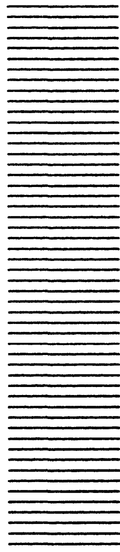
XEROX

Xerox System
Integration Standard

Courier:
The Remote Procedure Call Protocol

XEROX



Xerox System Integration Standard

Courier: The Remote Procedure Call Protocol

XSIS 038112
December 1981

Xerox Corporation
Stamford, Connecticut 06904

Notice

This *Xerox System Integration Standard* describes Courier, the Remote Procedure Call Protocol—the request/reply discipline used by many application protocols in Xerox Network Systems.

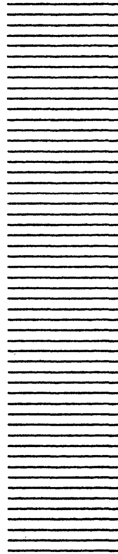
1. This standard includes subject matter relating to patent(s) of Xerox Corporation. No license under such patent(s) is granted by implication, estoppel, or otherwise, as a result of publication of this specification.

2. This standard is furnished for informational purposes only. Xerox does not warrant or represent that this standard or any products made in conformance with it will work in the intended manner or be compatible with other products in a network system. Xerox does not assume any responsibility or liability for any errors or inaccuracies that this document may contain, nor have any liabilities or obligations for any damages, including but not limited to special, indirect, or consequential damages, arising out of or in connection with the use of this document in any way.

3. No representations or warranties are made that this specification, or anything made in accordance with it, is or will be free of any proprietary rights of third parties.

Copyright© Xerox Corporation 1981.
All Rights Reserved.

XEROX®, Xerox Network Systems, and NS
are trademarks of XEROX CORPORATION.



Preface

This document is one of a family of publications that describes the network protocols underlying Xerox Network Systems.

Xerox Network Systems comprise a variety of digital processors interconnected by means of a variety of transmission media. System elements communicate both to transmit information between users and to economically share resources. For system elements to communicate with one another, certain standard protocols must be observed.

This document defines Version 3 of Courier, the Network Systems Remote Procedure Call Protocol. It is of interest both to Courier implementors and to designers of Courier-based application protocols. It introduces the remote procedure call paradigm and describes the purpose and layered nature of the protocol. It explains how two system elements establish and terminate connections between them, agree on the version of Courier that will govern their dialogue, and exchange data via the connection. It presents and gives an example of the intended use, standard representation, and standard notation for each Courier data type. It presents and gives an example of the format and use of each Courier message, and specifies the standard notation for defining a remote program. Appendices describe the procedure for obtaining remote program numbers, summarize the notation to be used in the documentation of Courier-based application protocols, present an example of such a protocol, and give examples of Courier messages and their binary encodings.

Comments and suggestions on this document and its use are encouraged. Please address communications to:

Xerox Corporation
Office Products Division
Network Systems Administration Office
3333 Coyote Hill Road
Palo Alto, California 94304

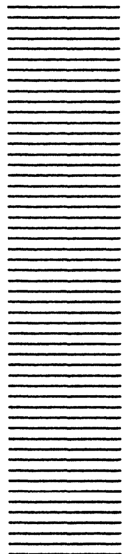


Table of contents

1 Introduction

1.1 Purpose . . . 1
1.2 The model . . . 1
1.3 Protocol layers . . . 3
1.4 Document organization . . . 3

2 Layer one: Transport

2.1 Introduction . . . 4
2.2 Establishing a connection . . . 5
2.3 Exchanging version numbers . . . 5
2.4 Transferring data . . . 6
2.5 Terminating a connection . . . 6

3 Layer two: Data types

3.1 Introduction . . . 8
3.2 Documentation conventions . . . 9
3.3 Type and constant declarations . . . 10
3.4 Predefined types . . . 10
- 3.4.1 Boolean . . . 10
- 3.4.2 Cardinal . . . 11
- 3.4.3 Long cardinal . . . 11
- 3.4.4 Integer . . . 12
- 3.4.5 Long integer . . . 12
- 3.4.6 String . . . 13
- 3.4.7 Unspecified . . . 14

3.5 Constructed types . . . 14
- 3.5.1 Enumeration . . . 14
- 3.5.2 Array . . . 15
- 3.5.3 Sequence . . . 16
- 3.5.4 Record . . . 17
- 3.5.5 Choice . . . 18

3.5.6 Procedure 19
3.5.7 Error 20

4 Layer three: Messages

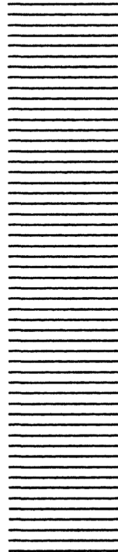
4.1 Introduction 22
4.2 Program declarations 22
4.3 Message types 23
4.3.1 Call 24
4.3.2 Reject 25
4.3.3 Return 26
4.3.4 Abort 26

Appendices

A References 27
B Program number assignment procedures 29
C Standard notation summary 30
D Sample application protocol 32
E Sample protocol exchanges 33

Figures

1.1 The model 2
1.2 Protocol layers 3
2.1 The block stream 4
3.1 The object stream 8
3.2 Graphical notation example 9
4.1 The message stream 22



1 Introduction

1.1 Purpose

One of the communication disciplines most frequently used by distributed system builders is that in which a request for service and its reply are exchanged by two system elements: a service provider and a service consumer. *Courier*, the Network System (NS) Remote Procedure Call Protocol, facilitates the construction of distributed systems by defining a single request/reply or transaction discipline for an open-ended set of higher-level application protocols. Courier standardizes the format of request and reply messages and the network representations for a family of data types from which request and reply parameters can be constructed.

Not all network communication is transaction-oriented. For example, the exchange of control information that typically precedes the transfer of a file between system elements might model naturally as a transaction. However, the transfer of the file's contents is more appropriately modeled as bulk data transfer.

Not all transaction-oriented communication is best accomplished using Courier. For example, the interrogation of a directory of network resources to locate a named resource might model naturally as a transaction. However, satisfying the performance requirements for that operation might necessitate the use of datagrams, rather than virtual circuits (upon which Courier is based).

Other NS protocols—for example, the Sequenced Packet Protocol and the Internet Datagram Protocol [5]—support applications for which Courier is inappropriate.

1.2 The model

As depicted in Figure 1.1, Courier specifies the manner in which a workstation or other active system element invokes operations provided by a server or other passive system element.

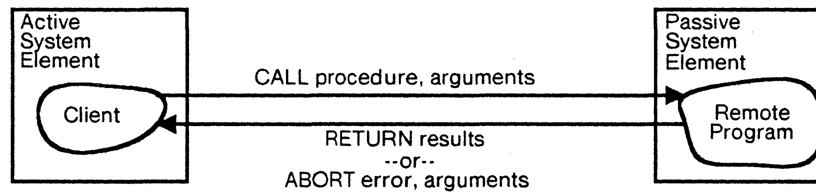


Figure 1.1 The model

Courier uses the subroutine or procedure call as a metaphor for the exchange of a request and its positive reply. An operation code is modeled as the name of a *remote procedure*, the parameters of the request as the arguments of that procedure, and the parameters of the positive reply as the procedure's results. Courier uses the raising of an exception condition or error as a metaphor for the return of a negative reply. An error code is modeled as the name of a *remote error* and the parameters of the negative reply as the arguments of that error. Courier uses the module or program as a metaphor for a collection of related operations and their associated exception conditions. A family of zero or more remote procedures and the zero or more remote errors those procedures can raise are said to constitute a *remote program.*

A remote program usually represents a complete service, and its remote procedures the primitives of that service. One remote program, for example, might provide a file service and contain remote procedures for opening a file, reading a page of an open file, closing an open file, etc. Another remote program might provide a mail service and contain remote procedures for delivering a message to a user, retrieving a user's recent mail, and so forth. A remote program actually represents a type or class of service (for example, a file service). In particular, it does not represent an instance of a service (for example, the instance serving a particular client or executing on a particular system element).

Courier does for distributed system builders some of what a high-level programming language does for implementors of more conventional, non-distributed systems. Pascal, for example, allows the system builder to think in terms of procedure calls, rather than in terms of base registers, save areas, and branch-and-link instructions. So Courier allows the distributed system builder to think in terms of remote procedure calls, rather than in terms of socket numbers, network connections, and message transmission. Pascal allows the system builder to think in terms of integers and strings, rather than in terms of sign bits, length fields, and character codes. Courier allows the distributed system builder to do the same.

