Guidelines is presented in this rationale, along with reasons for changes and points of interest.

B.2.2 Organisational Guidelines

2 Original ISO Guideline: *"Either the language committee or the system facility committee should have primary responsibility for the language binding. Different language bindings to a system facility should not be the cause of substantial differences in program structure."*

Changed to reflect the X/Open Interworking organisation, and avoid repetition of Guideline 6.

3 Original ISO Guideline: *"Whichever committee is responsible for a particular binding, the other committee needs to be consulted as early as possible. The two committees have complementary responsibilities and concerns."*

Changed to reflect the rewording of Guideline 2.

B.2.3 General Technical Guidelines

B.2.4 Recommendations for Functional Specifications

9 Original ISO Guideline: *"A functional specification should use an abstract description, and should avoid being influenced by a particular programming language."*

This is one of the greatest changes from the ISO Guidelines. The **X/Open Portability Guide** is expressed very largely in the C language at present, and it is premature to express one part using an abstract methodology. Refer back to **Section 1.2**, **Language-Independent Specification** for more details.

12 This Guideline has been introduced since the ISO draft used by MAP.

B.2.5 Procedural Interface Guidelines

16 It is intended that the binding should reserve part of the identifier namespace(s) for private use by the application service implementation, perhaps because internal procedures are unavoidably exposed through the linkage mechanism. This is made clear in the notes accompanying the ISO Guideline.

B.2.6 Suggested Actions For Standards Committees

B.2.7 Recommendations For Programming Language Committees

B.2.8 Procedural Language Binding Generic Issues

28 The MAP specification deletes the second sentence, but X/Open may wish to adopt a specification which includes a generic binding (cf SQL).

29 Original ISO Guideline: *"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."*

30 Original ISO Guideline: *"If a generic binding is required, only one should be developed, as one standard is generally better than two for a single purpose."*

44 Original ISO Guideline: *"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."*

The **X/Open Portability Guide** provides for the C and COBOL languages, and currently requires that external (linkable) names be unique in the first 6 characters and also not rely on case differences. This may change in the future, at which time this Guideline should be reviewed. Note that some interfaces specified in the **X/Open Portability Guide** already violate this restriction which suggests that all X/Open members' systems actually exceed the requirements of the **X/Open Portability Guide**. Examples of such interfaces are XTI which requires 8-character significance and X Windows which requires 16-character.

B.3 API CONTENT

B.3.1 Introduction

The topic of asynchronous results has been removed from the present version, but should be replaced when a suitable mechanism is developed. It will be appropriate to consider the Directory Services mechanism as a model when it is finalised.

B.3.2 Functional Interface

B.3.3 Event Management

Introduction

The need to manage data associated with an event is one of the reasons that X/Open needs its own mechanism for interworking. Another concern is the timescale, since X/Open wants to adopt several interworking APIs in the near future. During this time, the system interface to event notification is likely to be both vendor-specific and changing. The intent is to provide a single mechanism which will utilise any of the low-level facilities. This allows the design of APIs which will themselves need minimal change over time, and which allow application programs to be written which will need no change except at the few places where system facilities must be directly used.

The proposal assumes that the 'holy grail' of unified event management will not be found in the timescales considered.

Background

System Facilities for Event Notification

Overview of Mechanism

Access Points

Events

Programming Examples

B.3.4 Error Handling

Introduction

The design criteria used are presented to provide motivation for the mechanism in terms of initial requirements and consequent design decisions. These are:

1. the mechanism must not penalise writers of simple applications;

2. the mechanism must allow implementation freedom to vendors;
3. the mechanism must fit many API models;
4. the mechanism must allow errors to be common across APIs where possible;
5. the mechanism must allow all errors to be detected in a uniform way;
6. the mechanism must be extensible to multithreaded programs, and
7. the mechanism must be extensible to multiple concurrent operations.

The first requirement led to a decision that there should be an easy method for simple applications to detect all errors and terminate safely with a helpful error message (see *error_fatal*( ) below). However, sophisticated applications should be able to:

- recover where possible (e.g., by means of retries), and
- abort operations tidily (e.g., release all dynamically allocated memory).

One of the main decisions based on all the criteria was that a functional interface is preferred to a definition of data items, since this permits different implementations of the API, including vendor-specific implementations with extensions.

As an example, consider that an *error* could be specified as an aggregate [structure] containing an *error_number*, an *error_class* and a *reference* to the operation which caused the error. Instead, it is preferable to specify operations to return the *error_number*, *error_class* and *reference* when given a particular *error* as parameter.

This frees implementors to choose the best underlying representation, and enables individual vendors to add extra functionality if required without changing the standard interface. Additional functionality simply results in additional operations, rather than changed aggregate definitions.

It also makes it easier to integrate errors arising in different parts of the implementation (e.g., local system errors versus remote service errors).

Mechanism

Status Lifetime

Programming Examples

B.3.5 Object Management

B.3.6 Internationalisation

B.3.7 Resource Constraints

B.3.8 Level of API

B.3.9 Extensions

B.4 BINDINGS TO APIS

B.4.1 Introduction

B.4.2 Naming Conventions

COBOL programs in an X/Open environment are able to call C functions which means that the C naming convention is sufficient.

B.4.3 Internal and External Names

The **X/Open Portability Guide** currently requires that portable external names in C are unique in the first six characters and are case-independent, and this is also all that is guaranteed by the draft ANSI C Standard (as an obsolescent feature). ANSI intend in the future to require C implementations to recognise case differences and at least 31 characters in external names. See the section entitled **Portability - Variable Names** in the **X/Open Portability Guide**, **Issue 3**, **Volume 4**, **Programming Languages** and Sections 3.1.2 and 3.9.1 of the Draft C Standard.

However, some interfaces defined in the **X/Open Portability Guide** require more than six characters. For example, the XTI interface in **Volume 7**, **Networking Services** requires eight, while the Xlib interface in **Volume 6**, **Window Management** needs twenty-one characters to distinguish *XCirculateSubwindowsDown()* from *XCirculateSubwindowsUp()*. This suggests that all X/Open implementations of the Common Applications Environment actually provide at least this significance, since otherwise it would be impossible to link X applications. With this in mind, the X/Open Interworking Working Group has selected a sixteen-character limit. It further suggests that the recommended six-character limit in **Volume 3**, **XSI Supplementary Definitions** be reviewed and increased to at least this value.

