ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

STANDARD ECMA-127

REMOTE PROCEDURE CALL USING OSI (RPC)

Second Edition - June 1990

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

STANDARD ECMA-127

REMOTE PROCEDURE CALL USING OSI (RPC)

Second Edition - June 1990

BRIEF HISTORY

The first edition of this ECMA Standard was ratified in December 1987.

In 1988 the standard was submitted to ISO/IEC JTC1 for ratification by the "fast track" procedure, and was balloted as DIS 10148. The ballot closed in February 1989 with insufficient votes for acceptance. This experience yielded the following information:

(a) ISO Member Bodies generally recognize that there is a need for urgent standardisation in this field.

(b) the "fast track" procedure is apparently not suitable for introduction of new standardization which has complex relationships to other standards work items;

(c) in particular, the ballot was perceived to raise issues that span SC21 OSI work, SC21 ODP work, SC22 languages work, etc.;

(d) technical misunderstandings arose from the way the technical content was formulated in the original text, and from the way it used OSI terminology and concepts;

(e) some technical errors were found in the document.

Learning from this experience, this second edition clarifies the standard and corrects known errors and sources of misunderstanding. The main changes are as follows:

(f) Clarification that the scope is extending the use of existing OSI standards to support RPC interactions.

(g) Clarification that architectural structure, programming interfaces and language bindings are out of scope (i.e. this is not the specification of a comprehensive "RPC system"; the focus is use of interconnection standards).

(h) Recognition that more sophisticated and comprehensive RPC standardisation depends on ODP architecture (Open Distributed Processing). ODP architecture is currently being developed in ECMA and in JTC1/SC21/WG7, and is not yet sufficiently mature for the development of an ODP RPC standard.

(i) Deletion of the "RPC Model" and its "D1", "D2" and "D3" interface structure. This is a consequence of (g) and (h) above, and a source of many of the misunderstandings (d).

This second edition will be resubmitted to ISO/IEC JTC1 after consultation with SC21 experts.

This Second Edition of Standard ECMA-127 has been approved by the ECMA General Assembly of 28th June 1990.

Table of Contents

1. SCOPE

The subject of this Standard ECMA-127 is Remote Procedure Call (RPC) for distributed applications, using the OSI standards ISO 8649, ISO 8650 (ACSE), ISO 8824, ISO 8825 (ASN.1), and ISO/IEC 9072 (ROSE).

This standard:

(a) defines the model of computation that is supported (see 6);

(b) defines an Interface Definition Notation (see 7);

(c) defines the RPC Service (see 8);

(d) defines the RPC Protocol (see 9);

(e) provides tutorial explanation and guidance (see Appendices).

This standard:

(f) is restricted to an interaction structure which is common to the procedure call constructs and data types of ISO programming languages;

(g) does not address the implementation of the RPC Service Provider using a connectionless transport and/or unreliable communication services.

(h) is restricted to those matters which relate to the use of existing OSI standards;

(i) does not include the architectural structure of support environments for RPC, or RPC programming language interfaces, and language bindings;

(j) does not address security issues.

More comprehensive RPC standardization is left for future study in the context of the Reference Model of Open Distributed Processing currently under development in ISO/IEC JTC1/SC21 WG7 (see ECMA TR/49). This standard is intended as a first step by enabling existing OSI standards to be used for the interconnection of applications written in higher level programming languages.

2. FIELD OF APPLICATION

The field of application of this standard is specifications of distributed application interactions, and equipment implementing OSI interconnection to support such interactions.

3. CONFORMANCE

A definition of the remote interactions of a distributed application complies with this standard if it uses an Interface Definition Language consistent with the Interface Definition Notation defined in 7.

An implementation conforms with this Standard if it implements the protocol specified in 9.

An implementation of this standard shall use a connection oriented protocol stack which guarantees adequate reliability characteristics (e.g., negligible loss of data, sequential delivery of transmitted data, and aliveness control).

Means of conformance testing are not defined in this standard.

No conformance requirements are defined for the internal interfaces of products or for language bindings.

4. REFERENCES

ECMA TR/49	Support Environment for Open Distributed Processing (SE-ODP).
ISO 6093	Representation of Numerical Values in Character Strings for Information Interchange.
ISO 7498	Information Processing Systems - Open Systems Interconnection - Basic Reference Model.
ISO TR 8509	Information Processing Systems - Open Systems Interconnection - Service Conventions.
ISO 8649	Information Processing Systems - Open Systems Interconnection - Service Definition for the Association Control Service Element.
ISO 8650	Information Processing Systems - Open Systems Interconnection - Protocol Specification for the Association Control Service Element.
ISO 8824	Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)
ISO 8825	Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules of Abstract Syntax Notation One (ASN.1)
ISO/IEC 9072/1	Draft International Standard: Information Processing Systems - Text Processing - Remote Operations Part 1: Model, Notation and Service Definition.
ISO/IEC 9072/2	Draft International Standard: Information Processing Systems - Text Processing - Remote Operations Part 2: Protocol Specification.
ISO/IEC 9545	Information Processing Systems - Open Systems Interconnection - Application Layer Structure
ISO/IEC 10031-1	Information Processing Systems - Text communications - Distributed Office Applications Model Part 1: General Model.

References to documents that are not standards publications are listed in Appendix C.

5. DEFINITIONS

5.1 External Definitions

This Standard uses the following terms defined in other standard documents:

- Application Entity Title:	(ISO 7498/3)
- Application Entity Invocation:	(ISO/IEC 9545)
- Application Entity Invocation identifier:	(ISO/IEC 9545)
- Application Process Invocation:	(ISO/IEC 9545)
- Application Process Invocation identifier:	(ISO/IEC 9545)
- Application Service Element:	(ISO 7498)
- Association:	(ISO 8649)
- Association Control Service Element:	(ISO 8649)
- Client:	(ISO/IEC 10031)
- Remote Operation:	(ISO/IEC 9072/1)
- Remote Operations Service Element:	(ISO/IEC 9072/1)
- Server:	(ISO/IEC 10031)
- IMPORT:	(ISO 8824)
- EXPORT:	(ISO 8824)

NOTE 1:
The terms "IMPORT" and "EXPORT" are used in ASN.1 module specifications. In this standard, these terms are not used with any other meaning.

5.2 RPC Definitions

For the purposes of this Standard the following definitions apply.

5.2.1 Procedure

A closed sequence of instructions that is entered from, and returns control to, an external source.

5.2.2 Procedure Function

A procedure which returns a value.

5.2.3 Procedure Definition

A declaration specifying the procedure name, its formal parameters with their names and types, whether the parameters are input, output or input/output, and whether there are any procedures to be called back.

5.2.4 Procedure Call

The invocation of a procedure.

5.2.5 Calling Procedure

The procedure which invokes another procedure.

5.2.6 Called Procedure

The procedure which is invoked by a procedure call.

5.2.7 Procedure Return

The act of returning control to the calling procedure.

5.2.8 **Remote Procedure**

A procedure which is invoked in a different execution context than the execution context of the caller.

5.2.9 **Remote Procedure Call; RPC**

A procedure call in which the calling procedure and the called procedure exist in separate execution contexts.

5.2.10 **Remote Procedure Call Binding; RPC Binding**

A relationship between a client, a server and an interface definition, which is used for remote procedure calls.

5.2.11 **Remote Procedure Callback**

An RPC call, x, from within the execution of a remote procedure, y, to a procedure within the execution context of the caller of y.

5.2.12 **Interface**

The externally visible characteristics of a set of procedures. These characteristics include datatype declarations, procedure declarations, and error declarations.

5.2.13 **Interface Definition**

The specification of an interface.

5.2.14 **Interface Definition Notation**

A notation for specifying interface definitions.

5.2.15 **Input Parameter**

A value communicated to the called procedure on entry from the calling procedure.

5.2.16 **Output Parameter**

A value communicated from the called procedure on return to the calling procedure.

5.2.17 **Input/Output Parameter**

A pair of related values, one communicated to the called procedure on entry from the calling procedure, and the other from the called procedure on return to the calling procedure.

5.2.18 **Marshalling**

The process of collecting parameters, converting them to an agreed data representation, and assembling them for transmission.

5.2.19 **Unmarshalling**

The process of disassembling a list of parameters from the message in which they were transmitted and converting them to the format used by the procedure.

5.3 Acronyms

ACSE	Association Control Service Element
AE	Application Entity
AEI	Application Entity Invocation
AP	Application Process
API	Application Process Invocation
APDU	Application Protocol Data Unit
PDU	Protocol Data Unit
RO	Remote Operations
ROIV	RO-INVOKE APDU
RORE	RO-ERROR APDU
RORJ	RO-REJECT APDU
RORS	RO-RESULT APDU
ROSE	Remote Operations Service Element

6. COMPUTATIONAL MODEL

This clause describes how Remote Procedure Calls are related to OSI concepts. Tutorial information on remote procedures is given in Appendix B.

6.1 Interface Definition

An RPC package defines a set of logically related procedures which will be executed remotely. The definitions associated with a package are contained in an Interface Definition. In this standard, interface definitions are written in the Interface Definition Notation which is defined in 7. The Interface Definition identifies:

a) the name of the package. This is intended to be the name by which people refer to the package;

b) the AE-Type Title (the Interface Definition may only be part of the AE-Type);

c) the version of the package;

d) the procedures contained in the package;

e) the parameters and function result (if any) of the procedures and their datatypes and attributes;

f) the set of errors which may be returned by a procedure;

g) the definition and declaration of datatypes;

h) the definition of errors generated by the procedures.

The Interface Definition may be used to generate routines that marshal and unmarshall the remote procedure's parameters for the calling and called procedures.

6.2 The Application Process Invocation

When using RPC an Application Process Invocation contains:

a) one or more procedures;

b) routines to marshal and unmarshall parameters; and

c) one or more Application Entity Invocations to provide the communications facilities required by the application. An AEI for RPC only supports a single Interface Definition.

6.3 Underlying OSI Protocols

When using RPC, the Application Entity Invocation will contain ROSE and Connection Oriented ACSE.

The ACSE services will be used by the RPC Service Provider either directly or indirectly (e.g. using Transaction Processing services).

The RPC Service Provider also assumes that the OSI protocols supporting it provide a reliable communication service.

6.4 Remote Procedure Call Binding

Prior to invoking a remote procedure in a particular interface, an RPC Binding is necessary between the client and the server. This RPC Binding is identified by a Binding-Handle.

Establishing the RPC Binding requires that the client knows the AE-Title (and possibly the AP-Invocation identifier) of the server; however, the method by which the client discovers this is outside the scope of this standard. The client initiates the RPC Binding by invoking the RPC-BIND service which passes the AE-Title to the RPC Service Provider. In turn the RPC Service Provider, depending on the Application Context, may use an appropriate service to establish an association, e.g. A-ASSOCIATE or TP-BEGIN-DIALOGUE.

The RPC-BIND service may defer association establishment until the first remote procedure is invoked, or may use an existing association. There may be more than one Association within an RPC Binding for simultaneous remote procedure calls.

6.5 Use of Associations

There is at most one remote procedure call outstanding on an Association at any given moment. This is to avoid deadlock and problems of fairness. During this time, only callbacks and the Cancel operation occur on the Association. If an Application Process Invocation supports multi-threading, a single Association may be used serially for remote calls emanating from different threads. One or more associations may be used by each thread to support multiple outstanding calls. When there is no remote procedure call outstanding, an Association may be used for other communications without disrupting the RPC Binding.

6.6 Procedure Invocation

When a procedure invokes a remote procedure, routines local to the calling procedure marshal the parameters specified in the call. The local implementation of the invocation and the marshalling routines are beyond the scope of this standard.

The marshalling routines in turn transfer control to the appropriate Application Entity Invocation. This AEI then communicates with the designated Application Entity Invocation within the Application Process Invocation containing the called procedure.

When the called Application Entity Invocation receives the request to invoke a procedure (which corresponds to a RPC-INVOKE service primitive), it transfers control to the routines within the Application Process Invocation which unmarshall the called procedure's parameters. When the unmarshalling is completed, the called procedure is invoked. The nature of the unmarshalling and invocation are beyond the scope of this standard.

6.7 Procedure Return

There is always a response generated (at least locally) as a consequence of invoking a procedure. If the called procedure has been successfully invoked and has produced results, these results are marshalled and transmitted to the calling Application Entity Invocation. If the called procedure could not be invoked, the ROSE Reject Reason is transmitted to the calling Application Entity Invocation. If the called procedure has been successfully invoked, but has not executed successfully, an error indication is transmitted to the calling Application Entity Invocation. The method used to determine whether or not a procedure has executed successfully is dependent on the application and is beyond the scope of this standard.

All status information obtained by the calling Application Entity Invocation as part of the invocation response is merged so that it may be accessed by the calling program.

6.8 Remote Procedure Call Back

During its execution, a remote procedure may execute a "callback" to a procedure in the calling Application Process Invocation. The invocation request and response for such a call are transmitted using the same Association as was used for the call to that remote procedure. The ROSE Linked Operations are used to perform the callback. The level of nesting of callbacks is not limited by this standard.

6.9 Context Handles

Some servers, or clients in the case of callbacks, maintain state between calls. Successive remote calls may access and modify this state information. An Application Process Invocation performs the activities of a server and/or a client. Therefore, the state of a remote procedure call is part of the state of an Application Process Invocation. As an Application Process Invocation may crash between remote procedure calls, remote procedure state is not necessarily maintained between successive calls.

An application indicates that it wishes state (context) to be maintained between successive calls by the use of a **RPCContextHandle**. This uniquely identifies state stored within a called procedure which belongs to a particular RPC Binding. Its value is maintained between calls and becomes undefined should the server or the client crash and the associated state be lost. The value of an **RPCContextHandle** is unique across space and time. When the RPC Binding is terminated, the state associated with the **RPCContextHandle** is deleted and the value of the **RPCContextHandle** becomes undefined.

The recovery of state when restarting a server or client is outside the scope of RPC.

6.10 Cancel

During the execution of a remote procedure, the RPC Service Provider of the calling procedure may receive a request for *orderly termination*; this feature is called CANCEL. The cancel request must be passed to the RPC Service Provider of the called procedure. As a consequence:

1) the called procedure may be terminated;

2) the cancel request may be masked temporarily or permanently; or

3) the cancel request may be handled and execution of the remote procedure continued.