In some respects, remote procedure calling is a generalization of the protocol design techniques employed by early Arpanet and Ethernet designers. In other respects, however, it is a departure from them. The reader is referred to [4] for a careful exposition of the protocol design experience that led to the remote procedure call model and for a detailed discussion of its advantages over earlier approaches.

1.3 Protocol layers

Courier is a layered protocol, as suggested by Figure 1.2. *Layer one*, the lowest layer, defines a block stream which can carry blocks of arbitrary binary data between system elements. Block streams are defined in terms of the connection abstraction of the Sequenced Packet Protocol.

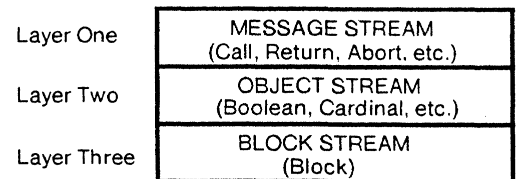


Figure 1.2 Protocol layers

Layer two defines an object stream capable of carrying structured data (for example, booleans and cardinals) between system elements. Object streams are defined in terms of the block stream abstraction of layer one.

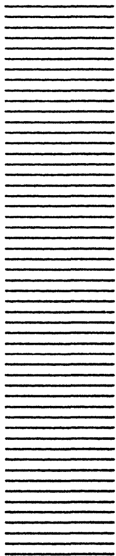
Layer three defines a message stream capable of carrying service requests (that is, call messages) and replies (for example, return and abort messages) between system elements. Message streams are defined in terms of the object stream abstraction of layer two.

1.4 Document organization

Section 2 of this document discusses the block stream, defining layer one of the protocol. Section 3 discusses the object stream, defining layer two. Section 4 discusses the message stream, defining layer three.

Appendix A lists other documents that supplement this protocol specification. Appendix B describes the procedure for obtaining remote program numbers. Appendix C summarizes the notation to be used in the documentation of Courier-based application protocols. Appendix D offers an example of such a protocol. Appendix E gives some examples of Courier messages and their binary encodings.

This document is of interest both to Courier implementors and to designers of Courier-based application protocols. Courier implementors should read the entire document. Application protocol designers need read only Section 3, Section 4.2, and Appendices B, C, and D.



2

Layer one: Transport

2.1 Introduction

Courier defines the manner in which one system element communicates with another to effect a remote procedure call. At layer one, Courier defines a bi-directional *block stream*, depicted in Figure 2.1, which transports a series of *data blocks* in each direction between system elements. Each data block comprises zero or more—*but always a multiple of 16*—bits of arbitrary data, consecutively numbered from zero to, say, *n*.

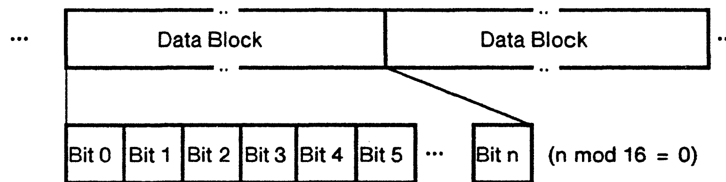


Figure 2.1 The block stream

The block stream is defined by imposing structure upon the data carried by a connection. A connection is a reliable, ordered, flow-controlled, bi-directional, logical communication channel (sometimes called a virtual circuit) between two software ports called sockets. The connection is an abstraction of the Sequenced Packet Protocol [5].

Connections are used only as a means for the reliable transport of data, and Courier-based application protocols should associate no state information with them. Applications that require the concept of a session should use remote procedures to initiate and terminate sessions, rather than using the establishment and termination of the connection to signal those events. The remote procedure whose invocation begins a session should return some sort of session handle. Such a handle can then be supplied as an argument to the other remote procedures that constitute the application, including, of course, the procedure that ends the session. In any case, the effect upon the application of a series of remote procedure calls should be independent of whether the series of calls is made via one connection or several.

This section defines the manner in which two system elements establish and terminate connections between them, agree on the version of the Courier Protocol that will govern their dialogue, and deduce the Courier block stream from the packets that traverse the connection.

2.2 Establishing a connection

A connection is established through the cooperation of an active user process on one system element and a passive listening process on another (or the same) system element. A user process is typically an application program whose lifetime is that of a session. Any number of user processes may coexist in a single machine. A listening process, on the other hand, is typically a system program and acts on behalf of an entire system element. Exactly one listening process is resident in each machine that implements Courier.

The listening process monitors well-known socket number 5. When a user process initiates a connection to that socket, the listening process spawns a server process to attend to the user process, and then resumes the monitoring of its contact socket. Any number of server processes may coexist in a single machine, and any number of connections may coexist between a pair of machines (subject to the resource limitations of each system element).

The details of connection establishment are specified by the Sequenced Packet Protocol [5].

2.3 Exchanging version numbers

Once a connection is established between them, user and server processes must decide which version of the Courier Protocol will govern their further interaction. This document defines Version 3 of the Courier Protocol. Occasionally, Courier may be expanded to provide new capabilities or changed to provide existing capabilities more effectively. In a large installation, one may be unable to guarantee that all system elements will advance simultaneously to each new version of the protocol. In fact, some system elements may have to support several versions of the protocol during transition periods.

User and server processes agree on a version of Courier by exchanging version number ranges. Each process tells the other the lowest and highest version numbers it supports; if only one version is supported, the two numbers are the same. If the two ranges overlap, the highest version number supported by both processes is selected and communication proceeds on that basis. If the ranges are disjoint, that is, if no version is supported by both processes, meaningful interaction is presumed impossible and the connection is terminated as described in Section 2.5.

The user process initiates the version number exchange by spontaneously sending its version number range to the server process. If the user process indicates support for several versions of the protocol, it must then wait for the server process to respond with its version number range; it can send nothing more on the connection without knowing which version of the protocol to employ. If the user process indicates support for just one version of the protocol, on the other hand, it can and must send additional data without waiting to hear from the server process; the server process will either correctly interpret that data in accordance with the protocol version specified, or terminate the connection.

The server process completes the version number exchange by sending its version number range to the user process. If the user process indicated support for several versions of the protocol, or if the server process does not support the one version for which the user process indicated support, the server process must immediately send its version number range to the user process. Otherwise, the server process may (but is not required to) continue receiving and processing data sent by the user process before sending its own version number range.

Version number ranges are encoded as the first 32 bits of system data sent in each direction on the connection (see Section 2.4 below). These bits are not considered part of the first data block. The first 16 bits represent the lowest supported version number, the second 16 bits the highest. Each version number is encoded as an unsigned binary number whose most and least significant bits are Bits 0 and 15, respectively. The message and packet that carry the version number range may, but need not, carry other system data.

The intent is that the version number exchange introduce no roundtrip delay when the user process supports a single version of the protocol. Conceptually, in this case, each process simply prefixes its version number range to the block stream.

2.4 Transferring data

The connection whose establishment is described above carries a series of packets, each of which contains data, datastream type, end-of-message, and other fields, as prescribed by the Sequenced Packet Protocol [5]. At a slightly higher level of abstraction, the connection carries a series of messages, each of which consists of one or more packets, the last—and only the last—of which has end-of-message set to TRUE. Courier requires that all packets of a message have the same datastream type. (The message abstraction of the Sequenced Packet Protocol, referred to here, is to be distinguished from the Courier abstraction of the same name, which is defined in Section 4.)

Apart from the version number exchange described in Section 2.3 above, each message whose packets are of datastream type zero represents a single Courier data block. Each packet of the message contributes to the data block the zero or more 8-bit bytes contained in its data field. Thus a data block is formed, at least conceptually, by simply concatenating the contents of the data fields of successive packets of a message. Packet boundaries are insignificant, that is, they carry no information. Using the numbering convention of the Internet Datagram Protocol [5], bit *b* of byte *B* of word *w* of the data field of each packet is the $(16*w+8*B+b+1)^{th}$ bit contributed to the data block by that packet.