FORTRAN 77 and previous versions of the language impose a six-character limit on the length of identifiers, but the POSIX FORTRAN binding committee (P1003.9) intend to require all implementations to provide thirty-one characters. The rationale for this is that it is necessary in order to obtain a reasonable binding to POSIX, and is in any case a very common extension in current implementations. The FORTRAN 8x standard will also require thirty-one-character significance to be supported.

The naming conventions should be reviewed when 31-character significance eventually becomes part of the **X/Open Portability Guide**.

B.4.4 Abstract Names

Because of differences between programming languages, no single set of names will work in all environments. Therefore each programming language must have a standard set of names defined for it.

Although a single convention suffices for X/Open COBOL and C, the abstract name idea is retained to allow bindings to other languages to be produced as required in individual cases or in the future. FORTRAN and Ada are examples of languages for which different conventions are appropriate.

B.4.5 General Naming Rules

B.4.6 Six-Character Naming Convention

The aim is to provide more readable names than the MAP scheme of always abbreviating to a single letter. FORTRAN 77 and previous versions of the language are examples of environments with very small limits on identifier length, but see the comments in **Section B.4.3**, **Internal and External Names** above.

B.4.7 Sixteen-Character Naming Convention

The sixteen-character convention is designed for use with external names.

B.4.8 Thirty-Character Naming Convention

The thirty-character convention is designed for use with internal names. For these, the **X/Open Portability Guide** states that a maximum of eight significant characters should be assumed for portability in C (see **Section 3.4** in **Volume 4**, **Programming Languages**). This assumption seems extremely pessimistic in view of existing interfaces adopted by X/Open which require longer names and so suggest that actual implementations provide them. The ANSI C standard requires at least 31 significant characters, which allows the complete abstract name to be used in most cases and results in much more readable identifiers. X/Open COBOL guarantees 30 significant characters in names, and is not case significant.

The suggested wording of the API Style Guide takes the view that C implementations provide 31-character capability already, and that it will in any case be required when X/Open adopts the ANSI C Standard.

B.4.9 Function Names

Chapter 5, **Layout of Specifications** requires all private but visible functions to be listed as part of the API specification, so users can detect obscure name clashes when using several packages.

Under the six-character naming scheme, the XTI functions used as examples of the naming scheme show little change from the original with careful choice of the abstract name, as can be seen in the full list given below. Other choices would result in more substantial changes, for example *receive_data_over_connection* would become *t_rdconnection* if *over* is counted as 'noise'. Note that the original names *t_rcvudata* and *t_rcvuderr* require 8 characters to be distinguished, and so exceed the maximum advised by **Section 3.4** in the **X/Open Portability Guide**, **Issue 3**, **Volume 4**, **Programming Languages**.

Under the sixteen-character naming scheme, all C function names are identical to the abstract names, and are considerably different to the existing names.

Abstract name	16-Character name	6-Character name	Original name
accept	t_accept	t_accept	t_accept
allocate	t_allocate	t_allocate	t_alloc
bind	t_bind	t_bind	t_bind
close	t_close	t_close	t_close
connect	t_connect	t_connect	t_connect
error	t_error	t_error	t_error
free	t_free	t_free	t_free
get_event	t_get_event	t_getevent	t_look
get_information	t_get_information	t_getinformation	t_getinfo
get_state	t_get_state	t_getstate	t_getstate
listen	t_listen	t_listen	t_listen
open	t_open	t_open	t_open
option_management	t_option_management	t_optmanagement	t_optmgmt
receive_connection	t_receive_connection	t_rcvconnection	t_rcvconnect
receive_data	t_receive_data	t_rcvdata	t_rcv
receive_disconnect	t_receive_disconnect	t_rcvdisconnect	t_rcvdis
receive_release	t_receive_release	t_rcvrelease	t_rcvrel
receive_unit_data	t_receive_unit_data	t_rudata	t_rcvudata
receive_unit_error	t_receive_unit_error	t_ruerror	t_rcvuderr
send_data	t_send_data	t_snddata	t_snd
send_disconnect	t_send_disconnect	t_snddisconnect	t_snddis
send_release	t_send_release	t_sndrelease	t_sndrel
send_unit_data	t_send_unit_data	t_sudata	t_sndudata
synchronise	t_synchronise	t_synchronise	t_sync
unbind	t_unbind	t_unbind	t_unbind

Abstract Name	Description
accept	accept a connect request
allocate	allocate a library structure
bind	bind an address to a transport endpoint
close	close a transport endpoint
connect	establish a connection with another transport user
error	produce error message
free	free a library structure
get_event	look at the current event on a transport endpoint
get_information	get protocol-specific service information
get_state	get the current state
listen	listen for a connect request
open	establish a transport endpoint
option_management	manage options for a transport endpoint
receive_connection	receive the confirmation from a connection request
receive_data	receive data or expedited data sent over a connection
receive_disconnect	retrieve information from disconnect
receive_release	acknowledge receipt of an orderly release indication
receive_unit_data	receive a data unit
receive_unit_error	receive a unit data error indication
send_data	send data or expedited data over a connection
send_disconnect	send user-initiated disconnect request
send_release	initiate an orderly release
send_unit_data	send a data unit
synchronise	synchronise transport library
unbind	disable a transport endpoint

B.4.10 External Variable Names

B.4.11 Internal Names

B.4.12 Error Names

B.4.13 Parameter Names

(From MAP 4.2.2.5.) No convention is needed for C or COBOL since parameter names are not part of a function's external interface, but this is not true of all languages. For example, an Ada binding would need a naming convention since parameter names are visible in the interface.

B.4.14 Parameter Placement

Consistency with ISO Guideline 35 and MAP 4.2.3.

B.4.15 Aggregate Parameters

This discussion replaces the text concerning DCBs in the MAP specification. The DCB is a concept introduced as part of the MAP interface model to which we have no corresponding concept. However some of their text is pertinent to the general notion of aggregates.

B.4.16 Parameter Usage

B.4.17 Guidelines for Other Languages

B.5 LAYOUT OF SPECIFICATIONS

B.5.1 Basic Principles

The layout is based on an amalgamation of the MAP specification (particularly A7A1.4.2.5), the POSIX proposal, the ISO Guidelines including the GKS example, the X.400 APIA specification (X.400 GW API), and existing **X/Open Portability Guide** practice. Examples of pages from each of these specifications are included as **Appendix C**, **Example Specifications**.