The procedure return contains an indication of whether the cancel request was handled or ignored. As there may be more than one cancel request during a remote procedure call, the procedure return contains a count of the number of cancels which were processed.

At the time a calling procedure invokes a remote procedure call, it may be processing a cancel request or a cancel request may be pending. In this case, the procedure invocation must indicate that a cancel is pending. This cancel may be handled in the called procedure. When a called procedure has invoked another remote procedure and receives a cancel request, its local RPC Service Provider must forward this cancel request to the remote procedure.

7. INTERFACE DEFINITION NOTATION

An Interface Definition defines a set of procedures and their parameters. It is also used to determine the format and content of the data transmitted between the calling and called procedures, see 9.2 and 9.3.

In this standard the Interface Definition is presented in the form of an abstract language. This allows this standard to formally present the capabilities of the language while still permitting multiple different concrete programming languages to be developed for it. An abstract language specifies no spelling of keywords, no punctuation and no ordering of elements; it simply specifies the components of the language and grouping of components.

Appendix E contains an example of concrete syntax. An example of an interface definition using this syntax is in Appendix F.

7.1 Conventions used in the Abstract Language Definition

7.1.1 Terminals and Non Terminals

The abstract language is defined using a modified Backus-Naur Form (BNF). Each terminal and non-terminal of the abstract language is denoted by an alphabetic identifier, possibly including hyphens. A non-terminal is denoted using only hyphens and lower case letters. A terminal of the abstract language is said to be an "absolute terminal", if it refers to a terminal in the concrete language, that is a keyword, a separator, or a combination thereof. An absolute terminal is denoted using only hyphens and upper case letters. A terminal of the abstract language is said to be a "relative terminal" if it refers to a non-terminal in the concrete language (e.g. <Identifier>.) A relative terminal is denoted using hyphens and both upper and lower case letters, starting with an upper case letter and containing at east one lower case letter. Realisation of a relative terminal in a concrete language may also generate additional keywords and/or separators.

7.1.2 Optional Language Components

Portions of the language that are optional are denoted by enclosing them within square brackets ("[" and "]").

7.1.3 Alternatives

Alternatives (language components where one of several shall be chosen) are denoted by separating them with vertical bars ("|").

7.1.4 Repetition

Repetition is denoted by following the component that may be repeated with an ellipsis ("..."). If an item within square brackets is followed by an ellipsis, it indicates that the item may be repeated zero or more times. Any other item followed by an ellipsis may be repeated one or more times.

7.1.5 Grouping

When a set of components replaces a single component in a production, the set is enclosed in curly braces ("{" and "}").

7.1.6 Ordering

The order in which two or more components appear in a production is not significant. A concrete language may represent them in a different order.

7.2 Package

A package is used to define the remote service interface between the calling procedure and the called procedure. The package declaration consist of procedure declarations and related specification declarations.

```
<package> ::=                 <package-attributes> <package-
                              body>
```

```
<package-attributes> ::=          <name-attribute> <AE-Type-Title>
                                  <version-attribute>

<version-attribute> ::=           <VERSION-SPECIFIER> <Digit>...
```

The <version-attribute> provides a mechanism for co-ordinating versions of a package between the clients and servers.

```
<name-attribute> ::=              <Identifier>

<package-body> ::=                [<type-section>] <procedure-section>
                                  [<error-section>]

<type-section> ::=                <type-declaration> ...

<procedure-section> ::=           <procedure-declaration> ...

<error-section> ::=               <error-declaration> ...
```

The <error-section> specifies the errors that may be reported by the procedures of the package

```
<procedure-declaration> ::=       <procedure-identification>
                                  {<procedure-specification> | <defined-
                                  datatype>} <procedure-residence>
```

The <defined-datatype> specified in the <procedure-declaration> shall be a <procedure-datatype>.

```
<procedure-identification> ::=    <PROCEDURE-INDICATOR>
                                  <procedure-identifier>

<procedure-identifier> ::=        <Identifier>

<procedure-specification> ::=     [<procedure-parameter>] ...
                                  [<function-result>] [<callbacks>]
                                  [<errors-reported>]

<procedure-residence> ::=         {<CLIENT-SPECIFIER> | <SERVER-
                                  SPECIFIER>}

<procedure-parameter> ::=         <Identifier> <parameter-attributes>
                                  <parameter-datatype>

<parameter-attributes> ::=        <PARAMETER-ATTRIBUTE-
                                  SPECIFIER> <parameter-access>

<parameter-access> ::=            {<IN> | <INOUT> | <OUT>}

<parameter-datatype> ::=          <type-specification>

<function-result> ::=             <FUNCTION-INDICATOR> <simple-
                                  type-specification>
```

<callbacks> ::= <CALLBACK-INDICATOR> <procedure-identifier> ...

The <procedure-identifier>s are the names of procedures that this procedure will call..

<errors-reported> ::= <ERRORS-REPORTED-INDICATOR> <error-identifier> ...

An <errors-reported> identifies the errors which may be returned by a procedure.

<error-identifier> ::= <Identifier>

An <error-identifier> shall appear in an <error-declaration>.

7.3 Type Declarations

Type Declaration statements are used to define types which will be used in the declaration of the remote procedures and their parameters. Type declarations can be built using the provided built-in types and user-defined types.

<type-declaration> ::= <Identifier> <type-specification>

<type-specification> ::= { <primitive-datatype> | <constructed-datatype> | <defined-datatype> }

<simple-type-specification> ::= <primitive-datatype> | <defined-datatype>

<primitive-datatype> ::= { <integer-datatype> | <real-datatype> | <character-string-datatype> | <bit-string-datatype> | <boolean-datatype> | <enumerated-datatype> | <complex-datatype> | <numeric-string-datatype> | <context-handle-datatype> }

<constructed-datatype> ::= { <array-constructor> | <varying-string-constructor> | <record-constructor> | <discriminated-union-constructor> | <pointer-constructor> | <procedure-datatype> }

<defined-datatype> ::= <Identifier>

The <Identifier> in the definition of <defined-datatype> shall appear in a <type-declaration>.

7.3.1 RPC Primitive datatypes

This subclause describes the primitive datatypes defined by the Interface Definition Notation.

<integer-datatype> ::= <INTEGER-DATATYPE-SPECIFIER> <SIZE-INDICATOR> [<range-specification>]

<real-datatype> ::= <REAL-DATATYPE-SPECIFIER> <precision-specifier>

<character-string-datatype> ::= <CHARACTER-STRING-DATATYPE-SPECIFIER> <static-or-conformant-string-length>

<bit-string-datatype> ::= <BIT-STRING-DATATYPE-SPECIFIER> <static-or-conformant-string-length>

<static-or-conformant-string-length> ::= <string-length> | <UNSPECIFIED-STRING-BOUNDS-INDICATOR> | <string-related-variable-specification>

A static string is a string whose length is fixed in the Interface Definition.

A conformant string is a string whose length is determined by the calling procedure.

<string-related-variable-specification> ::= <Identifier>

If a string is a conformant string, many languages require that the string length be passed to the called procedure as a separate procedure parameter. The <string-related-variable-specification> identifies the parameter containing the string length. This procedure-parameter shall be an <integer-datatype>.

<boolean-datatype> ::= <BOOLEAN-DATATYPE-SPECIFIER>

<enumerated-datatype> ::= <ENUMERATED-DATATYPE-SPECIFIER> <enumeration-item>... <ENUM-SIZE-INDICATOR>

<enumeration-item> ::= <Identifier>

<complex-datatype> ::= <COMPLEX-DATATYPE-SPECIFIER> <precision-specifier>

<numeric-string-datatype> ::= <NUMERIC-STRING-DATATYPE-SPECIFIER> <string-length>

A numeric string is a string whose values shall conform to ISO 6093. There may also be application specific restrictions on the values passed.

<context-handle-datatype> ::= <CONTEXT-HANDLE-SPECIFIER> <string-length>

<string-length> specifies the length of a value of type context-handle.

A context handle is used to uniquely identify information stored within the called procedure which belongs to a particular RPC Binding. Its value is maintained between calls and becomes undefined should the server or the client crash.

<precision-specifier> ::=	<Digit> ... | <PRECISION-INDICATOR>
<string-length> ::=	<Digit> ...
<range-specification> ::=	<lower-bound-constant> <upper-bound-constant>
<lower-bound-constant> ::=	<bound-constant>
<upper-bound-constant> ::=	<bound-constant>

7.3.2 RPC Constructed Data Types

A type that is composed of more than one primitive type is called a constructed type. There are six sorts of constructed types: arrays, varying strings, records, discriminated unions, pointers, and procedures.

7.3.2.1 Specification of Arrays

The Interface Definition for an array includes bounds information for each dimension of the array followed by the definition of the type of the elements in the array.

Each dimension of an array has a lower bound and an upper bound. These bounds can be constant values, i.e. known at the time the Interface Definition is written, or variable values that are determined when the program is executed. Variable bounds may be limited by a minimum value for a variable lower bound and a maximum value for a variable upper bound.

The bounds of an array are of a type based on the <integer-datatype>.

There are two different classes of array:

- **Static**
  Static arrays are arrays whose bounds are fixed in the Interface Definition. Neither the calling nor the called procedure has the ability to change the bounds.
- **Conformant**
  Conformant arrays are arrays whose bounds are determined by the calling procedure. The called procedure must be able to handle arrays of arbitrary bounds.

<array-constructor> ::=	<ARRAY-SPECIFIER> <array-bound> ... <type-specification>
<array-bound> ::=	<lower-bound> <upper-bound>

<lower-bound> ::= <bound-constant> | <conformant-bound>

<upper-bound> ::= <bound-constant> | <conformant-bound>

<conformant-bound> ::= <UNSPECIFIED-BOUND-INDICATOR> | <array-related-variable-specification>

The <UNSPECIFIED-BOUND-INDICATOR> is used when a bound is known at the time of the procedure call; but is not a constant.

<array-related-variable-specification> ::= <Identifier>

If an array has conformant bounds, many languages require that the bounds information be passed to the called procedure as separate procedure parameters. The <array-related-variable-specification> identifies the parameter containing the bound. This <procedure-parameter> shall be an <integer-datatype>

<bound-constant> ::= <Digit> ...

7.3.2.2 Specification of Varying Strings

A varying string is a string whose elements are <character-string-datatype> or <bit-string-datatype> and which has both a maximum length and a current length. If the string has no elements the current length is zero.

<varying-string-constructor> ::= {<defined-datatype>| <character-string-datatype>| <bit-string-datatype>} <varying-string-specification>

<varying-string-specification> ::= <maximum-string-length> <VARYING-INDICATOR>

<maximum-string-length> ::= <Digit> ... |<UNSPECIFIED-STRING-BOUNDS-INDICATOR>

The <defined-datatype> shall be a <character-string-datatype> or a <bit-string-datatype>.

7.3.2.3 Specification of Records

A record constructor is used to define a new type that is a collection of fields of various types. The record type declaration specifies the identifier and type of all fields in the record.

<record-constructor> ::= <RECORD-SPECIFIER> <record-field-list>

<record-field-list> ::= <record-field> ...

<record-field> ::=	<Identifier> <type-specification> [<IGNORE>]

A <record-field> which is based on a conformant or varying string or array shall be the last <record-field> in the record.

The <IGNORE> attribute specifies that the field so marked shall not be transmitted to the remote procedure, and that it shall not be overwritten on return from the called procedure. The called procedure can make no assumptions about the initial state of such a field.

7.3.2.4 Specification of Discriminated Unions

A discriminated union is used to build a datatype with alternative representations of its data. A tag field appearing with the definition of an alternative indicates which alternative is being defined.

<discriminated-union-constructor> ::=	<DISCRIMINATED-UNION-SPECIFIER> <union-tag-specifier> [<union-name>] <union-body>
<union-tag-specifier> ::=	<union-tag-datatype> <Identifier>
<union-tag-datatype> ::=	<integer-datatype> | <character-datatype> | <boolean-datatype> | <enumerated-datatype>
<union-name> ::=	<Identifier>
<union-body> ::=	<union-case>...[<CASE-ELSE> [<record-field>]]
<union-case> ::=	<case-label>...[<record-field>]
<case-label> ::=	<Union-Tag-Constant>

The types of all <Union-Tag-Constant>s in a discriminated union shall be the same, and shall match the type specified in the <union-tag-specifier>. A particular <case-label> shall appear at most once in a discriminated union. The <record-field> shall not contain any <varying-string-constructor>s, conformant arrays, or strings with an <UNSPECIFIED-BOUNDS-INDICATOR>.

7.3.2.5 Specification of Pointers

A pointer is a datatype whose value is a reference to some other data value. Since such a reference is only valid in the address space of the calling(called) procedure, pointer variables are dereferenced and the values that they reference are transmitted. In the receiving procedure, the local reference to the transmitted value is placed in the pointer variable.

<pointer-constructor> ::=	<POINTER-SPECIFIER> <referenced-datatype>
<referenced-datatype> ::=	<type-specification>

A <procedure-parameter> shall not have a <parameter-access> of <INOUT> if a) it is of type <pointer-constructor> or b) of a type which contains a <pointer-constructor>.

7.3.2.6 Specification of Procedure Data Type

The procedure datatype is used to specify a class of procedures. A procedure datatype may be used to specify a procedure instance, or may be used to declare a parameter of another procedure datatype.

<procedure-datatype> ::=	<PROCEDURE-DATATYPE-SPECIFIER>

A parameter of type procedure can only be an <IN> parameter. The names of the procedures corresponding to its values shall appear in <callbacks> in the <procedure-specification> for every procedure to which it is passed as a parameter.

7.3.3 Specification of Errors

Declaration of error messages allows the messages for a package to be enumerated in a single place within the Interface Definition.

<error-declaration> ::=	<error-identification> [<error-specification>]
<error-identification> ::=	<ERROR-INDICATOR> <error-identifier>
<error-specification> ::=	<diagnostic> ...
<diagnostic> ::=	<DIAGNOSTIC-INDICATOR> <Diagnostic-Code> [<Diagnostic-Message>]

8. SERVICE DEFINITION

8.1 General

This service description uses the Service Definition Conventions established by ISO TR 8509.

In the tables specifying which parameters are present in the service primitives (2 through 8 below), the following notation is used per ISO TR 8509:

M	means that the parameter is "mandatory" (it will always be present in that service primitive);
U	means that the parameter is an "user option" (it might not be provided by the user in a particular instance of that service primitive);

C means that the parameter is "conditional" (i.e., it will always be present in that indication-type primitive if it was present in the corresponding request-type primitive);

O means that the parameter is a "provider option" (it might not be provided by the service provider in a particular instance of that service primitive);

(=) means that when the parameter is present (in a particular instance of that indication-type primitive), it will take the same value it had in the corresponding request-type primitive.