Although its primary purpose is to carry the *system data* that constitutes the block stream, the Courier connection may also be used to carry *application data*. For example, a file retrieval primitive implemented as a remote procedure may use the connection to return the contents of a file. Doing so is generally more sensible than returning the file's contents as a result of the procedure, and may be more convenient than establishing a separate connection for the transfer. Apart from the *end* and *end-reply* packets mentioned in Section 2.5 below, packets of datastream type greater than zero are assumed to carry application data. The syntax and semantics of such data are the province of higher-level application-specific protocol documents.

Fine point: Whenever possible, application protocols should avoid multiplexing system and application data on the same connection and, instead, use a separate connection for the latter. The presence of large amounts of application data on the Courier connection may prevent the timely exchange of system data.

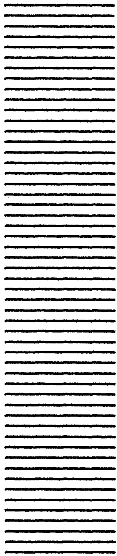
2.5 Terminating a connection

Both the user and server processes may initiate termination of the connection at any time. The former should do so as soon as it has no further need of the server process, the latter after the

connection has been inactive (that is, carried no data) for a period that the server process deems excessive. In either case, the server process should destroy itself once the connection is terminated.

The details of connection termination are specified elsewhere. Courier uses the three-way handshake of *end* and *end-reply* packets detailed in the specification of the Sequenced Packet Protocol [5].

Fine point: Because it is sent in-band, the *end* packet will be seen and acted upon immediately only if the destination process is actively reading data from the connection at the time. Therefore, connection termination cannot be counted upon to cancel an outstanding remote procedure call.



3

Layer two: Data types

3.1 Introduction

At layer one, Courier defines a block stream, which carries a series of blocks, each of which is arbitrary binary data. At layer two, Courier defines a bi-directional *object stream*, depicted in Figure 3.1, which transports a series of structured *data objects* in each direction between system elements. Each data object is of one of several standard *data types* (for example, boolean or cardinal).

Figure 3.1 The object stream

The object stream is defined by imposing structure upon the data carried by the block stream. Each data object is encoded as a single data block. Some data types (for example, array and record) combine several data objects to form a larger, composite data object. In such cases, the entire composite data object is encoded as a single data block. Block boundaries allow data objects to be read from the connection without knowledge of the corresponding objects' syntax or semantics.

This section presents and gives examples of the intended use, standard representation, and standard notation for each of the data types defined by Courier. A type's *standard representation* is the detailed encoding that Courier implementations employ when transmitting objects of that type via the block stream. A type's *standard notation* is the detailed conventions that application protocol designers employ to denote either the type itself, or a constant of that type, in higher-level protocol documents. Thus a type's standard representation and notation are, respectively, the run- and documentation-time descriptions of a class of data objects transportable via the object stream.

Although Courier data will occasionally be carried via 9,600 baud telephone lines and other low-speed media, more frequently it will be transported by 10 megabit-per-second Ethernet networks [1]. Therefore, the standard representations defined below are designed to minimize not the amount of communication bandwith required for their transportation, but rather the amount of computing bandwith required for their preparation and interpretation. Thus, for example, all standard representations are a multiple of 16 bits in length, even though, in some

cases, more compact representations could have been defined (for example, for boolean). An example of how data representations can be cleverly optimized for space can be found in [2].

Courier's standard representations are untyped at run time. That is, the standard representation of a data object encodes the value but not the type of that object. Knowledge of an object's type resides in the software that generates or interprets the representation, rather than in the representation itself. To properly interpret an incoming data object, the software must have run-time access to its type declaration (for example, in the form of a table).

Implementation note: Many high-level languages define data types that are semantically equivalent (or similar) to those defined by Courier. In such environments, it is often useful to define mappings between Courier data types and those of the host language. A Courier implementation can then provide software that converts a Courier data object (in its standard representation) to or from a form in which it can be manipulated using normal language or run-time facilities.

3.2 Documentation conventions

Standard representations are defined throughout this section using an informal graphical notation. For example, the standard representation of a data object of type boolean is a single bit that encodes its value, preceded by 15 zero bits. The value TRUE is encoded as one, the value FALSE as zero. This structure is depicted graphically as shown in Figure 3.2.

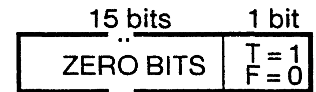


Figure 3.2 Graphical notation example

The representation of each data object is decomposed into one or more *fields*, each of which is depicted by a rectangle. A field's description and size in bits, *n*, appear within and above its rectangle, respectively. Although the bits that constitute a field are not explicitly numbered in the graphical depictions, Bits 0 and n-1 are understood to lie at the left and right edges of the rectangle, respectively. The abbreviations "MSB" and "LSB" stand for "most significant bit" and "least significant bit", respectively. Ellipses ("..") that break a rectangle's horizontal edges indicate that the field is not shown to scale.

When two or more fields constitute a representation, their order in the graphical depiction reflects their order in the data block. Ellipses ("...") appearing between rectangles indicate that the previous field or fields are replicated. In examples, standard representation values are partitioned into 16-bit units whose contents are specified in both octal and hexadecimal.

Standard notations are defined throughout this document using Backus-Naur Form (BNF) [3]. For example, the standard notation for the boolean type is simply the keyword BOOLEAN, and for boolean constants either the keyword TRUE or FALSE. These definitions are stated in BNF as follows:

```
BooleanType      ::= BOOLEAN
BooleanConstant  ::= TRUE | FALSE
```

Symbols rendered in **bold** are non-terminals; all other symbols are terminals. Non-terminals whose first character is upper-case are defined in the grammar; all other non-terminals, of which there are four—**identifier**, **number**, **string**, and **empty**—are informally defined outside of the grammar.

An **identifier** is a sequence of upper- and lower-case letters and digits; the first character must be a letter. Case differences are significant and distinguish one identifier from another. A **number** is a sequence of digits and upper-case letters in the range 'A through 'F, optionally followed by the letter 'D, 'B, or 'H. The suffix 'D, the default, signifies decimal notation (that is, radix 10). The suffix 'B signifies octal notation (that is, radix 8). The suffix 'H signifies hexadecimal notation (that is, radix 16). A **string** is a sequence of NS characters [6] (also, see Section 3.4.6). A quotation mark (") must be doubled to distinguish it from the one that always delimits a string in the standard notation. The non-terminal **empty** denotes the null or empty string of symbols.

Comments may be embedded in the documentation of higher-level protocols. They are preceded by the symbol "--" and terminated by either the symbol "--" or the end of a line.

3.3 Type and constant declarations

Data types, as well as values or *constants* of those types, may be defined or *declared* in higher-level protocol documents in the following manner. A declaration assigns a name (**identifier**) to a particular type or constant. Such names simplify written and verbal discussion of the protocol, but have no significance at run time. In particular, they never appear in the block stream:

```
TypeDecl      ::= identifier : TYPE = Type ;
ConstantDecl  ::= identifier : Type = Constant ;
```

The data types defined by Courier fall into two broad classes: predefined and constructed. A *predefined type* is one that is fully specified by Courier. A *constructed type* is one defined by the application protocol designer, in most cases using predefined or other constructed types:

```
Type      ::= PredefinedType | ConstructedType
Constant  ::= PredefinedConstant | ConstructedConstant
```

3.4 Predefined types

Courier defines seven predefined data types:

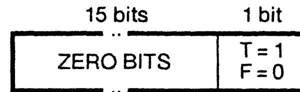
```
PredefinedType      ::= BooleanType | CardinalType | LongCardinalType |
                        IntegerType | LongIntegerType | StringType | UnspecifiedType

PredefinedConstant  ::= BooleanConstant | CardinalConstant | LongCardinalConstant |
                        IntegerConstant | LongIntegerConstant |
                        StringConstant | UnspecifiedConstant
```

The *boolean* type is used to model logical data. The *cardinal*, *long cardinal*, *integer*, and *long integer* types are used to model numeric data. The *string* type is used to model textual data. The *unspecified* type is used to model data which need not be interpreted by its recipient.

3.4.1 Boolean

A data object of type boolean represents a logical quantity that can assume either of two values, called TRUE and FALSE. Switch settings and answers to yes-or-no questions are among the entities that are appropriately modeled as booleans.

