Rational Networking—TCP/IP Reference Manual

Transport Layer (TRL)

Document Control Number: 8003A-02 (803-002331)

Rev. 1.0, November 1985
Rev. 2.0, July 1986
Rev. 3.0, July 1987 (Delta)

This document subject to change without notice.

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

How to Use This Book

The Transport Layer (TRL) book of the *Rational Networking—TCP/IP Reference Manual* describes the Ada® package specifications for the facilities provided by the Rational Networking—TCP/IP implementation of the Transport protocol. This book is intended for users who are familiar with the Rational Environment™ and with Ada programming.

Organization of the Networking Manual

The *Rational Networking—TCP/IP Reference Manual* (Networking Manual for brevity) includes the following volumes:

1	Telnet (TEL) File Transfer Protocol (FTP)
2	Transport Layer (TRL) Remote Procedure Call (RPC)

Each *volume* of the Networking Manual contains two *books* separated by large colored tabs. Each book contains information on particular features or areas of application in networking. The abbreviation for the name of each book (for example, TEL for Telnet) appears on the binder cover and spine, and this abbreviation is used in page numbers and cross-references. The books grouped into a given volume are logically related.

The Networking Manual provides reference information organized to efficiently answer specific questions about the Rational Networking product. Products other than the Networking product are documented in individual manuals (for example, the *Rational Environment Reference Manual* or the *Rational Target Build Utility Reference Manual*).

Volume 1

Volume 1 documents the commands used in the Rational Networking product. This volume contains the following two books:

- **Telnet**: The Telnet (TEL) book contains the commands used to establish and terminate connections to a remote host through a terminal logged into a local host.
- **File Transfer Protocol**: The File Transfer Protocol (FTP) book contains the commands used to transfer text and binary data files between two hosts. Packages File_Transfer, Ftp_Product, and Transfer_Generic can be used to develop programmatic interfaces for FTP.

Volume 1 also contains a Master Index that combines the index information from all four books of the Networking Manual.

Volume 2

Volume 2 documents the packages provided to develop new programmatic interfaces. This volume contains the following two books:

- **Transport Layer**: The Transport Layer (TRL) book describes the concepts and interfaces used to build networking tools for specific applications.
- **Remote Procedure Call**: The Remote Procedure Call (RPC) book describes the concepts and interfaces used to write clients and servers for remote procedure calls, in which an application running on one host can make procedure calls to applications running on different hosts.

Volume 2 also contains a Master Index that combines the index information from all four books of the Networking Manual.

Book Organisation

Each book begins with a colored tab on which the name of the book appears. Each book typically contains the following sections:

- **Key Concepts**: This section describes the key concepts that pertain to the networking facilities documented in the book. The section is located behind its own tab following the How to Use This Book section.
- **Unit sections**: Each of the commands, tools, and so on has a declaration within an Ada compilation unit (typically a package). For each unit, there is a section that contains reference entries for the declarations within that unit. Each section is preceded by a tab.

  The sections for units are alphabetized by the simple names of the units. For example, the section for package !Tools.Network.Revn.Units.Network_Product is alphabetized under Network_Product.

  For many units, introductory material and/or examples specific to the unit appear after the section tabs.

  Within the section of a given unit, the reference entries describing the unit's declarations are organized alphabetically after the section introduction. Appearing at the top of each page of a reference entry are the simple name of the name of the given declaration and the fully qualified pathname of the enclosing unit.
- **Index**: Preceded by a tab, the Index appears as the last section of each book. It contains entries for each unit or declaration, along with additional topical

references. Each book index covers only the material documented in that particular book. Each volume contains a Master Index that provides entries for the information documented in all the books within the Networking Manual.

Italic page numbers indicate the page on which the primary reference entry for a declaration appears; nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Suggestions for Finding Information

The following suggestions can help you find various kinds of information in the documentation for Rational's products.

Learning about New Facilities

If you are a novice user starting to use the Rational Environment, consult the *Rational Environment User's Guide.*

If you are familiar with the Rational Environment but are interested in learning about certain networking commands, for example, you might start by reading the Key Concepts for the specific book, which describes important concepts and gives helpful examples.

It can also be useful to glance through the introductions provided for some of the units in the book. These introductions, located immediately after the tabs for the units, often contain helpful examples.

Finding Information on a Specific Item

If you know the name of the item and the book in which it is documented, consult either the table of contents or the index for that book. You can also turn through the pages of the book using the names and pathnames of the reference entries to locate the entry that you want. Remember that the reference entries for a unit are organized alphabetically by simple name within a tabbed section.

If you know the simple name of the entry but do not know the book in which it is documented, look in the Master Index to find the book abbreviation and page number.

If you cannot find an item in the Master Index, the item either is not documented or is documented in manuals for a product other than the Rational Networking—TCP/IP product (for example, the Rational Environment or the Rational Target Build Utility). If you know the pathname, consult the World ! section of the Reference Summary in Volume 1 of the *Rational Environment Reference Manual* to determine whether the item is documented and in which manual.

Using the Index

The index of each book contains entries for each unit and its declarations, organized alphabetically by simple name. When using the index to find a specific item, consult the italic page number for the primary reference to that item. Nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Viewing Specifications On-Line

If you know the pathname of a declaration and want to see its specification in a window of the Rational Environment, provide its pathname to the !Commands.Common.Definition procedure—for example, `Definition ("!Tools.Ftp.Revn.Units-.Commands.Ftp");`. If you know the simple name of the unit in which the declaration appears, in most cases you can use searchlist naming as a quick way of viewing the unit—for example, `Definition ("\Ftp");`.

Using On-Line Help

Most of the information contained in the reference entries for each unit is available through the on-line help facilities of the Rational Environment. Press the Help on Help key or consult the *Rational Environment User's Guide* or the *Rational Environment Reference Manual*, EST, Help, for more information on using this on-line help facility.

Cross-Reference Conventions

The following conventions are used in cross-references to information:

- **Specific page/book**: For references to a specific place in a book, the book abbreviation is followed by the page number in the book (for example, RPC-23). If the book abbreviation is omitted, the current book is implied (for example, the page numbers in the table of contents for a book do not include the book prefix).
- **Declaration in same unit**: References to the documentation for a declaration in the same unit are indicated by the simple name of the desired declaration. For example, within the reference entry for the Ftp.Connect procedure, a reference to the Ftp.Disconnect procedure is "procedure Disconnect." Note that if the unit contains nested packages, references to nested declarations use qualified pathnames.
- **Declaration in different unit, same book**: References to the documentation for a declaration in another unit are indicated by the qualified pathname of the desired declaration. For example, within the reference entry for the Ftp.Connect procedure, a reference to the Ftp_Profile.Auto_Login function is "function Ftp_Profile.Auto_Login."
- **Declaration in different book**: References to the documentation for a declaration in another book are indicated by the addition of the abbreviation for that book. For example, within the reference entry for the Ftp.Connect procedure, a reference to the Transport.Connect procedure in the Transport Layer book would be "TRL, procedure Transport.Connect."

Feedback to Rational: Reader's Comments Form

Rational wants to make its documentation as useful and error-free as possible. Please provide us with feedback. The last page of each book contains a Reader's Comments form that you can use to send us comments or to report errors. You can also submit problem reports and make suggestions electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

RATIONAL

Key Concepts

The Rational Networking—TCP/IP product forms an intersystem interface for host/target development. Networking facilitates:

- Downloading Ada source code and target object code
- Debugging interactively
- Controlling target execution
- Updating target state

Transport (TCP/IP Ethernet) provides for byte-stream data transfer when it sets up connections serving as reliable data pipelines between two Rational R1000s or a Rational R1000 and another system. Transport does the work of transferring the data through these connections.

This book is designed for advanced users who may want to build other networking tools for their own applications. It discusses the basic concepts of the Rational implementation of the TCP/IP data transfer protocol. These concepts include:

- Host, connection, and socket identifiers
- Active and passive data connections

Host Names

Each host machine is identified by both a string name and an address. Host names and addresses are maintained in the database provided by package Transport_Name. Transport_Name provides mappings between the TCP/IP network addresses and the more familiar host string names.

Currently, the database for package Transport_Name is a text file in !Machine-.Transport_Name_Map. Each line of the file contains the network name, the host identifier, the host name, and the machine type. Sample entries are:

```
TCP/IP  89.4.27.1  Clem   Rational

TCP/IP  89.4.8.43  Logo   VMS
```

To change the database, change the entry in this file using the Rational Editor, !Commands.Library.Copy, !Commands.Archive.Restore, or FTP.

Connection and Socket Identifiers

A data *connection* is the first item built by the Transport Layer for interaction between computers. A connection is a bidirectional data pipeline set up between a pair of data ports, called *sockets*, one on each machine. The connection "ties up" the sockets only while the connection is being established. After the connection is made, the sockets are released so additional programs can connect on them.

To establish a connection, a program on machine A passively waits on a defined socket. A program on machine B is situated on an arbitrary socket and connects to the program on the waiting socket of machine A.

The Connection_Id uniquely identifies the connection so the higher-level tools can access it.

Active and Passive Connects

A connection is established by a passive connect, initiated by one machine, being followed by an active connect from the other machine to the same Socket_Id. The active and passive connects represent two halves that join together as a connection. Active and passive connects differ in that the passive connect is made first and the active connect is made only to an existing program waiting passively. Once the connection is completed, it is no longer considered active or passive, and either side can transmit and receive data in either direction. Either machine can disconnect a transport connection.

A general state diagram for the Transport Layer is shown in Figure 1-1.

Connection Ownership

A job terminates when all the tasks elaborated by the job terminate. Thus, a job can be unlimited in size. While a connection is being used by a job, the connection and all its resources are considered to be "owned" by that job. Upon job termination, the connection and its associated resources are released. If a job fails to terminate, the connection resources are tied up until the user enters a !Commands.Job.Kill command to terminate the job.

Pretransmission Buffering

After a request is made for data transfer, the request will return. The data may be transmitted immediately or, if there is a problem on the network (noise on the line, receiving computer accepting data too slowly, and so on), stored in a buffer for a short time until the problem goes away and transmission completes. The data size may exceed the buffer size, so the buffered transmission may have to proceed as several smaller transfers.

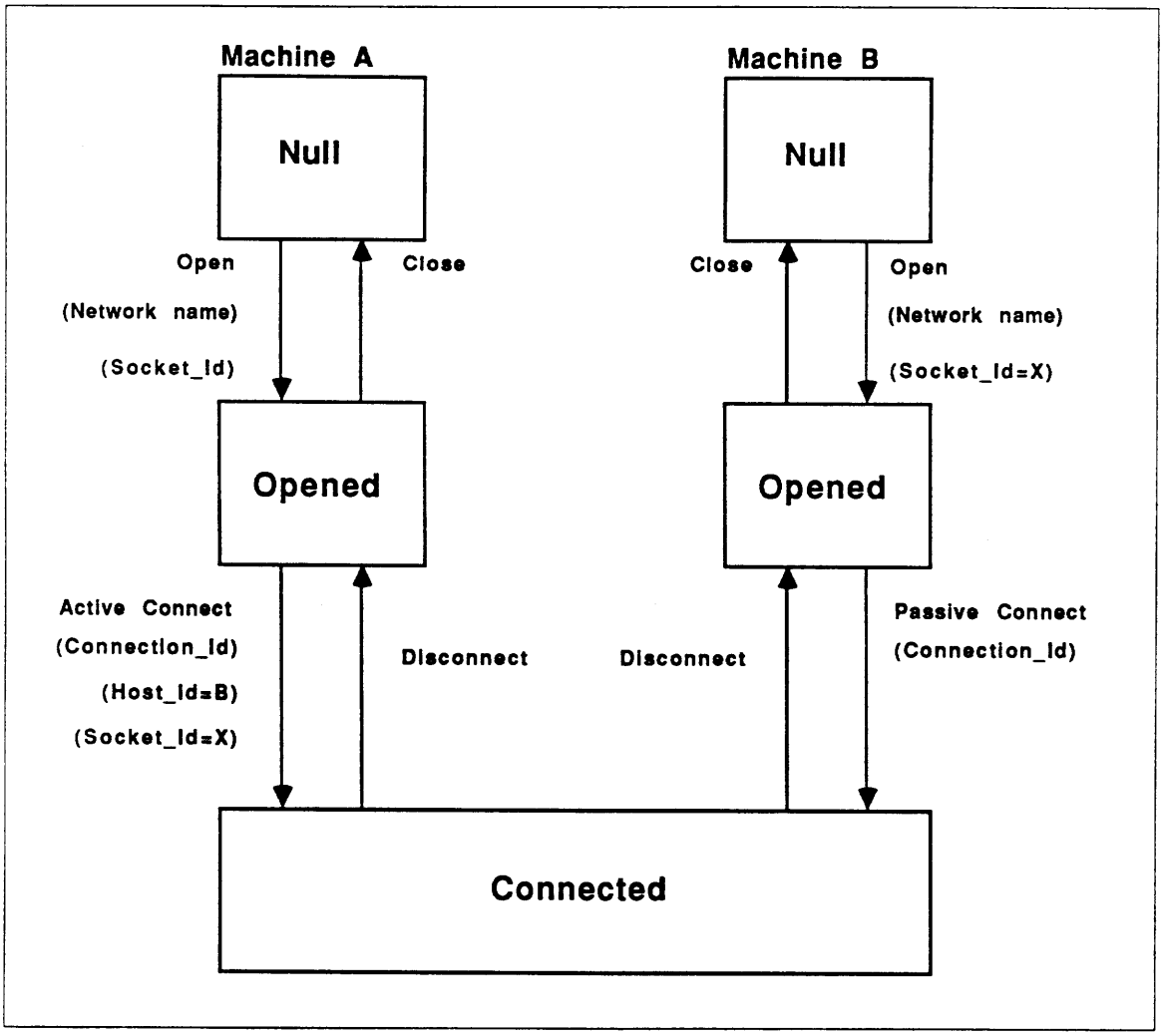

Figure 1-1. Transport Layer State Diagram

RATIONAL

package Byte_Defs

```
subtype Byte is System.Byte;

subtype Byte_String is System.Byte_String;

function "Mod" (X : Byte;
                Y : Byte) return Byte renames System."Mod";

function "Rem" (X : Byte;
                Y : Byte) return Byte renames System."Rem";

function ">" (X : Byte;
              Y : Byte) return Boolean renames System.">";
function ">" (X : Byte_String;
              Y : Byte_String) return Boolean renames System.">";

function "=" (X : Byte;
              Y : Byte) return Boolean renames System."=";
function "=" (X : Byte_String;
              Y : Byte_String) return Boolean renames System."=";

function ">=" (X : Byte;
               Y : Byte) return Boolean renames System.">=";
function ">=" (X : Byte_String;
               Y : Byte_String) return Boolean renames System.">=";

function "<" (X : Byte;
              Y : Byte) return Boolean renames System."<";
function "<" (X : Byte_String;
              Y : Byte_String) return Boolean renames System."<";

function "/" (X : Byte;
              Y : Byte) return Byte renames System."/";

function "*" (X : Byte;
              Y : Byte) return Byte renames System."*";

function "-" (X : Byte;
              Y : Byte) return Byte renames System."-";

function "<=" (X : Byte;
               Y : Byte) return Boolean renames System."<=";
function "<=" (X : Byte_String;
               Y : Byte_String) return Boolean renames System."<=";

function "+" (X : Byte;
              Y : Byte) return Byte renames System."+";

function "&" (X : Byte_String;
              Y : Byte_String) return Byte_String renames System."&";
function "&" (X : Byte;
              Y : Byte_String) return Byte_String renames System."&";
function "&" (X : Byte_String;
              Y : Byte) return Byte_String renames System."&";
```

```
function "&" (X : Byte;
              Y : Byte) return Byte_String renames System."&";
```

Description

Renames the System.Byte, the System.Byte_String, and the standard operations defined on these types.

The purpose of this package is to simplify porting applications software written on top of the Transport Layer to other (non-Rational) environments, where these types might be declared somewhere other than in package System. Networking software uses these definitions, not those in System. When porting, you need to modify only package Byte_Defs to make use of an alternative declaration.

```
subtype Byte is System.Byte;
```

Specifies an eight-bit byte.

In portable Ada, this is declared as:

```
Type Byte is range 0 .. 255;
```

```
subtype Byte_String is System.Byte_String;
```

Specifies an unconstrained array of bytes.

In portable Ada, this is declared as:

```
Type Byte_String is array (Integer range <>) of Byte;
pragma Pack (Byte_String);
```

```
function "Mod" (X : Byte;
                Y : Byte) return Byte renames System."Mod";
```

Specifies the standard modulus operator.

```
function "Rem" (X : Byte;
                Y : Byte) return Byte renames System."Rem";
```

Specifies the standard remainder operator.

```
function "=" (X : Byte;
              Y : Byte) return Boolean renames System."=";
function "=" (X : Byte_String;
              Y : Byte_String) return Boolean renames System."=";
```

Specifies the standard equality operators.

```
function ">" (X : Byte;
              Y : Byte) return Boolean renames System.">";
function ">" (X : Byte_String;
              Y : Byte_String) return Boolean renames System.">";

function "<" (X : Byte;
              Y : Byte) return Boolean renames System."<";
function "<" (X : Byte_String;
              Y : Byte_String) return Boolean renames System."<";

function ">=" (X : Byte;
               Y : Byte) return Boolean renames System.">=";
function ">=" (X : Byte_String;
               Y : Byte_String) return Boolean renames System.">=";

function "<=" (X : Byte;
               Y : Byte) return Boolean renames System."<=";
function "<=" (X : Byte_String;
               Y : Byte_String) return Boolean renames System."<=";
```

Specifies the standard comparison operators.

```
function "+" (X : Byte;
              Y : Byte) return Byte renames System."+";
```

Specifies the standard addition operator.

```
function "-" (X : Byte;
              Y : Byte) return Byte renames System."-";
```

Specifies the standard subtraction operator.

```
function "*" (X : Byte;
              Y : Byte) return Byte renames System."*";
```

Specifies the standard multiplication operator.

```
function "/" (X : Byte;
              Y : Byte) return Byte renames System."/";
```

Specifies the standard division operator.

```
function "&" (X : Byte_String;
              Y : Byte_String) return Byte_String renames System."&";
function "&" (X : Byte;
              Y : Byte_String) return Byte_String renames System."&";
function "&" (X : Byte_String;
              Y : Byte) return Byte_String renames System."&";
function "&" (X : Byte;
              Y : Byte) return Byte_String renames System."&";
```

Specifies the standard catenation operators.

---

package Host_Id_Io

Package Host_Id_Io provides commands that allow you to convert a Host_Id to a displayable string. The displayable string can be written either to Current_Output or to a specified file. The displayable output can be read from Current_Input or from a specified file and converted back into a Host_Id. For example, Host_Id (12,11,32,57) is converted to 12.11.32.57.

function Get

```
function Get return Transport_Defs.Host_Id;
```

Description

Reads a Host_Id from the Current_Input file.

The file should contain a Host_Id in symbolic form.

The Host_Id is returned.

Parameters

`return Transport_Defs.Host_Id;`
Returns Transport_Defs.Host_Id.

Example

The following example uses the Get function to read a Host_Id value from the Current_Input file. The value is assigned to Item. The Put procedure then writes the Host_Id to the Current_Output file.

```
with Host_Id_Io;
with Text_Io;
with Transport_Defs;
procedure Convert_Host_Ids is
begin
   declare
      Item : Transport_Defs.Host_Id := Host_Id_Io.Get;
   begin
      Text_Io.New_Line;
      Host_Id_Io.Put (Item);
   end;
   Text_Io.New_Line;
end Convert_Host_Ids;
```

function Get

```
function Get (File : Text_Io.File_Type) return Transport_Defs.Host_Id;
```

Description

Reads a Host_Id from the Current_Input file.

The file should contain a Host_Id in symbolic form.

The Host_Id is returned.

Parameters

`File : Text_Io.File_Type;`
Specifies the handle for a file.

`return Transport_Defs.Host_Id;`
Returns Transport_Defs.Host_Id.

Example

The following example uses the Get function to read a Host_Id value from the input file specified by the file handle. The file is opened, the Convert_Host_Ids procedure writes the Host_Id to the Current_Output file, and then the file is closed.

```
with Host_Id_Io;
with Text_Io;
with Transport_Defs;
procedure Convert_Host_Ids is
   Host_Id_Handle : Text_Io.File_Type;
   Host_Id_File_Name : Constant_String := !Users.Wjh.Host_Id_File;
begin
   Text_Io.Open (Host_Id_Handle,
                 In,
                 Host_Id_File_Name);
   Host_Id_Io.Put (Host_Id_Io.Get (Host_Id_Handle));
   Text_Io.Close(Host_Id_Handle);
end Convert_Host_Ids;
```

function Image

```
function Image (Item : Transport_Defs.Host_Id) return String;
```

Description

Converts the item into a human-readable string.

The format of the returned string is: ##.##.##.##

Parameters

`Item : Transport_Defs.Host_Id;`
Specifies the Host_Id to be converted to a string.

`return String;`
Returns the specified Host_Id as a string.

Example

The following program uses the Image function to convert the byte-string value of Clem into a human-readable form. The human-readable form is then assigned to Item. The !Io.Text_Io.Put_Line procedure writes Item to the Current_Output file.

```
with Host_Id_Io;
with Text_Io;
with Transport_Defs;
procedure Convert_Host_Ids is
   Clem : Transport_Defs.Host_Id := (0,14,31,86);
begin
   declare
      Item : Transport_Defs.Host_Id := Host_Id_Io.Image (Clem);
   begin
      Text_Io.Put_Line (Item);
   end;
   Text_Io.New_Line;
end Convert_Host_Ids;
```

procedure Put

```
procedure Put (Item : Transport_Defs.Host_Id);
```

Description

Displays a symbolic representation of the specified Host_Id.

This procedure writes to the Current_Output file (usually an output window).

Parameters

```
Item :  Transport_Defs.Host_Id;
```

Specifies the Host_Id to be written.

Example

The following program assigns Clem a byte-string value. The Put procedure writes the string to the Current_Output file.

```
with Host_Id_Io;
with Text_Io;
with Transport_Defs;
procedure Convert_Host_Ids is
   Clem : Transport_Defs.Host_Id := (0,14,31,86);
begin
   Text_Io.New_Line;
   Host_Id_Io.Put (Clem);
   Text_Io.New_Line;
end Convert_Host_Ids;
```

procedure Put

```
procedure Put (File : Text_Io.File_Type;
               Item : Transport_Defs.Host_Id);
```

Description

Displays a symbolic representation of the specified Host_Id.