8.2 Service Primitives.

Service	Type
RPC-BIND	Local
RPC-UNBIND	Local
RPC-INVOKE	Unconfirmed
RPC-RESULT	Unconfirmed
RPC-ERROR	Unconfirmed
RPC-CANCEL	Unconfirmed
RPC-REJECT	Provider-initiated

Table 1 - RPC Service Primitives

Note that RPC-BIND and RPC-UNBIND may result in a protocol exchange.

8.3 RPC-BIND Service

Parameter name	Req	Cnf
Destination-Server	M	
Interface-Definition-Name	M	
Binding-Handle		M
Max-Concurrent-Invokes	U	
Status		M

Table 2 - RPC-BIND parameters

RPC-BIND establishes an RPC Binding between the invoking client AP-Invocation and the named server AP-Invocation for a named Interface Definition. The service produces a Binding-Handle which references the RPC Binding. Optionally, it allocates Associations for the RPC Binding; but the service primitive may complete before any Association is allocated. The policy for allocating Associations is implementation-dependent (see 9.1.1.)

Destination-Server: this shall uniquely identify the server to which the RPC Binding will be established. It shall include the AE-Title of the server and may include the AP-Invocation-Identifier of a particular invocation of the server.

Interface-Definition-Name:	this shall uniquely name the Interface Definition of the RPC Binding.
Binding-Handle:	returned to the client to identify the RPC Binding. The scope of the Binding-Handle is a globally unique identifier of the RPC Binding.
Max-Concurrent-Invokes:	specifies the maximum number of concurrent remote procedure calls for this RPC Binding.
Status:	shall contain RpcStatusInfo (see 9.6).

8.4 RPC-UNBIND Service

Parameter name	Req	Cnf
Binding-Handle	M	
Status		M

Table 3 - RPC-Unbind Parameters

This service terminates the named RPC Binding and invalidates the Binding-Handle. Outstanding RPC calls on the RPC Binding are lost. The permitted parameters are:

Binding-Handle:	shall be present and specifies the RPC Binding to terminate.
Status:	shall contain RpcStatusInfo information (see 9.6).

8.5 RPC-INVOKE Service

Parameter name	Req	Ind
Binding-Handle	M	M(=)
Operation-Value	M	M(=)
Cancel-Flag	M	M(=)
Argument	U	C(=)
Call-id	U	O

Table 4 - RPC-INVOKE parameters

This service is used to call a remote procedure using the specified RPC Binding. Note that this service may be supported by both synchronous and asynchronous programming interfaces. The permitted parameter values are:

Binding-Handle:	shall specify the RPC Binding to be used for the RPC call.
Operation-Value:	shall be the ASN.1 integer value corresponding to the order in which the called procedure is specified in the Interface Definition; 1 for the first procedure, 2 for the second procedure and so on.
Cancel-Flag:	shall specify if a cancel is pending.

Argument:	shall contain the input and input/output parameters, if any, specified in the Interface Definition for the corresponding remote procedure.
Call-id	local identifier of service primitive. In the case of callbacks, this shall have the same value as that in a previous RPC-INVOKE.ind to identify the invocation to which the callback applies.

8.6 RPC-RESULT Service

Parameter name	Req	Ind
Binding-Handle	M	M(=)
Result	U	C(=)
Cancel-Flag	M	M(=)
Cancel-Count	M	M(=)
Status	M	M(=)
Call-id	U	O

Table 5 - RPC-RESULT parameters

This service is used to return the input/output and output parameters at the end of a normal execution of the invoked remote procedure. The permitted parameter values are:

Binding-Handle:	shall specify the RPC Binding.
Cancel-Flag:	shall specify if a cancel is pending.
Cancel-Count:	shall specify how many RPC-CANCEL.ind have been completely handled when the RPC-Result.req primitive is issued.
Result:	shall contain the input/output and output parameters, if any, specified in the Interface Definition for the corresponding remote procedure.
Status:	shall contain RpcStatusInfo (see 9.6).
Call-id	local identifier of service primitive. Its value shall be the same as the corresponding RPC-INVOKE primitive.

8.7 RPC-ERROR Service

Parameter name	Req	Ind
Binding-Handle	M	M(=)
Cancel-Flag	M	M(=)
Cancel-Count	M	M(=)
Status	M	M
Call-id	U	O

Table 6 RPC-ERROR parameters

This service is used to indicate an error which occurred during the course of the RPC call. The permitted values of parameters are:

Binding-Handle:	shall specify the RPC Binding.
Cancel-Flag:	shall specify if a cancel is pending.
Cancel-Count:	shall specify how many RPC-CANCEL.ind have been completely handled when the RPC-Error.req primitive is issued.
Status:	shall contain RpcStatusInfo (see 9.6).
Call-id	local identifier of service primitive. Its value shall be the same as the corresponding RPC-INVOKE primitive.

8.8 RPC-CANCEL Service

Parameter name	Req	Ind
Binding-Handle	M	M(=)
Call-id	U	O

Table 7 RPC-CANCEL parameters

This service is used to cancel the most recent RPC-INVOKE.req issued by the RPC user. It shall not be issued when there is no outstanding remote procedure call. An RPC-CANCEL.ind shall not be delivered to an RPC user if a response to the cancelled remote procedure call has already been issued. An RPC-CANCEL.ind shall not be delivered to the peer RPC Service User before the corresponding RPC-INVOKE.ind. The permitted parameter values are:

Binding-Handle:	shall specify the RPC Binding for which the cancel is requested.
Call-id	local identifier of service primitive. Its value shall be the same as the corresponding remote procedure call to be cancelled.

8.9 RPC-REJECT Service

Parameter name	Ind
Binding-Handle	M
Status	M
Call-id	O

Table 8 RPC-REJECT parameters

This service may be issued by the RPC service provider to indicate that the underlying interconnection has failed. The permitted parameter values are:

Binding-Handle:	shall specify the RPC Binding on which the reject occurred.
Status:	shall contain RpcStatusInfo information (see 9.6).
Call-id	local identifier of service primitive. Its value shall be the same as the corresponding remote procedure call.

9. RPC PROTOCOL SPECIFICATION

This clause describes the relationship between the RPC Service (see 8) and the resulting protocol. This protocol is represented by a subset of the ROSE Services (ISO/IEC 9072-1) and OSI association management services (e.g. ISO 8649 ACSE or ISO/IEC 10026-2 Distributed Transaction Processing).

The BIND and UNBIND macros from the RO-notation in ISO/IEC 9072-1 are not used in this standard. Therefore, the ROSE requirement of being the sole user of an Association is weakened such that the RPC service provider has exclusive use of an Association only for the duration of a remote procedure call.

RPC-BIND establishes an RPC Binding between the invoking client and the named server. There may be several concurrent Associations between the client and server. The management of Associations within a RPC Binding is the responsibility of the RPC Service Provider.

The provision of concurrency within the RPC service necessitates the co-ordination of the (sequential) RO service of each association that is established for the support of an RPC Binding.

Clause 9.1 defines the mapping of RPC Service primitives onto a given RO service. The co-ordination of multiple associations are only informally defined in clause 9.1.

9.1 Mapping of the RPC Service

9.1.1 Mapping of RPC-BIND

The RPC Service Provider is responsible for selecting suitable Associations for use in the RPC Binding. These may be pre-existing Associations or may be established by the RPC Service Provider. This is a local matter.

9.1.1.1 Mapping to A-ASSOCIATE

The RPC Service Provider uses the AE-Title component of the Destination-Server parameter to determine the server's Presentation-Address, and this value is passed in the corresponding parameter of the A-ASSOCIATE.req. If the API-identifier and AEI-identifier are specified in the Destination-Server parameter, they are passed in the corresponding called parameters of the first A-ASSOCIATE.req. They shall be provided in all subsequent A-ASSOCIATE.req for this RPC. The application context for the association shall identify the Interface Definition available on the association.

Once an Association, either pre-existing or explicitly created, has been included in an RPC Binding, it cannot be used in any other RPC Binding until the first RPC Binding has terminated with the RPC-UNBIND service.

9.1.1.2 Other Mappings

It is possible to map RPC-Bind onto other existing and future OSI Connection Oriented services in the Application layer. However, these mappings are for further study.

9.1.2 Mapping of RPC-Unbind.

The RPC service provider releases those Associations it established for the RPC Binding using the appropriate service. Associations established prior to the RPC Binding are not released. However, RO service indications related to the RPC Binding on these Associations will be ignored.

9.1.3 Mapping of RPC-Invoke.Req

The RPC service provider determines if there is a unused Association allocated to RPC Binding. If not, a new Association is established as specified in 9.1.1. The RPC-INVOKE.req is mapped onto the RO-INVOKE.req.

The mapping of parameters from RPC-INVOKE.req to RO-INVOKE.req is as follows:

- **Binding-Handle** is mapped to Invoke-ID if the RPC-INVOKE.req is not a callback (see 8.3);
- **Operation-value** is mapped to Operation-value;
- **Cancel-Flag** and **Argument** are concatenated and mapped to Argument, which is structured according to the rules in 9.2;
- **Operation-class** shall have the value 2.
- **Priority** shall not be used.

If the RPC-INVOKE.req is a callback (as indicated by the call-id parameter with the same value as that in a previously received RPC-INVOKE.ind), the RPC Service Provider shall select a suitable Invoke-Id (as defined in ISO/IEC 9072) and pass the Invoke-Id of the parent operation in the Linked ID.

Operation Class 2 is used only to allow the Cancel operation to use the same Association as the corresponding Invoke.

9.1.4 Mapping of RO-INVOKE.ind

RO-INVOKE.ind is mapped onto RPC-INVOKE.ind or RPC-CANCEL.ind depending on the Operation-Value.

If the Operation Value is not CANCEL, then the RO-INVOKE.ind is mapped into RPC-INVOKE.ind. The Operation-Value parameter of the RO-INVOKE.ind is mapped to the corresponding parameter of the RPC-INVOKE.ind.

Using the Argument parameter from the RO-INVOKE.ind, the Cancel-Flag and Argument parameters are passed to the user in the corresponding parameters of the RPC-INVOKE.ind. The Invoke-ID is mapped to Binding-Handle. The RPC Service Provider records the Operation-Value and Linked-ID parameters of the RO-INVOKE.ind.

If the Operation-Value is CANCEL and the RPC Service User is still executing the remote procedure identified in the RO-INVOKE.ind, then the RO-INVOKE.ind is mapped onto RPC-CANCEL.ind. If the execution of the remote operation has completed (indicated by an RPC-RESULT.req or RPC-ERROR.req primitive) then the RO-INVOKE.ind is ignored.

If the RO-INVOKE.ind is linked to an operation which has been previously cancelled, then the RO-INVOKE.ind is ignored.

9.1.5 Mapping of RPC-RESULT.req

RPC-RESULT.req is mapped onto RO-RESULT.req on the same association as the corresponding RO-INVOKE.ind was received. The Binding-Handle parameter is mapped to Invoke-ID. The Cancel-Flag, Cancel-Count, Status and Result parameters of the RPC-RESULT.req are concatenated to form the result parameter of the RO-RESULT.req, structured as specified in 9.3. The Priority parameter of the RO-RESULT.req shall not be used.

If the Association was terminated before issuing the RPC-RESULT.req, then the RPC Service Provider shall issue a RPC-REJECT.ind with value "crash problem" to its RPC Service User.

9.1.6 Mapping of RO-RESULT.ind

The RO-RESULT.ind is mapped onto the RPC-RESULT.ind. The Invoke-ID is mapped to the Binding-Handle. Using the Result parameter from the RO-RESULT.ind, the Cancel-Flag, Cancel-Count, Status and Result parameters are passed to the user in the corresponding parameters of the RPC-RESULT.ind.

9.1.7 Mapping of RPC-ERROR.req

The RPC-ERROR.req is mapped onto the RO-ERROR.req on the same association as the corresponding RO-INVOKE.ind was received. The Binding-Handle parameter is mapped to the Invoke-ID. The Cancel-Flag, Cancel-Count and Status parameters of the RPC-ERROR.req are concatenated and mapped to the Error-Parameter parameter or the RO-ERROR.req. The Error-value parameter of the RO-ERROR-req shall be type INTEGER and have value 1. The priority parameter of the RO-ERROR.req primitive shall not be used.

If the Association was terminated before issuing the RPC-ERROR.req, then the RPC Service Provider shall issue a RPC-REJECT.ind with the status parameter having the value "crash problem" to the RPC Service User.

9.1.8 Mapping of RO-ERROR.ind

The RO-ERROR.ind is mapped onto the RPC-ERROR.ind. The Invoke-ID parameter of the RO-ERROR.ind is mapped to the Binding-Handle parameter of the RPC-ERROR.ind. Using the Error-Parameter from the RO-ERROR.ind, the Cancel-Flag, Cancel-Count and Status are passed to the user in the corresponding parameters of the RPC-ERROR.ind.

9.1.9 Mapping of RPC-CANCEL.req

RPC-CANCEL is mapped onto the RO-INVOKE.req service on the same association as the RPC call to be cancelled. The association on which to issue the cancel operation is identified by the call-id parameter.

The parameters of the RO-INVOKE.req service primitives are:

- **Invoke-Id**: any value not currently used for an RO-INVOKE.req;
- **Operation-Value**: shall have the value 0 (CANCEL Operation);
- **Argument**: shall have the value of the Invoke-Id of the operation to be cancelled;
- **Operation-Class** shall have the value 5;
- **Priority** shall not be used.

9.1.10 Mapping of RO-REJECT.ind

The RO-REJECT.ind is mapped onto the RPC-REJECT.ind. The INVOKE-ID is mapped to the Binding-Handle. The Returned-parameters parameter is ignored. The Status parameter shall be RpcStatusInfo as specified in 9.6.

9.2 The Argument Field for the ROIV APDU

This subclause uses ASN.1 to describe the ROIV APDU.

The argument field of the ROIV APDU shall be constructed as follows:

1) The argument field is a SequenceType.

2) The first element type in the SequenceType is a BooleanType corresponding to the Cancel-Flag.

3) One or more element types are included for each
   a) input parameter,
   b) input/output parameter,
   c) output parameter or function-result containing an array or string whose bounds are not all constants.