The standard representation of a boolean is a single bit that encodes its value, preceded by 15 zero bits. The value TRUE is encoded as one, the value FALSE as zero:



The standard notation for the boolean type is simply the keyword BOOLEAN, and for boolean constants either the keyword TRUE or FALSE:

```
BooleanType      ::=  BOOLEAN
BooleanConstant  ::=  TRUE | FALSE
```

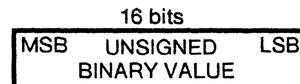
Example:

TRUE is encoded as: value 000001_8 0001_{16}

3.4.2 Cardinal

A data object of type cardinal represents an integer in the closed interval [0, 65535]. File counts and time intervals measured in seconds are among the entities that might be appropriately modeled as cardinals.

The standard representation of a cardinal is a single 16-bit field that encodes its value as an unsigned binary number whose MSB and LSB are Bits 0 and 15, respectively:



The standard notation for the cardinal type is simply the keyword CARDINAL, and for cardinal constants simply a number:

```
CardinalType      ::=  CARDINAL
CardinalConstant  ::=  number
```

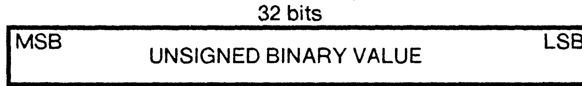
Example:

15D is encoded as: value 000017_8 $000F_{16}$

3.4.3 Long cardinal

A data object of type long cardinal represents an integer in the closed interval [0, 4294967295]. File sizes measured in bytes, and dates measured in seconds since the turn of the century, are among the entities that might be appropriately modeled as long cardinals.

The standard representation of a long cardinal is a single 32-bit field that encodes its value as an unsigned binary number whose MSB and LSB are Bits 0 and 31, respectively:



The standard notation for the long cardinal type is simply the keyword LONG CARDINAL, and for long cardinal constants simply a number:

```
LongCardinalType      ::= LONG CARDINAL
LongCardinalConstant  ::= number
```

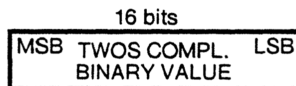
Example:

	value
65551D is encoded as:	000001 000017_8
	0001 $000F_{16}$

3.4.4 Integer

A data object of type integer represents a signed integer in the closed interval [-32768, 32767]. Personal checking account balances are among the entities that might be appropriately modeled as integers.

The standard representation of an integer is a single 16-bit field that encodes its value as a twos complement binary number whose MSB and LSB are Bits 0 and 15, respectively:



The standard notation for the integer type is simply the keyword INTEGER, and for integer constants a signed number:

```
IntegerType      ::= INTEGER
IntegerConstant  ::= number | - number
```

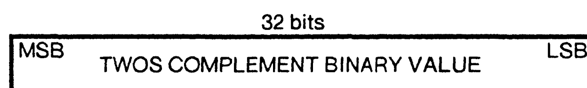
Example:

	value
-15D is encoded as:	177761_8
	$FFF1_{16}$

3.4.5 Long integer

A data object of type long integer represents a signed integer in the closed interval [-2147483648, 2147483647]. National treasury balances are among the entities that might be appropriately modeled as long integers.

The standard representation of a long integer is a single 32-bit field that encodes its value as a twos complement binary number whose MSB and LSB are Bits 0 and 31, respectively:



The standard notation for the long integer type is simply the keyword LONG INTEGER, and for long integer constants a signed number:

```
LongIntegerType      ::= LONG INTEGER
LongIntegerConstant  ::= number | - number
```

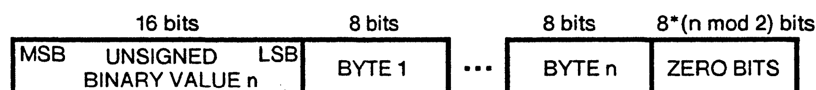
Example:

	value	
-65551D is encoded as:	177776	177761_8
	FFFE	$FFF1_{16}$

3.4.6 String

A data object of type string represents an ordered collection of NS characters, whose number need not be specified until run time. The 16-bit NS Character Set [6] supports multinational applications, including Japanese-language systems. User, directory, file, and mailbox names, as well as passwords, are among the entities that are often appropriately modeled as strings.

The standard representation of a string is a 16-bit field that encodes *n*, the number of 8-bit bytes required to encode the text of the string in accordance with the NS String Format [6], immediately followed by that encoding, followed in turn by eight zero bits if *n* is odd. The number *n*, whose maximum value is 65,535, is encoded as an unsigned binary number whose MSB and LSB are Bits 0 and 15, respectively:



The standard notation for the string type is simply the keyword STRING, and for string constants a string enclosed in quotation marks ("):

```
StringType      ::= STRING
StringConstant  ::= " string "
```

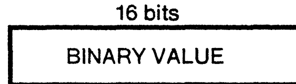
Examples:

	count	"Wh"	"it"	"e"	
"White" is encoded as:	000005	053550	064564	062400_8	
	0005	5768	6974	6500_{16}	
	count	"Su"	"m "	Greek	"Σ"
"Sum Σ" is encoded as:	000007	051565	066440	177401	031000_8
	0007	5375	6D20	FF01	3200_{16}

3.4.7 Unspecified

A data object of type unspecified represents a 16-bit quantity of unspecified interpretation. Session and file handles are among the entities that might be appropriately modeled as unspecifieds.

The standard representation of an unspecified is a single uninterpreted 16-bit field:



The standard notation for the unspecified type is simply the keyword UNSPECIFIED. The standard notation for unspecified constants is a number that, when interpreted as a cardinal constant, defines the standard representation's 16-bit binary value:

```
UnspecifiedType       ::=  UNSPECIFIED
UnspecifiedConstant   ::=  number
```

Example:

	value
16440B is encoded as:	016440_8
	$1D20_{16}$

3.5 Constructed types

Courier defines seven constructed data types:

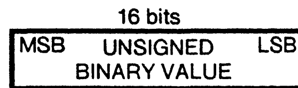
```
ConstructedType      ::=  EnumerationType | ArrayType | SequenceType |
                          RecordType | ChoiceType | ProcedureType | ErrorType

ConstructedConstant  ::=  EnumerationConstant | ArrayConstant | SequenceConstant |
                          RecordConstant | ChoiceConstant | ProcedureConstant |
                          ErrorConstant
```

Enumeration types are used to model severely restricted numeric data. *Array* and *sequence* types are used to model homogeneous collections of data. *Record* types are used to model heterogeneous collections of data. *Choice* types are used to model selections from among heterogeneous collections of data. *Procedure* types are used to model remotely performed operations. *Error* types are used to model exception conditions reported by such operations.

3.5.1 Enumeration

A data object of type enumeration represents a quantity that can assume any of a relatively few named integer values in the closed interval [0, 65535]. File types and error codes are among the entities that might be appropriately modeled as enumerations.

The standard representation of an enumeration is a single 16-bit field that encodes its value as an unsigned binary number whose MSB and LSB are Bits 0 and 15, respectively:



Fine point: Whenever possible, designers of higher-level protocols should choose consecutive enumeration values, beginning with zero.

The standard notation for enumeration types uses braces ('{ and '}) to delimit the set of defined names and their corresponding values. The standard notation for enumeration constants is simply one of the defined names:

```
EnumerationType        ::=   { CorrespondenceList }
EnumerationConstant    ::=   identifier

CorrespondenceList     ::=   Correspondence | CorrespondenceList , Correspondence
Correspondence         ::=   identifier ( number )
```

Example:

Given the declaration:

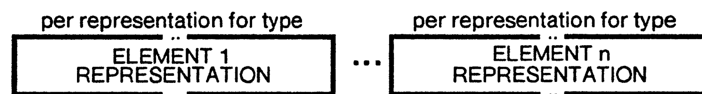
```
Mode: TYPE = {readPage(0), writePage(1), readAndOrWritePage(2)};
```

	value
writePage is encoded as:	000001_8
	0001_{16}

3.5.2 Array

A data object of type array represents an ordered, one-dimensional, homogeneous collection of data objects, whose type and number *n* are specified at documentation time. The *elements* of an array may be of any type, either predefined or constructed. In particular, the elements may themselves be arrays. File pages are among the entities that might be appropriately modeled as arrays.