Further guidelines for the content of specifications can be given if the X/Open Interworking Architecture provides a single model for X/Open Interworking APIs.

The original ISO Guideline reads as follows:

48 *"For readability and ease of maintenance, a single set of contents should be used for every binding developed to a functional interface standard. Guidelines concerning the format are:*

- *It is not necessary that there be one function description per page.*
- *Descriptive information should not be copied from the functional interface standard.*
- *The functional description schema should have titles for the sections relating to arguments, errors and references.*
- *This schema should not contain explicit references back to the functional interface standard document.*
- *A short description of the arguments, mapping back to the parameters from the functional interface standard, should be given within the schema.*
- *It is not necessary for there to be a page break, with corresponding page headers, for every major section in the functional interface standard.*
- *The structure definitions should not be split between two pages.*
- *Descriptions should not be broken across two pages.*
- *Error Messages, for binding errors only, should be a part of the function description schema.*
- *Along with the table of abbreviations, there should also be a table containing the function names alphabetically, by level."*

It has been modified chiefly because the C binding has been combined with the functional interface standard for current X/Open use. The resulting documents will have different typical sizes of sections, making different layout choices more appropriate. Consistency with existing **X/Open Portability Guide** practice has also been considered.

B.5.2 Introduction

B.5.3 Types

B.5.4 Constants

B.5.5 Error Mechanism and Error Codes

See ISO Guideline 11.

B.5.6 Parameters

B.5.7 Index of C Language Names

ISO Guideline 15 suggests reserving part of the name space for implementations.

B.5.8 Object Specifications

Class Definitions

C Binding of Classes

Example Class Specification - Message

B.5.9 Function Specifications

The X/Open Interworking Working Group felt that classifying functions in an API was of little value, and outweighed by the benefits of a single alphabetical list of functions. This also accords with current **X/Open Portability Guide** style.

C SYNOPSIS

C function declarations without prototypes are an obsolescent feature of ANSI C. The chosen style, which uses prototypes, is chosen in advance of a published version of the **X/Open Portability Guide** which has adopted ANSI C. The goal is to avoid rework of new APIs, and an anticipation of **X/Open Portability Guide** adoption of ANSI C.

ARGUMENTS

Inclusion of external state dependencies and state changes accords formal status to side-effects which are often only discussed in the DESCRIPTION. Inclusion of side-effects, return value and parameter subcomponents allows bindings to other languages with different procedure calling mechanisms to be generated more easily, and also has the benefit of describing the function more completely. The parameter characteristics are chosen to match those of the X.400 APIA specification.

ERRORS

The standard description of the error is not strictly necessary since it simply repeats the description given in the full list of errors, but it is present in the current **X/Open Portability Guide** and makes the specification slightly easier to read. It is important that the standard description is precisely the same as in the full list and that any specific text is additional to it.

SEE ALSO

The abstract name is used as the primary index to the document and so is a more appropriate reference than the C name.

B.5.10 Example Function Specification - Receive_Data

C SYNOPSIS

Notes:

- The function name was originally *t_rcv*.
- The order of *buffer_length* (was **nbytes**) and *buffer_return* (was **buf**) have been exchanged so that input parameters precede output ones (ISO Guideline 35).
- There is a problem with the original specification since the buffer length is **unsigned** whilst the function return value (containing the number of bytes read) is **int**. Presumably, the number of bytes of data and the buffer length are actually restricted to less than {INT_MAX} rather than {UINT_MAX}.

DESCRIPTION

A better function would guarantee that *more_data* is never set if the concept of a TSDU is not supported, to avoid conditional tests.

The original definition of this function returns status information in several parameters (*status*, *error_number* and even *transfer_length*) which interact and are not fully defined without access to additional *information* returned from *open* or *get_information* (see below). One benefit of the suggested explicit listing of all parameters is that this kind of confusion becomes more obvious.

B.6 RULES FOR C BINDINGS

This section is present as an example of the kind of material which can be presented in additional sections for bindings to other languages if and when they are deemed necessary.

Some duplication of information from earlier sections is a consequence of the decision to combine the C binding with the functional specifications.

B.6.1 General Rules

The output parameter convention clearly shows the underlying type of the parameter and the reason that a pointer is used, which is concealed by the possible alternative declaration:

```
typedef parameter_type * parameter_pointer_type;
...
parameter_pointer_type parameter_name;
```

B.6.2 Implementation Considerations

Appendix C

Example Specifications

This Appendix is part of the rationale and lists examples from other API Style Guides which were considered in arriving at the Style Guide defined in the document.

1. an example from the **X/Open Revised XTI Developers' Specification**, **X/Open 1990**;
2. an example from the POSIX 1003.1-N178 proposal;
3. a GKS example from the ISO Guidelines for Language Bindings;
4. an example from the X/Open and X.400 APIA X.400 API Specification, and
5. an example from MAP.

C.1 EXAMPLE ONE: REVISED XTI

The following example is taken from the **X/Open Revised XTI Developers' Specification**, **X/Open 1990**, page 67ff.

NAME

t_rcv - receive data or expedited data sent over a connection

SYNOPSIS

#include <xti.h>

int t_rcv(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned int nbytes;
int *flags;

DESCRIPTION

Parameters	Before call	After call
fd	x	/
buf	x	(x)
nbytes	x	/
flags	/	x

This function receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive, *buf* points to a receive buffer where user data will be placed, and *nbytes* specifies the size of the receive buffer. The argument *flags* may be set on return from *t_rcv*( ) and specifies optional flags as described below.

By default, *t_rcv*( ) operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NONBLOCK is set (via *t_open*( ) or *fcntl*( )), *t_rcv*( ) will execute in asynchronous mode and will fail if no data is available. (See **[TNODATA]** below.)

On return from the call, if T_MORE is set in *flags*, this indicates that there is more data, and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple *t_rcv*( ) calls. In the asynchronous mode, the T_MORE flag may be set on return from the *t_rcv*( ) call even when the number of bytes received is less than the size of the receive buffer specified. Each *t_rcv*( ) with the T_MORE flag set indicates that another *t_rcv*( ) must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a *t_rcv*( ) call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from *t_open*( ) or *t_getinfo*( ), the T_MORE flag is not meaningful and should be ignored. If *nbytes* is greater than zero on the call to *t_rcv*( ), *t_rcv*( ) will return 0 only if the end of a TSDU is being returned to the user.