4) The types corresponding to a parameter are described in the order specified in the Interface Definition. The type of the function-result, if present, is described last.

5) A parameter which has been declared as a <primitive-datatype> with <parameter-access> equal to <IN> or <INOUT> is described using the following correspondences:

<integer-datatype>	IntegerType with optional ValueRange
<real-datatype>	RealType
<character-string-datatype>	GeneralString
<bit-string-datatype>	BitStringType
<boolean-datatype>	BooleanType
<enumerated-datatype>	EnumeratedType values shall be sequential starting at 0
<complex-datatype>	two consecutive RealTypes representing the real part followed by the imaginary part
<numeric-string-datatype>	ISO646String
<context-handle-datatype>	OctetStringType

6) When a parameter of the type <procedure-datatype> is an input parameter, it is described as an IntegerType. Note that the value of the Integer is the order in which its corresponding <procedure-declaration> appears in the <procedure-section>, starting at 1.

7) A parameter which is a record is described by describing the element types corresponding to each element in the record.

   A <record-field> which was specified in the Interface Definition with <IGNORE> is not described.

8) A parameter which is a discriminated union is described as a <union-tag-datatype> value followed by the appropriate <record-field> value, when present. If the value of the <union-tag-datatype> does not match a value in any case label, only the <union-tag-datatype> is described.

9) A parameter which is an input or input/output array is described as list of elements.

Case 1 - All array bounds are constant. No bounds information is described.

Case 2 - Not all array bounds are constants. The lower bound and upper bound for every dimension are described as IntegerType in the order in which they appear in the <array-constructor>.

The array elements are described as a SequenceOfType where type is the type of the array's elements. If the array's elements are of type <record-constructor>, the array is described as a SequenceOfType whose type is SequenceType and where the types of the SequenceType are the element types corresponding to each element in the record. The elements of the array are described in row major order.

10) Programming language considerations require that bounds information for output arrays whose bounds are not all constants be described to the called procedure when it is invoked. This information is described in the same way as bounds information is described in Case 2 above.

11) A parameter which is an input or input/output varying string is described by describing its maximum string length as an IntegerType followed by the string. The maximum string length is described for output varying strings.

12) A parameter which is a pointer is described by describing the element to which it points. A null pointer is described as an element of the indicated type with a length field of 0. Dereferencing of tree structures shall be depth first. Where the reference type is a <record-constructor> containing more than one <pointer-constructor>, the first <pointer-constructor> will be dereferenced first. Dereferencing a pointer shall not cause any aliasing or infinite recursion.

9.3 The Result Field for the RORS APDU

This subclause uses ASN.1 to describe the RORS APDU.

The result of the RORS APDU shall be constructed as follows:

1) The result is SequenceType.

2) The first element type in the SequenceType is a BooleanType corresponding to Cancel-Flag.

3) The second element type in the SequenceType is an IntegerType corresponding to Cancel-Count.

4) The third element Type in the SequenceType is RpcStatusInfo (see 9.6)

5) One or more element types are included for each input/output parameter, each output parameter, and function-result.The type of the function-result, if present, is described last.

6) The types corresponding to a parameter are described in the order specified in the Interface Definition.

7) A parameter whose type is a <primitive-datatype> is described as specified in 9.2 item 5.

8) A parameter which is a record is described by describing the types corresponding to each element of the record.

9) A parameter which is a discriminated union is described according to the rules in 9.2 item 8.

10) A parameter which is an input/output or output array is described according to the rules in 9.2 item 9.

11) A parameter which is an input/output or output varying string is described as a GeneralString or a BitStringType. Note that the called procedure does not change the value corresponding to maximum string length.

12) A parameter which is a pointer is described according to the rules in 9.2 item 12.

13) In the case of an "abnormal" RpcStatusInfo, parameters may be replaced by the NullType.

9.4 Error

This subclause uses ASN.1 to describe ROER APDU.

The Error-Parameter of the ROER APDU shall be constructed as follows:

1) The Error-Parameter is SequenceType.

2) The first element type in the SequenceType is a BooleanType corresponding to Cancel-Flag

3) The second element type in the SequenceType is an IntegerType corresponding to Cancel-Count.

4) The third element Type in the SequenceType is RpcStatusInfo (see 9.6).

9.5 Reject Reason

The Reject-Reason parameter of the RORJ APDU shall consist of RpcStatusInfo

9.6 RPC Status Reporting

The status of an RPC invocation is reported by RPCStatusInfo. This type is defined in RPC-ErrorManagement.

```
RPC-ErrorManagement DEFINITIONS -- version 1 -- ::=
    BEGIN

      EXPORTS
        RpcStatusInfo, RpcError, RpcDiagnosticMessage,
                RpcDiagnosticCode
------------------------------------------------------------------------------------------------

  -- First level definitions:

  RpcStatusInfo  ::=
          SEQUENCE { RpcStatus,
                     RpcError OPTIONAL }

    RpcError      ::= SEQUENCE {RpcDiagnosticCode,
                     RpcDiagnosticMessage OPTIONAL}

-- RpcStatus distinguishes the first level of granularity within which the RpcDiagnostics provide
-- fine grained distinctions. RpcDiagnosticCode is intended for interpretation by programs.
-- RpcDiagnosticMessage should be a natural language equivalent of the diagnostic code, and is
-- intended for human readability.

------------------------------------------------------------------------------------------------

  -- Second level definitions:

    RpcStatus ::= ENUMERATED {

-- Positive values are used for status resulting directly from execution of the procedure. All the
-- other values (negative) report environmental error conditions arising from the call being to a
-- remote procedure.

          normal     (0),

-- this is the expected outcome. The remote procedure has executed exactly once, and has
-- returned reporting success.

          warning    (1),

-- The remote procedure has executed exactly once, has produced all the expected output values
-- and has returned; but the output values might not be what the user expected.For example, a
-- floating point underflow has occurred and a result has been set to zero.

          abnormal (2),

-- The remote procedure has executed exactly once, produced results and has returned; but an
-- error has occurred during the execution. The output produced is not all correct. To avoid
-- marshalling errors, incorrect output values have been set to the NULL type (or otherwise
-- modified).
```

error (3),

-- The remote procedure has executed exactly once, but is unable to complete and return output
-- values (e.g. involuntary termination due to an arithmetic overflow).

-- Environmental Error conditions:

-- In the following text the term "executes at most once" means that the RPC Service guarantees
-- only that the remote procedure will not execute more than once. Execution of it has or will
-- occur once, or never, and may be complete or incomplete.

rOSEGeneralProblem (-1),

-- Corruption of Remote Operations protocol control information has been detected at the
-- Remote Operations Service. The details are defined in ISO/IEC 9072/1. The remote procedure
-- executes at most once.

rOSEInvokeProblem (-2),

-- An unacceptable invocation has been detected at the Remote Operations Service, locally or
-- remotely (in the latter case it may be a protocol error). The details are defined in ISO/IEC
9072/1.
-- The remote procedure does not execute.

rOSEReturnResultProblem (-3),

-- An unacceptable RO-RESULT APDU has been detected as a Remote Operations protocol
-- error at the Remote Operations Service. The details are defined in ISO/IEC 9072/1. The remote
-- procedure executes at most once.

rOSEReturnErrorProblem (-4),

-- An unacceptable RO-ERROR APDU has been detected as a Remote Operations protocol error
-- at the Remote Operations Service. The details are defined in ISO/IEC 9072/1. The remote
procedure
-- executes at most once.

interconnectionProblem (-5),

-- The underlying interconnection is not available. This is detected by Connection management.
-- The remote procedure executes at most once.

crashProblem (-6),

-- The called procedure (and/or the environment in which it executes) has crashed. This has been
-- detected and reported by some implementation specific means, independently of the means
-- inherent in the other status values. The remote procedure does not execute.

invalidContextHandle (-7),

```
-- A context handle passed to the server or client during callback is invalid because the server
-- system has crashed, lost state, and restarted. The remote procedure does not execute.

                procedureCancelled        (-8),

-- The invoked remote procedure call has been cancelled.

                invalidBindingHandle      (-9) }

-- An RPC Binding handle passed to the RPC-Bind service is invalid.

          RpcDiagnosticCode ::= INTEGER

-- An optional value which gives the definitive details of what happened to the remote procedure
-- call. The context within which it is interpreted is implicit in the RpcStatus value, as follows:

   -- If RPC status is normal the RpcDiagnosticCode shall not be included.

   -- If RpcStatus is one of warning, abnormal or error: the interpretation of the
   -- RpcDiagnosticCode is either defined in the Interface Definition (by Importing RpcError into
   -- the Interface Definition), or not (in which case it is implementation-specific).

   -- If RpcStatus is one of: rOSEGeneralProblem, rOSEInvokeProblem,
   -- rOSEReturnResultsProblem or rOSEReturnErrorProblem: the interpretation of the
   -- RpcDiagnosticCode shall be the ROSE problem types defined in ISO/IEC 9072/1 and 9072/2.

-- interconnectionProblem: the interpretation of the RpcDiagnosticCode shall be the rejection and
-- abort types defined in ISO 8649.

-- crashProblem: the interpretation of the RpcDiagnosticCode shall be implementation-specific
-- and is not defined in this standard.

             RpcDiagnosticMessage ::= GeneralString

-- An optional character string, which describes in natural language the reason for the returned
-- status. When the RpcStatus value is normal this string (if present) shall start with "Normal
-- Result" (or its equivalent where the language used is not English), followed optionally by
-- additional information.

-- RpcStatus value is invalid Context Handle, this string (if present) shall contain "Invalid Context
-- Handle (or its equivalent where the language used is not English).

          END -- end of RPC Error Management definitions.
```

10. STATE TABLES

These state tables are only concerned with RPC over a single Association. Management of Associations belonging to an RPC Binding is outside the scope of this standard. However, RPC specific rules for Association management are given in 9.1.

10.1 Inbound Events

Ireq	RO-INVOKE.req
Rreq	RO-RESULT.req
Ereq	RO-ERROR.req
Creq	RO-CANCEL.req with operation-value = 0 (CANCEL)
ROIV	RO-INVOKE APDU
RORS	RO-RESULT APDU
RORE	RO-ERROR APDU
RORJ	RO-REJECT APDU
ROIVC	RO-INVOKE APDU for CANCEL operation

10.2 Outbound Events

ROIV	RO-INVOKE APDU
RORS	RO-RESULT APDU
RORE	RO-ERROR APDU
RORJ	RO-REJECT APDU
ROIVC	RO-INVOKE APDU containing the CANCEL operation
Iind	RO-INVOKE.ind
Rind	RO-RESULT.ind
Eind	RO-ERROR.ind
Cind	RO-CANCEL.ind with operation-value = 0 (CANCEL)
RJind	RPC-REJECT.ind

10.3 States

NoCall	Client bound to a server. No call in progress.
AwaitResult	Issued RPC-INVOKE.req and awaiting response.
IssuedCancel	Issued RPC-CANCEL.req and awaiting response.
ExecutingCall	Executing a remote procedure.
ReceivedCancel	Received RPC-CANCEL.ind
Exported	Server has announced its availability.
CancelCall	Collision of Cancels detected.

10.4 Variables

CB	type Integer Count of depth of nested callbacks.

10.5 Actions

[1]	Set CB = 0
[2]	Set CB = CB + 1
[3]	If CB > 0 set CB = CB - 1
[4]	If CB > 0 set CB = CB - 2

10.6 Predicates

p1	Callback active. CB $\geq$ 1.
p2	Cancel is for a child operation.
p3	Operation to be cancelled is valid. Invoke with same Call-Id.
p4	Callback nesting at least two deep. CB $\geq$ 2.

10.7 State Tables

Tables 9, 10 and 11 refer to the events concerning a single association.

	Nocall [1]	Exported [2]
ROIV		Iind ExecutingCall [1]
ROIVC	NoCall	Exported
Ireq	ROIV AwaitResult RJind NoCall	

1) NoCall is the initial state for the client end of an association.

2) Exported is the initial state for the server end of an association.

Table 9 - Initial State Table

	CancelCall	AwaitResult	Issued Cancel	ExecutingCall	ReceiveCancel
Ireq				ROIV [2] AwaitResult	
Creq		p3 ROIVC IssuedCancel	p3 ROIVC IssuedCancel		
Rreq	p4 RORS [4] AwaitResult ^p4 RORS [1] Exported			p1 RORS [3] AwaitResult ^p1 RORS [1] Exported	p1 RORS [3] AwaitResult ^p2 RORS [1] Exported
Ereq	p4 RORE [4] AwaitResult ^p4 RORE [1] Exported			p1 RORE [3] AwaitResult ^p1 RORE [1] Exported	p1 RORE [3] AwaitResult ^p2 RORE [1] Exported
EJreq	p4 RORJ [4] AwaitResult ^p4 RORJ [1] Exported			p1 RORJ [3] AwaitResult ^p1 RORJ [1] Exported	p1 RORJ [3] AwaitResult ^p2 RORJ [1] Exported

Table 10 - Request State Tables

	CancelCall	AwaitResult	IssuedCancel	ExecutingCall	ReceiveCancel
RORS		p1 Rind [3] ExecutingCall ^p1 Rind [1] NoCall	p1 Rind [3] ExecutingCall ^p1 Rind [1] NoCall		
RORE		p1 Eind [3] ExecutingCall ^p1 Eind [1] NoCall	p1 Eind [3] ExecutingCall ^p1 Eind [1] NoCall		
ROIV		Iind [2] ExecutingCall	IssuedCancel		
ROIVC	p2 IssuedCancel ^p2 Cind CancelCall	p1 Cind ReceiveCancel	p2 IssuedCancel ^p2 Cind CancelCall	Cind[1] ReceiveCancel	Cind[1] ReceiveCancel
RORJ		p1 RJind [3] ExecutingCall ^p1 RJind [1] NoCall	p1 RJind [3] ExecutingCall ^p1 RJind [1] NoCall		

1 *During the execution of a remote procedure, it may invoke another remote procedure. Thus one association A to an Application Process Invocation will be in state ExecutingCall and AwaitResult state on another association B. In this case, receiving a ROIVC on association A will have the same effect on association B as a Creq; i.e., the ROIVC will be forwarded on Association B and the state of the Application Process Invocation on association B will become IssuedCancel.*

Table 11 - PDU Receiving State Tables

APPENDICES

APPENDIX A
PROCEDURE CALL TUTORIAL

(This Appendix is not an integral part of the standard)

A.1. INTRODUCTION

As a preliminary step towards understanding procedure call which are remote, this tutorial explains local procedure calls characteristics. It assumes only a basic knowledge of programming techniques. This Appendix is intended to be complete in itself, therefore it repeats some information that is provided elsewhere in the standard.