The standard representation of an array is simply the standard representations of its elements, one following the other, in proper order. The number of bits required to encode each element is determined by the elements' type:



The standard notation for array types uses the keyword ARRAY to introduce the number and type of the array's elements. The standard notation for array constants uses brackets ('[and ']) to enclose an ordered (possibly empty) list of element values, separated by commas (','):

```
ArrayType       ::=  ARRAY number OF Type
ArrayConstant   ::=  [ ElementList ] | [ ]

ElementList     ::=  Constant | ElementList , Constant
```

Example:

Given the declaration:

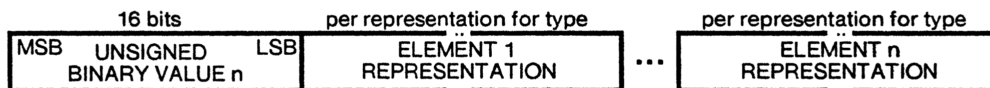
PageContents: TYPE = ARRAY 256 OF UNSPECIFIED;

	$elem_1$	... $elem_{256}$
[0, 7602B, ... 54553B] is encoded as:	000000	... 054553_8
	0000	... $596B_{16}$

3.5.3 Sequence

A data object of type sequence represents an ordered, one-dimensional, homogeneous collection of data objects, whose type and maximum number m are specified at documentation time, but whose actual number n need not be specified until run time. (The maximum value of m is 65,535.) The *elements* of a sequence may be of any type, either predefined or constructed. In particular, the elements may themselves be sequences. File access and mail distribution lists are among the entities that might be appropriately modeled as sequences.

The standard representation of a sequence is a 16-bit field that encodes the actual number of elements in the sequence, immediately followed by the standard representations of the elements, one following the other, in proper order. The number of elements in the sequence is encoded as an unsigned binary number whose MSB and LSB are Bits 0 and 15, respectively. The number of bits required to encode each element is determined by the elements' type:



The standard notation for sequence types uses the keyword SEQUENCE to introduce the maximum number and type of the sequence's elements. If unspecified, the former defaults to 65535, the largest cardinal. The standard notation for sequence constants uses brackets ('[and ']) to enclose an ordered (possibly empty) list of element values, separated by commas (','):

```
SequenceType       ::=  SEQUENCE MaximumNumber OF Type
SequenceConstant   ::=  [ ElementList ] | [ ]

MaximumNumber      ::=  number | empty
ElementList        ::=  Constant | ElementList , Constant
```

Example:

Given the declaration:

PageContents: TYPE = SEQUENCE 256 OF UNSPECIFIED;

	count	$elem_1$ ... $elem_{255}$
[7602B, ... 54553B] is encoded as:	000377	007602 ... 054553_8
	00FF	0F82 ... $596B_{16}$

3.5.4 Record

A data object of type record represents an ordered, possibly heterogeneous collection of data objects, whose types and number *n* are specified at documentation time. A record *component* may be of any type, either predefined or constructed. In particular, a record component may itself be a record. Predefined collections of file attributes are among the entities that might be appropriately modeled as records.

The standard representation of a record is simply the standard representations of its components, one following the other, in proper order. The number of bits required to encode each component is determined by the component's type:

The standard notation for record types uses the keyword RECORD to introduce the names, types, and (implicitly) order of the record's components (if any). The standard notation for record constants uses brackets ('[and ']) to enclose an ordered (possibly empty) list of component values, separated by commas (',). When a list of component names precedes the specified type or constant, that type or constant is understood to be assigned to each of the specified components, which are distinct:

```
RecordType      ::=  RECORD [ FieldList ] | RECORD [ ]
RecordConstant  ::=  [ ComponentList ] | [ ]

FieldList       ::=  Field | FieldList , Field
Field           ::=  NameList : Type
ComponentList   ::=  Component | ComponentList , Component
Component       ::=  NameList : Constant
NameList        ::=  identifier | NameList , identifier
```

Example:

Given the declaration:

Credentials: TYPE = RECORD **[user, password:** STRING**];**

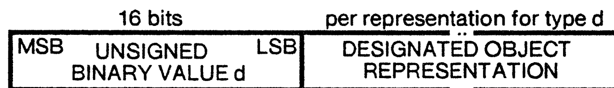
[user: "White", password: "vlw"] is encoded as:

count	"Wh"	"it"	"e"	count	"vl"	"w"
000005	053550	064564	062400	000003	073154	073400_8
0005	5768	6974	6500	0003	766C	7700_{16}

3.5.5 Choice

A data object of type choice represents a data object whose type is chosen at run time from a set of candidate types specified at documentation time. Candidate types are designated by named integer values in the closed interval [0, 65535]. Each candidate type may be either predefined or constructed and, in particular, may itself be a choice. Dates that may appear in either textual or binary form are among the entities that might be appropriately modeled as choices.

The standard representation of a choice is a 16-bit field that encodes a designator value *d* as an unsigned binary number whose MSB and LSB are Bits 0 and 15, respectively, immediately followed by an object of the designated type in its standard representation. The number of bits required to encode the designated object is determined by its type:



Fine point: Whenever possible, designers of higher-level protocols should choose consecutive designator values, beginning with zero.

As pointed out earlier, Courier's standard representations are untyped at run time. That is, the standard representation of a data object encodes its value but not its type. Using the choice type, however, the protocol designer can construct data objects whose representations are effectively typed at run time by means of the choice designator.

The standard notation for choice types uses the keyword CHOICE to introduce, and braces ('{ and '}) to delimit, the set of candidate types. Along with each type is specified its designator's name and value. The standard notation also permits the designator to be separately declared, as an enumeration type. In such cases, the name of that type appears in the choice specification in lieu of designator values. The standard notation for choice constants is simply a designator name, followed by a constant of the corresponding type:

ChoiceType	::=	CHOICE **DesignatorType** OF **{ CandidateList }**
ChoiceConstant	::=	**identifier Constant**
DesignatorType	::=	**empty | identifier**
CandidateList	::=	**Candidate | CandidateList , Candidate**
Candidate	::=	**DesignatorList => Type**
DesignatorList	::=	**Designator | DesignatorList , Designator**
Designator	::=	**identifier | identifier (number)**

Example:

Given the declaration:

```
FileIdentifier: TYPE = CHOICE OF {
 name(0)    => STRING,
 handle(1)  => UNSPECIFIED};
```

Or the declarations:

```
FileIdentifierType: TYPE = {name(0), handle(1)};
FileIdentifier: TYPE = CHOICE FileIdentifierType OF {
 name       => STRING,
 handle     => UNSPECIFIED};
```

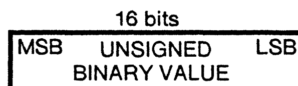
	desig	object
handle 7712B is encoded as:	000001	007712_8
	0001	$0FCA_{16}$

3.5.6 Procedure

A data object of type procedure represents the identifier or code for an operation that one system element will perform at the request of another (or the same) system element. The operation may require parameters when invoked, return parameters if it succeeds, and report exception conditions if it fails. The *arguments* and *results* of a procedure are data objects whose types and number are specified at documentation time; each argument and result may be of any type, either predefined or constructed. The *errors* raised by a procedure are represented by data objects of type error (see Section 3.5.7 below) whose constant values are specified at documentation time. Remotely accessible file manipulation primitives are among the entities that might be appropriately modeled as procedures.

Note: The use of procedure types is severely restricted. Procedures may not be passed as arguments or results of remote procedures, or as arguments of remote errors; procedure types may be used only to specify the remote procedures that constitute a remote program, as described in Section 4.2. More precisely, constants may be of type procedure, but procedure arguments, procedure results, error arguments, array and sequence elements, record components, and choice candidates may not.

The standard representation of a procedure is a single 16-bit field that encodes its value—an integer in the closed interval [0, 65535]—as an unsigned binary number whose MSB and LSB are Bits 0 and 15, respectively:



The standard notation for procedure types uses the keyword PROCEDURE to introduce the names, types, and (implicitly) order of the procedure's arguments; the keyword RETURNS to introduce the names, types, and (implicitly) order of the procedure's results; and the keyword REPORTS to introduce the names of the procedure's error constants. When a list of names

precedes the specified type, that type is understood to be assigned to each of the specified arguments or results, which are distinct. The standard notation for procedure constants is simply a number:

```
ProcedureType       ::=  PROCEDURE ArgumentList ResultList ErrorList
ProcedureConstant   ::=  number

ArgumentList        ::=  empty | [ FieldList ]
ResultList          ::=  empty | RETURNS [ FieldList ]
ErrorList           ::=  empty | REPORTS [ NameList ]
FieldList           ::=  Field | FieldList , Field
Field               ::=  NameList : Type
NameList            ::=  identifier | NameList , identifier
```

Example:

Given the declaration:

```
OpenFileProcedure: TYPE = PROCEDURE [user, password, filename: STRING]
 RETURNS [handle: UNSPECIFIED]
 REPORTS [NoSuchUser, IncorrectPassword, NoSuchFile];
```

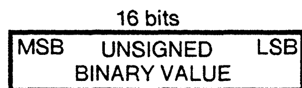
	value
0 is encoded as:	000000_8
	0000_{16}

3.5.7 Error

A data object of type error represents the identifier or code for an exception condition that one system element may report to another (or the same) system element in response to a request to perform an operation. Parameters may accompany the report. The *arguments* of an error are data objects whose types and number are specified at documentation time; each argument may be of any type, either predefined or constructed. Invalid password or nonexistent file are among the conditions that might be appropriately modeled as errors.