On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, *t_rcv*( ) will set T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have T_EXPEDITED set on return. The end of the ETSDU is identified by the return of a *t_rcv*( ) call with the T_MORE flag not set.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the *t_look*( ) function. Additionally, the process can arrange to be notified via the EM interface.

VALID STATES

T_DATAXFER, T_OUTREL

ERRORS

On failure, *t_errno* is set to one of the following:

[TBADF] The specified file descriptor does not refer to a transport endpoint.

[TNODATA] O_NONBLOCK was set, but no data is currently available from the transport provider.

[TLOOK] An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT] This function is not supported by the underlying transport provider.

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

[TSYSERR] A system error has occurred during execution of this function.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

RETURN VALUE

On successful completion, *t_rcv*( ) returns the number of bytes received. Otherwise, it returns -1 on failure and *t_errno* is set to indicate the error.

SEE ALSO

fcntl( ), *t_getinfo*( ), *t_look*( ), *t_open*( ), *t_snd*( ).

(The following is taken from Section 6.2.)

Key for Parameter Arrays

For each XTI function description, a table is given which summarises the contents of the input and output parameter. The key is given below:

x The parameter value is meaningful. (Input parameter must be set before the call and output parameter may be read after the call.)

(x) The content of the object pointed to by the x pointer is meaningful.

? The parameter value is meaningful but the parameter is optional.

(?) The content of the object pointed to by the ? pointer is optional.

/ The parameter value is meaningless.

= The parameter after the call keeps the same value as before the call.

C.2 EXAMPLE TWO: POSIX

The following example is taken from POSIX Draft 1003.1-N178 proposal, page 8ff.

Reproduced with permission.

4.4 POSIX-Specific Procedure Interfaces

4.4.1 *get-file-statistics*

The following represents the proposed format and contents for the description of the abstract equivalent of the current POSIX function whose name in the C binding is *stat*( ).

4.4.1.1 Procedure Parameters

Name	**Data Type**	**Access Type**		
		Read	**Written [Success]**	**Written [Exception]**
pathname	pathname-type	X		
file-kind	file-kind-type		X	
file-permissions	file-permission-type		X	
file-identifier	file-identifier-type		X	
link-count	link-count-type		X	
user-identification	user-identifier-type		X	
group-identification	group-identifier-type		X	
file-eof-offset	file-offset-type		X	
time-file-last-accessed	time-value-type		X	
time-file-last-modified	time-value-type		X	
time-file-last-status-change	time-value-type		X	
PROCEDURE-STATUS	procedure-status-type		X	X

4.4.1.2 Expected Procedure Status Values Returned

Error	**Description**	**Possible Recovery Action**
successful completion	Normal successful completion	N/A
error-no-access	?	?
error-name-too-long	?	?
error-no-entry	?	?
error-not-a-directory	?	?
error-invalid-file-descriptor	?	?

4.4.1.3 Associated Packages

Package Name	**Description**
system-type-definition-package	?
file-statistics-definition-package	?

4.4.2 *close-a-file*

The following represents the proposed format and contents for the description of the abstract equivalent of the current POSIX function whose name is *close*( ).

4.4.2.1 Procedure Parameters

Name	Data Type	Access Type		
		Read	Written [Success]	Written [Exception]
file-descriptor	file-descriptor-type	X		
PROCEDURE-STATUS	procedure-status-type		X	X

4.4.2.2 Expected Procedure Status Values Returned

Error	Description	Possible Recovery Action
successful completion	Normal successful completion	N / A
error-invalid-file-descriptor	?	?
error-signal-interrupt	?	?

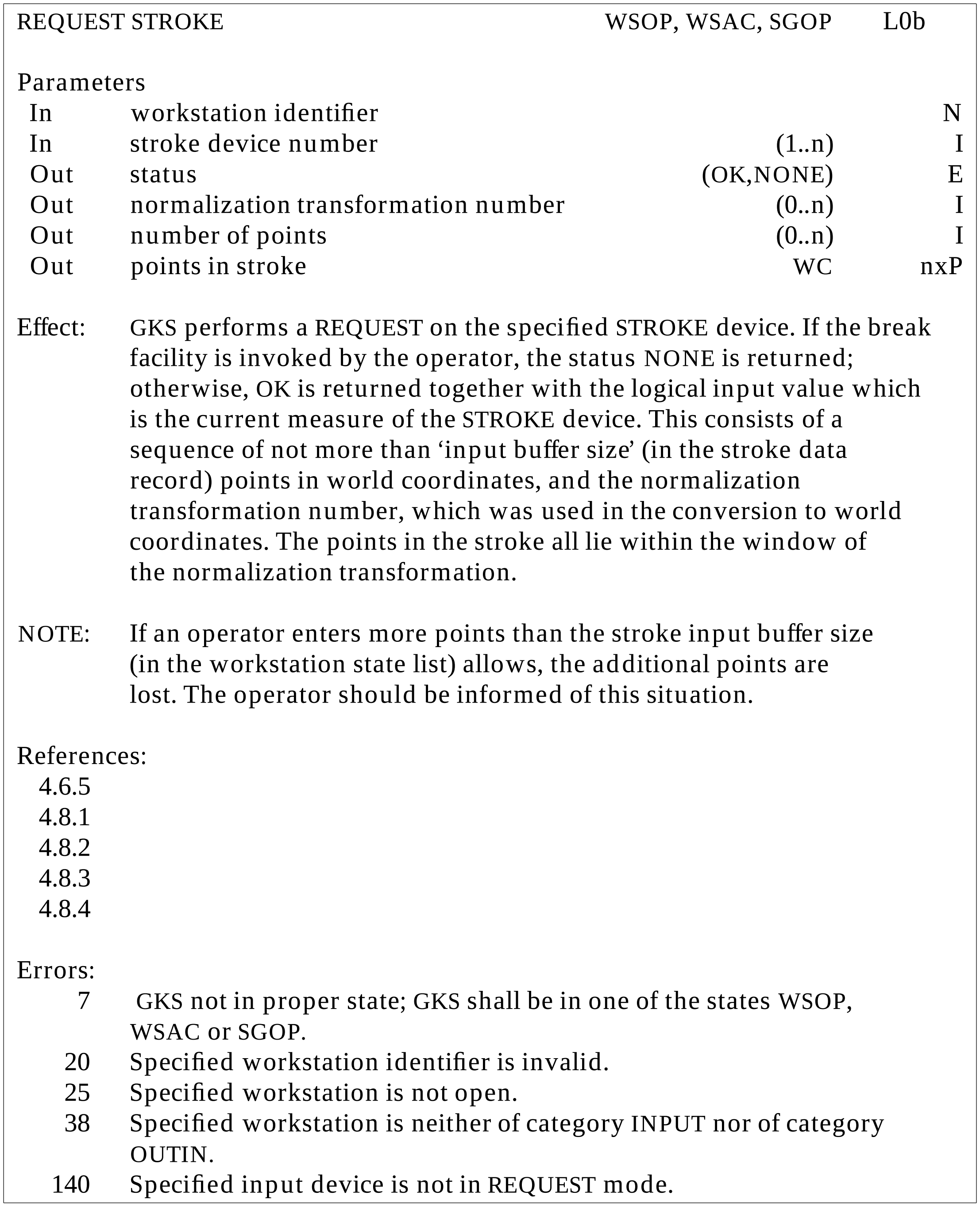
4.4.2.3 Associated Packages

None.