The references to standards publications are in 1.4. Other references are in Appendix C.

A.2. GENERAL STRUCTURE

We are concerned here with the general language structure and execution structure of procedure calls. Language-specific details are avoided.

- **Inter-module communication**. The procedure call mechanism exists as a means of orderly communication between logically separate procedures of a sequential computer program. The participants in a procedure call are referred to here as the calling procedure and the called procedure.
- **Procedure structure**. A procedure is a closed sequence of instructions that is entered from and returns control to an external source. It consists essentially of a procedure definition and a procedure body which is terminated by some kind of return statement.
- **Referencing**. The called procedure is referenced by its procedure name. This value is necessarily unique according to the naming scope rules of the programming language concerned. The name of the calling procedure is not visible to the called procedure.
- **Parameter passing**. Information is communicated between the procedures by parameters which are passed by the call, as explained in A.4.
- **Procedure definition**. The identity and parameters of a procedure are defined in its procedure definition. In most languages this definition is at the beginning of the procedure itself.
- **Sequential execution**. Procedure call execution has a request/response structure with exact synchronisation which is fundamental to the semantics of procedure calls. It is explained more fully in A.3.
- **Nesting**. A called procedure may itself make procedure calls to other procedures, which may call others, etc. This calling may be nested to arbitrary depth. Languages have scoping rules which restrict what can be called.

Figure A.1 (next page) illustrates an example of a modular program with procedure call structure.

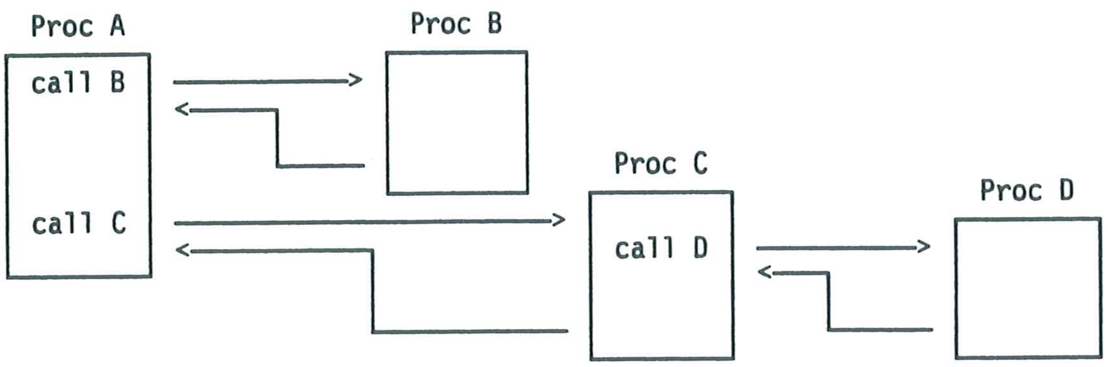

Figure A.1 - A modular program, structured into separate procedures integrated via procedure calls

A more complex example might include recursive calls in which a called procedure makes procedure calls to itself, to its caller, etc.

A.3. COMPUTATIONAL MODEL

The computational model of procedure calls is now explained in more detail.

With respect to any particular procedure call there is one thread of execution control, as illustrated in Figure A.2 (a). The execution passes from the calling procedure to the called procedure, and then back again. The execution of the calling procedure resumes only after the called procedure has been executed.

At the called procedure the thread of execution may itself make further (nested) procedure calls in order to accomplish its purpose. This same procedure then has "calling" and "called" roles with respect to different procedure calls. E.g. the procedure x in Figure A.2 (b) is the called procedure with respect to the procedure call from the first procedure, and the calling procedure with respect to the procedure call to the third procedure.

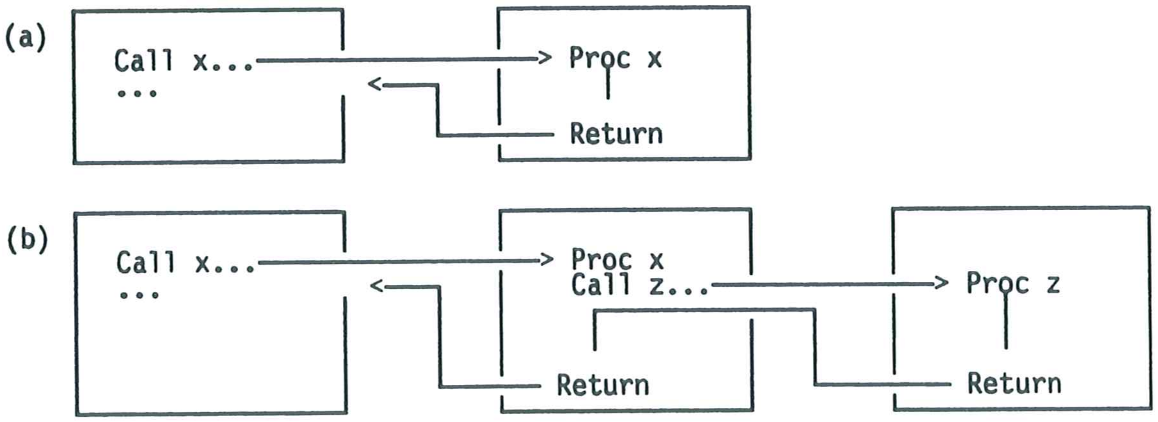

Figure A.2 - Procedure execution

The existence of procedure call linkage does not, of itself, extract specific obligations from the called procedure. The called procedure has general obligations inherent in being accessible to such linkage; but it does not know what linkage actually exists or is in active use until a procedure call uses it; nor is it obliged to remember this relationship outside the duration of the procedure call. The linkage only provides repeatable access to a predetermined destination (the called procedure).

The calling and called procedures should achieve any necessary continuity between successive procedure calls via the effects of the procedure calls themselves. This is done by the procedure calls referencing something that persists throughout this time (e.g. a reference "handle" value, or a transaction identifier). This kind of binding is rather different from that familiar to OSI experts (i.e. ACSE Associations).

A.4. PARAMETER STRUCTURE

Procedure calls exchange information explicitly via parameters. But in most languages there can also be hidden information flow between procedures via shared information which is accessible within a scope that is global to the procedures.

- **Parameter passing semantics.** The called procedure needs to be able to locate values of the parameters specified in the calling statement. This is usually done by passing the addresses of the parameters to the called procedure, and is termed "call by reference". Some languages, e.g. PASCAL, may pass values directly, and this is termed "call by value". A few languages use a further mechanism, termed "call by name". See [Gries]
- **Parameter quantity.** The procedure definition of a procedure fixes the number of parameters that can be passed to it.
- **Formal parameters.** The names, types and sequence of parameters are defined in the procedure definition. These are termed the "formal parameters" of the procedure.
- **Actual parameters.** The procedure call statement in the calling procedure identifies the locally accessible information corresponding to the formal parameters. These are termed the "actual parameters" of the procedure call.
- **Parameter direction.** For each parameter there is a defined direction of information flow. A parameter which passes information into the called procedure is referred to here as an "input" parameter; vice versa an "output" parameter; or both directions "input/output".

These parameter passing concepts are summarised in Figure A.3.

```
 The Calling procedure
with the actual parameters
```

```
...
Call B (IN x,y,z): OUT p
...
...
```

```
 The Called procedure
with the formal parameters
```

```
Proc B
(IN m, n, q): OUT s
Begin
...
End
```

```
Mapping used for these parameter values in program execution:
    x ──> parameter 1 ──> m    y ──> parameter 2 ──> n
    z ──> parameter 3 ──> q    p <── parameter 4 <── s
```

Figure A.3 - Procedure Call parameter passing

In most languages the programmer is left to define output parameters for status information and error diagnostics. This is explained more fully in A.6. Some languages, such as Mesa [Mitchell], include comprehensive exception handling in which each procedure call nominates exception procedures to handle defined error conditions.

A.5. IMPLEMENTATION CONSIDERATIONS

- **Type checking**. Compilers (and run time interpreters) may include type checking to ensure that the types of the actual parameters of the calling procedure are consistent with the types of the formal parameters in the procedure definition of the called procedure. Some languages guarantee that all procedure call parameters are "type complete" and "type safe"; e.g. Algol 68 [Wijngarden].
- **Linking**. The object-code of the calling and called procedures may be constructed independently. Tying them together is termed linking. This is the process of binding the object-code to the addresses of the relevant procedures, parameters and variables. Depending on the language system and the environment, the linkage may be pre-configured (static or early binding), or is determined when or after the software is loaded for execution (dynamic or late binding).
- **Mixed languages**. In some execution environments it is possible for procedure call linkage to exist between modules with source code in different languages, compiled to common target conventions.
- **IRDS**. Master copies of procedure definitions may be stored in a data-dictionary (e.g. an Information Resources Dictionary System - IRDS). The definitions are then maintained there separately from the program procedures.

A.6. ERROR MANAGEMENT

When a procedure executes, various things can happen to affect the output produced by it. The classification below is used in the main body of the standard.

(a) **Normal**. The procedure's execution is successful, all the values are computed as expected.

(b) **Warning**. The procedure has produced output; but detected a situation that should be brought to the caller's attention. For example, the procedure had to change the value of an input parameter, ignored an input parameter in order to complete execution, set a result to zero after floating point underflow, encountered the end-of-file condition, etc.

(c) **Abnormal**. The procedure detects a severe error which prevents successful completion of the computation. Part of the output may have been computed; but the termination is orderly, in that the procedure is able to indicate that not all the output is present.

(d) **Error**. The procedure is unable to complete execution and return output values. E.g. it detects an error in the input parameters and terminates abruptly without computing any results; or it is abnormally terminated by an execution problem such as arithmetic overflow. In such cases it is often impossible to transfer output parameters correctly, because output values will not exist, and whatever occupies their storage may not have the correct representation for the output's type.

The first three situations should be handled by returning a status value in the output parameters, along with any computed values. The fourth case (d) should be handled by returning a status value which indicates that parameters values are undefined. Typically this kind of situation (d) is detected and reported by the language execution system rather than the called procedure itself.

In some implementations, the occurrence of problems such as (c) and (d) may lead to the calling procedure being aborted without it being aware of them.

A.7. SUMMARY

Local procedure call characteristics of particular relevance are:

- **Asymmetric**. A procedure call interaction has a strict request/response discipline, and occurs at the initiative of the calling procedure.
- **Synchronous**. Execution is synchronous. Execution of the calling procedure waits until the return from the called procedure.
- **Restricted Parameters**. The information explicitly communicated via procedure calls is in parameters of defined quantity, sequence and type.
- **Variability**. Some details of procedure call structure are different in different languages and execution environments; but there is a core that is common to nearly all.
- **Software engineering**. Languages and software development processes may include highly developed structure to support procedure calls and related software modularity.

These procedure call characteristics of programming languages are the principal determinants of RPC structure.

APPENDIX B
RPC TUTORIAL

(This Appendix is not an integral part of the standard)

B.1. INTRODUCTION

This tutorial explains Remote Procedure Call (RPC) concepts. It assumes a basic understanding of procedure call structure as explained in Appendix A, and of current OSI standardisation. It assumes no prior knowledge of RPC. This Appendix is intended to be complete in itself; therefore it repeats some information that is provided elsewhere in the standard.

The references to standards publications are in 1.4. Other references are in Appendix C.

As explained in [Birrell 84], the idea of remote procedure calls is simple. It is based on the observation that procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single computer. Therefore, it is proposed that this same mechanism be extended to provide for transfer of control and data across communication networks.

When a remote procedure is invoked, execution of the calling procedure is suspended, the parameters are passed across the network to the remote environment, and the called procedure is executed there. When the called procedure terminates and produces its output, this is passed back to the calling environment, where execution of the calling procedure resumes as if returning from a single-machine call. While the calling procedure is suspended, other processes on that machine may still execute (depending on the details of the parallelism of that environment and the RPC implementation).

There are many attractive aspects to this idea. One is clean and simple semantics: these should make it easier to build distributed computations, and to get them right. Another is efficiency: procedure calls seem simple enough for the communication to be quite rapid. A third is generality: procedure call is already the most important mechanism for communication between the logically separate parts of software systems. A more general point is that an RPC approach helps to assure that applications investment by users is network-independent.

RPC concepts first became generally visible in 1976 [White], and were integrated into a proprietary networking architecture in 1981 [Xerox].

Since the early 1980s RPC techniques have been thoroughly evaluated and reported by the research community; e.g. in [Birrell 84], [Birrell 85], [Gibbons], [Hamilton], [Jones], [Nelson], [Panzieri]. Their general conclusion is that RPC is a vital ingredient of distributed interactive processing.

Since the mid 1980s proprietary RPC facilities have become available on various operating systems, e.g. on Unix [Sun] and on PC-DOS [IBM].

B.2. APPLICATION SYSTEMS

B.2.1 RPC Application Characteristics

RPC is oriented to distributed applications in which there is interactive communication, program to program, with short response times and relatively small amounts of data transfer.

A remote component of an application is usually modelled as a named and complete service. Where access is via RPC, the primitives of the service are implemented as a corresponding family of remote procedures. This family of procedures representing the service is a remote program, as illustrated in Figure B.1.

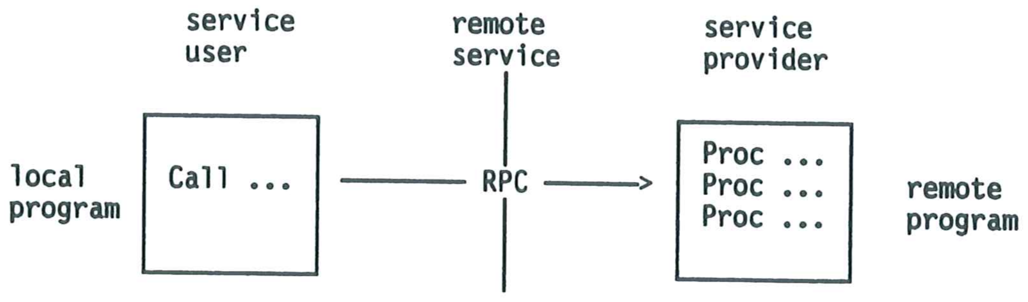

Figure B.1 - General structure

The remote service/program accessed via RPC may be application-specific (e.g. part of some financial application), or a generic shared resource (e.g. a file server).