Note: The use of error types is severely restricted. Errors may not be passed as arguments or results of remote procedures, or as arguments of remote errors; error types may be used only to specify the remote errors that constitute a remote program, as described in Section 4.2. More precisely, constants may be of type error, but procedure arguments, procedure results, error arguments, array and sequence elements, record components, and choice candidates may not.

The standard representation of an error is a single 16-bit field that encodes its value—an integer in the closed interval [0, 65535]—as an unsigned binary number whose MSB and LSB are Bits 0 and 15, respectively:



The standard notation for error types uses the keyword ERROR to introduce the names, types, and (implicitly) order of the error's arguments. When a list of names precedes the specified type, that type is understood to be assigned to each of the specified arguments, which are distinct. The standard notation for error constants is simply a number:

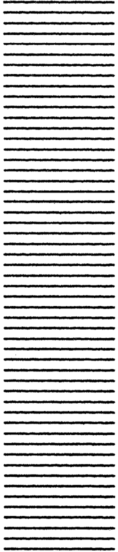
```
ErrorType      ::= ERROR ArgumentList
ErrorConstant  ::= number

ArgumentList   ::= empty | [ FieldList ]
FieldList      ::= Field | FieldList , Field
Field          ::= NameList : Type
NameList       ::= identifier | NameList , identifier
```

Example:

Given the declaration:

AccessDeniedError: TYPE = ERROR;

	value
3 is encoded as:	000003_8
	0003_{16}



4

Layer three: Messages

4.1 Introduction

At layer two, Courier defines an object stream, which carries a series of data objects, each of which is of one of several standard data types. At layer three, Courier defines a bi-directional, alternating *message stream*, depicted in Figure 4.1, which transports a series of *messages* in each direction between system elements. Each message represents either a request for service or a positive or negative reply to such a request.

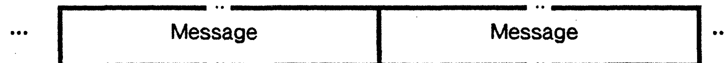


Figure 4.1 The message stream

The message stream is defined by imposing additional structure upon the data carried by the object stream. Each message is encoded as a single data object.

This section presents and gives examples of the format and use of each of the messages defined by Courier. Because messages are defined as data objects, the conventions employed in their definition are precisely the standard notation defined in Section 3. For the same reason, message encodings are the appropriate standard representations. This section also introduces a small amount of additional standard notation. In particular, conventions are defined below for declaring a remote program.

4.2 Program declarations

Every remote program is assigned a *program number*, which identifies it at run time. Program numbers are unique and unambiguous throughout all space and time. Every remote program is assigned exactly one number, and no two programs are assigned the same number. Once assigned, a program's number is never changed. The program number space is centrally administered; the procedures for obtaining a block of numbers are given in Appendix B.

Every remote program is also assigned a *program name*. A program's name facilitates written and verbal discussion of the protocol, but has no significance at run time. In particular, it never appears in the object stream. Program names are locally assigned by each development

organization and, therefore, are not globally unambiguous. Program numbers can always be used to disambiguate program names, but often it will be clear from context to what program a name refers.

Every remote program is further characterized by a *version number*, which distinguishes among successive versions of the program and helps to ensure at run time that caller and callee are agreed upon the calling sequences of the program's remote procedures. Each remote program has its own version number space, which is locally administered by the program's designer. The first version of a program is always numbered one. Whenever a program's declaration is changed in any way, its version number must be incremented by one.

A remote program consists of zero or more remote procedures and the zero or more remote errors they can raise. It is defined or *declared* in a higher-level protocol document as indicated below. A program declaration specifies the program's name (identifier) and its program and version numbers (number). It declares a procedure constant for each of the program's procedures, and declares an error constant for each of its errors. Within a program, no two procedures may have the same name or value, and no two errors may have the same name or value. Additional type and constant declarations may (but need not) be included to simplify or enhance the documentation:

```
Program                ::= identifier : PROGRAM number VERSION number =
                           BEGIN DependencyList DeclarationList END.

DependencyList         ::= empty | DEPENDS UPON ReferencedProgramList ;
ReferencedProgramList  ::= ReferencedProgram |
                           ReferencedProgramList , ReferencedProgram
ReferencedProgram      ::= identifier ( number ) VERSION number
DeclarationList        ::= empty | DeclarationList Declaration
Declaration            ::= TypeDecl | ConstantDecl
```

Appendix C generalizes the standard notation defined throughout this document to permit the declaration of one remote program to make use of types and constants defined in the declarations of other remote programs. As indicated above, a program declaration must list the name, number, and version number of every remote program upon which it depends in this way.

4.3 Message types

Courier defines four message types:

```
Message: TYPE = CHOICE OF {
  call(0)    => CallMessageBody,
  reject(1)  => RejectMessageBody,
  return(2)  => ReturnMessageBody,
  abort(3)   => AbortMessageBody};
```

The *call* message calls a remote procedure, that is, invokes a remote operation. The *reject* message rejects such a call, that is, reports an inability to even attempt a remote operation. The *return* message reports a procedure's return, that is, acknowledges the operation's successful completion. The *abort* message raises a remote error, that is, reports the operation's failure.

Only the user process may send a call message, and only the server process may send a reject, return, or abort message. A call message is sent whenever the user process desires service from the server process. A reject, return, or abort message is sent in eventual response to every call message. The user process is said to have a call *outstanding* between the time it sends a call message and the time it receives the acknowledging reject, return, or abort message. The user process may have at most one call outstanding at any point in time.

4.3.1 Call

The call message invokes, with the arguments supplied, the remote procedure whose program number, program version number, and procedure value are specified. The call message also contains a *transaction identifier*, which should have the value zero. Although not declared a procedure, the procedure value below is that assigned to the corresponding procedure constant in the program declaration:

```
CallMessageBody: TYPE = RECORD [
 transactionID: UNSPECIFIED,
 programNumber: LONG CARDINAL,
 versionNumber, procedureValue: CARDINAL,
 procedureArguments: RECORD [procedure-dependent]];
```

Fine point: The transaction identifier is unused. It is included to facilitate future expansion.

Fine point: Modeling procedure arguments, procedure results, and error arguments as record components requires the "*procedure-dependent*" and "*error-dependent*" departures from the standard notation that appear above and in following subsections. Strictly speaking, an argument or result list is more correctly modeled as a SEQUENCE OF UNSPECIFIED. However, so doing would require that implementations be able to determine the length of the entire list before outputting the first argument or result.

Example:

Given the declaration:

```
FileAccess: PROGRAM 13 VERSION 1 =

BEGIN
Credentials: TYPE = RECORD [user, password: STRING];
Mode: TYPE = {readPage(0), writePage(1), readAndOrWritePage(2)};
OpenFile: PROCEDURE [credentials: Credentials, filename: STRING, mode:
                    Mode]
 RETURNS [handle: UNSPECIFIED, pageCount: CARDINAL]
  REPORTS [... NoSuchFile, ...] = 0;
NoSuchFile: ERROR = 2;
...
END.
```

A call to procedure **OpenFile** with user "**White**", password "**vlw**", filename "**Data**", and mode **readPage** is encoded as:

```
call [
   transactionID: 0,
   programNumber: 13, versionNumber: 1, procedureValue: 0,
   procedureArguments: [
  credentials: [user: "White", password: "vlw"],
  filename: "Data",
  mode: readPage]];
```

4.3.2 Reject

The reject message rejects a call to a remote procedure, specifying the nature of the problem encountered. The reject message contains the transaction identifier specified in the rejected call message:

```
RejectMessageBody: TYPE = RECORD [
 transactionID: UNSPECIFIED,
 rejectionDetails: CHOICE OF {
  noSuchProgramNumber(0)      => RECORD [],
  noSuchVersionNumber(1)      => ImplementedVersionNumbers,
  noSuchProcedureValue(2),
  invalidArgument(3),
  unspecifiedError(FFFFH)     => RECORD []}];

ImplementedVersionNumbers: TYPE = RECORD [
 lowest, highest: CARDINAL];
```

The rejection code **noSuchProgramNumber** indicates that either the program number specified in the call message is invalid (that is, unassigned) or the program to which it is assigned is unimplemented by the system element.