C.3 EXAMPLE THREE: ISO GUIDELINES

The following example is taken from the ISO Guidelines for Language Bindings, PDTR 10182, page 43ff.

Reproduced with permission. Complete copies can be obtained through national standards bodies.



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 same data type need be bound to the same data type in the language.

E enumeration a data type comprizing 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 nx4nR

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 Example 5.

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

Type
GEInputClass                    =       (GVLocator..GVString);
GTWsId                          =       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(
            WsId                        :   GTWsId;
            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

C.4 EXAMPLE FOUR: X.400 API SPECIFICATION

The following example is taken from the **X/Open and X.400 APIA X.400 API Preliminary Specification**, **X/Open 1990**, **Chapter 3**, **Message Handling Interfaces**.

3.1 DATA TYPES

This section defines, and Table 3 lists, the data types of the MA and MT interfaces that are specific to MH. The data types of both the generic and C interfaces are specified. Those of the C interface are repeated in Section 3.5, which serves as a summary and a reference. The interfaces also include the Boolean, Object, Object Identifier, Private Object, Return Code, String and intermediate data types of the OM interface.

Data Type	Description
Feature	The features to be negotiated for a session.
Interval	An interval of time measured in milliseconds.
Object Count	A number of objects.
Sequence Number	The sequence number of an object in a retrieval queue.

Table 3 : Interface Data Types Specific to MH

3.1.1 Feature

NAME

Feature - type definition for requesting features

DECLARATION

```
typedef struct {
    OM_object_identifier feature;
    OM_boolean      activated;
} MH_feature;
```

DESCRIPTION

A data value of this type is used for negotiating the features of a session.

3.1.2 Interval

NAME

Interval - the integer that denotes an interval of time measured in milliseconds

DECLARATION

```
typedef OM_uint32 MH_interval;
```

DESCRIPTION

A data value of this data type is the integer in the interval $[0, 2^{32})$ that denotes an interval of time measured in milliseconds.

3.1.3 Object Count

NAME

Object Count - the integer that denotes a number of objects

DECLARATION

```
typedef OM_uint32 MH_object_count;
```

DESCRIPTION

A data value of this data type is the integer in the interval $[0, 2^{32})$ that denotes a number of objects.

3.1.4 Sequence Number

NAME

Sequence Number - the sequence number of an object in a retrieval queue

DECLARATION

typedef OM_uint32 MH_sequence_number;

DESCRIPTION

A data value of this data type is the integer in the interval [0, 2^{31}) (sic) that denotes a message or report in a retrieval queue.

Sequence numbers are assigned in ascending order, but not necessarily consecutively. An object's sequence number never changes, and no sequence number denotes two different objects, even at different times.

3.2 ACCESS FUNCTIONS

This section defines, and Table 4 lists, the functions of the MA interface. The functions of both the generic and C interfaces are specified. Those of the C interface are repeated in Section 3.5, which serves as a summary and a reference.

Function	Description
Cancel Submission	Cancel the deferred delivery of a submitted message.
Close	Terminate an MA session.
Finish Delivery	Conclude the delivery in progress in a session.
Finish Retrieval	Conclude the retrieval in progress in a session.
Open	Establish an MA session.
Size	Determine the size of the delivery or retrieval queue.
Start Delivery	Begin the delivery of a message or a report.
Start Retrieval	Begin the retrieval of a message or a report.
Submit	Submit a communique.
Wait	Return when an object is available for delivery or retrieval.

Table 4 : MA Interface Functions

As indicated in the table, the MA interface comprises a number of functions whose purpose and range of capabilities are summarised as follows:

1. **Cancel Submission.** This function cancels the deferred delivery of a message, without regard to the session in which it was submitted.

2. **Close.** This function terminates an MA session between the client and the service. If the delivery or retrieval of a message or a report is in progress, the service first unsuccessfully finishes that delivery or retrieval.

3. **Finish Delivery.** This function concludes the delivery in progress in a session. The client supplies delivery confirmations, as required, for users to which the object, a message, was delivered. It also indicates to which users the object, either a message or a report, is undeliverable.

4. **Finish Retrieval.** This function concludes the retrieval in progress in a session. The client indicates whether the service is to remove the retrieved message or report

from the retrieval queue, or leave it there.

5. **Open.** This function establishes an MA session between the client and the service, and makes the Basic Access FU and the OM Package initially available in that session. The client may also specify the other features required for the session. The client specifies either its own name or the O/R address of a local user. The session provides MTS access to the local user at the specified address, if the latter, or to a group of local users, statically associated with the client name, if the former.
6. **Size.** This function determines the number of messages and reports in the delivery or retrieval queue to which a session provides access.
7. **Start Delivery.** This function begins the delivery of a message or a report to one or more of the users associated with a session. If no messages or reports await delivery, the function reports an exception.
8. **Start Retrieval.** This function begins the retrieval of a message or a report from the retrieval queue to which a session provides access. If no messages or reports await retrieval, the function reports an exception.
9. **Submit.** This function submits a communique (see the Communique class) by adding it to the submission queue to which a session provides access. The function first verifies the communique's integrity.
10. **Wait.** This function returns when a message or a report is available for delivery or retrieval in the delivery or retrieval queue to which a session provides access, or when a period of time elapses, whichever occurs first.

The functions are grouped into four FUs - one basic and one each for submission, delivery, and retrieval - as indicated in Table 5. (As stated previously, the Delivery and Retrieval FUs are mutually exclusive.)