B.2.2 RPC Remote Interface Definitions

The remote interactions are defined in an *interface definition*. This is essentially a set of (abstract) procedure definitions of the remote service/program, combined with other relevant specification information. It defines the remote interface between the user and provider of the remote service. See Figure B.2.

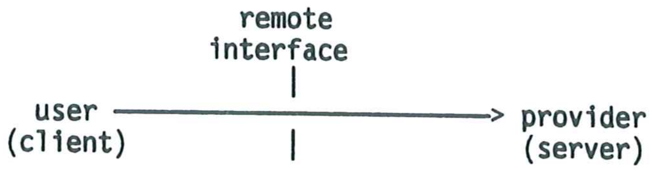

Figure B.2 - A Remote Interface

At this level of abstraction the parties to request/response interactions are usually termed *client* and *server*, as in ISO/IEC 10031-1. Use of client/server/service terminology is not particular to RPC.

As explained in B.3.4, the interface definition should be expressed in an *interface specification language* which allows heterogeneous implementations of the programs which provide and use the declared interface.

An interface specification language defines and enforce generic rules which would restrict call structure and parameter structure in ways generally applicable to heterogeneous RPC.

B.2.3 RPC Bindings

A server declares information about the services (procedures) it offers, and a client (directly or indirectly) makes use of such information about the services which it intends to use. This information is in terms of interface definitions, service names, qualitative controls, etc. Typically such information is stored in and retrieved from data-dictionaries (IRDS) and directories in ways general to all distributed processing (and not specific to RPC).

The term *RPC Binding* is used here for the access linkage which enables the client to access the server. It has essentially the same properties as local procedure linkage.

B.3. RPC Considerations

B.3.1 Remoteness

We are concerned here with direct implications of the procedures being in physically separate computers linked by telecommunications.

- **Naming scope**. The naming scope of procedure calls is usually restricted to procedures in the same machine. Some extra naming provisions are needed to call external procedures.
- **Binding and loading**. In most languages the binding between procedures is implicit in the program structure. It occurs as a normal consequence of program composition and compilation. Remoteness implies some need for explicit binding action under program control at run time. Specific provisions may also be needed to load the remote procedure when it is needed. This is analogous to the way programs bind to files at run time and cause external magnetic media volumes to be mounted and dismounted.
- **No shared memory**. Local procedure calls depend, implicitly or explicitly, on the use of shared variables which can be accessed by the calling and called procedures. By definition, RPCs have no shared memory: the two procedures are usually in physically separate computers. The general conclusion for RPC is that parameter values must be copied from machine to machine (even where the normal compiler generated object code for a corresponding local call would be call by reference). The ideal parameter passing semantics for RPC are therefore call by value.
- **Communications**. The underlying support system necessarily uses communications as part of the mechanization of the remote procedure calls. But this is not directly visible in the procedure definition, nor to the procedures using RPC.
- **Error Management**. Remoteness also introduces different error possibilities, as explained in B.3.3.
- **Degree of remoteness**. Much of the early experience with remote procedure calls was with local area networks. RPC implementation experience with wide area networks confirms that RPC techniques scale

up to arbitrary degrees of remoteness. The ability to scale down to low degrees of remoteness (e.g. RPC across a backplane bus) is inherent in the derivation of RPC from local procedure call techniques.

- **Security**. Remoteness also has many security implications.

A preliminary conclusion to be drawn from the above is that a remote procedure call cannot be exactly like a local procedure call. This is confirmed by the other factors now considered.

B.3.2 Distribution Transparency

A major system and application design issue is whether or not to hide distributedness and its consequences. The term *distribution transparency* is used here for discussing the visibility of distributedness within distributed systems.

- **Arguments for transparency**. It can be advantageous if all the consequences of distribution are made transparent (i.e. invisible). This hides complexity, simplifies the task of application designers, and enhances the re-usability of application code. The evolution of existing products based on centralised systems is then inherently straightforward. A successful experiment with such transparency for procedure calls is Unix United [Brownbridge].

- **Arguments against transparency**. Full transparency, which completely conceals distribution, can be relatively expensive in terms of underlying implementation effort and performance overheads. Moreover, it denies designers the opportunity to exploit the consequences of distribution via decentralisation and replication of control, or data, or both.

System design choices lead to different transparency requirements, and full distribution transparency is not always necessary. Therefore standards should not pre-empt these choices.

In this Basic RPC standard the visibility of distributedness at the user programmer interface is confined to particular parameters of the RPC service primitives. The user program can choose to ignore these.

B.3.3 Reliability

We are concerned here with the reliability implications of the calling procedure and called procedure being in physically separate machines.

- **Independent failures**. With a local procedure call there is one process (i.e. one execution context, one thread of execution) which either crashes or survives. But for a remote procedure call the process in one machine may crash while that at the other remains intact. This can lead to situations such as: remotely called procedures that are "orphaned" by calling procedure crashes; calling procedures which become "bereaved parents" waiting for replies from remotely called procedures that have crashed; and recovery situations in which the caller is uncertain whether the remote procedure has been executed (e.g. when network partitioning occurs during a remote interaction).

- **Communications failures.** The underlying communications system should hide the occurrence of communications failures, but cannot hide any prolonged inability to communicate (network partitioning).
- **Compatibility.** The physical separation of procedures emphasises the compatibility and version control problems latent in all modularity.
- **Error management.** Error possibilities particular to remote and heterogeneous interactions require special error management. See B.4.2.

It should be appreciated that the most difficult reliability issues arise not from communications failures, but from host overloads, host crashes and remote application failures. Perfect communications would not remove these problems, therefore the solution is not just a matter of using reliable communications connections etc. See [Saltzer].

An RPC system should have appropriate execution reliability semantics. This subject is explained in [Panzieri]. The following choices of guarantees may occur: remote execution occurs *exactly once*; remote execution occurs *at most once* (including possibilities of no execution and incomplete execution); remote execution occurs *at least once* (including possibilities of multiple complete and incomplete executions).

The exactly once semantics are the most difficult to guarantee, and the at least once semantics are the easiest. The degree of difficulty affects the amount of RPC protocol etc. needed to provide the guarantees.

The remote execution reliability guarantees inherent in the OSI Remote Operations standards (ISO/IEC 9072/1 and 9072/2) are exactly once for interactions returning to the calling procedure normally without error, and at most once for all other outcomes.

B.3.4 Heterogeneity

Much of the experience with using RPC has been in homogeneous environments; i.e. both ends use the same language and operating system.

An RPC system for OSI standards purposes must allow for heterogeneous environments: there will typically be different languages and different operating systems at each end.

These heterogeneity problems are now well understood after several years of practical experience with RPC in heterogeneous environments, e.g. Matchmaker [Jones] and HRPC [Bershad]. Powerful software tools systems have been developed to support RPC in heterogeneous environments, e.g. the stub generator in [Gibbons].

Some of the problems inherent in this heterogeneity are:

- **Semantics.** In different languages there are differences of semantics for what might seem to be the same data structures.
- **Language characteristics.** The data structures and procedure call structures supported in any one programming language are not all supported in exactly the same way in all other languages.

- **Concrete syntax**. Different machines use different conventions for the bit representation of data values, e.g. different byte ordering, size limits and floating point formats.

The basis for solving these problems is to specify the remote interactions in a canonical form which is independent of such variability. This requires use of an *interface specification language* and an *external data representation* which are independent of the choices of programming languages, operating systems, computers and networks, as fully explained in [Gibbons]. The next step is to define a language binding for each programming language of interest. This defines the mapping of the semantics and syntax of the programming language onto the RPC control structure and the external data representation structure.

B.4. RPC Structure

B.4.1 Call Structure

For the reasons discussed in B.3, remote calls via a heterogeneous RPC system need to be in some respects different from local procedure calls. However, they also need to be well integrated into the local language, software tools system and execution environment. The main points are:

- **Language structure**. The control structure and syntax of remote calls must be acceptable in the local language environment. Typically remote procedure calls will be implemented via a normal procedure call in the host language.
- **Call destination**. The immediate destination of the call is likely to be a local RPC support procedure (typically an "RPC stub procedure") which handles the remote interaction. The ultimate destination is a particular procedure in a particular server which is remote from the calling procedure.
- **Control structure**. As with local procedure calls, execution of the calling procedure waits until the called procedure replies, or until this is pre-empted by an exception condition (see B.4.2).
- **Actual Parameters**. The call statement will define the actual parameters (in ways specific to the particular programming language).

These parameters should be consistent with the formal parameters of the interface definition.

B.4.2 Marshalling

At the calling procedure, parameters that are input to the remote call are converted into the agreed external data representation, and are assembled into a protocol octet string for transmission.

At the called procedure, the protocol octet string is disassembled, and the value of each parameter is converted into the local data representation and put in the appropriate location in the called procedure's address space. Execution of the called procedure is then dispatched.

After the called procedure completes execution, the same process occurs in the opposite direction.

This process of converting, copying, assembling and disassembling RPC parameter values is termed "marshalling". It is illustrated in Figure B.3.

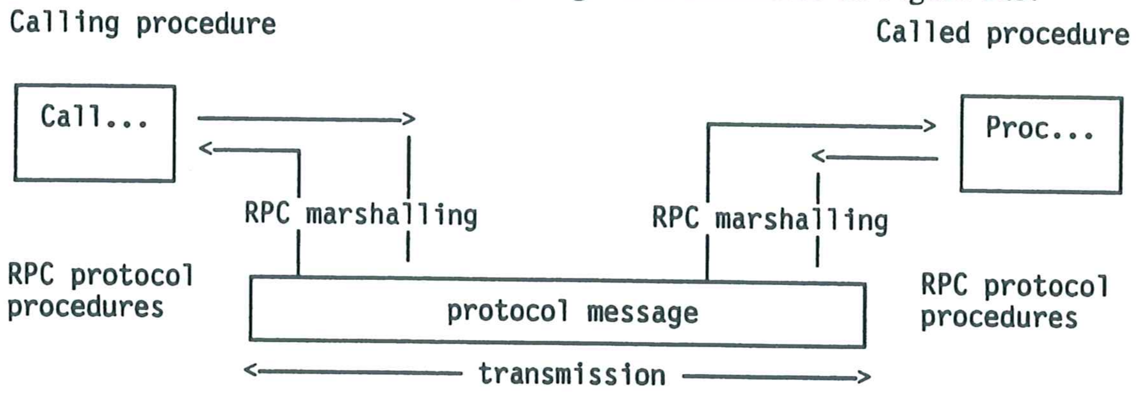

Figure B.3 - RPC Marshalling

In distributed systems the parsing, copying and converting of data for protocol interactions is often a major source of implementation complexity and of performance overheads. The systematic structure of RPC marshalling is amenable to automatic code generation (see B.5) and to specialised performance optimisations.

Where the client and server systems are homogeneous and have the same internal data representation, there is an opportunity for the parameter values to be transferred in native mode, with reduced marshalling overheads.

B.4.3 RPC Error Management

An RPC system should have comprehensive error reporting provisions. Most RPC systems distinguishes between:

- **execution errors** which arise from the procedure call itself (as explained in A.6); and
- **environmental errors** which arise from the called procedure being remote. Examples of environmental errors are: remote system crash, unrecoverable communications problems, naming and binding problems, security problems, protocol or syntax compatibility problems, etc.

A general way of distinguishing between these different error cases is to include in the procedure call definition extra parameters for status and diagnostics. This information would usually be examined by the calling procedure after the return, and could be used to decide what (if any) recovery action is necessary.

Another error management issue is the use of timeouts. The well researched solution adopted in [Birrell 84] is for the RPC system not to have user-visible timeouts. The client application program should have the usual kind of global timeouts for detecting all non-terminating execution (local or remote). This is consistent with the client responsibility for recovery. Internal to the RPC system there may be hidden timeouts for protocol error management, and for periodic checks that unusually long running calls are still executing.

B.4.4 RPC Process Structure

RPC does not require specialised process structures: implementations use whatever are the normal kinds of process structures for their language systems and execution environments. The general case is a single thread of execution per program.

Where a program requires concurrent remote interactions this may be achieved via multiple programmed asynchronous interactions across message read/write interfaces to the underlying protocol machines. But the remote interactions then depart from procedure call language semantics.

Procedure call structure is fully preserved if the language/systems structure allows program execution to spawn multiple processes capable of executing asynchronously and subsequently resynchronising together. These constructs are usually termed a "process fork" and a "process join". The separate asynchronous and logically concurrent threads of a computation can then make multiple concurrent procedure calls without directly delaying progress of the whole computation.

This kind of concurrency, built into the language/execution environment, is different from that traditionally used for interworking between data processing systems. Programming is simplified by using well-formed language constructs (fork and join), and protocols are simplified by decomposition into multiple logically separate interactions, each with a simple synchronous structure. There are also potential performance gains through opportunities for specialised local hardware and software support for handling the asynchrony and synchronisation internally, instead of across the network.

B.4.5 RPC Protocols

Protocols to support RPC consist mainly of request/response message pairs. The request message transfers the procedure invocation and the input values, and the response message transfers the output values (or is an error management message). Simple disciplines for sequence control, flow control and error management are inherent in this strict request/response structure. Each RPC protocol message (PDU) may have a complete self-identifying structure.

Connectionless communications services may be used to carry such protocols. But the usual practice is to introduce RPC techniques by using existing connection-oriented communications, and to evolve later to selective use of connectionless services where appropriate. The [Xerox] and [Sun] RPCs have followed this evolutionary course. Another possible evolution is to carry the RPC data structures over specialised protocols optimised for process-to-process communication, as in [Birrell 84] and the Rex protocol [Otway] evolved from it.

B.5. RPC SOFTWARE ENGINEERING PROCESS

An RPC system may be integrated into comprehensive software engineering processes which exploit the structuredness and relative simplicity of RPC.

The first stage of the process is *interface definition*. The main ingredients are the interface definition language, together with means of documenting, maintaining and retrieving them. This could include use of an Information Resources Dictionary Systems (IRDS), and software tools for syntax checking, specification animation, etc. This is an iterative process, as illustrated in Figure B.4.