This procedure writes the value of Item to the specified output file.

Parameters

`File : Text_Io.File_Type;`
Specifies the handle for a file to which the Host_Id is written.

`Item : Transport_Defs.Host_Id;`
Specifies the Host_Id to be written.

Example

The following program assigns Clem a byte-string value. The Put procedure writes the string to the output file that was opened for output.

```
with Host_Id_Io;
with Text_Io;
with Transport_Defs;
procedure Convert_Host_Ids is
   Clem : Transport_Defs.Host_Id := (0,14,31,86);
   Host_Id_Handle : Text_Io.File_Type;
   Host_Id_File_Name : Constant_String := !Users.Wjh.Host_Id_File;
begin
   Text_Io.Open (Host_Id_Handle,
                 Out,
                 Host_Id_File_Name);
   Text_Io.Put  (Host_Id_Io.Image(Clem));
   Text_Io.Close(Host_Id_Handle);
end Convert_Host_Ids;
```

end Host_Id_Io;

package Network

Package Network provides interactive commands entered through Command windows and allows user queries about the transport connections and hosts.

procedure Close_All

```
procedure Close_All;
```

Description

Closes all open transport connections.

This procedure calls the Transport.Close procedure on all currently open transport connections. A listing of the connections closed is written into the Current_Output file (using package !Io.Text_Io). For each connection, various key items of information (obtained from package Transport) are displayed.

This is a fairly dangerous operation. If other people on your machine are using transport connections, their work will be rudely interrupted by calling this procedure. Use it with care.

References

procedure Transport.Close

type Transport.Connection_Id_Iterator

procedure Show

```
procedure Show;
```

Description

Shows all open transport connections.

This procedure displays a listing of all currently open transport connections to the Current_Output file (using package !Io.Text_Io). For each connection, various key items of information (obtained from package Transport) are displayed.

References

type Transport.Connection_Id_Iterator

procedure Show_Host

```
procedure Show_Host (Host_Name : String := "");
```

Description

Shows the address of the specified host.

This procedure selects the host specified and displays various key items of information (obtained from package Transport_Name).

Parameters

```
Host_Name :  String := "";
```

Specifies the name of a host in the network. The default, "", indicates that no host is defined.

Example

In the following two examples, the first shows the output from a call to the Show_Host procedure when the default is used. The second example shows the output from a call to the Show_Host procedure when a host name is specified.

```
with Network;
begin
    Show_Host;
end;

"" is undefined (Transport_Name.Undefined).

with Network;
begin
    Show_Host ("logo");
end;

logo  TCP/IP  89.64.1.69
```

References

type Transport_Name.Host_Iterator

procedure Show_Hosts

```
procedure Show_Hosts;
```

Description

Shows the names and addresses of all known hosts.

This procedue iterates over all hosts in the database managed by package Transport_Name—that is, all hosts whose names are known by this system. For each host, various key items of information (obtained from package Transport_Name) are displayed.

By default, the display is written to the current output window.

References

type Transport_Name.Host_Iterator

procedure Time

```
procedure Time (From_Host : in String := "");
```

Description

Displays the time of day, as reported by the given host.

This procedure initiates a connection to the time server on the given host, receives the time of day, converts it to local time, and writes it into the Current_Output file (using package !Io.Text_Io).

Parameters

```
From_Host :  in String := "";
```

Specifies the name of a host in the network. The default, "", indicates that no host is defined.

Errors

If the default, "", is used (host is not known), the network is not up, or no time server is running on the host, various combinations of error messages and exceptions occur.

References

package Transport

package Transport_Name

end Network;

package Network_Product

Package Network_Product provides a way to check whether the Transport Layer utilities have been installed on your machine.

function Is_Installed

```
function Is_Installed return Boolean;
```

Description

Returns true if the Transport Layer utilities have been installed on your machine.

exception Is_Not_Installed

```
Is_Not_Installed : exception;
```

Description

Occurs when a Transport Layer subprogram is called if the Networking product is not installed on your machine.

end Network_Product;

RATIONAL

procedure Tcp_Ip_Boot

Procedure Tcp_Ip_Boot provides the Rational system the necessary information to boot the TCP/IP hardware controller board.

procedure Tcp_Ip_Boot

```
procedure Tcp_Ip_Boot
              (Use_Arp            : Boolean := True;
               Enable_Link_Level  : Boolean := True;
               Exos_Prefix        : String  := "!Tools.Network.";
               Host_Id_File       : String  := "!Machine.Tcp_Ip_Host_Id";
               Ether_Id_File      : String  := "!Machine.Ethernet_Host_Id";
               Use_Checksums      : Boolean := True;
               Diagnostic         : Boolean := False);
```

Description

Provides the Rational machine the necessary information to boot the TCP/IP board.

Parameters

`Use_Arp : Boolean := True;`

Specifies whether the Address Resolution Protocol (ARP) is used when TCP/IP is booted. The default value is true.

`Enable_Link_Level : Boolean := True;`

Specifies whether the TCP/IP controller sends Ethernet packages. The default value is true.

`Exos_Prefix : String := "!Tools.Network.";`

Specifies the prefix that is added to !Tools.Network to specify the file containing information about the Exos controller. The default is "!Tools.Network.".

`Host_Id_File : String := "!Machine.Tcp_Ip_Host_Id";`

Specifies the file that contains the Internet address of this specific machine. The file contains the address in decimal.dotted notation. The default is "!Machine.Tcp-_Ip_Host_Id".

`Ether_Id_File : String := "!Machine.Ethernet_Host_Id";`

Specifies the file that contains the Ethernet address of this specific machine. The file contains the address in decimal.dotted notation. If the file does not exist, the Ethernet address is taken from the address in PROM. The default is "!Machine.Tcp-_Ip_Host_Id".

```
Use_Checksums : Boolean := True;
```

Specifies whether the TCP checksum is used in addition to the Ethernet checksum. The default is true.

```
Diagnostic : Boolean := False;
```

Specifies whether the TCP_IP_Boot diagnostics are used. If true, the controller follows the standard booting procedures but does not actually boot the network. The default is false.

RATIONAL

package Transport

This package provides reliable, connection-oriented data communication between machines. In terms of the International Standards Organization Open Systems Interconnection reference model, package Transport is the service interface to the Transport Layer.

The underlying protocol is currently TCP/IP, running on an Ethernet local area network. In the future, this package may provide access to X.25 or other protocols that provide a transport service.

Transport services are based on the notions of a *connection* and a *socket.* A connection is a reliable data pipe with two ends, usually in different machines. A socket is a location at which connections are established.

Two programs must cooperate to form a connection. One program (the passive program) waits for incoming connections on some socket. The other program (the active program), usually on another machine, initiates a connection to the same socket. A connection thus is formed between the two programs, and they can transmit data to each other. Once connected, it does not matter which program was active and which was passive; either one can transmit data or disconnect the connection at any time.

In each machine, only one program can be waiting for a passive connect on each socket at a time. There can be many programs simultaneously waiting on different sockets.

In TCP/IP, both ends of each connection are associated with sockets. The two sockets do not have to be the same, and they usually are not.

procedure Close

```
procedure Close (Connection : Transport.Connection_Id);
```

Description

Closes the given connection.

The underlying resources associated with the connection are released and made available for use by other connections.

If the given connection is connected, it is disconnected. If the given connection is already closed or has never been opened, this procedure does nothing.

Parameters

`Connection : Transport.Connection_Id;`
Specifies the connection to be closed.

References

procedure Close_All

procedure Open

procedure Close_All

```
procedure Close_All (Owner : Machine.Job_Id);
```

Description

Closes all connections owned by a job.

Each connection is owned by some job. When a job terminates, all connections owned by it are automatically closed (in effect, the Close_All procedure is called whenever a job terminates).

Parameters

`Owner : Machine.Job_Id;`
Specifies a job that may own connections.

References

procedure Close

function Get_Owner

procedure Set_Owner

procedure Connect

```
procedure Connect
        (Connection     :     Transport.Connection_Id;
         Status         : out Transport_Defs.Status_Code;
         Remote_Host    :     Transport_Defs.Host_Id;
         Remote_Socket  :     Transport_Defs.Socket_Id;
         Max_Wait       :     Duration                    := Duration'Last);

procedure Connect
           (Connection  :     Transport.Connection_Id;
            Status      : out Transport_Defs.Status_Code;
            Max_Wait    :     Duration                    := Duration'Last);
```

Description

Connects to some remote machine.

The first form of this procedure (which specifies Remote_Host and Remote_Socket parameters) is an active connect; that is, it initiates a connection to a particular remote socket.

The second form (which does not specify a Remote_Host or a Remote_Socket parameter) is a passive connect; that is, it waits for a connection to arrive on the local socket associated with the specified Connection_Id.

Parameters

```
Connection : Transport.Connection_Id;
```

Specifies the local connection object used to establish the connection. If the connection is not open, the Connect procedure returns the status value Transport_Defs.Not_Open.

```
Status : out Transport_Defs.Status_Code;
```

Returns the outcome of the operation. Possible values are:

- Transport_Defs.Connection_Refused (active connect only)
- Transport_Defs.No_Free_Memory
- Transport_Defs.No_Local_Resources
- Transport_Defs.No_Such_Host (active connect only)
- Transport_Defs.Not_Open
- Transport_Defs.Ok
- Transport_Defs.Timed_Out
- Transport_Defs.Too_Many_Clients

```
Remote_Host : Transport_Defs.Host_Id;
```

Specifies the host to which the connection is to be formed for an active connect.

```
Remote_Socket : Transport_Defs.Socket_Id;
```

Specifies the socket to which the connection is to be formed for an active connect.

```
Max_Wait : Duration := Duration'Last;
```

Specifies the maximum amount of time to wait. If this period of time expires and no connection has been formed, the Connect procedure returns the status value Transport_Defs.Timed_Out. TCP/IP ignores this parameter and sets its own timeout value.

References

procedure Disconnect

---

type Connection_Id

```
type Connection_Id is private;
```

Description

Specifies the resources in the machine associated with a transport connection.

This type denotes various resources that are required to build and maintain a transport connection.

References

procedure Close

procedure Connect

procedure Disconnect

procedure Open

type Connection_Id_Iterator

```
type Connection_Id_Iterator is limited private;
```

Description

Denotes an iterator over all open Connection_Ids on all networks to which the local machine is currently connected.

If you want Connection_Ids on only one network, you can determine the network associated with each Connection_Id produced by this iterator and ignore the ones you do not want.

References

type Connection_Id

function Done

procedure Init

procedure Next

function Value

procedure Disconnect

```
procedure Disconnect (Connection : Transport.Connection_Id);
```

Description

Disconnects the specified connection.

If the specified connection is not connected, the Disconnect procedure does nothing.

Parameters

```
Connection : Transport.Connection_Id;
```

Specifies the connection that is to be disconnected.

References

procedure Connect

function Done

```
function Done (Iter : Network_Name_Iterator) return Boolean;

function Done (Iter : Connection_Id_Iterator) return Boolean;
```

Description

Indicates that the given iterator is done.

An iterator is done when it has been moved past the last element in the collection over which it iterates. If function Done (Iter) is true, function Value (Iter) is undefined.

Parameters

`Iter : Network_Name_Iterator;`
Specifies an iterator.

`Iter : Connection_Id_Iterator;`
Specifies an iterator.

`return Boolean;`
Returns true if the iterator is done.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

function Done
package !Tools.Networking.Transport

References

procedure Init

procedure Next

function Value

---

function Get_Owner

```
function Get_Owner (Connection : Transport.Connection_Id)
                                                    return Machine.Job_Id;
```

Description

Returns the owner of the connection.

Each connection is owned by some job. Whenever a job terminates, all connections owned by it are closed automatically. By default, a connection is owned by the job that opens it.

Parameters

`Connection : Transport.Connection_Id;`

Specifies a connection.

`return Machine.Job_Id;`

Returns the owner of the connection or, if the connection is not open, Machine.Nil_Job_Id.

References

procedure Close_All

procedure Set_Owner

function Hash

```
function Hash (Connection : Transport.Connection_Id) return Natural;
```

Description

Calculates a number suitable for indexing into a hash table.

The same connection always hashes to the same number. Different connections tend to hash to different numbers.

Parameters

`Connection : Transport.Connection_Id;`
Specifies a connection.

`return Natural;`
Returns a number derived from the connection. This number (modulo the size of your hash table) can be used as a hash table index.

Example

The following code fragment returns the number resulting from hashing the connection. That number (modulo the size of the hash table) is used as the hash table index for The_Bucket.

```
declare
    subtype Index is Natural range 0 .. Table_Size - 1;
Table : array (Index) of Bucket;
begin
    The_Bucket := Table (Hash (X) mod Table'length);
end;
```

procedure Init

```
procedure Init (Iter : in out Network_Name_Iterator);

procedure Init (Iter : in out Connection_Id_Iterator);
```

Description

Initializes an iterator to the first element of a collection.

Parameters

```
Iter : in out Network_Name_Iterator;
```
Specifies an iterator.

```
Iter : in out Connection_Id_Iterator;
```
Specifies an iterator.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

References

function Done

procedure Next

function Value

function Is_Connected

```
function Is_Connected (Connection : Transport.Connection_Id)
                                                    return Boolean;
```

Description

Determines whether the specified connection is connected.

If the connection is closed or has never been opened, a value of false is returned.

Parameters

`Connection : Transport.Connection_Id;`
Specifies a connection.

`return Boolean;`
Returns true if the specified connection is connected.

References

procedure Connect

procedure Disconnect

function Is_Connecting_Active

```
function Is_Connecting_Active (Connection : Transport.Connection_Id)
                                                    return Boolean;
```

Description

Determines whether the specified connection is in the process of establishing an active connection.

Parameters

`Connection : Transport.Connection_Id;`

Specifies a connection.

`return Boolean;`

Returns true if the specified connection is in the middle of an active connection. If the connection is connected, closed, or has never been opened, a value of false is returned.

References

procedure Connect

function Is_Connecting_Passive

```
function Is_Connecting_Passive (Connection : Transport.Connection_Id)
                                                    return Boolean;
```

Description

Determines whether the specified connection is in the process of establishing a passive connection.

Parameters

`Connection : Transport.Connection_Id;`

Specifies a connection.

`return Boolean;`

Returns true if the specified connection is in the middle of a passive connection. If the connection is connected, closed, or has never been opened, a value of false is returned.

References

procedure Connect

function Is_Open

```
function Is_Open (Connection : Transport.Connection_Id) return Boolean;
```

Description

Determines whether the specified connection is currently open.

If the connection is closed or has never been opened, a value of false is returned.

Parameters

`Connection : Transport.Connection_Id;`
Specifies a connection.

`return Boolean;`
Returns true if the specified connection is open.

References

procedure Close

procedure Open

function Local_Host

```
function Local_Host (Connection : Transport.Connection_Id)
                                                  return Transport_Defs.Host_Id;

function Local_Host (Network : Transport_Defs.Network_Name)
                                                  return Transport_Defs.Host_Id;
```

Description

Returns the Host_Id associated with the local end of the connection.

If the specified connection is not connected, the Null_Host_Id is returned.

Parameters

`Connection : Transport.Connection_Id;`
Specifies a connection.

`Network : Transport_Defs.Network_Name;`
Specifies the network.

`return Transport_Defs.Host_Id;`
Returns the Host_Id of the machine.

function Local_Socket

```
function Local_Socket (Connection : Transport.Connection_Id)
                                        return Transport_Defs.Socket_Id;
```

Description

Returns the Socket_Id associated with the end of the specified connection.

If the connection is not open, the Null_Socket_Id is returned.

Parameters

`Connection : Transport.Connection_Id;`
Specifies an open connection.

`return Transport_Defs.Socket_Id;`
Returns the Socket_Id associated with the end of the specified connection.

References

procedure Open

function Network

```
function Network (Connection : Transport.Connection_Id)
                                        return Transport_Defs.Network_Name;
```

Description

Returns the name of the network associated with the given connection.

If the connection is not open, the Null_Network_Name is returned.

Parameters

`Connection : Transport.Connection_Id;`
Specifies an open connection.

`return Transport_Defs.Network_Name;`
Returns the Network_Name associated with the end of the specified connection.

References

procedure Open

type Network_Name_Iterator

```
type Network_Name_Iterator is limited private;
```

Description

Specifies an iterator over the names of all networks known to the local system.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Text_Io.Put_Line ("all known networks:");
Init (Iter);
while not Done (Iter) loop
    Text_Io.Put_Line (String (Value (Iter)));
    Next (Iter);
end loop;
```

References

function Done

procedure Init

procedure Next

function Value

type Transport_Defs.Network_Name

procedure Next

```
procedure Next (Iter : in out Network_Name_Iterator);

procedure Next (Iter : in out Connection_Id_Iterator);
```

Description

Moves the iterator to the next value.

Parameters

```
Iter :  in out Network_Name_Iterator;
```
Specifies an iterator.

```
Iter :  in out Connection_Id_Iterator;
```
Specifies an iterator.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

References

function Done

procedure Init

function Value

constant Null_Connection_Id

```
Null_Connection_Id : constant Connection_Id;
```

Description

Denotes no particular connection.

The function Is_Open (Null_Connection_Id) is false.

procedure Open

```
procedure Open (Connection    : out Transport.Connection_Id;
                Status        : out Transport_Defs.Status_Code;
                Network       :     Transport_Defs.Network_Name;
                Local_Socket  :     Transport_Defs.Socket_Id     :=
                                        Transport_Defs.Null_Socket_Id);
```

Description

Allocates the resources required to form a connection.

The Network and Local_Socket parameters must be specified when the connection is opened.

Once opened, the connection can be connected and disconnected many times. It continues to be associated with the same Network and Local_Socket parameters.

Parameters

```
Connection :  out Transport.Connection_Id;
```

Returns a Connection_Id that can be used to denote the connection in subsequent calls to other procedures.

```
Status :  out Transport_Defs.Status_Code;
```

Returns the outcome of the operation. Possible values are:

- Transport_Defs.No_Free_Connections
- Transport_Defs.No_Free_Memory
- Transport_Defs.No_Free_Sockets
- Transport_Defs.No_Hardware
- Transport_Defs.No_Local_Resources
- Transport_Defs.No_Such_Network
- Transport_Defs.Not_Downloaded
- Transport_Defs.Not_Initialized
- Transport_Defs.Ok

```
Network : Transport_Defs.Network_Name;
```

Specifies the network to be associated with the connection. This parameter determines the interpretation of the Host_Id specified in a subsequent call to the Connect procedure. Refer to type Transport_Defs.Host_Id.

```
Local_Socket : Transport_Defs.Socket_Id := Transport_Defs.Null_Socket_Id;
```

Specifies the local socket to be associated with the connection. If Local_Socket equals Null_Socket_Id, the transport service invents a socket identification that is not currently in use and assigns it to this connection. The invented socket identification is not in the range of socket identifications that are reserved for well-known services.

References

procedure Close

procedure Connect

type Transport_Defs.Host_Id

---

procedure Receive

```
procedure Receive
          (Connection :     Transport.Connection_Id;
           Status     : out Transport_Defs.Status_Code;
           Data       : out Byte_Defs.Byte_String;
           Count      : out Natural;
           Max_Wait   :     Duration                    := Duration'Last);
```

Description

Receives some data.

This procedure does not wait to fill the data buffer. As soon as ***any*** data are received, the procedure returns the data.

The transport service can store received data in its own buffer before you call the Receive procedure. If there is a lot of received data in this buffer when you call the Receive procedure, you get as much of the buffered data as will fit in the buffer you supply.

Parameters

`Connection : Transport.Connection_Id;`

Specifies the connection from which to receive data. The connection must be connected.

`Status : out Transport_Defs.Status_Code;`

Returns the outcome of the receive operation. Possible values are:

- Transport_Defs.Connection_Broken
- Transport_Defs.Disconnected
- Transport_Defs.Not_Connected
- Transport_Defs.Not_Open
- Transport_Defs.Ok
- Transport_Defs.Timed_Out
- Transport_Defs.Too_Many_Clients

`Data : out Byte_Defs.Byte_String;`

Specifies the buffer into which to store received data.

```
Count :  out Natural;
```

Returns the number of bytes actually received (may be less than Data'Length).

```
Max_Wait :  Duration := Duration'Last;
```

Specifies the maximum amount of time to wait to receive data. The Receive procedure returns when one or more Data bytes have been received or when Max_Wait expires, whichever comes first.

Example

If you want to wait for a certain number of bytes to arrive, use the following code fragment:

```
declare
    Count : Natural;
    Total : Natural := 0;
begin
    while Total < Data'Length loop
        Receive (Connection, Status,
                 Data (Data'first + Total .. Data'last),
                 Count);
        Total := Total + Count;
        exit when Transport_Defs."/=" (Status, Transport_Defs.Ok);
    end loop;
end;
```

References

procedure Transmit

---

function Remote_Host

```
function Remote_Host (Connection : Transport.Connection_Id)
                                        return Transport_Defs.Host_Id;
```

Description

Returns the Host_Id of the machine at the other end of the specified connection.

If the connection is not connected, the Null_Host_Id is returned.

Parameters

`Connection : Transport.Connection_Id;`
Specifies a connection.

`return Transport_Defs.Host_Id;`
Returns the Host_Id of the machine at the other end of the specified connection.

References

procedure Connect

function Remote_Socket

```
function Remote_Socket (Connection : Transport.Connection_Id)
                                        return Transport_Defs.Socket_Id;
```

Description

Returns the Socket_Id of the socket at the other end of the specified connection.

If the connection is not connected, the Null_Socket_Id is returned.

Parameters

`Connection : Transport.Connection_Id;`
Specifies a connection.

`return Transport_Defs.Socket_Id;`
Returns the Socket_Id of the socket at the other end of the specified connection.

References

procedure Connect

procedure Set_Owner

```
procedure Set_Owner (Connection : Transport.Connection_Id;
                     Owner      : Machine.Job_Id);
```

Description

Sets the owner of a connection.

If the connection is not open, do nothing.

Each connection is owned by some job. When a job terminates, all connections owned by it are closed automatically. By default, a connection is owned by the job that opens it.

The Set_Owner procedure can be used to give a connection to some job other than the one that opened it.

Parameters

`Connection : Transport.Connection_Id;`
Specifies a connection.

`Owner : Machine.Job_Id;`
Specifies the new owner of the connection.

References

procedure Close_All

function Get_Owner

procedure Transmit

```
procedure Transmit
          (Connection :     Transport.Connection_Id;
           Status     : out Transport_Defs.Status_Code;
           Data       :     Byte_Defs.Byte_String;
           Count      : out Natural;
           Max_Wait   :     Duration                    := Duration'Last;
           More       :     Boolean                     := False);
```

Description

Transmits data on a connection.

The Transmit procedure returns when any of the following occur:

- Transmit has transmitted the data you supplied.
- Transmit has stored the data you supplied in its own buffers and has taken responsibility for delivering it.
- The timeout (Max_Wait parameter) you specified has expired.

You can determine how many bytes were transmitted or buffered from the Count parameter.

Parameters

```
Connection : Transport.Connection_Id;
```

Specifies the connection on which to transmit the data. If the connection is not connected, the operation fails.