Basic Access	Submission	Delivery	Retrieval
Open	Submit	Size	Size
Close	Cancel Submission	Start Delivery	Start Retrieval
OM API		Finish Delivery	Finish Retrieval

Table 5 : MA Interface Functional Units

Note: OM API is defined in [1]

The intent of the interface definition is that each function is atomic, i.e., that it either carries out its assigned task in full and reports success, or fails to carry out even a portion of the task and reports an exception. However, the service does not guarantee that a task will not occasionally be carried out in part but not in full.

Note: Making such a guarantee might be prohibitively expensive.

Whether a function detects and reports each of the exceptions listed in the Errors clause of its specification is unspecified. If a function detects two or more exceptions, which it reports is unspecified. If a function reports an exception for which a return code is defined, however, it uses that (rather than another) return code to do so.

NAME

Cancel Submission - cancel the deferred delivery of a submitted message

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_cancel_submission (
    OM_private_object      session,
    OM_object              mts_identifier
);
```

DESCRIPTION

This function cancels the deferred delivery of a message, without regard to the session in which it was submitted.

ARGUMENTS

Session (Private Object)

An established MA session between the client and the service; an instance of the Session class.

MTS Identifier (Object)

The MTS identifier assigned to the message whose delivery is to be cancelled; an instance of the MTS Identifier class.

RESULTS

Return Code (Return Code)

Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-message, no-such-representation, no-such-session, no-such-syntax, no-such-type, not-private, permanent-error, pointer-invalid, system-error, temporary-error, too-late, too-many-values, wrong-class, wrong-value-length, wrong-value-makeup, wrong-value-number, wrong-value-syntax or wrong-value-type.

NAME

Close - terminate an MA session between the client and the service

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_close (
   OM_private_object       session
);
```

DESCRIPTION

This function terminates an MA session between the client and the service. If the delivery or retrieval of a message or a report is in progress in the session, the service first unsuccessfully finishes that delivery or retrieval.

ARGUMENTS

Session (Private Object)
An established MA session between the client and the service; an instance of the Session class.

RESULTS

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

ERRORS

function-interrupted, memory-insufficient, network-error, no-such-session, not-private, permanent-error, pointer-invalid, system-error, temporary-error or wrong-class.

NAME

Finish Delivery - conclude the delivery in progress in a session

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_finish_delivery (
    OM_private_object       session,
    OM_object               delivery_confirmations,
    OM_object               non_delivery_reports
);
```

DESCRIPTION

This function concludes the delivery in progress in a session. The client supplies delivery confirmations, as required, for users to which the object, a message, was delivered. It also indicates to which users the object, either a message or a report, is undeliverable, and thus, by implication, to which users the object was delivered.

The client indicates to which users the message or report is temporarily, rather than permanently, undeliverable. The circumstances that cause temporary undeliverability are client-defined. However, whether the service will accept any circumstances as causes of temporary failure is service implementation-defined. If the service does not support temporary failures, it treats them as permanent.

If the object is a message, the service issues delivery reports (DRs), as required, for the users to which it has been delivered; issues NDRs, as required, for the users to which it is permanently undeliverable; and returns the message to the delivery queue for the users to which it is temporarily undeliverable. In the first case, the service considers that it has transferred responsibility for the message to the client.

If the object is a report, the service returns the report to the delivery queue for the users to which it is temporarily undeliverable. For the users to which the report has been delivered, the service considers that it has transferred responsibility for the report to the client.

ARGUMENTS

Session (Private Object)

An established MA session between the client and the service; an instance of the Session class. A delivery shall be in progress.

Delivery Confirmations (Object)

One or more delivery confirmations, as required, for users to which the object, a message, was delivered; an instance of the Local Delivery Confirmations class.

This argument is omitted if there are no confirmations.

In the C interface, the argument's absence is signalled by the null pointer.

Non-delivery Reports (Object)

Indicates the one or more users to which the object cannot be delivered; an instance of the Local NDR class.

This argument is omitted if there are no such users.

In the C interface, the argument's absence is signalled by the null pointer.

RESULTS

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-object, no-such-representation, no-such-session, no-such-syntax, no-such-type, not-private, permanent-error, pointer-invalid, session-not-busy, system-error, temporary-error, too-many-values, wrong-class, wrong-value-length, wrong-value-makeup, wrong-value-number, wrong-value-syntax or wrong-value-type.

NAME

Finish Retrieval - conclude the retrieval in progress in a session

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_finish_retrieval (
    OM_private_object      session,
    OM_boolean             remove
);
```

DESCRIPTION

This function concludes the retrieval in progress in a session. The client indicates whether the service is to remove the message, report from the retrieval queue, or leave it there. In the former case, the service considers that it has transferred responsibility for the object from the service to the client. In the latter case, the service makes the objects inaccessible (i.e., the object handles are made invalid); the associated communique or report, however, can be obtained again in a subsequent Start Retrieval function invocation.

ARGUMENTS

Session (Private Object)

An established MA session between the client and the service; an instance of the Session class. A retrieval shall be in progress.

Remove (Boolean)

Whether the service is to remove the retrieved message or report from the retrieval queue, rather than leave it there.

RESULTS

Return Code (Return Code)

Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-object, no-such-session, not-private, permanent-error, pointer-invalid, session-not-busy, system-error, temporary-error or wrong-class.