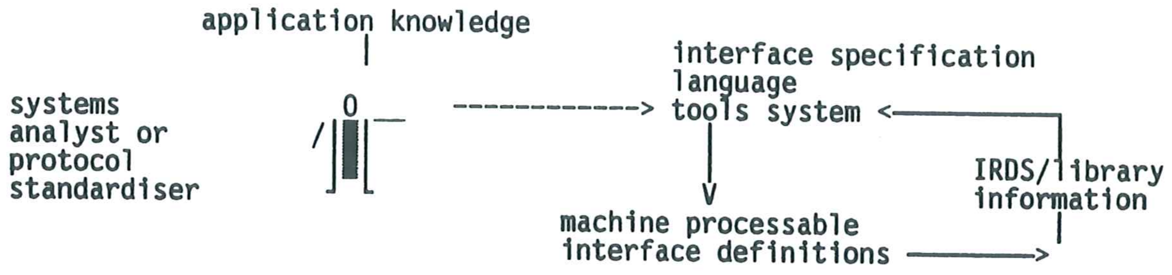
application knowledge
systems analyst or protocol standardiser
interface specification language tools system
machine processable interface definitions
IRDS/library information

Figure B.4 - Interface definition system

The next stage of the process is stub generation. This takes an interface definition and generates the RPC procedures to marshal and unmarshal the parameters of procedure calls using that interface specification. It may also produce the appropriate generic procedures for handling RPC binding primitives for that interface, and code for the RPC buffering and the RPC protocol machine. The stub at the server end also includes a dispatching procedure which distinguishes between calls to different procedures within the server program.

Figure B.5 illustrates a highly automated stub generation system which is fully described in [Gibbons].

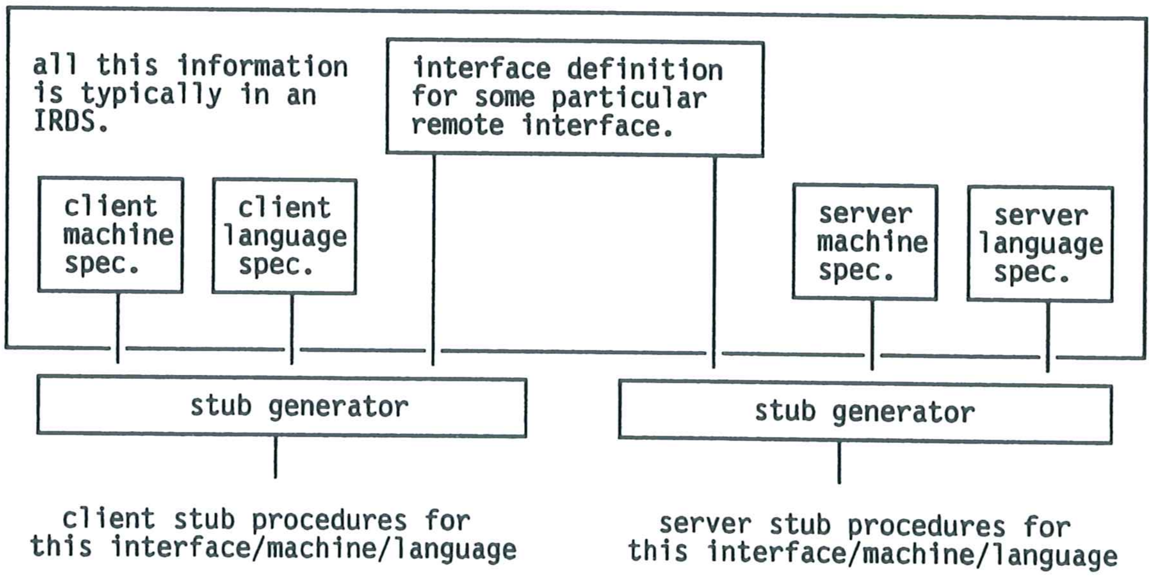
all this information is typically in an IRDS.
interface definition for some particular remote interface.
client machine spec.
client language spec.
server machine spec.
server language spec.
stub generator
stub generator
client stub procedures for this interface/machine/language
server stub procedures for this interface/machine/language

Figure B.5 - An automated system for stub generation

The final stage of the automated development route brings together the application procedures using the RPC system and the generic stub code for the remote interfaces concerned. Program development at the client and server ends typically occurs independently (and possibly many times over, and in many separate business enterprises which use and provide the defined service). Automated development routes, based on the same machine processable text of the interface definition, can assure a high probability of correct and compatible implementations. This may also reduce the amount and cost of validation and conformance testing.

This kind of RPC automation is also relevant for the construction of distributed application's software where the user-visible languages do not have procedure call constructs (e.g. Fourth Generation language systems).

B.6. RPC Summary

The essence of RPC is summarised as:

- program-to-program interworking;
- via programming language constructs;
- maximal de-coupling from communications matters;
- minimal intrusion into the user program source code.

There is consequent scope for automated development routes and specialised design and implementation.

APPENDIX C
BIBLIOGRAPHY

(This Appendix is not an integral part of the standard)

[Bershad]	BN Bershad, DT Ching, ED Lazowska, J Sanislo, M Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems, i.e Transactions on Software Engineering, Vol. SE-13, No. 8, August 1987.
[Birrell 84]	AD. Birrell and BJ. Nelson. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, vol 2, no. 1, pp 39-59, Feb 1984.
[Birrell 85]	AD. Birrell. Secure Communications using Remote Procedure Calls. ACM Transactions on Computer Systems, vol.3, no. 1, pp1-14, Feb.1985.
[Brownbridge]	DR. Brownbridge, LF. Marshal and B. Randell. The Newcastle Connection or UNIXes of the World Unite! Software Practice and Experience, 12 (12), 1147-1162 (December 1982).
[Cristian]	F. Cristian. Robust Data Types. Acta Informatica 17, pp 365-397 (1982).
[Gibbons]	PH. Gibbons. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. i.e Transactions on Software Engineering, Vol. SE-13, No. 1, Jan 1987.
[Gries]	D. Gries. Compiler Construction for Digital Computers. John Wiley 1971. ISBN 0-471-32776-X.
[Hamilton]	KG. Hamilton. A Remote Procedure Call System. Ph.D dissertation, Computer Laboratory, University of Cambridge, UK.
[IBM]	Server Requestor Programming Interface (SRPI).
[Jones]	MB. Jones, RF. Raschid, MR. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. Proceedings 12th. ACM Symposium on Principles of Programming Languages, Jan. 1985.
[Mitchell]	JG. Mitchell, W. Maybury, R. Sweet. Mesa Language Manual (Version 5.0). Tech. Rep. CSL-79-3, Xerox Palo Alto Research Centre, Palo Alto, Calif.1979.
[Nelson]	BJ. Nelson. Remote Procedure Call. Ph.D dissertation. Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Tech. Rep. CMU-CS-81-119, 1981.
[Otway]	D. Otway. Rex: a model for process to process interactions. Online Open Systems Conference. March 1987.

[Panzieri]	F. Panzieri, SK. Shrivastava. Rajdoot: a remote procedure call mechanism supporting orphan detection and orphan killing. Technical Report 200. Computing Laboratory, University of Newcastle upon Type.
[Saltzer]	JH. Saltzer, DP. Reid and DD. Clark: The End to End Argument. ACM Transactions on Computer Systems, Vol.2 No. 4, November 1987, pp 277-288.
[Sun]	Network File System: Remote Procedure Call Protocol Specification. Sun Microsystems Inc. 1984.
[White]	JE White. A High Level Framework for Network Based Resource Sharing. AFIPS Conference Proceedings, NCC, 45: pp 561-570 1976.
[Wijngarden]	A. Van Wijngarden (ed). Report on the Algorithmic Language Algol 68. Mathematisch Centrum, Amsterdam. MR 101.
[Xerox]	Courier: Remote Procedure Call Protocol. Xerox Corporation, Stamford, CT, USA. Xerox Systems Integration Standard 038112, Dec. 1981.

APPENDIX D
FUTURE EXTENSIONS

(This Appendix is not an integral part of the standard)

D.1. INTRODUCTION

This clause describes some potential extensions to the basic RPC protocol, and their language semantics and mappings onto ROSE and other OSI Application Service Elements.

In this standard the dialogue structure via the RPC binding between a client and a server has been restricted in the following ways:

- **asymmetric** - procedure calls are initiated in the direction client to server;
- **request/response** - each remote procedure call always consists of an end-to-end handshake (an invocation and its terminating response);
- **synchronous** - at any given time there is at most one outstanding call on an Association except when a client requests cancellation of the execution of a remote procedure.

These restrictions also apply when there is nested callback within a procedure call. To whatever depth the callback is nested, it is still serial request/response activity, and the outermost call is always invoked by the client.

These restrictions are made here because they are inherent in the procedure call semantics of most programming languages. Furthermore, these dialogue structures are sufficient for many kinds of distributed applications (although certainly not all).

In terms of the ISO/IEC 9072/1 and 9072/2 Remote Operations standards, this standard uses Operation Class 2 (including linked child Operations) within Association Class 1. The Remote Operations standards offer other dialogue structures via Operation Classes 3, 4, and 5 and Association Classes 2 and 3. These dialogue structures are not consistent with the computational model of a procedure call which always has a response.

D.2. CALLS WITH IMMEDIATE RETURN

In some distributed applications there are one-way interactions with no output parameters.

In such cases, the calling procedure would continue execution immediately after making the call, but without confirmation that execution of the remote procedure occurred and was successful.

This could be mapped onto Class 5 Operations. Similarly for the intermediate cases where only success or only failure is reported; they could be mapped onto Class 3 and Class 4 Operations.

But this is a departure from the language semantics of procedure calls, where return means that execution of the called procedure took place. In terms of distributed systems structure, these one-way interactions also tend to have producer/consumer structure, which is different from the client/server structure natural to RPC.

D.3. MUTUAL CALLS

Two components of a distributed application may want to interact with each other via independent calls which are mutual calls, not nested callback.

This standard does not preclude mutual calls, but they would use separate RPC Bindings (mapped onto separate ACSE Associations, initiated in opposite directions). Use of the same Association can result in a deadlock.

D.4. ORPHAN DETECTION

The called procedure is termed an "orphan" if it becomes unintentionally isolated from the calling procedure. This can occur where the calling procedure (the "parent") fails, or if there is prolonged communications failure (network partitioning).

The orphan has nowhere to send its output parameters, and may be uncertain what to do with any partial results which it retains.

The parent procedure may not remember what has happened when it recovers. Continued undetected existence of the orphan(s) may cause difficulties when the parent makes further calls. See [Panzieri]. (Note, however, that context handles may be used to identify state information retained in the called procedure.)

When a procedure is orphaned, the component containing the orphaned procedure may detect this as a break in the ACSE Association (visible at the service interface as an A-P-Abort). But this detection mechanism requires communication to be connection-oriented, which may not always be the case in future. Another possible approach would be use of the OSI CCR, OSI TP.

The computational model needs to be extended to describe the action taken by the orphan. In the simplest case, the orphan completes execution of the remote procedure but the return parameters are not returned to the calling procedure.

D.5. MULTI-ENDPOINT INTERACTIONS

Distributed applications may include multi-endpoint interactions in which an event at one component triggers action at several other components.

This is outside the scope of the current Remote Operations standards, and a multi-endpoint procedure call does not have normal procedure call semantics.

But linguistic structure for this kind of requirement is relevant to a general computational model for Open Distributed Processing (ODP).

D.6. TOTAL OPERATIONS

There may be requirements to exploit the explicit ERRORS structure of the Remote Operations notation more fully than in this standard. The aim would be to achieve the Total Operations structure and Robust Type characteristics explained in [Christian].

This is a problem because of inherent differences between this style of exception handling and the procedure call semantics and structure of most programming language. Therefore, it is left for future study.

D.7. PARAMETER TYPES

The current standard includes parameter types common to ISO Programming Languages. It does not include all types, e.g. COBOL Picture type, that might be desired in the future. Furthermore, there are restrictions on <function-result> and <pointer-constructor> that might be relaxed if the problems concerned with allocation of storage for unmarshalling can be solved.

D.8. ERROR HANDLING

In this standard the method of placing values in RpcError is implementation defined as are the possible values. Establishing standard values of RpcError for error conditions specified in programming language standards, e.g. floating point overflow, is for future study.

D.9. RPC USE OF OTHER OSI TRANSPORT SERVICES

There might be requirements for implementing the RPC Service Provider on a connectionless transport and/or unreliable OSI communication service. The main reason for doing this would be a possible improvement in overall performance.

In this case the RPC Service Provider should be modified to include such mechanisms as retry, segmenting/reassembling, sequence numbering, aliveness control, etc.

However, it is not clear if the resulting overall performance would be better than that obtained when using a transport protocol Class 4, for example.

D.10. SECURITY

There may be requirements to augment this Standard by adding security mechanisms. Such mechanisms should use general OSI security standards.

D.11. INTERFACE SUBTYPING

It may be desirable to extend the Interface Definition Notation to include Inheritance. This will allow an interface definition to be specified as a subtype of other interface definitions. The new interface definition will inherit the properties of its supertypes. The sort of inheritance (single, multiple, strict, etc.) is for further study.

APPENDIX E
SAMPLE CONCRETE SYNTAX FOR THE INTERFACE DEFINITION NOTATION

(This Appendix is not an integral part of the standard)

E.0. INTRODUCTION

This appendix presents a concrete representation which complies to the abstract language defined in section 7.1. This particular concrete representation is in the style of the programming language C.

The notation used is that of section 7.1 with the following exceptions and additions:

1) Within a production, the order in which the components appear is significant.

2) Language keywords are presented in upper case for ease in reading, the concrete representation of keywords is case insensitive.

3) Language punctuation that does not conflict with punctuation characters used in the BNF, appear in a production in the appropriate position. Language punctuation that does conflict with punctuation characters used in the BNF, is enclosed in less than and greater than symbols, e.g. <{>.

4) An abbreviation notation is used for lists of elements of the same category that are separated by commas. The expansion of <category-name-comma-list> is:

   <category-name-comma-list> ::= <category-name>[, <category-name>]...

5) A language keyword, an <Identifier>, a list of <digit>s not preceded or succeeded by a punctuation character shall be preceded or succeeded by at least one space or tab character. Any punctuation character may be preceded or succeeded by one or more space and/or tab characters.