```
Status : out Transport_Defs.Status_Code;
```

Returns the outcome of the transmit operation. Possible values are:

- Transport_Defs.Connection_Broken
- Transport_Defs.Disconnected
- Transport_Defs.Not_Connected
- Transport_Defs.Not_Open
- Transport_Defs.Not_Registered
- Transport_Defs.Ok
- Transport_Defs.Timed_Out
- Transport_Defs.Too_Many_Clients

```
Data : Byte_Defs.Byte_String;
```

Specifies the data to be transmitted.

```
Count : out Natural;
```

Specifies the number of bytes actually transmitted or buffered. Usually, this number will be Data'Length, but it may be less if the connection breaks or if the operation times out.

```
Max_Wait : Duration := Duration'Last;
```

Specifies the maximum amount of time to spend trying to transmit or buffer the data.

```
More : Boolean := False;
```

Specifies, if true, that the service can hold the data in its local buffers, to be combined with more data that you are about to transmit.

This is a performance hint only. The service is free to ignore it and to transmit all data as soon as you supply it.

Use this feature with care. If you specify More = True, your data may not be transmitted. This can cause a deadlock, depending on the structure of your dialogue with the program at the other end of the connection. For example, if the dialogue is structured as strictly two-way alternate (request/response) and you send a message with More = True, the remote program may not see your message (it is still in a local transmit buffer) and therefore may not send a response. A rule of thumb is to specify More = False whenever you transmit data that you expect to elicit a response from the remote program.

A call to Transmit with a null (zero length) data string and More = False has the effect of forcing transmission of any data that may be in a local transmit buffer.

References

procedure Receive

---

function Value

```
function Value (Iter : Network_Name_Iterator)
                                        return Transport_Defs.Network_Name;

function Value (Iter : Connection_Id_Iterator) return Connection_Id;
```

Description

Returns the current value of the iterator.

Parameters

`Iter : Network_Name_Iterator;`

Specifies an iterator over all known Network_Names.

`return Transport_Defs.Network_Name;`

Returns the current value of the iterator.

`Iter : Connection_Id_Iterator;`

Specifies an iterator over all open Connection_Ids.

`return Connection_Id;`

Returns the current value of the iterator.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

References

function Done

procedure Init

procedure Next

end Transport;

package Transport_Defs

Package Transport_Defs provides types that are used by package Transport and by other networking software. Package Transport_Defs also provides a few simple utility operations on those types—for example, hash functions and normalization functions.

function Hash

```
function Hash (Value : Network_Name) return Natural;

function Hash (Value : Host_Id) return Natural;

function Hash (Value : Socket_Id) return Natural;
```

Description

Produces a value suitable for indexing into a hash table.

The argument need not be normalized; it is normalized before the hash value is computed. So, if two values have the same normalized form, they will hash to the same place.

Parameters

`Value : Network_Name;`
Specifies the network name to be hashed. It need not be normalized.

`Value : Host_Id;`
Specifies the host identifier to be hashed. It need not be normalized.

`Value : Socket_Id;`
Specifies the socket identifier to be hashed. It need not be normalized.

`return Natural;`
Returns a number derived from the given value. This number (modulo the size of your hash table) can be used as a hash bucket index.

Example

The following code fragment returns the number resulting from hashing Network_Name. That number (modulo the size of the hash table) is used as the hash table index for The_Bucket.

```
declare
    subtype Index is Natural range 0 .. Table_Size - 1;
    Table : array (Index) of Bucket;
begin
    The_Bucket := Table (Hash (X) mod Table'Length);
end;
```

References

function Normalize

type Host_Id

```
type Host_Id is new Byte_Defs.Byte_String;
```

Description

Denotes a computer system that may be accessible via some network.

Essentially, this address is the network address of a machine. The interpretation of a Host_Id depends on the network. For a TCP/IP network, the Host_Id is an Internet address. It should be four bytes long and should contain the bytes of the Internet address, beginning with the most significant byte—that is, the first byte of the network number.

Host_Ids that are shorter than four bytes are right-justified and zero-padded to form an Internet address. Host_Ids that are longer than four bytes are truncated, retaining the first four bytes.

References

function Hash

function Normalize

type Socket_Id

function Transport_Name.Host_To_Host_Id

function Image

```
function Image (Status : Status_Code) return String;
```

Description

Converts a status code into a printable string.

Parameters

`Status : Status_Code;`
Specifies the status code to be converted.

`return String;`
Returns a printable string describing the status code.

Example

The following code fragment converts the status code into a printable string and then uses the !Io.Text_Io.Put_Line procedure to write it to the Current_Output file.

```
Text_Io.Put_Line ("status => " & Image (Status));
```

type Network_Name

```
type Network_Name is new String;
```

Description

Denotes one of several networks to which this machine can be attached.

Only TCP/IP is currently defined.

TCP/IP is the U.S. Department of Defense TCP and IP protocols, running on an IEEE 802.3 (Ethernet) local area network.

References

function Hash

function Normalize

function Transport_Name.Host_To_Network_Name

function Normalize

```
function Normalize (Value : Network_Name) return Network_Name;

function Normalize (Value : Host_Id) return Host_Id;

function Normalize (Value : Socket_Id) return Socket_Id;
```

Description

Converts the given value to an equivalent normalized form.

Normalization removes nonsignificant filler (for example, leading zeros or blanks) and converts characters to a single case.

Two values are equivalent when their normalized forms are equal.

Parameters

`Value : Network_Name;`
Specifies the value to be normalized.

`return Network_Name;`
Returns the normalized form of the given value. Leading and trailing blanks are removed, and characters are converted to uppercase.

`Value : Host_Id;`
Specifies the value to be normalized.

`return Host_Id;`
Returns the normalized form of the given value. Leading zeros are removed.

`Value : Socket_Id;`
Specifies the value to be normalized.

`return Socket_Id;`
Returns the normalized form of the given value. Leading zeros are removed.

Example

The following code fragment uses the Normalize function to convert the Network_Name foo_BAR to the same case: FOO_BAR:

```
Normalize (Network_Name'("   foo_BAR ")) = Network_Name'("FOO_BAR")
```

References

function Hash

constant Null_Host_Id

```
Null_Host_Id : constant Host_Id (1 .. 0) := (others => 0);
```

Description

Specifies a value of Host_Id type that denotes no particular host.

constant Null_Network_Name

```
Null_Network_Name : constant Network_Name := "";
```

Description

Specifies a value of Network_Name type that denotes no particular network.

constant Null_Socket_Id

```
Null_Socket_Id : constant Socket_Id (1 .. 0) := (others => 0);
```

Description

Specifies a value of Socket_Id type that denotes no particular socket.

References

procedure Transport.Open

type Socket_Id

```
type Socket_Id is new Byte_Defs.Byte_String;
```

Description

Denotes a socket within the context of a machine.

A socket is an abstract object, defined by the transport service. Conceptually, it is the location at which a connection is established. When a program does a passive connect, it must supply a Socket_Id and is said to *wait* or *listen* on that socket. When a program does an active connect, it must supply a Host_Id and a Socket_Id and is said to *call* that socket. Two such programs are connected to each other when the caller calls the same socket on which the waiter is waiting.

Many connections can be established on each socket. However, only one program can be waiting on each socket at a time. There can be many programs simultaneously waiting on different sockets.

The interpretation of a Socket_Id depends on the network. For TCP/IP, the Socket_Id is a TCP port number. It should be two bytes long and should contain the bytes of the port number, beginning with the most significant byte.

Socket_Ids that are shorter than two bytes are right-justified and zero-padded to form a port number. Socket_Ids that are longer than two bytes are truncated, retaining the first two bytes.

References

function Hash

type Host_Id

function Normalize

procedure Transport.Connect

procedure Transport.Open

type Status_Code

```
type Status_Code is new Integer;
```

Description

Describes the outcome of a transport operation.

The meaningful values of this type are declared as constants. They are:

- Access_Denied: The operation failed because the caller was not authorized to perform it.
- Connection_Broken: The operation failed because the given connection was disconnected by a failure somewhere in the network.
- Connection_Refused: The operation failed because the specified host refused the connect request. This usually means that there is no program waiting on the specified socket in the specified host.
- Disconnected: The operation failed because the given connection was disconnected by the other host.
- No_Free_Connections: The operation failed because it was not possible to obtain a local connection (specified by Transport.Connection_Id).
- No_Free_Memory: The operation failed because it was not possible to obtain memory space.
- No_Free_Sockets: The operation failed because it was not possible to obtain a socket.
- No_Hardware: The operation failed because the networking hardware is not installed in this machine.
- No_Local_Resources: The operation failed because it was not possible to obtain some local resource—for example, memory space or a control block.
- No_Such_Host: The operation failed because there is no known host denoted by the given Host_Id. The specified host may in fact exist but may be unreachable.
- No_Such_Network: The operation failed because there is no known network denoted by the given Network_Name.
- Not_Connected: The operation failed because the given Connection_Id denoted a connection that was not connected.
- Not_Downloaded: The operation failed because the networking hardware in this machine has not been successfully downloaded.
- Not_Initialized: The operation failed because the networking hardware and software on this machine have not been initialized.
- Not_Open: The operation failed because the given Connection_Id denoted a connection that was not open.

- Not_Registered: The operation failed because the calling task was not registered to perform it.
- Ok: The operation completed normally.
- Socket_In_Use: The operation failed because the specified socket is already in use by another task.
- Timed_Out: The operation did not complete because the specified Max_Wait period expired. This is not necessarily an error; if the Max_Wait period is short or zero, it may simply indicate that the desired event has not yet occurred.
- Too_Many_Clients: The operation failed because some other task was simultaneously attempting to do the same operation.

References

function Image

---

end Transport_Defs;

---

package Transport_Name

Package Transport_Name defines a mapping from machine names (strings) to machine addresses (Transport_Defs.Network_Names and Transport_Defs.Host_Ids). For users who prefer to refer to machines by their mnemonic names rather than their addresses, this package is useful for building user-interface programs.

This package does not provide a way to update the database of names. Currently, the Transport_Name database is a text file named !Machine.Transport_Name_Map. Each line of the file contains the network name, the host identifier, the host name, and the machine type. For example:

```
TCP/IP  89.4.27.1  aspen  R1000
TCP/IP  89.4.8.43  oak    VMS
```

To change the database, change this file. Any tool will do, including the Rational Editor, !Commands.Library.Copy, !Commands.Archive.Restore, or FTP.

function Done

```
function Done (Iter : Host_Iterator) return Boolean;
```

Description

Indicates that the given iterator is done.

An iterator is done when it has been moved past the last element in the collection over which it iterates. If function Done (Iter) is true, function Value (Iter) is undefined.

Parameters

`Iter : Host_Iterator;`
Specifies an iterator.

`return Boolean;`
Returns true if the iterator is done.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

References

procedure Init

procedure Next

function Value

function Host_Id_To_Host

```
function Host_Id_To_Host (Network : Transport_Defs.Network_Name;
                          Host    : Transport_Defs.Host_Id) return String;

function Host_Id_To_Host (Host : Transport_Defs.Host_Id) return String;
```

Description

Converts a host address to a string name.

Parameters

`Network : Transport_Defs.Network_Name;`
Specifies the name of a network.

`Host : Transport_Defs.Host_Id;`
Specifies the address of a machine on the given network.

`return String;`
Returns the printable name of the given host.

Errors

The Undefined exception is raised when the specified host is unknown—that is, it is not in the database.

type Host_Iterator

```
type Host_Iterator is limited private;
```

Description

Specifies an iterator over all named hosts on all networks.

If you want hosts on only one network, you can determine the network associated with each host name produced by this iterator and ignore the ones that you do not want.

References

function Done

procedure Init

procedure Next

function Value

function Host_To_Host_Id

```
function Host_To_Host_Id (Host_Name : String)
                                        return Transport_Defs.Host_Id;
```

Description

Converts a string name to a Host_Id.

Parameters

`Host_Name : String;`
Specifies the printable name of a machine somewhere in the network.

`return Transport_Defs.Host_Id;`
Returns the Host_Id used to connect to the given host. This value can be passed as a parameter to the Transport.Connect procedure.

Errors

The Undefined exception is raised when the given Host_Name is unknown—that is, it is not in the database.

References

procedure Transport.Connect

function Host_To_Machine_Type

```
function Host_To_Machine_Type (Host_Name : String) return String;
```

Description

Reads the name of a machine and returns a string specifying the type of machine.

Parameters

`Host_Name : String;`

Specifies the printable name of a machine somewhere in the network—for example, "Logo".

`return String;`

Returns the name of the type of machine—for example, "R1000".

function Host_To_Network_Name

```
function Host_To_Network_Name (Host_Name : String)
                                        return Transport_Defs.Network_Name;
```

Description

Converts a string name to a Network_Name.

Parameters

`Host_Name : String;`

Specifies the printable name of a machine somewhere in the network.

`return Transport_Defs.Network_Name;`

Returns the Network_Name used to connect to the given host. This value can be passed as a parameter to the Transport.Open procedure.

Errors

The Undefined exception is raised when the given Host_Name is unknown—that is, it is not in the database.

References

procedure Transport.Open

procedure Init

```
procedure Init (Iter : in out Host_Iterator);
```

Description

Initializes an iterator.

Parameters

```
Iter :  in out Host_Iterator;
```

Specifies an iterator.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

References

function Done

procedure Next

function Value

function Local_Host_Name

```
function Local_Host_Name (Network : Transport_Defs.Network_Name)
                                                    return String;
```

Description

Reads the name of the network and returns the name of the machine in which the caller is running.

Parameters

`Network : Transport_Defs.Network_Name;`

Specifies the name of the network—for example, "TCP/IP".

`return String;`

Returns the name of this machine—for example, "Universe".

Errors

The Undefined exception is raised when the given Host_Name is unknown—that is, it is not in the database of Host_Id names (refer to the package introduction).

procedure Next

```
procedure Next (Iter : in out Host_Iterator);
```

Description

Moves the iterator to the next value.

Parameters

```
Iter :  in out Host_Iterator;
```

Specifies an iterator.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

References

function Done

procedure Init

function Value

exception Undefined

```
Undefined : exception;
```

Description

Indicates that the caller has tried to resolve a mapping that is not defined (not in the database).

References

function Host_Id_To_Host

function Host_To_Host_Id

function Host_To_Network_Name

function Value

```
function Value (Iter : Host_Iterator) return String;
```

Description

Returns the current value of the iterator.

Parameters

`Iter : Host_iterator;`
Specifies an iterator over all known hosts.

`return String;`
Returns the current value of the iterator. This is the printable name of a host.

Example

The following code fragment initializes an iterator and then obtains the value of the iterator for each step of an iterative loop. Looping continues until the iterator has moved passed the last element in the collection over which it iterates.

```
Init (Iter);
while not Done (Iter) loop
    ... Value (Iter) ...
    Next (Iter);
end loop;
```

References

function Done

procedure Init

procedure Next

end Transport_Name;

package Transport_Route

The Rational system maintains a table used for routing IP packets, including TCP/IP and UDP/IP packets. When a packet is sent to a machine on some other network, the packet must be routed through a gateway and not directly to the destination machine. Some gateways do not respond to ARP (Address Resolution Protocol) queries for destination machines whose traffic they carry; therefore, the sending machine must know the Internet address of the gateway in order to transmit packets to it.

The routing table contains a list of entries, each containing a route (the Internet address of a gateway) with a destination that can be reached by the route. The destination may be a specific Host_Id (Internet address), a network number (signifying all hosts in that network), or the Null_Host_Id (signifying any remote host). There can be multiple entries for each route, identifying multiple hosts or networks accessible by way of the specified route. The table is kept ordered, starting with all host-specific entries, followed by all network-specific entries, followed by the default entry. Within each group, entries are maintained in the order in which they were defined. When the host is deciding where to send an outgoing packet, it searches the table entries in order.

The specification of the procedures in package Transport_Route follows:

```
with Transport_Defs;

package Transport_Route is

   procedure Show (Route : String := "";
                   Destination : String := "";
                   Network : Transport_Defs.Network_Name  := "";
                   Response : String := "<PROFILE>");

   procedure Load (Table : String := "!Machine.Transport_Routes";
                   Form : String := "";
                   Response : String := "<PROFILE>");
```

```
    procedure Define (Route : String;
                      Destination : String := "";
                      Network : Transport_Defs.Network_Name  := "IP";
                      Response : String := "<PROFILE>");

    procedure Undefine (Route : String;
                        Destination : String := "";
                        Network : Transport_Defs.Network_Name  := "IP";
                        Response : String := "<PROFILE>");

end Transport_Route;
```

procedure Define

```
procedure Define (Route       : String;
                  Destination : String                          := "";
                  Network     : Transport_Defs.Network_Name     := "IP";
                  Response    : String                          := "<PROFILE>");
```

Description

Adds one entry to the routing table, with the given values.

If there is already such an entry in the table, the procedure does nothing.

Parameters

```
Route : String;
```

Specifies the Internet address of the gateway. When Network = "IP", Route is the Internet address of an IP gateway.

Route can be a name or a Transport_Defs.Host_Id in decimal.dotted notation—for example, "89.64.1.22".

If Route is a name, the name is resolved to a Host_Id by Transport_Name.Host_To_Host_Id.

If Route is "", the null string is resolved to Transport_Defs.Null_Host_Id.

```
Destination : String := "";
```

Specifies a destination to which packets are routed. When Network = "IP", Destination can be:

- The complete Internet address of a machine, or
- The network number of a remote network, or
- The Null_Host_Id

A complete Internet address is used to route packets to that specific destination machine. A network number is used to route packets to any machine in that network. The Null_Host_Id is used to indicate a default route for machines that do not match any other table entry.

Destination can be a name or a Transport_Defs.Host_Id in decimal.dotted notation —for example, "128.33".

If Destination is a name, the name is resolved to a Host_Id by Transport_Name-.Host_To_Host_Id.

If Destination is "", the null string is resolved to Transport_Defs.Null_Host_Id.

```
Network : Transport_Defs.Network_Name := "IP";
```

Specifies the name of a transport service.

The default, "IP", represents transport services based on the Internet Protocol and Ethernet (for example, TCP/IP and UDP/IP).

```
Response : String := "<PROFILE>";
```

Specifies the options controlling logging and error handling. See the *Rational Environment Reference Manual*, SJM, package Profile.

Example

The following example adds an entry indicating that packets destined for the machine named Logo should be routed to the IP gateway at Internet address 89.64.1.22:

```
Transport_Route.Define ("89.64.1.22", "logo");
```

The following example adds an entry indicating that packets destined for network number 128.33 should be routed to the IP gateway named Shemp:

```
Transport_Route.Define ("shemp", "128.33");
```

The following example adds an entry indicating that, by default, packets destined for other networks should be routed to the IP gateway named Fred:

```
Transport_Route.Define ("fred");
```

procedure Load

```
procedure Load (Table    : String := "!Machine.Transport_Routes";
                Form     : String := "";
                Response : String := "<PROFILE>");
```

Description

Loads routing information from the named table into the system's routing table.

The procedure reads the object named by the Table parameter, using package !Io.Text_Io, and passes the specified form to Text_Io.Open. For each line in the table, the Load procedure calls the Define procedure with parameter values parsed from the line.

Parameters

`Table : String := "!Machine.Transport_Routes";`

Specifies the name of a file or other object that contains routing information. Within this object, each text line must contain the Host_Id or name of a route, optionally followed by the Host_Id or name of a destination, optionally followed by a Network_Name. These values must be separated by spaces. If the Network_Name is omitted, "IP" is assumed. If the destination Host_Id is omitted, the Null_Host_Id is assumed.

`Form : String := "";`

Specifies the options controlling how the table is read. This string is passed to Text_Io.Open.Form.

`Response : String := "<PROFILE>";`

Specifies the options controlling logging and error handling. See the *Rational Environment Reference Manual*, SJM, package Profile.

procedure Show

```
procedure Show (Route       : String                        := "";
                Destination : String                        := "";
                Network     : Transport_Defs.Network_Name   := "";
                Response    : String                        := "<PROFILE>");
```

Description

Creates a text listing of routing table entries and writes it to the Current_Output file.

Only entries that match the given Route, Destination, and Network values are listed. In each case, the value "" is a wildcard that matches all entries.

Parameters

```
Route :  String := "";
```

Specifies the Internet address of the gateway. When Network = "IP", Route is the Internet address of an IP gateway.

Route can be a name or a Transport_Defs.Host_Id in decimal.dotted notation—for example, "89.64.1.22".

If Route is a name, the name is resolved to a Host_Id by Transport_Name.Host-_To_Host_Id.

If Route is "", entries for all routes are shown.

```
Destination : String := "";
```

Specifies a destination to which packets are routed. When Network = "IP", Destination can be:

- The complete Internet address of a machine, or
- The network number of a remote network, or
- The Null_Host_Id

A complete Internet address is used to route packets to that specific destination machine. A network number is used to route packets to any machine in that network. The Null_Host_Id is used to indicate a default route for machines that do not match any other table entry.

Destination can be a name or a Transport_Defs.Host_Id in decimal.dotted notation —for example, "128.33".

If Destination is a name, the name is resolved to a Host_Id by Transport_Name-.Host_To_Host_Id.

If Destination is "", entries for all destinations are shown.

```
Network : Transport_Defs.Network_Name := "";
```

Specifies the name of a transport service.

The default, "", causes entries for all networks to be shown. "IP" represents transport services based on the Internet Protocol and Ethernet (for example, TCP/IP and UDP/IP).