NAME

Open - establish an MA session between the client and the service

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_open (
   OM_object           user_address,
   OM_string           client_name,
   MH_feature          feature_list[],
   OM_private_object   *session,
   OM_workspace              *workspace
);
```

DESCRIPTION

This function establishes an MA session between the client and the service, and makes the Basic Access FU and the OM Package initially available in that session. The client may also specify the other features required for the session.

The client specifies either its own name or the O/R address of a local user. The session provides MTS access to the local user at the specified address, if the latter, or to a group of local users, statically associated with the client name, if the former. The Retrieval FU can be requested only if a single user is designated, but that user may be designated in either of these two ways.

The client always designates a particular user in the same way. The choice between O/R address and client name is made by means outside the scope of this document. How users are associated with a client name is outside the document's scope. The maximum number of users in a group is implementation-defined (and may be one).

Opening an MA session also creates a workspace. A workspace contains objects returned as a result of functions invoked within that session. The workspace is used as an argument in the OM Create and Copy functions.

The maximum number of sessions that may exist simultaneously is implementation-defined and may vary with time.

ARGUMENTS

User Address (Object)

Explicitly identifies the local user to which the session is to provide MTS access; an instance of the OR Address class.

If this argument is absent, the Client Name argument shall be present.

In the C interface, the argument's absence is signalled by the null pointer.

Client Name (String)

The name by which the service knows the client, interpreted as a value whose syntax is String (IA5). It implicitly identifies one or more local users to which the session is to provide MTS access.

If this argument is absent, the User Address argument shall be present.

In the C interface, the argument's absence is signalled by the null pointer.

Feature-List (Feature-List)
An ordered sequence of features, each represented by an object identifier. The sequence is terminated by an object identifier having no components (a length of zero and any value of the data pointer in the C representation).

RESULTS

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

Activated (Boolean-List)
If the function completed successfully, this result contains an ordered sequence of Boolean values, with the same number of elements as the Feature-List. If true, each value indicates that the corresponding feature is now part of the interface. If false, each value indicates that the corresponding feature is not available.

In the C binding, this result is combined with the Feature-List argument as a single array of structures of type MH_feature.

Session (Private Object)
The established MA session between the client and the service; an instance of the Session class. The service prevents the client from modifying this object subsequently. This result is present if and only if the Return Code result is success.

Workspace (OM_workspace)
The workspace that will contain all objects returned as a result of the functions invoked in the session.

ERRORS

feature-conflicts, feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-client, no-such-representation, no-such-syntax, no-such-type, no-such-user, permanent-error, pointer-invalid, system-error, temporary-error, too-many-sessions, too-many-values, wrong-class, wrong-value-length, wrong-value-makeup, wrong-value-number, wrong-value-syntax or wrong-value-type.

NAME

Size - determine the number of messages and reports in the delivery or retrieval queue

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_size (
    OM_private_object      session,
    MH_object_count        *number
);
```

DESCRIPTION

This function determines the number of messages and reports in the delivery or retrieval queue to which a session provides access.

ARGUMENTS

Session (Private Object)

An established MA session between the client and the service; an instance of the Session class.

RESULTS

Return Code (Return Code)

Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

Number (Object Count)

The number of objects in the delivery or retrieval queue. However, if that number exceeds 2^{16}-1, 2^{16}-1 is returned. This result is present if and only if the Return Code result is success.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-session, not-private, permanent-error, pointer-invalid, system-error, temporary-error or wrong-class.

NAME

Start Delivery - begin the delivery of a message or a report to one or more of the users associated with a session

SYNOPSIS

[#include <xmh.h> |]

```
OM_return_code
ma_start_delivery (
    OM_private_object      session,
    OM_private_object      *object
);
```

DESCRIPTION

This function begins the delivery of a message or a report to one or more of the users associated with a session (see the Open function). If no objects await delivery, the function reports an exception. The client shall finish the delivery of one object before it starts the delivery of another in the same session. The delivery of a particular object cannot simultaneously be in progress in two sessions.

Whether the service begins the delivery of an object addressed to several users in one function invocation, or in one invocation per user, is dependent on the Multiple-delivery feature being available for the session.

Which qualifying object in the delivery queue (if there are several such objects) the service selects for delivery is implementation-defined.

Note: The invocation of this function initiates but does not consummate delivery. That is, it does not transfer responsibility for the object from the service to the client. That is accomplished by means of the Finish Delivery function.

Note: The fact that the Wait or Size function indicated that the delivery queue contained an object immediately prior to invocation of this function does not guarantee this function's success. For example, another process might have begun the object's delivery.

ARGUMENTS

Session (Private Object)

An established MA session between the client and the service; an instance of the Session class.

RESULTS

Return Code (Return Code)

Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

Object (Private Object)

The object whose delivery is started; an instance of the Delivered Message or the Delivered Report class. If the former, the object includes one envelope for each of one or more of the users associated with the session. The service prevents the client from modifying this object subsequently. This result is present if and only if the Return Code result is success.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-session, not-private, permanent-error, pointer-invalid, queue-empty, session-busy, system-error, temporary-error or wrong-class.

NAME

Start Retrieval - begin the retrieval of a message or a report

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_start_retrieval (
    OM_private_object          session,
    MH_sequence_number         minimum_sequence_number,
    MH_sequence_number         *selected_sequence_number,
    OM_private_object          *object
);
```

DESCRIPTION

This function begins the retrieval of a message or a report from the retrieval queue to which a session provides access. The client shall finish the retrieval of one object before it starts the retrieval of another in the same session. The retrieval of a particular object cannot simultaneously be in progress in two sessions.

The service selects for retrieval the object whose sequence number is nearest to but no less than a sequence number specified by the client. If no object has a sequence number greater than or equal to that specified, the function reports an exception.

Note: The invocation of this function initiates but does not consummate retrieval. That is, it does not transfer responsibility for the object from the service to the client. That is accomplished, if desired, by means of the Finish Retrieval function.

Note: The fact that the Wait or Size function indicated that the retrieval queue contained an object immediately prior to invocation of this function does not guarantee this function's success. For example, another process might have begun the object's retrieval.

ARGUMENTS

Session (Private Object)

An established MA session between the client and the service; an instance of the Session class.

Minimum Sequence Number (Sequence Number)

The sequence number of the first message or report to be considered for retrieval.

RESULTS

Return Code (Return Code)

Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

Selected Sequence Number (Sequence Number)

The sequence number of the message or report selected for retrieval. This result is present if and only if the Return Code result is success.

Object (Private Object)

The object whose retrieval is started; an instance of the Delivered Message or the Delivered Report class. If the former, the object includes a single envelope, for the user associated with the session. The service prevents the client from modifying this object subsequently. This result is present if and only if the Return Code result is success.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-session, not-private, permanent-error, pointer-invalid, queue-empty, session-busy, system-error, temporary-error or wrong-class.

NAME

Submit - submit a communique

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_submit (
    OM_private_object      session,
    OM_object              communique,
    OM_private_object      *submission_results
);
```

DESCRIPTION

This function submits a communique by adding it to the submission queue to which a session provides access. This transfers responsibility for the communique from the client to the service. The function first verifies the communique's integrity.

ARGUMENTS

Session (Private Object)

An established MA session between the client and the service; an instance of the Session class.

Communique (Object)

The object to be submitted; an instance of the Submitted Communique class. Its purported originator shall be among the users associated with the session. The communique is made inaccessible.