E.1. PACKAGES, PROCEDURES, AND PARAMETERS

<package> ::=	<package-attributes> <{> <package-body> <}>
<package-attributes> ::=	<[>VERSION(<digit>...), <AE-Type-Title> <]> INTERFACE <name-attribute>
<name-attribute> ::=	<Identifier>
<package-body> ::=	[<type-section>] <procedure-section>
<type-section> ::=	[<import>...] <interface-component>...
<interface-component> ::=	<type-declaration> | <constant-declaration>

<procedure-section> ::=	<procedure-declaration>...
<procedure-declaration> ::=	[<procedure-attributes>] <function-result> <procedure-identifier> ([<procedure-parameter-comma-list>])
<procedure-attributes> ::=	<[> {<procedure-residence> [,<callbacks>] | <callbacks> [,<procedure-residence>] }<]>
<procedure-identifier> ::=	<Identifier>
<procedure-residence> ::=	CLIENT | SERVER

If <procedure-residence> is not specified SERVER is assumed.

<callbacks> ::=	CALLBACKS(<procedure-identifier-comma-list>)

The <procedure-identifier>s are the names of procedures that this procedure will call.

<function-result> ::=	<simple-type-specification> | VOID

If the procedure is a function procedure, then <function-result> indicates the datatype of the function's result. If the procedure is not a function procedure, then the keyword "VOID" is written.

<procedure-parameter> ::=	<[> <parameter-attributes> <]> <type-specification> <declarator>
<parameter-attributes> ::=	<directional-attribute> [,<array-bound-attribute-comma-list>]

If there is more than one <parameter-attributes>, they may be written in any order. The keywords MIN__IS and MAX__IS shall appear at most once.

<directional-attribute> ::=	IN | IN , OUT | OUT

E.2. IMPORTING DECLARATIONS

<import> ::=	IMPORT <import-specification> ;
<import-specification> ::=	"<interface-name-attribute>" FROM "<Implementation-Dependent-Source>"
<interface-name-attribute> ::=	<name-attribute>

The <interface-component>s in the package(interface) whose name is <name-attribute> are imported into this package.

E.3. TYPE DECLARATIONS

<type-declaration> ::=	TYPEDEF <type-specification> <declarators> ;
<declarators> ::=	<declarator> [, <declarator>]

<declarator> ::= <Identifier> | <array-declaration> | <pointer-declaration>

By default input and input/output parameters are passed to the called procedure by value. If the called procedure is written in a language that does not call by value, the * symbol in a <pointer-declaration> is used to indicate that the parameter is to be passed by reference.

All output parameters are passed by reference. Unless the parameter is an array, a <pointer-declaration> shall be used to describe the parameter. An output array may also be described using a <pointer-declaration.>

<type-specification> ::= <primitive-datatype> | <constructed-datatype> | <defined-datatype>

<simple-type-specification> ::= <primitive-datatype> | <defined-datatype>

The <defined-datatype> shall identify a <primitive-datatype>.

<primitive-datatype> ::= <integer-datatype> | <real-datatype> | <character-string-datatype> | <bit-string-datatype> | <boolean-datatype> | <complex-datatype> | <numeric-string-datatype> | <enumerated-datatype> | <context-handle-datatype>

<constructed-datatype> ::= <varying-string-constructor> | <record-constructor> | procedure-datatype> | discriminated-union-constructor>

<defined-datatype> ::= <Identifier>

The <Identifier> in the <defined-datatype> shall appear in a <type-declaration>.

E.4. RPC PRIMITIVE DATATYPES

<integer-datatype> ::= {<size-indicator> [UNSIGNED][INT] | [UNSIGNED]<size-indicator>[INT] } [<range-specification>]

<range-specification> ::= <[> <lower-bound-constant>..<upper-bound-constant> <]>

<size-indicator> ::= HYPER | LONG | SHORT | SMALL

HYPER specifies an eight-octet integer, LONG specifies a four-octet integer, SHORT specifies a two-octet integer, and SMALL specifies a one-octet integer.

<real-datatype> ::= REAL(<precision-specifier>)

<precision-specifier> ::= <Digit>...

<precision-specifier> indicates the desired number of decimal digits, although the data is represented in base 2.

<character-string-datatype> ::= CHAR [(<integer-constant>)]

The <integer-constant> is the length of the string. The default string length is 1. A fixed length string whose length is variable is represented as an array described using an <array-bound-attribute>.

<bit-string-datatype> ::= BIT [(<integer-constant>)]

The <integer-constant> is the length of the string. The default string length is 1. A fixed length string whose length is variable is represented as an array described using an <array-bound-attribute>.

<boolean-datatype> ::= BOOLEAN

<complex-datatype> ::= COMPLEX(<precision-specifier>)

<numeric-string-datatype> ::= NUMERIC(<Digit>...)

<context-handle-datatype> ::= CONTEXT(<Digit>...)

<enumerated-datatype> ::= <size-indicator> ENUM <{> <enum-literal-comma-list> <}>

<enum-literal> ::= <Identifier>

E.5. SPECIFICATION OF ARRAYS

In C an array is not a datatype; rather an ordered set of elements of a given datatype may be contained in an array.

<array-declaration> ::= <Identifier> <array-bounds-list>

<array-bounds-list> ::= <array-bounds>

<array-bounds> ::= <[> <]> | <[> <upper-bound> <]> | <[> <lower-bound> .. <upper-bound> <]>

Omitting both array-bounds is equivalent to using a conformant bound of "*". If the <lower-bound> is omitted it is assumed to be 0.

<lower-bound> ::= <bound-constant> | <conformant-bound>

<upper-bound> ::= <bound-constant> | <conformant-bound>

<conformant-bound> ::= *

<bound-constant> ::= <integer-constant>

E.6. SPECIFICATION OF VARYING STRINGS

<varying-string-constructor> ::= {<defined-datatype> | <character-string-datatype> | <bit-string-datatype> } <varying-indicator>

The <defined-datatype> shall be a <character-string-datatype> or a <bit-string-datatype>.

<varying-indicator> ::= MAX_IS (<integer-constant> | *)

The <integer-constant> is the maximum length of the string.

E.7. SPECIFICATION OF RECORDS

```
<record-constructor> ::=          STRUCT <tag> | STRUCT [<tag>] <{>
                                  <record-field>...<}>

<tag> ::=                         <Identifier>

<record-field> ::=                [<[> <member-attribute> <]>] <type-
                                  specification> <declarators> ;

<member-attribute> ::=            IGNORE | <array-bound-attribute-comma-
                                  list>

<array-bound-attribute> ::=       MIN_IS(<attr-var-comma-list>) |
                                  MAX_IS(<attr-var-comma-list>)
```

<array-bound-attribute>s are used to specify the lower and/or upper bounds for an array that has that bound unspecified in at least one dimension in the <type-declaration> for the array. The <attr-var-comma-list> shall have one list position per dimension. If the array bound is unspecified for a particular dimension, the <attr-var> shall be present and the parameter or <record-field> identified by <attr-var> contains the value of the bound.

```
<attr-var> ::=                    [ [*] <Identifier>]
```

E.8. SPECIFICATION OF DISCRIMINATED UNIONS

```
<discriminated-union-constructor> ::=    UNION <tag> | UNION [<tag>]
                                  SWITCH(<union-tag-specifier>) [<union-
                                  name>]<{> <union-body<}>

<union-tag-specifier> ::=         <union-tag-datatype> <Identifier>

<union-tag-datatype> ::=          <integer-datatype> | <character-datatype>
                                  | <boolean-datatype> | <enumerated-
                                  datatype>

<union-name> ::=                  <Identifier>

<union-body> ::=                  <union-case> ... [DEFAULT:[<record-
                                  field>]]

<union-case> ::=                  <case-label>...[<record-field>]

<case-label> ::=                  CASE <case-constant>:

<case-constant> ::= <integer-constant> | <Character> | <enum-literal> |
                                  {TRUE | FALSE}
```

E.9. SPECIFICATION OF POINTER DATA TYPES

```
<pointer-declaration> ::=         * <declarator>
```

E.10. SPECIFICATION OF PROCEDURES AND FUNCTIONS FOR CALLBACK

C does not have procedures, rather it has functions which return no value.

```
<procedure-datatype> ::=        FUNC
```

E.11 SPECIFYING CONSTANTS

```
<constant-declaration> ::=      CONST INT <constant-identifier> =
                                <Digit> ... ;

<constant-identifier> ::=       <Identifier>

<integer-constant> ::=          <constant-identifier> | <Digit> ...
```

E.12 AE-TYPE-TITLES

```
<AE-Type-Title> ::=             <object-identifier>

<object-identifier> ::=         <obj-id-component> ...

<obj-id-component> ::=          { <Identifier> | <Digit> ... |
                                <Identifier> ( <Digit> ...) }
```

APPENDIX F
EXAMPLE

(This Appendix is not an integral part of the standard)

F.1. Interface Definition

This example illustrates the Interface Definition for two procedures that manipulate arrays whose bounds are not known until the calling procedure is executed. Callback is also used.

```
[version 1 { ISO(1) identified-organization(3) icd-ecma(0012) standard(0)
RPC(127) example (0) }] interface Example

  {
  TYPEDEF long N;
  TYPEDEF real  A[*,*], B[*];

  TYPEDEF char max__is (200) ErrorMessage;

[server, callbacks( French, English, Italian )]

  void Invert(

    [in, max__is( Rows, Rows)]  A InputMatrix,
      /* InputMatrix is a conformant array whose bounds information is
         contained in the variable Rows. */

    [out, max__is(Rows, Rows)] A *OutputMatrix,
      /*OutputMatrix is a conformant array. Output arrays may optionally be
      denoted as pointers. */

    [in]   N Rows,
      /*Rows is the value of the bounds of InputMatrix and OutputMatrix, it
         appears explicitly in the calling sequence, whenever client procedures
         and/or server procedures are written in programming languages, e.g.
         Fortran, that require conformant bounds to be  explicit parameters.*/

    [in]   func LanguageUsed
      /*The value of this parameter will determine which procedure will be
         called back.*/

    [out] ErrorMessage *Diagnostic
         --Since Diagnostic is an output parameter it is passed by reference.

         )

[server, callbacks (English, French, Italian)]

  void MultiplyVectors (
[in]  B InputVector1,
   /*InputVector1 is an array with a conformant bound. The bound
      information will be derived from the calling procedure at
      execution time.*/
[in]  B InputVector2,
[out] A CrossProduct,
   /*CrossProduct is an output array with conformant bounds. The bounds
      information will be derived from the calling procedure at execution
      time.*/
[in]  func LanguageUsed,
[out] ErrorMessage *Diagnostic
      -- Since Diagnostic is an output parameter it is passed by reference.
      )
[client] ErrorMessage French (
   /*French is a function which returns a value of type ErrorMessage.
      It resides in the client.*/
[in]  ErrorMessage BeforeTranslation )
```

The functions English and Italian have the same description as the procedure French.

```
      } /*end of interface definition.*/
```

F.2. ARGUMENT Field for the ROIV APDU for Procedure Invert

```
SEQUENCE { Cancel-Flag BOOLEAN, InputMatrixRows, InputMatrixColumns,
      SEQUENCE OF InputMatrixElement, OutputMatrixRows,
      OutputMatrixColumns, Rows, LanguageUsed, DiagnosticMaxLength }

   InputMatrixRows ::=    INTEGER -- First dimension of InputMatrix.
         The value is the same as the value of Rows.

   InputMatrixColumns ::=   INTEGER --Second dimension of
         InputMatrix.The value is the same as the value of Rows.

   InputMatrixElement ::=   REAL  --An element of InputMatrix. Elements
         are transmitted in row major order.

   OutputMatrixRows ::= INTEGER --Array bounds information for the
         output array. The value is the same as the value of Rows.

   OutputMatrixColumns ::= dimension

   Rows ::= dimension INTEGER   --The number of rows and columns
         in the input and output matrices.
```

LanguageUsed ::= INTEGER {3} --When the value is 3, the procedure named Italian may be called back.

DiagnosticMaxLength ::= INTEGER {200} --The maximum length of the varying character string ErrorMessage.

F.3. The RESULT Field for the RORS APDU for the Procedure Invert

SEQUENCE { Cancel-Flag BOOLEAN, Cancel-Count INTEGER, SEQUENCE OF OutputMatrixElement, Diagnostic }

OutputMatrixElement ::= REAL --Elements of the inverted matrix in row major order.

Diagnostic ::= GeneralString --The length of this string is not greater than 200.

F.4. The ARGUMENT Field for the ROIV APDU FOR Procedure MultiplyVectors

SEQUENCE { Cancel-Flag BOOLEAN, InputVector1Length, SEQUENCE OF InputVector1Element, InputVector2Length, SEQUENCE OF InputVector2Element, CrossProductRows, CrossProductColumns, LanguageUsed, DiagnosticMaxLength }

InputVector1Length ::= INTEGER --Number of elements in InputVector1.

InputVector1Element ::= REAL --Element of InputVector1.

InputVector2Length ::= INTEGER --Number of elements in InputVector2.

InputVector2Element ::= REAL --Element of InputVector2.

CrossProductRows ::= INTEGER --Number of rows in the output array.

CrossProductColumns ::= INTEGER --Number of columns in the output array.

LanguageUsed ::= INTEGER {1} --A value of 1 corresponds to the function named French.

DiagnosticMaxLength ::= INTEGER {200} --Maximum length of the output varying character string.

F.5 RESULT Field for RORS APDU for Procedure MultiplyVectors

SEQUENCE { Cancel-Flag BOOLEAN, Cancel-Count INTEGER,
SEQUENCE OF CrossProductElement, Diagnostic }

CrossProductElement ::= REAL --Element of the output array CrossProduct.

Diagnostic ::= GeneralString --The length of this string is not greater than 200.

F.6. ARGUMENT Field for the ROIV APDU for Function French

SEQUENCE { Cancel-Flag BOOLEAN, BeforeTranslationMaxLength,
BeforeTranslation, FunctionResultMaxLength }

BeforeTranslationMaxLength ::= INTEGER {200}--Maximum Length of the input varying character string.

BeforeTranslation ::= GeneralString--Value of the input varying characterstring.

FunctionResultMaxLength ::= INTEGER {200} --Maximum Length of the output varying character string.

F.7. RESULT Field for the RORS APDU for Function French

SEQUENCE { Cancel-Flag BOOLEAN, Cancel-Count INTEGER,
FunctionResult }

FunctionResult ::= GeneralString --The length of the string is not greater than 200.

The functions English and Italian are described exactly like the function French.