```
Response : String := "<PROFILE>";
```

Specifies the options controlling logging and error handling. See the *Rational Environment Reference Manual*, SJM, package Profile.

---

procedure Undefine

```
procedure Undefine
                (Route       : String;
                 Destination : String                          := "";
                 Network     : Transport_Defs.Network_Name     := "IP";
                 Response    : String                          := "<PROFILE>");
```

Description

Deletes the entry with the given values from the routing table.

If there is no such entry in the table, the procedure does nothing.

Parameters

```
Route : String;
```

Specifies the Internet address of the gateway. When Network = "IP", Route is the Internet address of an IP gateway.

Route can be a name or a Transport_Defs.Host_Id in decimal.dotted notation—for example, "89.64.1.22".

If Route is a name, the name is resolved to a Host_Id by Transport_Name.Host-_To_Host_Id.

If Route is "", the null string is resolved to Transport_Defs.Null_Host_Id.

```
Destination :  String := "";
```

Specifies a destination to which packets are routed. When Network = "IP", Destination can be:

- The complete Internet address of a machine, or
- The network number of a remote network, or
- The Null_Host_Id

A complete Internet address is used to route packets to that specific destination machine. A network number is used to route packets to any machine in that network. The Null_Host_Id is used to indicate a default route for machines that do not match any other table entry.

Destination can be a name or a Transport_Defs.Host_Id in decimal.dotted notation —for example, "128.33".

If Destination is a name, the name is resolved to a Host_Id by Transport_Name-.Host_To_Host_Id.

If Destination is "", the null string is resolved to Transport_Defs.Null_Host_Id.

```
Network :  Transport_Defs.Network_Name  := "IP";
```

Specifies the name of a transport service.

The default, "IP", represents transport services based on the Internet Protocol and Ethernet (for example, TCP/IP and UDP/IP).

```
Response :  String := "<PROFILE>";
```

Specifies the options controlling logging and error handling. See the *Rational Environment Reference Manual*, SJM, package Profile.

---

end Transport_Route;

Index

This index contains entries for each unit and its declarations as well as definitions, topical cross-references, exceptions raised, errors, enumerations, pragmas, switches, and the like. The entries for each unit are arranged alphabetically by simple name. An italic page number indicates the primary reference for an entry.

C

D

E

G

H

I

T

U

V

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____
_____
_____
_____
_____
_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____
_____
_____
_____

Indicate any additions or changes you would like to see in the index.

_____
_____
_____
_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____
Company _____
Address _____
City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043

Rational Networking—TCP/IP Reference Manual

Remote Procedure Call Facility (RPC)

Document Control Number: 8003A-02 (803-002332)

Rev. 1.0, November 1985
Rev. 2.0, July 1986
Rev. 3.0, July 1987 (Delta)

This document subject to change without notice.

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

RATIONAL

How to Use This Book

The Remote Procedure Call Facility (RPC) book of the *Rational Networking—TCP/IP Reference Manual* describes the Ada® package specifications for the facilities provided by the Rational® Networking—TCP/IP implementation of the RPC protocol. This book is intended for users who are familiar with the Rational Environment™ and with Ada programming.

Organization of the Networking Manual

The *Rational Networking—TCP/IP Reference Manual* (Networking Manual for brevity) includes the following volumes:

1	Telnet (TEL) File Transfer Protocol (FTP)
2	Transport Layer (TRL) Remote Procedure Call (RPC)

Each *volume* of the Networking Manual contains two *books* separated by large colored tabs. Each book contains information on particular features or areas of application in networking. The abbreviation for the name of each book (for example, TEL for Telnet) appears on the binder cover and spine, and this abbreviation is used in page numbers and cross-references. The books grouped into a given volume are logically related.

The Networking Manual provides reference information organized to efficiently answer specific questions about the Rational Networking product. Products other than the Networking product are documented in individual manuals (for example, the *Rational Environment Reference Manual* or the *Rational Target Build Utility Reference Manual*).

Volume 1

Volume 1 documents the commands used in the Rational Networking product. This volume contains the following two books:

- **Telnet**: The Telnet (TEL) book contains the commands used to establish and terminate connections to a remote host through a terminal logged into a local host.
- **File Transfer Protocol**: The File Transfer Protocol (FTP) book contains the commands used to transfer text and binary data files between two hosts. Packages File_Transfer, Ftp_Product, and Transfer_Generic can be used to develop programmatic interfaces for FTP.

Volume 1 also contains a Master Index that combines the index information from all four books of the Networking Manual.

Volume 2

Volume 2 documents the packages provided to develop new programmatic interfaces. This volume contains the following two books:

- **Transport Layer**: The Transport Layer (TRL) book describes the concepts and interfaces used to build networking tools for specific applications.
- **Remote Procedure Call**: The Remote Procedure Call (RPC) book describes the concepts and interfaces used to write clients and servers for remote procedure calls, in which an application running on one host can make procedure calls to applications running on different hosts.

Volume 2 also contains a Master Index that combines the index information from all four books of the Networking Manual.

Book Organization

Each book begins with a colored tab on which the name of the book appears. Each book typically contains the following sections:

- **Key Concepts**: This section describes the key concepts that pertain to the networking facilities documented in the book. The section is located behind its own tab following the How to Use This Book section.
- **Unit sections**: Each of the commands, tools, and so on has a declaration within an Ada compilation unit (typically a package). For each unit, there is a section that contains reference entries for the declarations within that unit. Each section is preceded by a tab.

  The sections for units are alphabetized by the simple names of the units. For example, the section for package !Tools.Network.Revn.Units.Network_Product is alphabetized under Network_Product.

  For many units, introductory material and/or examples specific to the unit appear after the section tabs.

  Within the section of a given unit, the reference entries describing the unit's declarations are organized alphabetically after the section introduction. Appearing at the top of each page of a reference entry are the simple name of the name of the given declaration and the fully qualified pathname of the enclosing unit.
- **Index**: Preceded by a tab, the Index appears as the last section of each book. It contains entries for each unit or declaration, along with additional topical

references. Each book index covers only the material documented in that particular book. Each volume contains a Master Index that provides entries for the information documented in all the books within the Networking Manual.

Italic page numbers indicate the page on which the primary reference entry for a declaration appears; nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Suggestions for Finding Information

The following suggestions can help you find various kinds of information in the documentation for Rational's products.

Learning about New Facilities

If you are a novice user starting to use the Rational Environment, consult the *Rational Environment User's Guide.*

If you are familiar with the Rational Environment but are interested in learning about certain networking commands, for example, you might start by reading the Key Concepts for the specific book, which describes important concepts and gives helpful examples.

It can also be useful to glance through the introductions provided for some of the units in the book. These introductions, located immediately after the tabs for the units, often contain helpful examples.

Finding Information on a Specific Item

If you know the name of the item and the book in which it is documented, consult either the table of contents or the index for that book. You can also turn through the pages of the book using the names and pathnames of the reference entries to locate the entry that you want. Remember that the reference entries for a unit are organized alphabetically by simple name within a tabbed section.

If you know the simple name of the entry but do not know the book in which it is documented, look in the Master Index to find the book abbreviation and page number.

If you cannot find an item in the Master Index, the item either is not documented or is documented in manuals for a product other than the Rational Networking—TCP/IP product (for example, the Rational Environment or the Rational Target Build Utility). If you know the pathname, consult the World ! section of the Reference Summary in Volume 1 of the ***Rational Environment Reference Manual*** to determine whether the item is documented and in which manual.

Using the Index

The index of each book contains entries for each unit and its declarations, organized alphabetically by simple name. When using the index to find a specific item, consult the italic page number for the primary reference to that item. Nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Viewing Specifications On-Line

If you know the pathname of a declaration and want to see its specification in a window of the Rational Environment, provide its pathname to the !Commands-.Common.Definition procedure—for example, `Definition ("!Tools.Ftp.Revn-.Units.Commands.Ftp");`. If you know the simple name of the unit in which the declaration appears, in most cases you can use searchlist naming as a quick way of viewing the unit—for example, `Definition ("\Ftp");`.

Using On-Line Help

Most of the information contained in the reference entries for each unit is available through the on-line help facilities of the Rational Environment. Press the Help on Help key or consult the *Rational Environment User's Guide* or the *Rational Environment Reference Manual*, EST, Help, for more information on using this on-line help facility.

Cross-Reference Conventions

The following conventions are used in cross-references to information:

- **Specific page/book**: For references to a specific place in a book, the book abbreviation is followed by the page number in the book (for example, RPC-23). If the book abbreviation is omitted, the current book is implied (for example, the page numbers in the table of contents for a book do not include the book prefix).
- **Declaration in same unit**: References to the documentation for a declaration in the same unit are indicated by the simple name of the desired declaration. For example, within the reference entry for the Ftp.Connect procedure, a reference to the Ftp.Disconnect procedure is "procedure Disconnect." Note that if the unit contains nested packages, references to nested declarations use qualified pathnames.
- **Declaration in different unit, same book**: References to the documentation for a declaration in another unit are indicated by the qualified pathname of the desired declaration. For example, within the reference entry for the Ftp.Connect procedure, a reference to the Ftp_Profile.Auto_Login function is "function Ftp_Profile.Auto_Login."
- **Declaration in different book**: References to the documentation for a declaration in another book are indicated by the addition of the abbreviation for that book. For example, within the reference entry for the Ftp.Connect procedure, a reference to the Transport.Connect procedure in the Transport Layer book would be "TRL, procedure Transport.Connect."

Feedback to Rational: Reader's Comments Form

Rational wants to make its documentation as useful and error-free as possible. Please provide us with feedback. The last page of each book contains a Reader's Comments form that you can use to send us comments or to report errors. You can also submit problem reports and make suggestions electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

RATIONAL

Key Concepts

The Rational Networking—TCP/IP product forms an intersystem interface for host/target development. Networking facilitates:

- Downloading of Ada source code and target object code
- Interactive debugging
- Control of target execution
- Updating of target state
- Building of test scaffolds
- Distributed applications

The Remote Procedure Call (RPC) Facility allows an application running on one host to access databases and software on another host (both R1000s and non-R1000s). RPC is based on the Ada procedural call model, extending calls between hosts.

The RPC facility allows a user application to make intersystem procedure calls. Through RPC, an application running on one computer can execute software on another computer.

This book is designed for advanced users capable of writing clients and servers for remote procedure calls. Generalized examples provide strategies for users to write their own software.

Users who have a system that already implements the RPC protocol need read only "RPC Function and Structure" and "Examples," below, for help in building their own RPC clients and servers. Users who want to implement RPC on a system that does not currently implement the RPC protocol should read at least the sections mentioned above as well as "RPC Protocol."

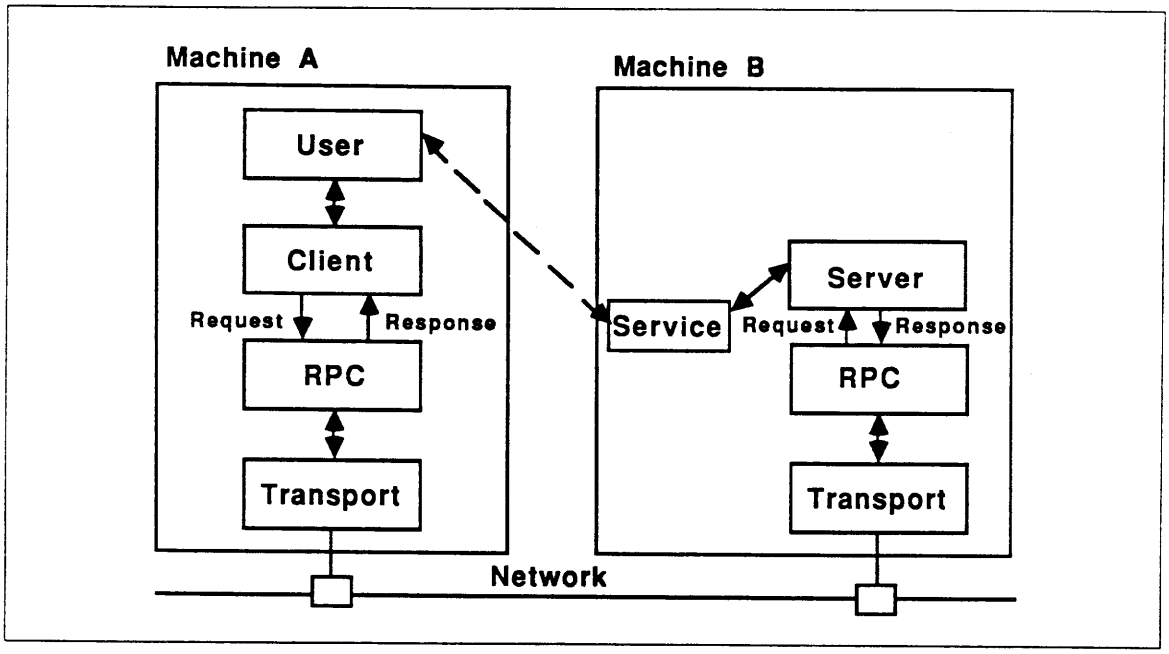

Figure 1-1. Remote Procedure Call Model

RPC Function and Structure

The RPC facility supports communication between tasks running on different machines. As a command link between computers, RPC provides access to procedures residing on remote computers. A general RPC diagram in Figure 1-1 shows two machines, A and B, connected by transport connections over an Ethernet network. The user application invokes a client on local machine A. The client makes requests for procedures to be performed (a service) on remote machine B and sends these requests using the RPC utilities to the server on machine B. The server invokes these procedures and sends status responses and procedure results back to the user application through the client.

A group of related operations made accessible by RPC is called a *service*. A client requests that a server perform an application by sending a message to the server. This service request message contains certain information outlined in "RPC Protocol," below. The server returns to the client a response message describing the result of the requested service. Every request (whether accepted or not) must be matched by a response message in the opposite direction. From the time the client sends a request until the corresponding response returns, the client is not permitted to transmit further requests. If the user application uses multitasking, the application can make simultaneous remote procedure calls, each through different client task threads, using multiple connections.

This communications model is similar to Ada procedural linkage. A client (caller) requests an operation (calls a procedure) and passes certain data (the *in* parameters) with the request. The server performs the operation and passes back other data (the *out* parameters) with the response.

In summary, the RPC facility defines:

- A naming standard providing conventions for identifying services and operations.
- An interchange standard of conventions for representing data in a machine-independent way. The standard includes guidelines for extension to new data types. Values of any Ada type except access values and task types can be interchanged.

The pairing of a client and a server lasts only for the duration of a request/response exchange. A client task can request the same service several times, but it does not necessarily reach the same server task each time. Depending on the implementation of the RPC facility, a server can receive request messages from any client package implementation in the network.

Limitations

The RPC facility does not support other models of communication. Examples of unsupported models are:

- A unidirectional stream of messages between two tasks.
- A sequence of request/response pairs between a pair of tasks bound to each other throughout the sequence.

The RPC facility does not provide a complete implementation of any particular client or server. It does, however, provide a set of Ada software tools that are helpful for writing clients and servers. By themselves, these tools do not implement client or server software; that must be done by the user. General implementation guidelines are provided in "Implementing Clients and Servers," below.

The RPC facility does not provide a way to interchange:

- Private types
- Task types
- Access types

Only values that can be represented as a sequence of bytes with no implicit shared storage can be interchanged.

Implementing Clients and Servers

The following steps are the guidelines for implementing an RPC server and client. A sample server and client are included in "Examples," below. The examples can be used as templates for your own clients and servers.

1. Define the service and implement the package that performs it on the server machine.

   If the service defines new types, these types should be grouped together in another package (for example, *Service_Defs*). Because these types are used by

both the service and the (remote) client interface to the service, it is simpler to share the types between these packages by defining them in their own package.

2. Implement interchange operations for the types used by the service. If the service uses types for which there are no extant interchange operations, you must write the Put and Get interchange operations for those types. These operations define a machine-independent representation for the type values. Refer to package Interchange for guidelines in writing new interchange operations. Because these interchange operations are used by both the client and the server machines, it is most convenient to put them in package *Service_Defs*.

3. Assign a socket to the service. Each service must have a Transport_Defs-.Network_Name and a Transport_Defs.Socket_Id. These values are used to establish connections to the server. The Network_Name and Socket_Id are used by both the client and the server machines, so they should be put in package *Service_Defs*.

   Two services cannot share the same socket identifier on the same machine. The assignment of socket identifiers must be managed to avoid conflict. Two services should not share a socket identifier even if they are on different machines. This separation simplifies any future transfers of the two services onto the same machine.

4. Assign program, version, and procedure numbers to the service. Each service should have a unique program number. Version numbers should be assigned to successive (incompatible) versions of the same service. Each subprogram in the service must be assigned a unique procedure number. These numbers are used by both the client and the server machines, so they should be put in package *Service_Defs*.

5. Implement a Process_Call procedure. The visible part of the procedure is specified by Rpc_Server.Serve.Process_Call. The procedure checks the version number and the case of the procedure number. Each clause in the case gets arguments from the transport stream, calls the appropriate procedure in the service package, calls Rpc_Server.Begin_Response, and then puts the results back into the stream.

   Call Begin_Response after calling the service subprogram but before putting any result values. If Begin_Response is called before the service subprogram is called, exceptions raised by the service subprogram are not passed back to the client correctly. If Begin_Response is called after the results have been put, the client interprets the result bytes as a response header (with unpredictable results).

6. Create a collection of server tasks. Instantiate Rpc_Server.Serve on the Process_Call procedure. Then instantiate Transport_Server on the resulting Serve procedure. Finally, create a pool of server tasks using the appropriate Network_Name and Socket_Id. When the pool is created, the number of simultaneously active server tasks can be limited.

7. Implement a client interface package. Each subprogram in the package allocates a stream, puts the procedure number and arguments into the transport stream, and gets the results.

Examples

The following examples illustrate usages of the RPC facility.

Client Interface

To invoke the following service through the RPC facility, use this program:

```
with Complex_Defs;

package Complex_Service is

    subtype Number is Complex_Defs.Number;

    function Make (Real, Imag : Float) return Number;
    function Real_Part (X : Number) return Float;
    function Imaginary_Part (X : Number) return Float;

    function Plus (Left, Right : Number) return Number;
    function Minus (Left, Right : Number) return Number;
    function Image (X : Number) return String;
    function Value (X : String) return Number;

    Overflow : exception;
    Underflow : exception;

end Complex_Service;
```

If your Environment already contains a similar package (the client interface) whose operations correspond to the operations in Complex_Service, use the following program:

```
with Complex_Defs;

package Complex_Client is
    type Number is private;
    function Make (Real, Imag : Float) return Number;
    function Real_Part (X : Number) return Float;
    function Imaginary_Part (X : Number) return Float;
    function Plus (Left, Right : Number) return Number;
    function Minus (Left, Right : Number) return Number;
    function Image (X : Number) return String;
    function Value (X : String) return Number;
    procedure Finalize;

    Overflow : exception;
    Underflow : exception;

private
    type Number is new Complex_Defs.Number;
end Complex_Client;
```

To invoke an operation, call the corresponding subprogram in the client interface.

Declarations Used by Client and Server

The client and the server packages share some definitions and procedures:

```
with Interchange;
with Rpc;
with Transport_Defs;

package Complex_Defs is

    type Number is
        record
            Real, Imag : Float;
        end record;


    -- Interchange operations:

    generic
        type Stream_Id is limited private;
        with procedure Put
            (Into : Stream_Id; Data : Interchange.Float) is <>;
        with procedure Get
            (From : Stream_Id; Data : out Interchange.Float) is <>;
    package Interchange_Operations is
        procedure Put (Into : Stream_Id; Data : Number);
        procedure Get (From : Stream_Id; Data : out Number);
    end Interchange_Operations;


    -- Shared definitions for remote procedure call:

    Network : constant Transport_Defs.Network_Name := "TCP/IP";
    Socket : constant Transport_Defs.Socket_Id (1..2) := (10, 45);

    Program : constant Rpc.Program_Number := 45;
    Version : constant Rpc.Version_Number := 0;

    package Proc_Number is
        Make : constant Rpc.Procedure_Number := 0;
        Real_Part : constant Rpc.Procedure_Number := 1;
        Imaginary_Part : constant Rpc.Procedure_Number := 2;
        Plus : constant Rpc.Procedure_Number := 3;
        Minus : constant Rpc.Procedure_Number := 4;
        Image : constant Rpc.Procedure_Number := 5;
        Value : constant Rpc.Procedure_Number := 6;
    end Proc_Number;

    package Exception_Number is
        Overflow : constant Rpc.Exception_Number := 1;
        Underflow : constant Rpc.Exception_Number := 2;
    end Exception_Number;

end Complex_Defs;
```

Client Implementation

If the client interface for a service has not been implemented:

```
with Complex_Defs;
with Interchange;
with Rpc;
with Rpc_Client;
with Transport_Complex;
with Transport_Defs;
with Transport_Interchange;
with Transport_Name;
with Transport_Stream;

package body Complex_Client is

    Remote_Host : constant Transport_Defs.Host_Id
                  := Transport_Name.Host_To_Host_Id ("shemp");

    procedure Start_Request is new Rpc_Client.Start_Request_Generic
                   (Complex_Defs.Network,
                    Remote_Host,
                    Complex_Defs.Socket,
                    Complex_Defs.Program,
                    Complex_Defs.Version);

    procedure Raise_Exception (Excep : Rpc.Exception_Number) is
    begin
        case Excep is
            when Complex_Defs.Exception_Number.Overflow  =>
                raise Overflow;
            when Complex_Defs.Exception_Number.Underflow  =>
                raise Underflow;
            when others =>
                raise Rpc.Other_Error;
        end case;
    end Raise_Exception;

    procedure End_Request is new Rpc_Client.End_Request_With_Exception
                   (Raise_Exception);

    procedure Finalize is
    begin
        Transport_Stream.Finalize;
    end Finalize;

    function Make (Real, Imag : Float) return Number is
        Stream : Transport_Stream.Stream_Id;
        Answer : Number;
    begin
        Start_Request (Stream, Complex_Defs.Proc_Number.Make);
        Transport_Interchange.Put (Stream, Interchange.Float (Real));
        Transport_Interchange.Put (Stream, Interchange.Float (Imag));
        End_Request (Stream);
        Transport_Complex.Get (Stream, Complex_Defs.Number (Answer));
        Rpc_Client.End_Response (Stream);
        return Answer;
    end Make;
```