The rejection code **noSuchVersionNumber** indicates that either the version number specified in the call message is invalid (that is, unassigned) or the program version to which it is assigned is unimplemented by the system element. If this rejection code is specified, the caller may infer that some version of the specified program, albeit not the one requested, is supported by the system element. Included in the reject message are the numbers of the lowest and highest versions of the program implemented by the system element. If only one version is implemented, the two numbers are the same. The caller may *not* assume that the system element implements *every* version in the specified range, but in practice that will usually be the case.

The rejection code **noSuchProcedureValue** indicates that the procedure value specified in the call message is invalid (that is, unassigned). If this rejection code is specified, the caller may infer that the specified version of the specified program is implemented by the system element.

The rejection code **invalidArgument** indicates that one (or more) of the procedure arguments specified in the call message is of incorrect type or fails to conform to its type's standard representation. If this rejection code is specified, the caller may infer that the specified version of the specified program is implemented by the system element.

The rejection code **unspecifiedError** indicates that the call message was rejected for a reason other than those described above.

Example:

The response to a call to an unsupported remote program is encoded as:

```
reject [
   transactionID: 0,
   rejectionDetails: noSuchProgramNumber []];
```

4.3.3 Return

The return message reports a remote procedure's return and supplies its results. The return message contains the transaction identifier specified in the call message that invoked the procedure:

```
ReturnMessageBody: TYPE = RECORD [
 transactionID: UNSPECIFIED,
 procedureResults: RECORD [ procedure-dependent ]];
```

Example:

Given the declaration and call of Section 4.3.1, a return with handle **16440B** and page count **511** is encoded as:

```
return [
 transactionID: 0,
 procedureResults: [
  handle: 16440B,
  pageCount: 511]];
```

4.3.4 Abort

The abort message raises, with the arguments supplied, the remote error whose error value is specified. The abort message contains the transaction identifier specified in the call message that invoked the procedure. Although not declared an error, the error value below is that assigned to the corresponding error constant in the program declaration:

```
AbortMessageBody: TYPE = RECORD [
 transactionID: UNSPECIFIED,
 errorValue: CARDINAL,
 errorArguments: RECORD [ error-dependent ]];
```

Example:

Given the declaration and call of Section 4.3.1, an abort with error **NoSuchFile** is encoded as:

```
abort [
 transactionID: 0,
 errorValue: 2,
 errorArguments: []];
```

A

Appendix A References

The following documents supplement this protocol specification. References [1-4] are informational; they contain helpful motivational and explanatory material, but Courier can be understood without them. References [5-6] are mandatory; they describe other protocols upon which Courier depends.

Reference [1] contains the data link and physical layer specifications for the Ethernet, the transmission medium for which Courier's standard representations are optimized:

[1] Digital Equipment Corporation; Intel Corporation; Xerox Corporation. The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications. 1980 September 30; Version 1.0.

Reference [2] provides an example of a data representation that is cleverly optimized to minimize space, rather than processing overhead:

[2] Haverty, Jack. MSDTP—Message Services Data Transmission Protocol. Arpa Network Working Group Request for Comments 713, NIC 34739. Laboratory for Computer Science, Massachusetts Institute of Technology; 1976 April 6.

Reference [3] defines Backus-Naur Form (BNF), the notation used throughout this document to formally define Courier's standard notation for the documentation of higher-level protocols:

[3] Naur, P., ed. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*. 6(1): 1-17; 1963.

Reference [4] recounts the protocol design experience that led to the remote procedure call model:

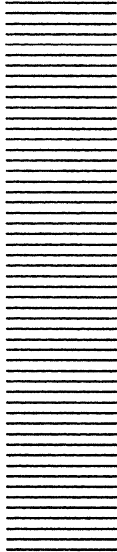
[4] White, James E. A High-Level Framework for Network-Based Resource Sharing. AFIPS Conference Proceedings, National Computer Conference. 45: 561-570; 1976.

Reference [5] defines the Sequenced Packet Protocol, upon which Courier relies for data transport:

[5] Xerox Corporation. Internet Transport Protocols. Xerox System Integration Standard. Stamford, Connecticut; 1981 December; XSIS-028112.

Reference [6] defines the NS Character Set and the NS String Format, which provide the basis for Courier's string data type:

[6] Xerox Corporation. NS Character Set Specification. Version 1.0; in preparation.

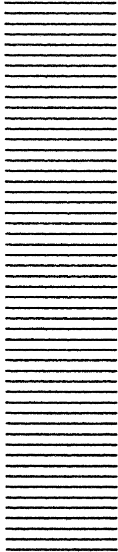


B

Appendix B
Program number assignment procedures

As stated in Section 4, every remote program is assigned a number that uniquely identifies it throughout the distributed system. The program number space is administered by Xerox Corporation. To obtain a block of program numbers, submit a written request to:

Xerox Corporation
Office Products Division
Network Systems Administration Office
3333 Coyote Hill Road
Palo Alto, California 94304



C

Appendix C
Standard notation summary

Because the details of parameter encodings and request and reply message formats are defined by Courier, higher-level application protocols based upon Courier can be specified at a very high level. The standard notation for the documentation of such protocols is defined in the body of this document and summarized below in compressed and reorganized form. Courier's standard notation is commonly referred to as the Courier *language.*

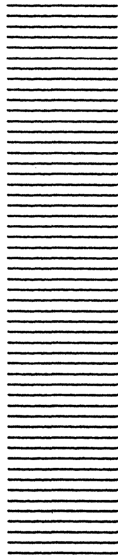
Two generalizations are introduced in the summary. First, the documentation of one remote program may reference a type (ReferencedType) or constant (ReferencedConstant) defined in the documentation of another (or the same) remote program. In such a reference, the name of the remote program and the name of the type or constant drawn from it are separated by a period ('.). Second, in most places where a numeric value (NumericValue) is required, either a number or a declared numeric constant may be supplied.

```
Program                ::= identifier : PROGRAM number VERSION number =
                           BEGIN DependencyList DeclarationList END.

DependencyList         ::= empty | DEPENDS UPON ReferencedProgramList ;
ReferencedProgramList  ::= ReferencedProgram | ReferencedProgramList , ReferencedProgram
ReferencedProgram      ::= identifier ( number ) VERSION number

DeclarationList        ::= empty | DeclarationList Declaration
Declaration            ::= identifier : TYPE = Type ; | identifier : Type = Constant ;

Type                   ::= PredefinedType | ConstructedType | ReferencedType

PredefinedType         ::= BOOLEAN | CARDINAL | LONG CARDINAL |
                           INTEGER | LONG INTEGER | STRING | UNSPECIFIED
ConstructedType        ::= { CorrespondenceList } | ARRAY NumericValue OF Type |
                           SEQUENCE MaximumNumber OF Type | RECORD [ FieldList ] | RECORD [ ] |
                           CHOICE DesignatorType OF { CandidateList } |
                           PROCEDURE ArgumentList ResultList ErrorList | ERROR ArgumentList
ReferencedType         ::= identifier | identifier . identifier

CorrespondenceList     ::= Correspondence | CorrespondenceList , Correspondence
Correspondence         ::= identifier ( NumericValue )

MaximumNumber          ::= NumericValue | empty
NumericValue           ::= number | ReferencedConstant

DesignatorType         ::= empty | ReferencedType
```

CandidateList	::=	Candidate | CandidateList , Candidate
Candidate	::=	DesignatorList => Type
DesignatorList	::=	Designator | DesignatorList , Designator
Designator	::=	identifier | Correspondence
ArgumentList	::=	empty | [FieldList]
ResultList	::=	empty | RETURNS [FieldList]
ErrorList	::=	empty | REPORTS [NameList]
FieldList	::=	Field | FieldList , Field
Field	::=	NameList : Type