RESULTS

Return Code (Return Code)

Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

Submission Results (Private Object)

The results of the submission; an instance of the Submission Results class. This result is present if and only if the Return Code result is success.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-class, no-such-object, no-such-representation, no-such-session, no-such-syntax, no-such-type, not-private, originator-improper, permanent-error, pointer-invalid, system-error, temporary-error, too-many-values, wrong-class, wrong-value-length, wrong-value-makeup, wrong-value-number, wrong-value-syntax or wrong-value-type.

NAME

Wait - return when a message or a report is available for delivery or retrieval, or when a period of time elapses, whichever occurs first

SYNOPSIS

```
[#include <xmh.h> | ]

OM_return_code
ma_wait (
    OM_private_object      session,
    MH_interval            interval,
    OM_boolean             *available
);
```

DESCRIPTION

This function returns when a message or a report is available for delivery or retrieval in the delivery or retrieval queue to which a session provides access, or when a period of time elapses, whichever occurs first.

The function manipulates an event flag associated with the session. It returns when the event flag is true or after a specified interval has elapsed, whichever occurs first. If the interval is zero, the client is not blocked in any circumstance.

A session's *event flag* is a Boolean maintained by the service. It is set to false when the session is first established and again whenever the present function finds it true. Asynchronously, the service may set the flag to true whenever it places an object in the delivery or retrieval queue to which the session provides access. If several sessions provide access to that queue (see the Open function), the service sets to true the flag associated with at least one. Which and how many sessions the service notifies in this manner are implementation-defined.

Note: This function is designed to be easily implemented using the event signalling primitives of many operating systems, including the primitives whose inclusion in POSIX is currently under discussion within IEEE.

ARGUMENTS

Session (Private Object)
An established MA session between the client and the service; an instance of the Session class.

Interval (Interval)
The maximum length of time that the service is to block the client before returning.

RESULTS

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be **success** or one of the values listed under ERRORS below.

Available (Boolean)
Whether the event flag was ever found true. This result is present if and only if the Return Code result is success.

ERRORS

feature-unavailable, function-interrupted, memory-insufficient, network-error, no-such-session, not-private, permanent-error, pointer-invalid, system-error, temporary-error or wrong-class.

C.5 EXAMPLE FIVE: MAP SPECIFICATION VERSION 3.0

The following example is taken from the MAP Specification Version 3.0, Appendix 7, Attachment 3, page A7A3-5.13ff.

Reproduced with permission.

5.4 LISTEN

5.4.1 Description

The purpose of this function is to declare a willingness of the AE to accept an association indication from another AE invocation. It also informs the network service provider that any additional association should be queued for this AE.

This function corresponds to one primitive, i.e., A_ASSOCIATE indication.

A successful completion of this function will provide a CONNECTION_ID which can be used for rejecting or honoring the peer request for association.

An A_ASSOCIATE indication received by this function has been approved by both the ACSE and the corresponding ASE. An OUTPUT BUFFER OVERFLOW error will destroy an A_ASSOCIATE indication and there is no mechanism to recover it.

It is possible for an outstanding asynchronous Listen function to be terminated by a Stop Listen function. As a result, an error code STOP LISTEN PURGE will be generated by the STOP LISTEN Service Provider and the return_event_name of the Listen function will be noted. Refer to the Stop Listen function detail description on the Service Provider for more information.

5.4.2 References

Section 2.2.1 Number of AE Invocations Per Presentation Address
Section 2.3 Responding AE Invocation Selection
Section 2.5.1 Receiving Association Indications
Section 3.2 Connection Establishment Phase

5.4.3 Parameters

name	(short description)	default
Exposed Input:		
AE_LABEL		
RETURN_EVENT_NAME		SYNCHRONOUS
In / Out DCB:		
Input:		

IN / OUT_DCB_SIZE:

Output:

CONTEXT_NAME
CALLING_AE_NAME
CALLED_AE_NAME
CALLING_PRESENTATION_ADDRESS
CALLED_PRESENTATION_ADDRESS
RETURN_CODE
ASE_SPECIFIC_ASSO_IND_INFORMATION
(The exact structure of this information must be defined by each ASE interface specification)

Exposed Output:

CONNECTION_ID

5.4.4 Errors

DUPLICATE RETURN_EVENT_NAME
INVALID RETURN_EVENT_NAME
INVALID AE_LABEL
SERVICE UNAVAILABLE
NO RESOURCES TO QUEUE REQUEST
OUTPUT BUFFER OVERFLOW
ASSOCIATIONS PER AE INVOCATION EXCEEDED
STOP LISTEN PURGE

5.4.5 Detail Description

5.4.5.1 Library Function

1. Check the range of RETURN_EVENT_NAME. If it is out of range, indicate INVALID RETURN_EVENT_NAME error and return to the user.
2. If the RETURN_EVENT_NAME is "in use", indicate DUPLICATE RETURN_EVENT_NAME error and return to the user.
3. Perform General Error Checks that can be carried out by the Library Function.
4. If RETURN_EVENT_NAME is SYNCHRONOUS, generate a unique Return_Event.

5. Queue the service request for the network service provider. If the request cannot be queued because the provider is not present, indicate NETWORK SERVICE UNAVAILABLE. If the request cannot be queued because no system resource is available, indicate NO RESOURCES TO QUEUE REQUEST.
6. If the request was synchronous, wait for the Return_Event to be noted with no time out.
7. Return to the user.

5.4.5.2 General Error Checks

If AE_LABEL is not recognized by the server, indicate UNKNOWN AE_LABEL error. If the error checking was performed in the library routine, then return to the user program. Otherwise, note Return_Event and terminate further processing on the request.

5.4.5.3 HLSP Function

If previous general error checks were successful, the following must be performed.

1. Record the AE_Invocation as interested in receiving indications.
2. If the Number of Associations Per AE Invocation is exceeded, indicate ASSOCIATIONS PER AE INVOCATION EXCEEDED.
3. Issue an indication solicitation to the PSP.
4. Wait for the response from the PSP with indefinite time.
5. If there is a STOP LISTEN PURGE error, indicate it and skip the next two steps.
6. Put all output information from the PSP into the designated output area. If the output buffer area is insufficient, indicate OUTPUT BUFFER OVERFLOW.
7. If no error, increment the Number of Associations per AE invocation for AE_LABEL.
8. Note Return_Event.

5.4.5.4 PSP Function

The PSP behavior is consistent with the PSP specification in the Interface Model and Specifications Requirements document.

This particular PSP function, however, is AE specific, not AE invocation specific.

Index