```
function Real_Part (X : Number) return Float is
    Stream : Transport_Stream.Stream_Id;
    Answer : Float;
begin
    Start_Request (Stream, Complex_Defs.Proc_Number.Real_Part);
    Transport_Complex.Put (Stream, Complex_Defs.Number (X));
    End_Request (Stream);
    Transport_Interchange.Get (Stream, Interchange.Float (Answer));
    Rpc_Client.End_Response (Stream);
    return Answer;
end Real_Part;

function Imaginary_Part (X : Number) return Float is
    Stream : Transport_Stream.Stream_Id;
    Answer : Float;
begin
    Start_Request (Stream, Complex_Defs.Proc_Number.Imaginary_Part);
    Transport_Complex.Put (Stream, Complex_Defs.Number (X));
    End_Request (Stream);
    Transport_Interchange.Get (Stream, Interchange.Float (Answer));
    Rpc_Client.End_Response (Stream);
    return Answer;
end Imaginary_Part;

function Plus (Left, Right : Number) return Number is
    Stream : Transport_Stream.Stream_Id;
    Answer : Number;
begin
    Start_Request (Stream, Complex_Defs.Proc_Number.Plus);
    Transport_Complex.Put (Stream, Complex_Defs.Number (Left));
    Transport_Complex.Put (Stream, Complex_Defs.Number (Right));
    End_Request (Stream);
    Transport_Complex.Get (Stream, Complex_Defs.Number (Answer));
    Rpc_Client.End_Response (Stream);
    return Answer;
end Plus;

function Minus (Left, Right : Number) return Number is
    Stream : Transport_Stream.Stream_Id;
    Answer : Number;
begin
    Start_Request (Stream, Complex_Defs.Proc_Number.Minus);
    Transport_Complex.Put (Stream, Complex_Defs.Number (Left));
    Transport_Complex.Put (Stream, Complex_Defs.Number (Right));
    End_Request (Stream);
    Transport_Complex.Get (Stream, Complex_Defs.Number (Answer));
    Rpc_Client.End_Response (Stream);
    return Answer;
end Minus;

function Image (X : Number) return String is
    Stream : Transport_Stream.Stream_Id;
begin
    Start_Request (Stream, Complex_Defs.Proc_Number.Image);
    Transport_Complex.Put (Stream, Complex_Defs.Number (X));
    End_Request (Stream);
```

```
            declare
                Answer : constant String
                        := Transport_Interchange.Get_String (Stream);
            begin
                Rpc_Client.End_Response (Stream);
                return Answer;
            end;
        end Image;

        function Value (X : String) return Number is
            Stream : Transport_Stream.Stream_Id;
            Answer : Number;
        begin
            Start_Request (Stream, Complex_Defs.Proc_Number.Value);
            Transport_Interchange.Put_String (Stream, X);
            End_Request (Stream);
            Transport_Complex.Get (Stream, Complex_Defs.Number (Answer));
            Rpc_Client.End_Response (Stream);
            return Answer;
        end Value;

    end Complex_Client;
```

Server Implementation

To implement a server, create tasks that execute the operations of the service on behalf of clients in other hosts:

```
    with Complex_Defs;
    with Rpc;
    with Rpc_Server;
    with Transport;
    with Transport_Defs;
    with Transport_Server;
    with Transport_Stream;

    package Complex_Server is

        procedure Serve (Connection : Transport.Connection_Id);

        package Server is new Transport_Server (Serve);

        Std : constant Server.Pool_Id := Server.Create (Complex_Defs.Network,
                                                        Complex_Defs.Socket);

        procedure Finalize (Abort_Servers : Boolean := False)
                        renames Server.Finalize;

    end Complex_Server;


    with Complex_Service;
    with Interchange;
    with Text_Io;
    with Transport_Complex;
    with Transport_Interchange;
```

```
package body Complex_Server is

    procedure Process_Call (Stream : Transport_Stream.Stream_Id;
                            Id : Rpc.Transaction_Id;
                            Version : Rpc.Version_Number;
                            Proc : Rpc.Procedure_Number) is
    begin
        case Proc is
            when Complex_Defs.Proc_Number.Make =>
                declare
                    Real, Imag : Float;
                    Answer : Complex_Defs.Number;
                begin
                    Transport_Interchange.Get
                        (Stream, Interchange.Float (Real));
                    Transport_Interchange.Get
                        (Stream, Interchange.Float (Imag));
                    Answer := Complex_Service.Make (Real, Imag);
                    Rpc_Server.Begin_Response (Stream, Id);
                    Transport_Complex.Put (Stream, Answer);
                end;
            when Complex_Defs.Proc_Number.Real_Part =>
                declare
                    X : Complex_Defs.Number;
                    Answer : Float;
                begin
                    Transport_Complex.Get (Stream, X);
                    Answer := Complex_Service.Real_Part (X);
                    Rpc_Server.Begin_Response (Stream, Id);
                    Transport_Interchange.Put
                        (Stream, Interchange.Float (Answer));
                end;
            when Complex_Defs.Proc_Number.Imaginary_Part =>
                declare
                    X : Complex_Defs.Number;
                    Answer : Float;
                begin
                    Transport_Complex.Get (Stream, X);
                    Answer := Complex_Service.Imaginary_Part (X);
                    Rpc_Server.Begin_Response (Stream, Id);
                    Transport_Interchange.Put
                        (Stream, Interchange.Float (Answer));
                end;
            when Complex_Defs.Proc_Number.Plus =>
                declare
                    Left, Right : Complex_Defs.Number;
                    Answer : Complex_Defs.Number;
                begin
                    Transport_Complex.Get (Stream, Left);
                    Transport_Complex.Get (Stream, Right);
                    Answer := Complex_Service.Plus (Left, Right);
                    Rpc_Server.Begin_Response (Stream, Id);
                    Transport_Complex.Put (Stream, Answer);
                end;
```

```
            when Complex_Defs.Proc_Number.Minus  =>
                declare
                    Left, Right : Complex_Defs.Number;
                    Answer : Complex_Defs.Number;
                begin
                    Transport_Complex.Get (Stream, Left);
                    Transport_Complex.Get (Stream, Right);
                    Answer := Complex_Service.Minus (Left, Right);
                    Rpc_Server.Begin_Response (Stream, Id);
                    Transport_Complex.Put (Stream, Answer);
                end;
            when Complex_Defs.Proc_Number.Image  =>
                declare
                    X : Complex_Defs.Number;
                begin
                    Transport_Complex.Get (Stream, X);

                    declare
                        Answer : constant String
                                := Complex_Service.Image (X);
                    begin
                        Rpc_Server.Begin_Response (Stream, Id);
                        Transport_Interchange.Put_String (Stream, Answer);
                    end;
                end;
            when Complex_Defs.Proc_Number.Value  =>
                declare
                    X : constant String
                            := Transport_Interchange.Get_String (Stream);
                    Answer : Complex_Defs.Number;
                begin
                    Answer := Complex_Service.Value (X);
                    Rpc_Server.Begin_Response (Stream, Id);
                    Transport_Complex.Put (Stream, Answer);
                end;
            when others =>
                raise Rpc.No_Such_Procedure;
        end case;
    exception
        when Complex_Service.Overflow  =>
            Rpc_Server.Return_Exception
                (Stream, Id, Complex_Defs.Exception_Number.Overflow);
        when Complex_Service.Underflow  =>
            Rpc_Server.Return_Exception
                (Stream, Id, Complex_Defs.Exception_Number.Underflow);
    end Process_Call;

    procedure Serve_Guts is new Rpc_Server.Serve
                (Program => Complex_Defs.Program);

    procedure Serve (Connection : Transport.Connection_Id) is
    begin
        Serve_Guts (Connection);
    end Serve;

end Complex_Server;
```

```
package body Complex_Defs is

    package body Interchange_Operations is

        procedure Put (Into : Stream_Id; Data : Number) is
        begin
            Put (Into, Interchange.Float (Data.Real));
            Put (Into, Interchange.Float (Data.Imag));
        end Put;

        procedure Get (From : Stream_Id; Data : out Number) is
        begin
            Get (From, Interchange.Float (Data.Real));
            Get (From, Interchange.Float (Data.Imag));
        end Get;

    end Interchange_Operations;

end Complex_Defs;


with Complex_Defs;
with Transport_Stream;
with Transport_Interchange;

package Transport_Complex is new Complex_Defs.Interchange_Operations
     (Transport_Stream.Stream_Id,
      Transport_Interchange.Put,
      Transport_Interchange.Get);
```

RPC Protocol

This section describes the data exchange protocol for remote procedure calls between machines.

It is recommended that the user implement the protocol by porting the Ada source provided by Rational. In principle, the protocol can be implemented on any system in any language. In practice, however, it is more difficult to implement the protocol in another language. This specification document makes frequent reference to an implementation in portable Ada. Accordingly, Ada can be thought of as a protocol specification language. If the implementation is to be done in another language, the following sections should be read carefully.

Connecting and Disconnecting

A client requesting a service must first initiate a connection to the machine offering that service. To establish the connection, the client must supply network, host, and socket identifiers. A service's network, host, and socket identifiers must be published as static information by that service. The connection must be connected before a request is transmitted and must remain connected from the time a request is transmitted until the corresponding response is received.

Once the connection is established, the client and server must exchange the range of allowable RPC protocol versions. For example, the client might initially transmit Rpc.Version_Range'(3, 4), indicating that it wants to use protocol version 3 or 4 (that is, ranging from 3 to 4). The server, after checking that it supports only protocol version 4, gives the response Rpc.Version_Range'(4, 4). Other value ranges can be transmitted, indicating support for other versions of the RPC protocol.

Only the client can disconnect the connection between requests. The server cannot disconnect at any time.

Exchanging Data

Each remote procedure call is an exchange of messages over a single connection. A message (the request) is transmitted from the client to the server, after which another message (the response) is transmitted from the server to the client.

Once a client has transmitted a request on a connection, the client cannot transmit any other data on that connection until the corresponding response is received.

Request and response messages can be of arbitrary length depending on the type of the data values in them. By sequentially parsing the interchange representation of each message component, the message recipient can find the end of the message. No special out-of-band marker is used to mark the end of a message.

Each message is a value of the Rpc.Message_Header type, followed by the argument values (in a request) or by the result values (in a response). The formats of request and response messages are defined by package Rpc. Refer to the table in the reference entry for the Rpc.Get_Message function for individual messages.

The interchange form of a message is defined in package Interchange.

Before any communication can begin, RPC requires a protocol version number to ensure that both machines are using the same communication protocol.

One complexity of intermachine procedure calls is that local procedure calls can be mixed with remote calls. This means that program pathnames are not necessarily unique. To avoid such ambiguities, RPC assigns a program number to the remote package and uses this number rather than the pathname. In addition, each procedure in the remote package is given a procedure number. Finally, each package is given a version number.

Thus, an RPC call requires:

- The value Rpc.Message_Kind'(Call_Message), identifying the message as a request.
- A protocol version number, establishing protocol conformity.
- A program number, referencing the remote package.
- A procedure number, referencing the procedure in the package.
- A version number of the remote package.

- A transaction identifier, which is always zero. This identifier is reserved for future use.
- Subprogram-dependent parameter values corresponding to the *in* parameters of an Ada subprogram call.

A response message sent in reply to a request message can be:

- Reject, indicating some problem with the requested program number, version number, or procedure number.
- Return, indicating normal completion of the requested operation.
- Abort, indicating that an exception was raised by the called subprogram.

A *reject* message contains:

- The value Rpc.Message_Kind'(Reject_Message), identifying it as a reject message.
- A Transaction_Id copied from the Transaction_Id of the corresponding request message.
- A value of type Rpc.Reject_Details, describing the error in the request message.

A *return* message contains:

- The value Rpc.Message_Kind'(Return_Message), identifying it as a return message.
- A Transaction_Id copied from the Transaction_Id of the corresponding request message.
- The results of the operation, the type of which depends on the called operation.

An *abort* message contains:

- The value Rpc.Message_Kind'(Abort_Message), identifying it as an abort message.
- A Transaction_Id copied from the Transaction_Id of the corresponding request message
- A value of type Rpc.Error_Type, indicating the exception raised. RPC defines values for each of the Ada standard exceptions as well as the value "others" for all other exceptions. Programs that return more exception information must encode it as part of their return data.

RPC Ada Packages

The Ada packages helpful in implementing the RPC facility are listed and described at the beginning of the RPC reference pages. If you are not familiar with those packages, refer to them before reading this section. These packages are tools only and are not themselves working clients or servers. Each client or server requires the development of a new package. An example client and server package is given in "Examples," above.

The RPC Ada packages are designed to be portable to any computer system with an Ada compiler and an implementation of package Transport.

Partial Implementations

Normally, the RPC facility supports both clients and servers. It is possible, however, to implement an RPC subset that supports only clients or only servers. An implementation that supports only clients can omit packages Transport_Server and Rpc_Server. An implementation that supports only servers can omit package Rpc_Client.

Server Tasks

The RPC facility depends on there being at least one server process for each service on the server machine. This process receives request messages from clients in other machines, makes the appropriate local subprogram calls, and returns response messages back to the clients. The operator of each server machine must maintain the appropriate server process in running condition.

Each server process on each machine includes an instantiation of package Transport_Server. When the instantiation elaborates, it creates a waiter task that waits for incoming connections. Each time a connection arrives, the waiter allocates a new server task (from a collection). The server handles the connection's request messages and terminates itself when the connection is disconnected. The Finalize procedure destroys all server tasks. Package Transport_Server provides operations limiting the number of servers active at one time and allowing termination of the waiter and optional termination of running servers.

Shared Elaboration

Some programming environments allow the sharing of a single elaboration of a package among many users. The RPC facility does not require shared elaboration for correct operation. Shared elaboration can offer the following advantages:

- If several users share one instance of a client interface, they then share the underlying Transport_Stream.Pool_Id and so get their transport streams from a common pool. If enough clients use the transport streams infrequently, the total number of transport streams required may be reduced.
- Packages Transport_Server and Transport_Stream contain worker tasks that synchronize references to the package state. Shared elaboration allows these tasks to serve multiple users.

Resource Allocation

Package Transport_Stream has no way to recover resources held by terminated clients. In writing a client, the user should incorporate some means for streams held by inactive clients to be reclaimed by package Transport_Stream.

Termination

Several RPC packages provided contain tasks in their bodies. All these tasks must terminate before a program can exit from a scope in which one of these packages is elaborated. See the Ada rules for task dependence and termination (*Reference Manual for the Ada Programming Language*, Section 9.4).

To support termination using standard Ada, each package containing tasks provides a Finalize procedure. Finalize terminates all tasks contained in the package and cleans up other kinds of state. The Finalize procedure should be used with care, because the package will be left useless for subsequent operations.

In the Rational Environment, when a subprogram executes in a Command window, all packages the subprogram depends on (except system interface packages) are elaborated for the duration of the command. A subprogram cannot complete until all tasks it elaborated in the supporting packages have terminated. If a subprogram depends on the RPC facility, a command running that subprogram cannot terminate normally until it calls the appropriate Finalize procedures. If the Finalize procedures are not called, the subprogram will hang without returning control to the user. Clearly, the best way to avoid this is to include all Finalize calls in the subprogram. If the program still hangs, the !Commands.Job.Kill command can terminate the subprogram.

Porting Guidelines

Here are some hints for porting the RPC facility Ada packages to a new environment.

The types and operators in package Interchange_Defs should be modified to correspond to the appropriate local types and operators. Refer to the description of package Interchange_Defs in the RPC reference pages.

Implementing the body of package Transport requires:

- Purchasing or writing an implementation of a reliable data communications protocol—for example, TCP/IP.
- Writing software to map the Transport interface onto the interface provided by the communications software mentioned above. This software must allow the assignment of a Transport.Connection_Id. This assignment supports multiple copies of the same value denoting the same connection.
- I/O operations (transmit and receive) with optional timeout parameters.

package Interchange

This package defines a collection of types for data interchange. Each type has:

- A local representation dependent on the characteristics of the local architecture and compiler
- A machine-independent interchange representation (as a byte sequence)

This package provides operations for converting between the local and interchange representations. The types provided are analogous to types in the Ada predefined language environment (packages Standard and Calendar; see the *Rational Environment Reference Manual*, Programming Tools (PT book). The conversion operations are generic on a procedural byte stream, so that they can be used on a variety of interchange media (communication channels, tapes, and disks).

The interchange representation does not carry type information. Thus, to convert interchanged data back to their local representations, their types must be known beforehand.

The interchange representation largely follows the rules of Courier, the Xerox System Integration Standard for remote procedure call protocol. In particular, all interchange values are a multiple of two bytes (16 bits) in length. Values that occupy several bytes are represented with their most significant byte first.

The algorithms for conversion to and from the interchange form can be extended to any Ada type except access values and task types. The rules for forming new type conversion algorithms are:

- A discrete (integer subtype or enumeration type) is represented by its 'Pos, converted to a Short_Integer type. This representation rule is implemented by the generic package Operations.Discrete.
- A vector (one-dimensional array) is represented by its 'Length, followed by its elements in index order. The 'Length is represented as a Natural subtype. This kind of representation loses information about the 'First of the vector (as does Ada slice assignment). This representation rule is implemented by the generic package Operations.Vector.

- A record is represented by the sequence of its fields, in the order of their declaration. Inaccessible fields in variant records are omitted. In other words, the interchange representation is ordered in the same way as its value would be written as an Ada aggregate.

subtype Byte

```
subtype Byte is Byte_Defs.Byte;
```

Description

Defines an eight-bit byte.

The interchange representation is like a Short_Integer type.

References

type Short_Integer

subtype Byte_String

```
subtype Byte_String is Byte_Defs.Byte_String;
```

Description

Defines a string of bytes indexed by Standard.Integer.

The interchange representation is a byte count (represented as an Integer type) followed by the bytes themselves in index order. The bytes are packed so that consecutive bytes in the string are represented as consecutive bytes in the interchange form. An extra padding byte is added if the string contains an odd number of bytes. This padding makes the entire value a multiple of two bytes in length.

References

TRL, type Transport.Byte_Defs.Byte_String

exception Constraint_Error

```
Constraint_Error : exception;
```

Description

Occurs when an interchange conversion procedure encounters an out-of-range value.

This exception is analogous to the Standard.Constraint_Error exception.

function Convert

```
function Convert (X : Calendar.Time) return Interchange.Time;
function Convert (X : Interchange.Time) return Calendar.Time;
function Convert (X : Standard.Duration) return Interchange.Duration;
function Convert (X : Interchange.Duration) return Standard.Duration;
```

Description

Converts between time representations.

Parameters

`X : Calendar.Time;`
Specifies a time to be converted.

`return Interchange.Time;`
Specifies the equivalent time in interchange representation.

`X : Interchange.Time;`
Specifies a time to be converted.

`return Calendar.Time;`
Specifies the equivalent time in calendar representation.

`X : Standard.Duration;`
Specifes a duration to be converted.

`return Interchange.Duration;`
Specifies the equivalent duration in interchange representation.

`X : Interchange.Duration;`
Specifies a duration to be converted.

`return Standard.Duration;`
Specifies the equivalent duration in standard representation.

subtype Day_Duration

```
subtype Day_Duration is Interchange.Duration;
```

Description

Defines the interchange analog of Calendar.Day_Duration.

subtype Day_Number

```
subtype Day_Number is Interchange.Short_Integer range 1 .. 31;
```

Description

Defines the interchange analog of Calendar.Day_Number.

type Duration

```
type Duration is
  record
      Seconds      : Interchange.Integer;
      Nanoseconds  : Interchange.Nanosecond_Count;
  end record;
```

Description

Defines the interchange analog of Standard.Duration.

The interchange representation is the sequence of the fields in the order of their declaration (seconds, nanoseconds).

type Float

```
type Float is new Interchange_Defs.Float;
```

Description

Defines the interchange analog of Standard.Float.

The interchange representation is IEEE single-precision (32-bit) floating point.

type Integer

```
type Integer is new Interchange_Defs.Longest_Integer
            range - (2 ** 30) - (2 ** 30) .. (2 ** 30) + ((2 ** 30) - 1);
```

Description

Defines the interchange analog of a Standard.Integer.

The interchange representation is 32 bits, given in two's-complement notation.

type Long_Float

```
type Long_Float is new Interchange_Defs.Long_Float;
```

Description

Defines the interchange analog of a Standard.Long_Float.

The interchange representation is IEEE double-precision (64-bit) floating point.

type Long_Integer

```
type Long_Integer is new Interchange_Defs.Longest_Integer;
```

Description

Defines the interchange analog of Standard.Long_Integer.

The range of Long_Integer is machine-dependent and so should be chosen to be as large as possible.

The interchange representation is of arbitrary precision. Thus, as many bytes are used as are required to contain the value, represented as a Byte_String subtype (a byte count followed by the bytes). Two's-complement notation is used.

subtype Long_Natural

```
subtype Long_Natural is Long_Integer range 0 .. Long_Integer'Last;
```

Description

Defines the interchange analog of Standard.Long_Natural.

subtype Long_Positive

```
subtype Long_Positive is Long_Integer range 1 .. Long_Integer'Last;
```

Description

Defines the interchange analog of Standard.Long_Positive.

subtype Month_Number

```
subtype Month_Number is Interchange.Short_Integer range 1 .. 12;
```

Description

Defines the interchange analog of Calendar.Month_Number.

subtype Nanosecond_Count

```
subtype Nanosecond_Count is Interchange.Natural range 0 .. 10 ** 9 - 1;
```

Description

Defines the set of digits to the right of the decimal point in a Duration type.

References

type Duration

subtype Natural

```
subtype Natural is Integer range 0 .. Integer'Last;
```

Description

Defines the interchange analog of Standard.Natural.

subtype Positive

```
subtype Positive is Integer range 1 .. Integer'Last;
```

Description

Defines the interchange analog of Standard.Positive.

type Short_Integer

```
type Short_Integer is range - (2 ** 14) - (2 ** 14) .. (2 ** 14) =
                                                        ((2 ** 14) - 1);
```

Description

Defines the interchange analog of Standard.Short_Integer.

The interchange representation is 16 bits, given in two's-complement notation.

subtype Short_Natural

```
subtype Short_Natural is Short_Integer range 0 .. Short_Integer'Last;
```

Description

Defines the interchange analog of Standard.Short_Natural.

subtype Short_Positive

```
subtype Short_Positive is Short_Integer range 1 .. Short_Integer'Last;
```

Description

Defines the interchange analog of Standard.Short_Positive.

type Time

```
type Time is
    record
        Year     : Year_Number;
        Month    : Month_Number;
        Day      : Day_Number;
        Seconds  : Day_Duration;
    end record;
```

Description

Defines the interchange analog of Calendar.Time.

The interchange representation is the sequence of the field values given in the order of their declaration (year, month, day, seconds).

References

Rational Environment Reference Manual, PT, package Calendar

subtype Year_Number

```
subtype Year_Number is Interchange.Short_Integer;
```

Description

Defines the interchange analog of Calendar.Year_Number.

References

Rational Environment Reference Manual, PT, package Calendar

generic package Discrete

Given a discrete type, this package provides operations for interchanging values of that type.

The specification of the package is:

```
generic
    type Discrete_Type is (<>);
package Discrete is
    ...
end Discrete;
```

generic formal type Discrete_Type

```
type Discrete_Type is (<>);
```

Description

Defines the type to be interchanged.

The interchange representation of a value of this type is the 'Pos of the value converted to a Short_Integer type.

Restrictions

The 'Pos and 'Val attributes must be defined for this type. These attributes are used to convert to and from this type's interchange representation.

procedure Get

```
procedure Get (From :     Stream_Id;
               Data : out Discrete_Type);
```

Description

Converts a value of the Discrete_Type type from its interchange representation to its local representation.

Parameters

`From : Stream_Id;`

Specifies the stream from which to get the interchange representation.

`Data : out Discrete_Type;`

Specifies the local representation.

procedure Put

```
procedure Put (Into : Stream_Id;
               Data : Discrete_Type);
```

Description

Converts a value of the Discrete_Type type from its local representation to its interchange representation.

Parameters

`Into : Stream_Id;`
Specifies the stream into which to put the interchange representation.

`Data : Discrete_Type;`
Specifies the local representation.

end Discrete;

generic package Operations

Given a facility for interchanging bytes, package Operations provides a facility for interchanging values of other types.

For each type, Put and Get operations are defined. Put converts any value to a sequence of bytes; Get reconstructs the original value.

The specification of the package is:

```
generic
    type Stream_Id is limited private;
    with procedure Put (Into : Stream_Id;
                        Data : Byte_String) is <>;
    with procedure Get (From : Stream_Id;
                        Data : out Byte_String) is <>;
package Operations is
    ...
end Operations;
```

The generic formal parameters define an abstract byte stream and operations for interchanging bytes on it. To function, a stream must not lose bytes and must deliver bytes in order.

A stream must perform fragmentation and reassembly. That is, the sequence of operations:

```
Put (Into, Data (1 .. X));
Put (Into, Data (X + 1 .. N));
```

must be equivalent to the single operation:

```
Put (Into, Data (1 .. N));
```

for any value of **X** in the range **1 .. N**. Likewise, the sequence of operations:

```
Get (From, Data (1 .. X));
Get (From, Data (X + 1 .. N));
```

must be equivalent to the single operation:

```
Get (From, Data (1 .. N));
```

for any value of **X** in the range **1 .. N.**

procedure Get

```
procedure Get (From :     Stream_Id;
               Data : out Byte);

procedure Get (From :     Stream_Id;
               Data : out Interchange.Duration);

procedure Get (From :     Stream_Id;
               Data : out Interchange.Float);

procedure Get (From :     Stream_Id;
               Data : out Interchange.Integer);

procedure Get (From :     Stream_Id;
               Data : out Interchange.Long_Float);

procedure Get (From :     Stream_Id;
               Data : out Interchange.Long_Integer);

procedure Get (From :     Stream_Id;
               Data : out Interchange.Short_Integer);

procedure Get (From :     Stream_Id;
               Data : out Interchange.Time);

procedure Get (From :     Stream_Id;
               Data : out Standard.Boolean);

procedure Get (From :     Stream_Id;
               Data : out Standard.Character);
```

Description

Converts a value from its interchange representation to its local representation.

The interchange representation of a Standard.Boolean is given as a Short_Integer type, with the value 0 for false and 1 for true. The interchange representation of a Standard.Character is its 'Pos converted to a Short_Integer type.

Parameters

`From : Stream_Id;`

Specifies the stream from which to get the interchange representation.

`Data : out Byte;`

Returns the local representation.

```
Data : out Interchange.Duration;
```
Returns the local representation.

```
Data : out Interchange.Float;
```
Returns the local representation.

```
Data : out Interchange.Integer;
```
Returns the local representation.

```
Data : out Interchange.Long_Float;
```
Returns the local representation.

```
Data : out Interchange.Long_Integer;
```
Returns the local representation.

```
Data : out Interchange.Short_Integer;
```
Returns the local representation.

```
Data : out Interchange.Time;
```
Returns the local representation.

```
Data : out Standard.Boolean;
```
Returns the local representation.

```
Data : out Standard.Character;
```
Returns the local representation.

Errors

If the interchange representation is out of range, the Constraint_Error exception will be raised. This usually indicates that the data in the stream at this point are of some other type.

References

generic formal procedure Get

---

generic formal procedure Get

```
with procedure Get (From :      Stream_Id;
                    Data : out Byte_String) is <>;
```

Description

Gets some bytes from a stream.

Parameters

`From : Stream_Id;`
Specifies the stream.

`Data : out Byte_String;`
Specifies the bytes.

Errors

Any exceptions raised by this procedure will be propagated.

function Get_Byte_String

```
function Get_Byte_String (From : Stream_Id) return Byte_String;
```

Description

Converts a byte string from its interchange representation to its local representation.

Get_Byte_String is a function because Byte_String is an unconstrained subtype.

Parameters

`From : Stream_Id;`
Specifies the stream from which to get the interchange representation.

`return Byte_String;`
Returns the local representation.

References

procedure Get

procedure Put_Byte_String

function Get_String

```
function Get_String (From : Stream_Id) return Standard.String;
```

Description

Converts a string from its interchange representation to its local representation.

Get_String is a function because String is an unconstrained type.

The interchange representation of a Standard.String is given as a Byte_String subtype, with one character per byte. Each character is represented by its 'Pos (its ASCII code).

Parameters

`From : Stream_Id;`
Specifies the stream from which to get the interchange representation.

`return Standard.String;`
Returns the local representation.

Errors

If the interchange representation is out of range, the Constraint_Error exception will be raised. This usually indicates that the data in the stream at this point are of some other type.

References

procedure Get

procedure Put_String

procedure Put

```
procedure Put (Into : Stream_Id;
               Data : Byte);

procedure Put (Into : Stream_Id;
               Data : Interchange.Duration);

procedure Put (Into : Stream_Id;
               Data : Interchange.Float);

procedure Put (Into : Stream_Id;
               Data : Interchange.Integer);

procedure Put (Into : Stream_Id;
               Data : Interchange.Long_Float);

procedure Put (Into : Stream_Id;
               Data : Interchange.Long_Integer);

procedure Put (Into : Stream_Id;
               Data : Interchange.Short_Integer);

procedure Put (Into : Stream_Id;
               Data : Interchange.Time);

procedure Put (Into : Stream_Id;
               Data : Standard.Boolean);

procedure Put (Into : Stream_Id;
               Data : Standard.Character);
```

Description

Converts a value from its interchange representation to its local representation.

The interchange representation of a Standard.Boolean is given as a Short_Integer type, with the value 0 for false and 1 for true. The interchange representation of a Standard.Character is its 'Pos converted to a Short_Integer type.

Parameters

`Into : Stream_Id;`

Specifies the stream into which to put the interchange representation.

`Data : Byte;`

Specifies the local representation.

Data : Interchange.Duration;
Specifies the local representation.

Data : Interchange.Float;
Specifies the local representation.

Data : Interchange.Integer;
Specifies the local representation.

Data : Interchange.Long_Float;
Specifies the local representation.

Data : Interchange.Long_Integer;
Specifies the local representation.

Data : Interchange.Short_Integer;
Specifies the local representation.

Data : Interchange.Time;
Specifies the local representation.

Data : Standard.Boolean;
Specifies the local representation.

Data : Standard.Character;
Specifies the local representation.

References

procedure Get

---

generic formal procedure Put

```
with procedure Put (Into : Stream_Id;
                    Data : Byte_String) is <>;
```

Description

Puts some bytes into a stream.

Parameters

`Into : Stream_Id;`
Specifies the stream.

`Data : Byte_String;`
Specifies the bytes.

Errors

Any exceptions raised by this procedure will be propagated.

procedure Put_Byte_String

```
procedure Put_Byte_String (Into : Stream_Id;
                           Data : Byte_String);
```

Description

Converts a Byte_String from its local representation to its interchange representation.

Parameters

`Into : Stream_Id;`

Specifies the stream into which to put the interchange representation.

`Data : Byte_String;`

Specifies the local representation.

References

function Get_Byte_String

procedure Put

procedure Put_String

```
procedure Put_String (Into : Stream_Id;
                      Data : Standard.String);
```

Description

Converts a Standard.String from its local representation to its interchange representation.

The interchange representation of a Standard.String is a Byte_String subtype, with one character per byte. Each character is represented by its 'Pos (its ASCII code).

Parameters

`Into : Stream_Id;`
Specifies the stream into which to put the interchange representation.

`Data : Standard.String;`
Specifies the local representation.

References

function Get_String

procedure Put

generic formal type Stream_Id

```
type Stream_Id is limited private;
```

Description

Specifies that data be taken from the selected stream of bytes.

The interchange representation of values is put into or taken from a stream.

end Operations;

RATIONAL

generic package Vector

A vector is a one-dimensional array of elements. Given a vector type (an unconstrained array type), this package provides interchange operations on that type.

The specification of the package is:

```
generic
    type Element_Type is private;
    with procedure Put (Into : Stream_Id;
                        Data : Element_Type) is <>;
    with procedure Get (From : Stream_Id;
                        Data : out Element_Type) is <>;

    type Index_Type is (<>);
    with procedure Put (Into : Stream_Id;
                        Data : Index_Type) is <>;
    with procedure Get (From : Stream_Id;
                        Data : out Index_Type) is <>;

    type Vector_Type is array (Index_Type range <>) of Element_Type;

package Vector is
    ...
end Vector;
```

generic formal type Element_Type

```
type Element_Type is private;
```

Description

Specifies the type of each element of a vector.

The interchange representation of this type is supplied by the user as Put and Get generic formal procedures.

function Get

```
function Get (From : Stream_Id) return Vector_Type;
```

Description

Converts a vector from its interchange representation to its local representation.

Parameters

`From : Stream_Id;`
Specifies the stream from which to get the interchange representation.

`return Vector_Type;`
Returns the local representation.

generic formal procedure Get

```
with procedure Get (From :      Stream_Id;
                    Data : out Element_Type) is <>;

with procedure Get (From :      Stream_Id;
                    Data : out Index_Type) is <>;
```

Description

Converts a value from its interchange representation to its local representation.

Parameters

`From : Stream_Id;`

Specifies the stream from which to get the interchange representation.

`Data : out Element_Type;`

Returns a user-supplied data type.

`Data : out Index_Type;`

Returns the local representation.

generic formal type Index_Type

```
type Index_Type is (<>);
```

Description

Specifies the index type of a vector.

The interchange representation of this type is supplied by the user as Put and Get generic formal procedures.

procedure Put

```
procedure Put (Into : Stream_Id;
               Data : Vector_Type);
```

Description

Converts a vector from its local representation to its interchange representation.

Parameters

`Into : Stream_Id;`
Specifies the stream into which to put the interchange representation.

`Data : Vector_Type;`
Specifies the local representation.

generic formal procedure Put

```
with procedure Put (Into : Stream_Id;
                    Data : Element_Type) is <>;

with procedure Put (Into : Stream_Id;
                    Data : Index_Type) is <>;
```

Description

Converts a value from its local representation to its interchange representation.

Parameters

`Into : Stream_Id;`
Specifies the stream into which to put the interchange representation.

`Data : Element_Type;`
Specifies a user-supplied data type.

`Data : Index_Type;`
Specifies the local representation.

generic formal type Vector_Type

```
type Vector_Type is array (Index_Type range <>) of Element_Type;
```

Description

Represents a one-dimensional array of elements.

This is the type for which package Vector provides interchange operations.

The interchange representation of a value of this type is the number of elements (given as an Integer type) followed by the elements in index order.

end Vector;

```
end Interchange;
```

RATIONAL

package Interchange_Defs

Package Interchange_Defs defines numeric types and operators for package Interchange. When porting this package, you should modify it to indicate the appropriate types on your system.

function "="

```
function "=" (X, Y : Longest_Integer) return Boolean renames Standard."=";
```

Description

Renames the standard operator.

Parameters

```
X, Y : Longest_Integer;
```
Specifies the highest-precision signed integer.

```
return Boolean;
```
Returns true if **X = Y**.

function ">"

```
function ">" (X, Y : Longest_Integer) return Boolean renames Standard.">";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Boolean;`
Returns true if **X** = **Y**.

function "<"

```
function "<" (X, Y : Longest_Integer) return Boolean renames Standard."<";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Boolean;`
Returns true if **X** = **Y**.

function ">="

```
function ">=" (X, Y : Longest_Integer) return Boolean
                                                renames Standard.">=";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Boolean;`
Returns true if **X = Y**.

function "<="

```
function "<=" (X, Y : Longest_Integer) return Boolean
                                                renames Standard."<=";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Boolean;`
Returns true if **X = Y**.

function "+"

```
function "+" (X, Y : Longest_Integer) return Longest_Integer
                                                renames Standard."+";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`

Specifies the highest-precision signed integer.

`return Longest_Integer;`

Returns **X + Y**.

function "-"

```
function "-" (X, Y : Longest_Integer) return Longest_Integer
                                                    renames Standard."-";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Longest_Integer;`
Returns **X - Y**.

function "*"

```
function "*" (X, Y : Longest_Integer) return Longest_Integer
                                                renames Standard."*";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Longest_Integer;`
Returns **X * Y.**

function "/"

```
function "/" (X, Y : Longest_Integer) return Longest_Integer
                                                renames Standard."/";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Longest_Integer;`
Returns **X / Y.**

function "mod"

```
function "mod" (X, Y : Longest_Integer) return Longest_Integer
                                                renames Standard."mod";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`
Specifies the highest-precision signed integer.

`return Longest_Integer;`
Returns X mod Y.

function "rem"

```
function "rem" (X, Y : Longest_Integer) return Longest_Integer
                                                renames Standard."rem";
```

Description

Renames the standard operator.

Parameters

`X, Y : Longest_Integer;`

Specifies the highest-precision signed integer.

`return Longest_Integer;`

Returns X rem Y.

function Duration_Magnitude

```
function Duration_Magnitude return Standard.Integer;
```

Description

This function returns a constant that is:

- An integral power of 10 (10, 100, 1,000, and so on)
- <= 10 ** 9
- <= Standard.Duration'Last
- <= Standard.Integer'Last

This constant is used by the Interchange.Convert procedure.

type Float

```
type Float is new Standard.Float;
```

Description

Represents IEEE floating-point, single-precision format.

type Long_Float

```
type Long_Float is new Standard.Float;
```

Description

Represents IEEE floating-point, double-precision format.

subtype Longest_Integer

```
subtype Longest_Integer is Standard.Long_Integer;
```

Description

Acts as a signed integer of the highest available precision.

This type is used for internal arithmetic by the Interchange conversion algorithms.

Package Interchange_Defs must define the following arithmetic and comparison operators for type Longest_Integer:

=, >, <, >=, <=, +, -, *, /, mod, rem

In the Rational implementation, these are defined by renaming the operators from package Standard.

References

type Interchange.Long_Integer

end Interchange_Defs;

package Rpc

This package defines the data types used to represent programs, versions, procedures, and exceptions.

constant Defined_Versions

```
Defined_Versions : constant Version_Range := (3, 5);
```

Description

Defines RPC protocol versions.

type Error_Type

```
type Error_Type is (Error_Other, Error_Constraint, Error_Numeric,
                    Error_Program, Error_Storage, Error_Tasking,
                    Status_Error, Mode_Error, Name_Error, Use_Error,
                    Device_Error, End_Error, Data_Error, Layout_Error,
                    Error_Server_Defined, Error_Username_Or_Password);
```

Description

Represents exceptions in RPC response message headers.

Each value corresponds to an exception or class of exceptions. Refer to the table of "Error Messages" in the reference entry for the Get_Message function.

This type indicates a server-defined error. These exceptions indicate the general cause of an error. The exact cause depends on the particulars of the user's server.

Enumerations

`Error_Other`
Indicates any of a variety of errors, excluding the following errors.

`Data_Error`
Indicates data error.

`Device_Error`
Indicates device error.

`End_Error`
Indicates end error.

`Error_Constraint`
Indicates contraint error.

`Error_Numeric`
Indicates numeric error.

`Error_Program`
Indicates program error.

Error_Server_Defined
Indicates other server-defined error.

Error_Storage
Indicates storage error.

Error_Tasking
Indicates tasking error.

Error_Username_Or_Password
Indicates username or password error.

Layout_Error
Indicates layout error.

Mode_Error
Indicates mode error.

Name_Error
Indicates name error.

Status_Error
Indicates status error.

Use_Error
Indicates use error.

References

type Exception_Number

function Get_Message

---

type Exception_Number

```
type Exception_Number is new Interchange.Integer;
```

Description

Represents server-defined exceptions in RPC response headers.

Each value corresponds to an exception. The correspondence is determined by the service that raises the exception.

References

type Message_Header

constant Exception_Versions

```
Exception_Versions : constant Version_Range := (4, Version_Number'Last);
```

Description

Defines RPC protocol versions that support server-defined exceptions.

procedure Get

```
procedure Get (From :     Stream_Id;
               Data : out Version_Range);

procedure Get (From :     Stream_Id;
               Data : out Exception_Number);
```

Description

Gets a value of the Version_Range or Exception_Number types from a stream.

Parameters

`From : Stream_Id;`
Specifies the stream from which to get the value.

`Data : out Version_Range;`
Returns the value got.

`Data : out Exception_Number;`
Returns the value got.

References

procedure Put

function Get_Message

```
function Get_Message (From : Stream_Id) return Message_Header;
```

Description

Gets a value of the Message_Header type from a stream.

Refer to the table of "Error Messages" on the following page.

Parameters

`From : Stream_Id;`
Specifies the stream from which to get the value.

`return Message_Header;`
Returns the value got.

Errors

If the message header has Kind => Reject_Message or Kind => Abort_Message, an exception will be raised. The particular exception is determined by the rest of the message header.

Error Messages

Message Header	*Exception Raised*
(Reject_Message,(Rej_No_Such_Program))	No_Such_Program
(Reject_Message,(Rej_No_Such_Version))	No_Such_Version
(Reject_Message,(Rej_No_Such_Procedure))	No_Such_Procedure
(Reject_Message,(Rej_Invalid_Argument))	Invalid_Argument
(Abort_Message,(Error_Other))	Other_Error
(Abort_Message,(Error_Constraint))	Standard.Constraint_Error
(Abort_Message,(Error_Numeric))	Standard.Numeric_Error
(Abort_Message,(Error_Program))	Standard.Program_Error
(Abort_Message,(Error_Storage))	Standard.Storage_Error
(Abort_Message,(Error_Tasking))	Standard.Tasking_Error
(Abort_Message,(Status_Error))	Io_Exceptions.Status_Error
(Abort_Message,(Mode_Error))	Io_Exceptions.Mode_Error
(Abort_Message,(Name_Error))	Io_Exceptions.Name_Error
(Abort_Message,(Use_Error))	Io_Exceptions.Use_Error
(Abort_Message,(Device_Error))	Io_Exceptions.Device_Error
(Abort_Message,(End_Error))	Io_Exceptions.End_Error
(Abort_Message,(Data_Error))	Io_Exceptions.Data_Error
(Abort_Message,(Layout_Error))	Io_Exceptions.Layout_Error
(Abort_Message,(Error_Server_Defined))	Server_Defined_Error
(Abort_Message,(Error_Username_Or_Password))	Username_Or_Password_Error

Error_Server_Defined corresponds to a class of exceptions defined by particular services. When a response header indicates an exception, with Error => Error-_Server_Defined, it is followed immediately by a value of the Exception_Number type, which identifies the exception.

References

procedure Put_Message

exception Invalid_Argument

```
Invalid_Argument : exception;
```

Description

Occurs when the interchange form of an argument is invalid.

An example would be the encoding of a value outside the legal range.

References

function Get_Message

function Max

```
function Max (X, Y : Version_Range) return Version_Number;
```

Description

Returns the largest version that is common to both **X** and **Y**.

If the server supports a range of protocol versions and the client supports a separate range of protocol versions, this function returns the largest version of the protocol that is common to both the client and the server.

Parameters

`X, Y : Version_Range;`
Specifies a version number.

`return Version_Number;`
Returns the largest version that is common to both **X** and **Y**.

type Message_Header

```
type Message_Header (Kind : Message_Kind := Return_Message) is
    record
        Id : Transaction_Id :=Ø;
        case Kind is
           when Call_Message =>
             Program : Program_Number;
             Version : Version_Number;
             Proc    : Procedure_Number;
           when Reject_Message =>
             Details : Reject_Details;
           when Return_Message =>
             null;
           when Abort_Message =>
             Error : Error_Type;
         end case;
     end record;
```

Description

Begins each RPC message (request or response) with introductory information about the message.

References

function Get_Message

procedure Put_Message

type Message_Kind

```
type Message_Kind is (Call_Message, Reject_Message, Return_Message,
                                                    Abort_Message);
```

Description

Acts as the discriminant of the Message_Header type.

Each RPC message (request or response) must be one of the above kinds.

Enumerations

Abort_Message
Indicates that an exception was raised by the called subprogram.

Call_Message
Indicates the initial request for service sent by client to server.

Reject_Message
Indicates some problem with the requested program number, version number, or procedure number.

Return_Message
Indicates normal completion of the requested operation.

References

type Message_Header

exception No_Such_Procedure

```
No_Such_Procedure : exception;
```

Description

Occurs when a nonexistent procedure is called.

This exception is raised when a server receives a request message containing a procedure number that does not identify an existing procedure. The exception may indicate a disagreement between client and server about procedure numbering.

References

function Get_Message

exception No_Such_Program

```
No_Such_Program : exception;
```

Description

Occurs when a nonexistent program is called.

This exception is raised when a server receives a request message containing a program number that does not identify an existing program. The exception may indicate a disagreement between client and server about program numbering.

References

function Get_Message

exception No_Such_Version

```
No_Such_Version : exception;
```

Description

Occurs when a nonexistent version is called.

This exception is raised when a server receives a request message containing a version number that does not identify a supported version.

References

function Get_Message

exception Other_Error

```
Other_Error : exception;
```

Description

Occurs under a variety of circumstances.

This exception is raised by the Get_Message function to indicate that an unknown exception has been raised in the server.

References

function Get_Message

function Overlaps

```
function Overlaps (X, Y : Version_Range) return Boolean:
```

Description

Checks whether two ranges of version numbers overlap.

Parameters

`X, Y : Version_Range;`
Specifies a range of version numbers.