Constant

Constant	::=	PredefinedConstant | ConstructedConstant | ReferencedConstant
PredefinedConstant	::=	TRUE | FALSE | number | - number | " string "
ConstructedConstant	::=	identifier | [ElementList] | [ComponentList] | [ ] | identifier Constant | number
ReferencedConstant	::=	identifier | identifier . identifier
ElementList	::=	Constant | ElementList , Constant
ComponentList	::=	Component | ComponentList , Component
Component	::=	NameList : Constant
NameList	::=	identifier | NameList , identifier

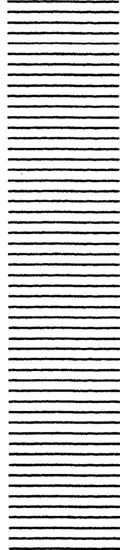


D

Appendix D
Sample application protocol

The standard notation summarized in Appendix C is illustrated below. The simple Courier-based application protocol defined below enables page-level access to remote files. Four remote procedures and ten remote errors constitute this particular, hypothetical remote program.

```
FileAccess: PROGRAM 13 VERSION 1 =

BEGIN
-- types and constants
Credentials: TYPE = RECORD [user, password: STRING];
Mode: TYPE = {readPage(0), writePage(1), readAndOrWritePage(2)};
PageContents: TYPE = ARRAY 256 OF UNSPECIFIED;

-- procedures
OpenFile: PROCEDURE [credentials: Credentials, filename: STRING, mode: Mode]
 RETURNS [handle: UNSPECIFIED, pageCount: CARDINAL] REPORTS [NoSuchUser,
  IncorrectPassword, NoSuchFile, AccessDenied, FileInUse, InvalidMode] = 0;

ReadPage: PROCEDURE [handle: UNSPECIFIED, pageNumber: CARDINAL]
 RETURNS [pageContents: PageContents] REPORTS [InvalidHandle, IncorrectMode,
  NoSuchPageNumber] = 1;

WritePage: PROCEDURE [handle: UNSPECIFIED, pageNumber: CARDINAL,
 pageContents: PageContents] REPORTS [InvalidHandle, IncorrectMode,
  FileTooLarge] = 2;

CloseFile: PROCEDURE [handle: UNSPECIFIED] REPORTS [InvalidHandle] = 3;

-- errors
NoSuchUser:          ERROR = 0;   -- user unrecognized by server; user unregistered
IncorrectPassword:   ERROR = 1;   -- password specified not that of specified user
NoSuchFile:          ERROR = 2;   -- filename unrecognized by server; file doesn't exist
AccessDenied:        ERROR = 3;   -- user unentitled to access file in specified mode
FileInUse:           ERROR [user: STRING] = 4; -- file already open for specified user
InvalidMode:         ERROR = 5;   -- invalid mode; not read..., write..., or
                                       readAndOrWritePage
InvalidHandle:       ERROR = 6;   -- invalid handle; perhaps obsoleted by CloseFile
IncorrectMode:       ERROR = 7;   -- requested operation inconsistent with open mode
NoSuchPageNumber:    ERROR = 8;   -- requested page unreadable; not present in file
FileTooLarge:        ERROR = 9;   -- requested page unwritable; file would be too large
END.
```



E

Appendix E
Sample protocol exchanges

The standard representation defined in Section 3 is illustrated below. The simple protocol exchanges shown assume the sample application protocol defined in Appendix D. The representation of each message is partitioned into 16-bit units whose values are specified in both octal and hexadecimal. The version number exchange described in Section 2.3 is assumed to have occurred previously and is not depicted here.

1. Call procedure OpenFile with user "White", password "vlw", filename "Data", and mode readPage:

call	tid	program#		ver#	proc	count	"Wh"	"it"	"e"
000000	000000	000000	000015	000001	000000	000005	053550	064564	062400_8
0000	0000	0000	000D	0001	0000	0005	5768	6974	6500_{16}

count	"vl"	"w"	count	"Da"	"ta"	readPg
000003	073154	073400	000004	042141	072141	000000_8
0003	766C	7700	0004	4461	7461	0000_{16}

Return with handle 16440B and pageCount 511:

return	tid	handle	pgCnt
000002	000000	016440	000777_8
0002	0000	1D20	$01FF_{16}$

2. Call procedure ReadPage with handle 16440B and pageNumber 15:

call	tid	program#		ver#	proc	handle	pgNum
000000	000000	000000	000015	000001	000001	016440	000017_8
0000	0000	0000	000D	0001	0001	1D20	$000F_{16}$

Return with pageContents [7602B, ... 54553B]:

return	tid	$word_1$	... $word_{256}$
000002	000000	007602	... 054553_8
0002	0000	0F82	... $596B_{16}$

3. Call procedure **CloseFile** with handle **16440B**:

call	tid	program#		ver#	proc	handle
000000	000000	000000	000015	000001	000003	016440_8
0000	0000	0000	000D	0001	0003	$1D20_{16}$

Return with no results:

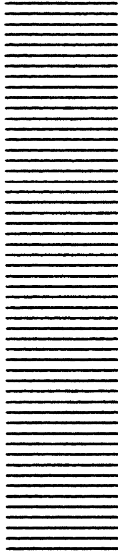
return	tid
000002	000000_8
0002	0000_{16}

4. Call procedure **CloseFile** with handle **16440B**:

call	tid	program#		ver#	proc	handle
000000	000000	000000	000015	000001	000003	016440_8
0000	0000	0000	000D	0001	0003	$1D20_{16}$

Abort with error **InvalidHandle**:

abort	tid	error
000003	000000	000006_8
0003	0000	0006_{16}

XEROX 

Xerox System Integration
Bulletin

OPD B018112
December 1981

NS Character Set Specification (Interim)

This specification is issued to allow users to implement the "Courier" and other Xerox NS protocols before the final NS Character Set Standard is released. This final document, now in preparation, will be an extension of this interim specification.

In most cases of interest, the NS Character Set is identical to the 8-bit Teletex character set defined by C.C.I.T.T. [B], except that it places the number sign ('#) and dollar sign ('$) in the same national use positions (2/3 and 2/4 respectively) assigned to them by ANSI [A], and leaves the C.C.I.T.T. positions for those characters empty. Consistent with the intent of the ISO standard [C], upon which the Teletex standard is based, this adjustment ensures that *the 7-bit graphic characters of the NS Character Set are precisely ASCII-standard.*

In most cases of interest, the NS String Format encodes a string as a series of zero or more 8-bit characters.

Together, the NS Character Set and String Format have the important property that *the overwhelming majority of ASCII, ISO, and C.C.I.T.T. strings are identical to the corresponding NS STRINGS.*

References:

[A] American National Standards Institute. *American National Standard Code for Information Interchange.* New York; 1977 June 9; ANSI X3.4-1977.

[B] Consultative Committee, International Telegraph and Telephone [C.C.I.T.T.]. *Recommendation S.61, Character Repertoire and Coded Character Sets for the International Teletex Service.* Geneva: International Telecommunications Union. v. 7 [yellow book]; 1980.

[C] International Organization for Standardization. *7-Bit Coded Character Set for Information Processing Interchange.* Geneva; 1973 July 1; First Edition, ISO 646-1973 (E).

Notice

This *Xerox System Integration Bulletin* describes the interim NS Character Set used by many application protocols in Xerox Network Systems.

1. This bulletin is furnished for informational purposes only. Xerox does not warrant or represent that this document or any products made in conformance with it will work in the intended manner or be compatible with other products in a network system. Xerox does not assume any responsibility or liability for any errors or inaccuracies that this document may contain, nor have any liabilities or obligations for any damages, including but not limited to special, indirect, or consequential damages, arising out of or in connection with the use of this document in any way.

2. No representations or warranties are made that this document, or anything made in accordance with it, is or will be free of any proprietary rights of third parties.

Copyright© 1981 Xerox Corporation
Stamford, Connecticut 06904
All Rights Reserved.

XEROX®, Xerox Network Systems, and NS
are trademarks of XEROX CORPORATION.

XEROX

XEROX

Courier: The Remote Procedure Call Protocol

Xerox System Integration Standard

XSIS-038112

Xerox Corporation
Stamford, Connecticut 06904

XEROX® is a trademark of
XEROX CORPORATION.

Printed in U.S.A.