`return Boolean;`
Returns true if the ranges **X** and **Y** overlap (have at least one version in common).

type Procedure_Number

```
type Procedure_Number is new Interchange.Short_Integer;
```

Description

Identifies a procedure within a program.

The correspondence between procedure numbers and procedures is determined by each service.

References

type Message_Header

type Program_Number

```
type Program_Number is new Interchange.Integer;
```

Description

Identifies a program.

References

type Message_Header

exception Protocol_Error

```
Protocol_Error : exception;
```

Description

Occurs when an event happens that is not legal in the RPC protocol.

An example event would be a server receiving a response message.

procedure Put

```
procedure Put (Into : Stream_Id;
               Data : Exception_Number);
```

Description

Puts a value of the Exception_Number type into a stream.

Parameters

`Into : Stream_Id;`
Specifies the stream into which to put the value.

`Data : Exception_Number;`
Specifies the value to put.

procedure Put

```
procedure Put (Into : Stream_Id;
               Data : Version_Range);
```

Description

Puts a value of the Version_Range type into a stream.

Parameters

`Into : Stream_Id;`
Specifies the stream into which to put the value.

`Data : Version_Range;`
Specifies the value to put.

procedure Put_Message

```
procedure Put_Message (Into : Stream_Id;
                       Data : Message_Header);
```

Description

Puts a value of the Message_Header type into a stream.

Parameters

`Into : Stream_Id;`
Specifies the stream into which to put the value.

`Data : Message_Header;`
Specifies the value to put.

type Reject_Details

```
type Reject_Details (Kind : Reject_Kind := Rej_Invalid_Argument) is
    record
      case Kind is
          when Rej_No_Such_Version =>
              Supported : Version_Range;
          when other =>
              null;
      end case;
    end record;
```

Description

Identifies the reason for rejecting an RPC request.

References

type Message_Header

type Reject_Kind

```
type Reject_Kind is (Rej_No_Such_Program, Rej_No_Such_Version,
                     Rej_No_Such_Procedure, Rej_No_Such_Argument);
```

Description

Acts as the discriminant of the Reject_Details type.

References

type Reject_Details

exception Server_Defined_Error

```
Server_Defined_Error : exception;
```

Description

Indicates that some server-defined exception has been raised.

subtype Stream_Id

```
subtype Stream_Id is Transport_Stream.Stream_Id;
```

Description

Identifies a bidirectional stream of bytes.

type Transaction_Id

```
type Transaction_Id is new Interchange.Short_Integer;
```

Description

Identifies a transaction.

A transaction is associated with each remote procedure call. This type is not used currently.

References

type Message_Header

exception Username_Or_Password_Error

```
Username_Or_Password_Error  : exception;
```

Description

Occurs when the username and/or password supplied with a remote procedure call is rejected by the server machine.

This exception is raised when either there is no such username or the password is incorrect.

constant Username_Versions

```
Username_Versions : constant Version_Range := (5, Version_Number'Last);
```

Description

Defines RPC protocol versions that support passing username and password information with each call.

type Version_Number

```
type Version_Number is new Interchange.Short_Integer;
```

Description

Identifies a particular version of a program (service).

type Version_Range

```
type Version_Range is
      record
          First, Last : Version_Number;
      end record;
```

Description

Identifies a range of versions.

By convention, a value X of the Version_Range type represents versions in the Ada range: X.First .. X.Last.

end Rpc;

RATIONAL

package Rpc_Access_Utilities

Package Rpc_Access_Utilities is used by RPC client programs to obtain usernames and passwords to control the identity of RPC servers running on other machines.

function Remote_Password

```
function Remote_Password
      (Host_Name : String;
       Response  : Profile.Response_Profile := Profile.Get) return String;
```

Description

Returns a password to be used by the machine named by the Host_Name parameter.

This function reads the object named by the Remote_Passwords switch in the given response profile, searching for a line that matches the given Host_Name. If a matching line is found, and the password field in that line is "<PROMPT>", the calling user is prompted to enter a password in an I/O window, and the entered password is returned. If the password field is not "<PROMPT>", the field is returned. If there is no line in the Remote_Passwords file that matches the Host_Name, the null string ("") is returned.

Parameters

`Host_Name : String;`

Specifies the name of the machine for which a password is needed.

`Response : Profile.Response_Profile := Profile.Get;`

Specifies the response profile by which to obtain a password. In this profile, the value of the Remote_Passwords switch, if nonnull, is the name of the object from which to read the password.

`return String;`

Returns a password on the machine named by the Host_Name parameter.

function Remote_Session

```
function Remote_Session
      (Host_Name : String;
       Response  : Profile.Response_Profile  := Profile.Get) return String;
```

Description

Returns a session (account) name to be used by the machine named by the Host_Name parameter.

This function reads the object named by the Remote_Sessions switch in the given response profile, searching for a line that matches the given Host_Name. If a matching line is found, and the session field in that line is "<PROMPT>", the calling user is prompted to enter a session in an I/O window, and the entered session is returned. If the session field is not "<PROMPT>", the field is returned. If there is no line in the Remote_Sessions file that matches the Host_Name, the null string ("") is returned.

Parameters

`Host_Name : String;`

Specifies the name of the machine for which a session (account) name is needed.

`Response : Profile.Response_Profile := Profile.Get;`

Specifies the response profile by which to obtain a session (account) name. In this profile, the value of the Remote_Sessions switch, if nonnull, is the name of the object from which to read the session (account) name.

`return String;`

Returns a session (account) name on the machine named by the Host_Name parameter.

function Remote_Username

```
function Remote_Username
      (Host_Name : String;
       Response  : Profile.Response_Profile := Profile.Get) return String;
```

Description

Returns a username to be used by the machine named by the Host_Name parameter.

This function reads the object named by the Remote_Passwords switch in the given response profile, searching for a line that matches the given Host_Name. If a matching line is found, and the username field in that line is "<PROMPT>", the calling user is prompted to enter a username in an I/O window, and the entered username is returned. If the username field is not "<PROMPT>", the field is returned. If there is no line in the Remote_Passwords file that matches the Host_Name, the null string ("") is returned.

Parameters

`Host_Name : String;`

Specifies the name of the machine for which a username is needed.

`Response : Profile.Response_Profile := Profile.Get;`

Specifies the response profile by which to obtain a username. In this profile, the value of the Remote_Passwords switch, if nonnull, is the name of the object from which to read the username.

`return String;`

Returns a username on the machine named by the Host_Name parameter.

procedure Start_Request_Generic

```
generic
    Default_Host_Name  : String;
    Default_Socket     : Transport_Defs.Socket_Id;
    Default_Program    : Rpc.Program_Number;
    Default_Version    : Rpc.Version_Number;
    Default_Username   : String                    := "";
    Default_Password   : String                    := "";

procedure Start_Request_Generic
         (Stream     : out Transport_Stream.Stream_Id;
          Proc       :     Rpc.Procedure_Number;
          Host_Name  :     String                      := Default_Host_Name;
          Socket     :     Transport_Defs.Socket_Id    := Default_Socket;
          Program    :     Rpc.Program_Number          := Default_Program;
          Version    :     Rpc.Version_Number          := Default_Version;
          Username   :     String                      := Default_Username;
          Password   :     String                      := Default_Password;
          Response   :     Profile.Response_Profile    := Profile.Get);
```

Description

Allocates a stream and transmits a request header with the specified program, version, procedure, username, and password values.

Host_Name is resolved to a Host_Id and a Network_Name by package Transport_Name. If Username = "", it is replaced by Remote_Username (Host_Name, Response). If Password = "", it is replaced by Remote_Password (Host_Name, Response).

This procedure is similar to the Rpc_Client.Start_Request_With_Username generic procedure, with the addition of Host_Name resolution and Remote_Password file processing.

Parameters

`Default_Host_Name : String;`

Specifies the default value of the host name.

`Default_Socket : Transport_Defs.Socket_Id;`

Specifies the default value of the socket.

`Default_Program : Rpc.Program_Number;`

Specifies the default value of the program.

```
Default_Version : Rpc.Version_Number;
```
Specifies the default value of the version.

```
Default_Username : String := "";
```
Specifies the default value of the username.

```
Default_Password : String := "";
```
Specifies the default value of the password.

```
Stream : out Transport_Stream.Stream_Id;
```
Returns the allocated stream.

```
Proc : Rpc.Procedure_Number;
```
Specifies the procedure number to transmit in the request message header.

```
Host_Name : String := Default_Host_Name;
```
Specifies the name of the host to which to connect. This name is resolved to a Network_Name and a Host_Id, using package Transport_Name, and these values are used, together with the specified socket, to establish a connection.

```
Socket : Transport_Defs.Socket_Id := Default_Socket;
```
Specifies the socket to which to connect.

```
Program : Rpc.Program_Number := Default_Program;
```
Specifies the program number to transmit in the request message header.

```
Version : Rpc.Version_Number := Default_Version;
```
Specifies the version number to transmit in the request message header.

```
Username : String := Default_Username;
```
Specifies the username to transmit in the request message header. If this value is null (""), Remote_Username (Host_Name, Response) is transmitted instead.

The remote RPC server uses the transmitted value to set its identity, thereby determining its access rights while it services this request.

```
Password : String := Default_Password;
```

Specifies the password to transmit in the request message header. If this value is null (""), Remote_Password (Host_Name, Response) is transmitted instead.

The remote RPC server uses the transmitted value to set its identity, thereby determining its access rights while it services this request.

```
Response : Profile.Response_Profile := Profile.Get;
```

Specifies the reponse profile by which to read the remote username and/or password.

References

function Remote_Password

function Remote_Username

generic procedure Rpc_Client.Start_Request_With_Username

---

end Rpc_Access_Utilities;

RATIONAL

package Rpc_Client

Package Rpc_Client provides operations used by clients to initiate remote procedure calls.

procedure End_Request

```
procedure End_Request (Stream : Transport_Stream.Stream_Id);
```

Description

Flushes the transmit buffer and gets a response header.

Either this procedure or the End_Request_With_Exception procedure must be called by each client procedure after transmitting all request parameters but before receiving any response parameters.

Parameters

`Stream : Transport_Stream.Stream_Id;`

Specifies the stream into which request parameters are put and from which response parameters are got.

Errors

If the stream is disconnected, the Transport_Stream.Not_Connected exception is raised.

generic procedure End_Request_With_Exception

Flushes the transmit buffer and gets a response header.

If the reponse header contains a server-defined exception, it is raised by the Raise-_Exception procedure. Either this procedure or the End_Request procedure is called by each client procedure after transmitting all request parameters but before receiving any response parameters.

The formal parameters to the generic procedure are:

```
generic
    with procedure Raise_Exception (Excep : Rpc.Exception_Number);
procedure End_Request_With_Exception
     (Stream : Transport_Stream.Stream_Id);
```

procedure End_Request_With_Exception

```
procedure End_Request_With_Exception
                  (Stream : Transport_Stream.Stream_Id);
```

Description

Flushes the transmit buffer and gets a response header.

If the reponse header contains a server-defined exception, it is raised by the Raise-_Exception procedure. Either this procedure or the End_Request procedure is called by each client procedure after transmitting all request parameters but before receiving any response parameters.

Parameters

`Stream : Transport_Stream.Stream_Id;`
Specifies the transport stream.

generic formal procedure Raise_Exception

```
with procedure Raise_Exception (Excep : Rpc.Exception_Number);
```

Description

Raises a server-defined exception.

Parameters

Excep : Rpc.Exception_Number;
Specifies the number identifying the exception to be raised.

procedure End_Response

```
procedure End_Response (Stream : Transport_Stream.Stream_Id);
```

Description

Deallocates the stream.

This procedure is called by each client procedure after getting all response parameters.

Parameters

```
Stream : Transport_Stream.Stream_Id;
```

Specifies the stream from which to get response parameters.

generic procedure Start_Request_Generic

Allocates a stream from the pool and transmits a request message header with the given program, version, and procedure values.

The Stream parameter returns the identification number of the stream into which to put request parameters and from which to get response parameters.

The Proc parameter specifies the number of the remote procedure called.

The formal parameters to the generic procedure are:

```
generic
    Default_Network : Transport_Defs.Network_Name;
    Default_Host : Transport_Defs.Host_Id;
    Default_Socket : Transport_Defs.Socket_Id;
    Default_Program : Rpc.Program_Number;
    Default_Version : Rpc.Version_Number;
procedure Start_Request_Generic
     (Stream : out Transport_Stream.Stream_Id;
      Proc : Rpc.Procedure_Number;
      Network : Transport_Defs.Network_Name := Default_Network;
      Host : Transport_Defs.Host_Id := Default_Host;
      Socket : Transport_Defs.Socket_Id := Default_Socket;
      Program : Rpc.Program_Number := Default_Number;
      Version : Rpc.Version_Number := Default_Version);
```

generic formal object Default_Host

```
Default_Host : Transport_Defs.Host_Id;
```

Description

Defines the host identifier that is to be used as the default host by the Start_Request_Generic procedure.

generic formal object Default_Network

```
Default_Network  : Transport_Defs.Network_Name;
```

Description

Defines the network name that is to be used as the default name by the Start_Request_Generic procedure.

generic formal object Default_Program

```
Default_Program : Rpc.Program_Number;
```

Description

Defines the program number that is to be used as the default number by the Start-_Request_Generic procedure.

generic formal object Default_Socket

```
Default_Socket : Transport_Defs.Socket_Id;
```

Description

Defines the socket identifier that is to be used as the default socket by the Start_Request_Generic procedure.

generic formal object Default_Version

```
Default_Version : Rpc.Version_Number;
```

Description

Defines the version number that is to be used as the default number by the Start-_Request_Generic procedure.

procedure Start_Request_Generic

```
procedure Start_Request_Generic
          (Stream : out Transport_Stream.Stream_Id;
           Proc : Rpc.Procedure_Number;
           Network : Transport_Defs.Network_Name := Default_Network;
           Host : Transport_Defs.Host_Id := Default_Host;
           Socket : Transport_Defs.Socket_Id := Default_Socket;
           Program : Rpc.Program_Number := Default_Number;
           Version : Rpc.Version_Number := Default_Version);
```

Description

Allocates a stream from the pool and transmits a request message header with the given program, version, and procedure values.

Parameters

`Stream : out Transport_Stream.Stream_Id;`

The Stream parameter returns the identification number of the stream into which to put request parameters and from which to get response parameters.

`Proc : Rpc.Procedure_Number;`

The Proc parameter specifies the number of the remote procedure called.

RATIONAL

generic procedure Start_Request_With_Username

Allocates a stream from the pool and transmits a request message header with the given program, version, and procedure values.

The Stream parameter returns the identification number of the stream into which to put request parameters and from which to get response parameters.

The Proc parameter specifies the number of the remote procedure called.

The Username and Password parameters can be passed to the server when the server must assume an identity in the access control system of the server machine.

The formal parameters to the generic procedure are:

```
generic
    Default_Network  : Transport_Defs.Network_Name;
    Default_Host  : Transport_Defs.Host_Id;
    Default_Socket  : Transport_Defs.Socket_Id;
    Default_Program  : Rpc.Program_Number;
    Default_Version  : Rpc.Version_Number;
    Default_Username  : String := "";
    Default_Password  : String := "";
procedure Start_Request_With_Username
     (Stream : out Transport_Stream.Stream_Id;
      Proc : Rpc.Procedure_Number;
      Network  : Transport_Defs.Network_Name  := Default_Network;
      Host  : Transport_Defs.Host_Id  := Default_Host;
      Socket  : Transport_Defs.Socket_Id  := Default_Socket;
      Program  : Rpc.Program_Number  := Default_Program;
      Version  : Rpc.Version_Number  := Default_Version;
      Username  : String := Default_Username;
      Password  : String := Default_Password);
```

generic formal object Default_Host

```
Default_Host : Transport_Defs.Host_Id;
```

Description

Defines the host identifier that is to be used as the default host by the Start_Request_With_Username procedure.

generic formal object Default_Network

```
Default_Network  : Transport_Defs.Network_Name;
```

Description

Defines the network name that is to be used as the default name by the Start_Request_With_Username procedure.

generic formal object Default_Password

```
Default_Password : String := "";
```

Description

Defines the password that is to be used as the default by the Start_Request_With-_Username procedure.

generic formal object Default_Program

```
Default_Program : Rpc.Program_Number;
```

Description

Defines the program number that is to be used as the default number by the Start-_Request_With_Username procedure.

generic formal object Default_Socket

```
Default_Socket : Transport_Defs.Socket_Id;
```

Description

Defines the socket identifier that is to be used as the default socket by the Start-_Request_With_Username procedure.

generic formal object Default_Username

```
Default_Username : String := "";
```

Description

Defines the username that is to be used as the default by the Start_Request_With-_Username procedure.

generic formal object Default_Version

```
Default_Version : Rpc.Version_Number;
```

Description

Defines the version number that is to be used as the default number by the Start-_Request_With_Username procedure.

procedure Start_Request_With_Username

```
procedure Start_Request_With_Username
          (Stream : out Transport_Stream.Stream_Id;
           Proc : Rpc.Procedure_Number;
           Network : Transport_Defs.Network_Name := Default_Network;
           Host : Transport_Defs.Host_Id := Default_Host;
           Socket : Transport_Defs.Socket_Id := Default_Socket;
           Program : Rpc.Program_Number := Default_Program;
           Version : Rpc.Version_Number := Default_Version;
           Username : String := Default_Username;
           Password : String := Default_Password);
```

Description

Allocates a stream from the pool and transmits a request message header with the given program, version, and procedure values.

The Username and Password parameters can be passed to the server when the server must assume an identity in the access control system of the server machine.

Parameters

```
Stream : out Transport_Stream.Stream_Id;
```

The Stream parameter returns the identification number of the stream into which to put request parameters and from which to get response parameters.

```
Proc : Rpc.Procedure_Number;
```

The Proc parameter specifies the number of the remote procedure called.

end Rpc_Client;

RATIONAL

package Rpc_Product

Package Rpc_Product provides a way to check whether the RPC product has been installed on your machine.

function Is_Installed

```
function Is_Installed return Boolean;
```

Description

Returns true if the RPC product has been installed on your machine.

exception Is_Not_Installed

```
Is_Not_Installed : exception;
```

Description

Occurs when an RPC subprogram is called, if the RPC product is not installed on your machine.

end Rpc_Product;

RATIONAL

package Rpc_Server

Package Rpc_Server provides operations used by servers to handle remote procedure calls.

procedure Begin_Response

```
procedure Begin_Response (Stream : Transport_Stream.Stream_Id;
                          Id     : Rpc.Transaction_Id);
```

Description

Transmits a response header.

This procedure is called from the body of the Process_Call procedure after request parameters are got from the stream but before response parameters are put into the stream.

Parameters

`Stream : Transport_Stream.Stream_Id;`

Specifies the stream from which request parameters are got and into which response parameters are put.

`Id : Rpc.Transaction_Id;`

Specifies the transaction identification number for a remote procedure call. This value is copied from the Id parameter to the Process_Call procedure.

Restrictions

This procedure must ***not*** be called before returning an exception.

References

procedure Return_Exception

procedure Return_Exception

```
procedure Return_Exception (Stream : Transport_Stream.Stream_Id;
                            Id     : Rpc.Transaction_Id;
                            Excep  : Rpc.Exception_Number);
```

Description

Transmits an exception message in response to an RPC request.

This procedure is called by the Process_Call procedure.

Parameters

`Stream : Transport_Stream.Stream_Id;`

Specifies the stream from which the request is got and into which the exception message is put.

`Id : Rpc.Transaction_Id;`

Specifies the transaction identification number for a remote procedure call. This value is copied from the Id parameter to the Process_Call procedure.

`Excep : Rpc.Exception_Number;`

Specifies the exception number to be returned to the caller.

RATIONAL

generic procedure Serve

Services an incoming RPC connection.

This procedure:

- Allocates a transport stream.
- Checks compatibility of incoming package and version. An exception message is returned if there is a mismatch.
- Processes calls until the connection is disconnected.
- Catches, on each call, any propagated exceptions, transmitting them and flushing the transmit buffer.
- Deallocates resources after disconnection.

The formal parameters to the generic procedure are:

```
generic
    Program : Rpc.Program_Number;
    Supported : Rpc.Version_Range := (0, Rpc.Version_Number'Last);
    with procedure Process_Call (Stream : Transport_Stream.Stream_Id;
                                 Id : Rpc.Transaction_Id;
                                 Version : Rpc.Version_Number;
                                 Proc : Rpc.Procedure_Number) is <>;
procedure Serve (Connection : Transport.Connection_Id);
```

generic formal procedure Process_Call

```
with procedure Process_Call (Stream  : Transport_Stream.Stream_Id;
                             Id      : Rpc.Transaction_Id;
                             Version : Rpc.Version_Number;
                             Proc    : Rpc.Procedure_Number) is <>;
```

Description

Processes one remote procedure call.

This procedure is supplied by the RPC server user and is called by an instantiation of the Serve_With_Username procedure.

Parameters

`Stream : Transport_Stream.Stream_Id;`

Specifies the stream from which request parameters are got and into which response parameters are put.

`Id : Rpc.Transaction_Id;`

Specifies the transaction identification number for a remote procedure call.

`Version : Rpc.Version_Number;`

Specifies the version of the RPC service (package) called.

`Proc : Rpc.Procedure_Number;`

Specifies the number of the procedure called.

Restrictions

The body of this procedure must get the request parameters from the stream and then do one of the following:

- Raise an exception
- Call the Return_Exception procedure and return
- Call the Begin_Response procedure, put response parameters into the stream, and return

---

Errors

Exceptions raised by this procedure are caught and transmitted as exception response messages.

References

procedure Begin_Response

procedure Return_Exception

procedure Serve

---

generic formal object Program

```
Program : Rpc.Program_Number;
```

Description

Supplies the number of the program (package) supported by this RPC server.

Errors

If the Serve procedure gets a request message that specifies some other program number, it responds with an exception message, without calling the Process_Call procedure.

References

procedure Serve

procedure Serve

```
procedure Serve (Connection : Transport.Connection_Id);
```

Description

Serves an incoming RPC connection by allocating a transport stream, checking the incoming package and version for a match, and processing calls until the connection is disconnected.

For each RPC call, this procedure catches any propagated exceptions, transmits them, and then flushes the transmit buffer. When the connection is disconnected or a protocol error occurs, the procedure deallocates the transport stream and returns.

Parameters

`Connection : Transport.Connection_Id;`

Specifies the identifier of the connection being served.

References

generic procedure Serve_With_Username

generic formal object Supported

```
Supported : Rpc.Version_Range := (0, Rpc.Version_Number'Last);
```

Description

Indicates the range of versions supported by this RPC server.

Errors

If a request for an unsupported version is received, an exception message is returned without calling the Process_Call procedure.

generic procedure Serve_With_Username

Services an incoming RPC connection.

This procedure:

- Allocates a transport stream
- Checks compatibility of incoming package and version. An exception message is returned if there is a mismatch
- Processes calls until the connection is disconnected
- Catches, on each call, any propagated exceptions, transmitting them and flushing the transmit buffer
- Deallocates resources after disconnection

The formal parameters to the generic procedure are:

```
generic
    Program : Rpc.Program_Number;
    Supported : Rpc.Version_Range := (0, Rpc.Version_Number'Last);
    with procedure Process_Call (Stream : Transport_Stream.Stream_Id;
                                 Id : Rpc.Transaction_Id;
                                 Version : Rpc.Version_Number;
                                 Proc : Rpc.Procedure_Number;
                                 Username : String;
                                 Password : String) is <>;
procedure Serve_With_Username (Connection : Transport.Connection_Id);
```

generic formal procedure Process_Call

```
with procedure Process_Call (Stream   : Transport_Stream.Stream_Id;
                             Id       : Rpc.Transaction_Id;
                             Version  : Rpc.Version_Number;
                             Proc     : Rpc.Procedure_Number;
                             Username : String;
                             Password : String) is <>;
```

Description

Processes one remote procedure call.

This procedure is supplied by the RPC server user and is called by an instantiation of the Serve procedure.

Parameters

`Stream : Transport_Stream.Stream_Id;`
Specifies the stream from which request parameters are got and into which response parameters are put.

`Id : Rpc.Transaction_Id;`
Specifies the transaction identification number for a remote procedure call.

`Version : Rpc.Version_Number;`
Specifies the version of the RPC service (package) called.

`Proc : Rpc.Procedure_Number;`
Specifies the number of the procedure called.

`Username : String;`
Specifies the username to be used with the procedure call.

`Password : String;`
Specifies the password to be used with the procedure call.

Restrictions

The body of this procedure must get the request parameters from the stream and then do one of the following:

- Raise an exception
- Call the Return_Exception procedure and return
- Call the Begin_Response procedure, put response parameters into the stream, and return

Errors

Exceptions raised by this procedure are caught and transmitted as exception response messages.

References

procedure Begin_Response

procedure Return_Exception

procedure Serve

generic formal object Program

```
Program : Rpc.Program_Number;
```

Description

Supplies the number of the program (package) supported by this RPC server.

Errors

If the Serve procedure gets a request message that specifies some other program number, it responds with an exception message, without calling the Process_Call procedure.

References

procedure Serve

---

procedure Serve_With_Username

```
procedure Serve_With_Username (Connection : Transport.Connection_Id);
```

Description

Serves an incoming RPC connection by allocating a transport stream, checking the incoming package and version for a match, and processing calls until the connection is disconnected.

For each RPC call, this procedure catches any propagated exceptions, transmits them, and then flushes the transmit buffer. When the connection is disconnected or a protocol error occurs, the procedure deallocates the transport stream and returns.

Parameters

`Connection : Transport.Connection_Id;`

Specifies the identifier of the connection being served.

References

procedure Serve

generic formal object Supported

```
Supported : Rpc.Version_Range := (Ø, Rpc.Version_Number'Last);
```

Description

Indicates the range of versions supported by this RPC server.

Errors

If a request for an unsupported version is received, an exception message is returned without calling the Process_Call procedure.

end Rpc_Server;

package Transport_Interchange

Instantiates package Interchange for use on a transport stream.

Package Transport_Interchange provides procedures for interchanging various standard types over transport connections.

This package is used by RPC programs for interchanging the parameters and results of remote procedure calls.

The specification of the package is:

```
with Interchange;
with Transport_Stream;

package Transport_Interchange is new Interchange.Operations
     (Stream_Id => Transport_Stream.Stream_Id,
      Put => Transport_Stream.Transmit,
      Get => Transport_Stream.Receive);
```

---

end Transport_Interchange;

RATIONAL

generic package Transport_Server

This package handles the creation of server tasks in response to incoming connections. The algorithm to be used for serving each connection is supplied by the user as a generic parameter.

This package is used by RPC servers to manage the tasks providing an RPC service.

The specification of the package is:

```
with Transport;
with Transport_Defs;

generic
    with procedure Serve (Connection : Transport.Connection_Id) is <>;
package Transport_Server is
    type Pool_Id is private;
    function Create (Network : Transport_Defs.Network_Name;
                     Local_Socket : Transport_Defs.Socket_Id;
                     Max_Servers : Natural := Natural'Last)
                                                    return Pool_Id;
    procedure Finalize (Abort_Servers : Boolean := False);
    function Local_Socket (Pool : Pool_Id)
                                          return Transport_Defs.Socket_Id;
    function Max_Servers (Pool : Pool_Id) return Natural;
    function Network (Pool : Pool_Id)
                                       return Transport_Defs.Network_Name;
    procedure Set_Max_Servers (Pool : Pool_Id;
                               Max_Servers : Natural);
    function Servers (Pool : Pool_Id) return Natural;
    end Transport_Server;
```

function Create

```
function Create
              (Network       : Transport_Defs.Network_Name;
               Local_Socket  : Transport_Defs.Socket_Id;
               Max_Servers   : Natural                      := Natural'Last)
                                                             return Pool_Id;
```

Description

Creates a pool of server tasks.

Each pool contains a waiter task that waits for incoming connections on the specified network and local socket. The waiter task starts waiting when the pool is created.

When a connection arrives, a server task is allocated to serve it. The server task calls the Serve procedure and passes it the newly arrived connection. When the Serve procedure returns, there is nothing more to be done with the connection. The server task then closes the connection and terminates.

The waiter task continues to wait for more incoming connections. In addition, the waiter task keeps track of the number of active servers. If this number reaches the given Max_Servers limit, the waiter task stops waiting for incoming connections.

Parameters

`Network : Transport_Defs.Network_Name;`

Specifies the network on which to wait for incoming connections.

`Local_Socket : Transport_Defs.Socket_Id;`

Specifies the local socket on which to wait for incoming connections.

`Max_Servers : Natural := Natural'Last;`

Specifies the maximum number of servers that can be active at one time. This number is an initial value and can be changed using the Set_Max_Servers procedure.

`return Pool_Id;`

Specifies the newly created pool.

References

procedure Set_Max_Servers

TRL, procedure Transport.Connect (passive form)

---

procedure Finalize

```
procedure Finalize (Abort_Servers : Boolean := False);
```

Description

Terminates all tasks dependent on this instantiation of package Transport_Server and closes all transport connections.

This procedure permits the caller to exit the scope in which package Transport_Server was instantiated. See the *Reference Manual for the Ada Programming Language*, Section 9.4.

Parameters

```
Abort_Servers : Boolean := False;
```

Specifies, if true, that all server tasks are to be aborted immediately. The default, false, specifies that each server is allowed to continue running until it is done serving its connection.

Restrictions

Calling this procedure renders package Transport_Server useless for subsequent operations.

This procedure does not function in the cross-compiled version of this package.

function Local_Socket

```
function Local_Socket (Pool : Pool_Id) return Transport_Defs.Socket_Id;
```

Description

Returns the local socket on which the given pool is waiting for incoming connections.

This value is set when the pool is created.

Parameters

`Pool : Pool_Id;`
Specifies a pool.

`return Transport_Defs.Socket_Id;`
Returns the local socket on which the given pool is waiting for incoming connections.

References

function Create

function Max_Servers

```
function Max_Servers (Pool : Pool_Id) return Natural;
```

Description

Returns the maximum number of server tasks that can be active simultaneously in the given pool.

This value is initialized when the pool is created and subsequently can be set by the user.

Parameters

`Pool : Pool_Id;`
Specifies a pool.

`return Natural;`
Returns the maximum number of server tasks that can be active simultaneously in the given pool.

References

function Create

procedure Set_Max_Servers

function Network

```
function Network (Pool : Pool_Id) return Transport_Defs.Network_Name;
```

Description

Returns the network on which the given pool waits for incoming connections.

This value is set when the pool is created.

Parameters

`Pool : Pool_Id;`
Specifies a pool.

`return Transport_Defs.Network_Name;`
Returns the network on which the given pool waits for incoming connections.

References

function Create

type Pool_Id

```
type Pool_Id is private;
```

Description

Identifies a pool of server tasks.

References

function Create

generic formal procedure Serve

```
procedure Serve (Connection : Transport.Connection_Id) is <>;
```

Description

Serves a connection.

This procedure is called once for each incoming connection. The tasks that call the Serve procedure are created and managed by manipulating pool objects.

Parameters

```
Connection :  Transport.Connection_Id;
```

Specifies the connection to serve. When the Serve procedure is called, this connection has been opened and connected, but no data have been transmitted or received on it. After the Serve procedure returns, this connection is closed.

References

function Create

function Servers

```
function Servers (Pool : Pool_Id) return Natural;
```

Description

Returns the number of server tasks currently active in the given pool.

This number is initially 0. The value increases each time a connection arrives and decreases each time a server terminates.

Parameters

`Pool : Pool_Id;`
Specifies a pool.

`return Natural;`
Returns the number of server tasks currently active in the given pool.

procedure Set_Max_Servers

```
procedure Set_Max_Servers (Pool        : Pool_Id;
                           Max_Servers : Natural);
```

Description

Sets the maximum number of server tasks that can be active in the given pool.

Once the number of active servers reaches this limit, the pool's waiter task stops waiting for incoming connections.

If the Max_Servers parameter is set to a number less than the current number of active servers, the existing servers will continue to run until they terminate by themselves.

Parameters

`Pool : Pool_Id;`

Specifies a pool.

`Max_Servers : Natural;`

Specifies the new limit on the number of active servers in a given pool.

References

function Max_Servers

function Servers

end Transport_Server;

RATIONAL

package Transport_Server_Job

This package handles the creation of server jobs in response to incoming connections. The algorithm used for serving each connection is supplied by the user as a generic parameter.

This package is used by RPC servers to manage the jobs providing a RPC service.

The specification of the package is:

```
with Transport;
with Transport_Defs;

package Transport_Server_Job is
    generic
        Network : Transport_Defs.Network_Name;
        Local_Socket : Transport_Defs.Socket_Id;
        with procedure Server_Start;
        with procedure Serve (Connection : Transport.Connection_Id);
    procedure Server_Generic;
    procedure Change_Identity (Username : String;
                               Password : String);
end Transport_Server_Job;
```

procedure Change_Identity

```
procedure Change_Identity (Username : String;
                           Password : String);
```

Description

Sets the calling job's access control identity to the specified username and password.

If Username = Password = "", the calling job's identity is set back to its original value—that is, the identity with which the job was first run.

Parameters

`Username : String;`
Specifies the username to be used by the calling job.

`Password : String;`
Specifies the password for this user.

Errors

If the username does not identify a user or the password is incorrect, the Rpc.Username_Or_Password_Error exception is raised.

generic formal object Local_Socket

```
Local_Socket : Transport_Defs.Socket_Id;
```

Description

Denotes a socket within the context of a machine.

For TCP/IP, the socket identifier is a TCP port number.

generic formal object Network

```
Network : Transport_Defs.Network_Name;
```

Description

Denotes one of several networks to which this machine can be attached.

Only TCP/IP is currently defined.

generic formal procedure Serve

```
procedure Serve (Connection : Transport.Connection_Id);
```

Description

Processes incoming requests from the connection.

This procedure is called once for each incoming connection.

Parameters

`Connection : Transport.Connection_Id;`

Specifies the connection to serve. When the Serve procedure is called, this connection has been opened and connected, but no data have been transmitted or received on it. After the Serve procedure returns, this connection is closed.

procedure Server_Generic

```
procedure Server_Generic;
```

Description

Waits for an incoming request for connection on the specified network and local socket.

When a connection request arrives, this procedure executes procedures Server_Start and Serve(Connection). The procedure repeats this process until the local socket is in use, and then it returns.

generic formal procedure Server_Start

```
procedure Server_Start;
```

Description

Starts another server job, using the network specified by the Network parameter at the socket specified by the Local_Socket parameter.

end Transport_Server_Job;

RATIONAL

package Transport_Stream

This package handles the details of allocating and using transport connections.

A transport stream combines a transport connection with some buffering and provides a simplified transmit/receive interface.

This package provides a facility for creating pools of streams. Allocating and deallocating streams occurs very quickly and introduces little overhead. If a stream remains deallocated for a few minutes, the transport connection associated with it is closed.

This package is useful for programs needing to communicate with another machine, without setting up and disconnecting transport connections, or for handling transport status codes.

procedure Allocate

```
procedure Allocate (Stream     : out Stream_Id;
                    Connection :     Transport.Connection_Id);
```

Description

Creates a stream without a pool around the given connection.

The connection is closed when the stream is deallocated.

Parameters

`Stream : out Stream_Id;`
Returns the identification number of the newly allocated stream.

`Connection : Transport.Connection_Id;`
Specifies the transport connection to which the stream is bound.

procedure Allocate

```
procedure Allocate (Stream   : out Stream_Id;
                    Is_New   : out Boolean;
                    Network  :     Transport_Defs.Network_Name;
                    Host     :     Transport_Defs.Host_Id;
                    Socket   :     Transport_Defs.Socket_Id);
```

Description

Locates or sets up a stream to the given network/host/socket combination.

Parameters

`Stream : out Stream_Id;`
Returns the identification number of the newly allocated stream.

`Is_New : out Boolean;`
Checks whether a stream is already in the pool.

`Network : Transport_Defs.Network_Name;`
Specifies the network name.

`Host : Transport_Defs.Host_Id;`
Specifies the host identification number.

`Socket : Transport_Defs.Socket_Id;`
Specifies the socket identification number.

procedure Allocate

```
procedure Allocate (Stream : out Stream_Id;
                    Pool   :     Pool_Id;
                    Is_New : out Boolean);
```

Description

Locates an idle stream from the specified pool.

If no idle stream is available, the procedure sets up an idle stream, opens and connects a transport connection, and returns Is_New = True.

Parameters

`Stream : out Stream_Id;`
Returns the identification number of the newly allocated stream.

`Pool : Pool_Id;`
Specifies the identification of the pool.

`Is_New : out Boolean;`
Checks whether a stream is already in the pool.

function Connection

```
function Connection (Stream : Stream_Id) return Transport.Connection_Id;
```

Description

Returns the transport connection to which the stream is bound.

Parameters

`Stream : Stream_Id;`
Specifies the stream identification number.

`return Transport.Connection_Id;`
Returns the connection identification number.

function Create

```
function Create (Network        : Transport_Defs.Network_Name;
                 Remote_Host    : Transport_Defs.Host_Id;
                 Remote_Socket  : Transport_Defs.Socket_Id;
                 Local_Socket   : Transport_Defs.Socket_Id
                         := Transport_Defs.Null_Socket_Id)  return Pool_Id;
```

Description

Creates a pool of active streams.

Parameters

`Network : Transport_Defs.Network_Name;`

Specifies the network name to be used for connecting to the remote host.

`Remote_Host : Transport_Defs.Host_Id;`

Specifies the identification number of the remote host.

`Remote_Socket : Transport_Defs.Socket_Id;`

Specifies the identification number of the remote socket.

`Local_Socket : Transport_Defs.Socket_Id := Transport_Defs.Null_Socket_Id;`

Specifies the identification number of the local socket to be used for connecting to the remote host.

`return Pool_Id;`

Returns the newly allocated pool.

procedure Deallocate

```
procedure Deallocate (Stream : Stream_Id);
```

Description

Returns a stream to its pool.

Once a stream has been deallocated, it must not be used subsequently because it may be reallocated to another program.

Parameters

`Stream : Stream_Id;`
Specifies the stream to be deallocated.

procedure Destroy

```
procedure Destroy (Pool : Pool_Id);
```

Description

Disconnects all streams in the pool and terminates all tasks associated with the pool.

Parameters

`Pool : Pool_Id;`
Specifies the pool to be destroyed.

procedure Disconnect

```
procedure Disconnect (Stream : Stream_Id);
```

Description

Disconnects the transport connection to which the stream is bound.

Parameters

`Stream : Stream_Id;`
Specifies the stream identification number.

procedure Finalize

```
procedure Finalize;
```

Description

Destroys all pools and terminates all tasks.

This procedure is used when exiting a subprogram in which package Transport_Stream (and by extension the tasks within it) was elaborated. See the *Reference Manual for the Ada Programming Language*, Section 9.4.

Restrictions

After this procedure is called, a subprogram in which package Transport_Stream was elaborated cannot be used.

function Flush_Receive_Buffer

```
function Flush_Receive_Buffer (Stream : Stream_Id)
                                        return Byte_Defs.Byte_String;
```

Description

Empties and gives as the return value the contents of a stream's receive buffer.

Parameters

`Stream : Stream_Id;`
Specifies the stream identification number.

`return Byte_Defs.Byte_String;`
Returns the current contents of the stream's receive buffer.

Restrictions

The receive buffer is left empty.

References

procedure Receive

procedure Flush_Transmit_Buffer

```
procedure Flush_Transmit_Buffer (Stream : Stream_Id);
```

Description

Transmits any data currently in the stream's transmit buffer.

This procedure does not return until all buffered data have been transmitted.

Parameters

`Stream : Stream_Id;`
Specifies the stream identification number.

Errors

If the stream is not connected or becomes disconnected in the middle of this operation, the Not_Connected exception is raised.

References

procedure Transmit

function Get_User_Id

```
function Get_User_Id (Stream : Stream_Id) return Integer;
```

Description

Returns the User_Id integer that is associated with the current stream.

Parameters

`Stream : Stream_Id;`
Specifies the stream identification number.

`return Integer;`
Returns an integer associated with the current stream.

References

procedure Set_User_Id

exception Not_Connected

```
Not_Connected : exception;
```

Description

Occurs when an attempt is made to operate on a unconnected stream.

References

procedure Receive

procedure Transmit

type Pool_Id

```
type Pool_Id is private;
```

Description

Identifies a pool of streams.

References

procedure Allocate

function Create

procedure Deallocate

procedure Receive

```
procedure Receive (From :     Stream_Id;
                   Data : out Byte_Defs.Byte_String);
```

Description

Receives data over a transport stream.

This call does not return until Data'Length bytes are received.

Parameters

`From : Stream_Id;`
Specifies the stream identification number.

`Data : out Byte_Defs.Byte_String;`
Returns the received data.

Errors

If the stream is not connected or becomes disconnected in the middle of this operation, the Not_Connected exception is raised.

References

function Flush_Receive_Buffer

procedure Scavenge

```
procedure Scavenge (Pool : Pool_Id);
```

Description

Disconnects any deallocated streams in the pool.

Parameters

`Pool : Pool_Id;`
Specifies the pool to be scavenged for unused streams.

procedure Scavenge

```
procedure Scavenge;
```

Description

Scavenges all pools for unused streams.

Periodically, a task within package Transport_Stream automatically scavenges all pools.

procedure Set_User_Id

```
procedure Set_User_Id (Stream  : Stream_Id;
                       User_Id : Integer   := 0);
```

Description

Maps the stream identification number to a user identification number.

The user identification number is not used by package Transport_Stream but is provided so that you can associate higher-level information with each stream.

Parameters

`Stream : Stream_Id;`
Specifies the stream identification number.

`User_Id : Integer := 0;`
Specifies the user identification number. By default, it is initialized to 0.

References

function Get_User_Id

type Stream_Id

```
type Stream_Id is private;
```

Description

Identifies a data pipe, called a stream, consisting of a transport connection and some associated buffers.

References

procedure Allocate

procedure Deallocate

procedure Receive

procedure Transmit

procedure Transmit

```
procedure Transmit (Into : Stream_Id;
                    Data : Byte_Defs.Byte_String);
```

Description

Transmits data over a transport stream.

The given data are either transmitted or stored in a buffer for later transmission. The buffer can be cleared using the Flush_Transmit_Buffer procedure.

Parameters

`Into : Stream_Id;`
Specifies the stream identification number.

`Data : Byte_Defs.Byte_String;`
Specifies the data to be transmitted. This procedure does not return until all the data are either transmitted or stored in the stream's transmit buffer.

Errors

If the stream is not connected or becomes disconnected in the middle of this operation, the Not_Connected exception is raised.

References

procedure Flush_Transmit_Buffer

function Unique

```
function Unique (Stream : Stream_Id) return Unique_Id;
```

Description

Converts the stream identification number to a unique number.

Stream_Id is a private type and its image cannot be obtained directly. Converting Stream_Id to Unique_Id provides a handle to allow you to use the identification number for printing hash tables, for example.

Parameters

`Stream : Stream_Id;`
Specifies the stream identification number.

`return Unique_Id;`
Returns a unique number used to represent the stream.

subtype Unique_Id

```
subtype Unique_Id is Integer;
```

Description

Identifies a unique value of the stream identification number.

The unique identification number can be used as a handle for printing hash tables, for example.

end Transport_Stream;

RATIONAL

Index

This index contains entries for each unit and its declarations as well as definitions, topical cross-references, exceptions raised, errors, enumerations, pragmas, switches, and the like. The entries for each unit are arranged alphabetically by simple name. An italic page number indicates the primary reference for an entry.

A

B

C

D

F

G

I

K

L

M

N

O

P

R

S

T

U

V

V

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____
_____
_____
_____
_____
_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____
_____
_____
_____

Indicate any additions or changes you would like to see in the index.

_____
_____
_____
_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____
Company _____
Address _____
City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043

Rational Networking—TCP/IP Reference Manual

Master Index: Networking

Document Control Number: 8003A

Rev. 1.0, July 1987 (Delta)

This document subject to change without notice.

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
1501 Salado Drive
Mountain View, California 94043

Master Index

The Master Index combines the indexes for the four books in the *Rational Networking—TCP/IP Reference Manual.* Thus, the prefix to the page number is the abbreviation for the book in which the item can be found. This index contains entries for each unit and its declarations as well as definitions, topical cross-references, exceptions raised, errors, enumerations, pragmas, switches, and the like. The entries for each unit are arranged alphabetically by simple name. An italic page number indicates the primary reference for an entry.

B

C

F

G

H

J

K

L

M

O

P

Q

R

S

U

V

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____
_____
_____
_____
_____
_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____
_____
_____
_____

Indicate any additions or changes you would like to see in the index.

_____
_____
_____
_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____
Company _____
Address _____
City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043

Rational Networking--TCP/IP Reference Manual, Master Index: Networking, 8003A