Internet Inter-ORB Protocol Connect for TPF

Reference

Release 1

SH31-0188-01

Internet Inter-ORB Protocol Connect for TPF

Reference

Release 1

SH31-0188-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page ix.

Second Edition (July 2000)

This is a major revision of, and obsoletes, SH31-0188-01 and all associated technical newsletters.

This edition applies to Version 1 Release 1 Modification Level 0 of IBM Internet Inter-ORB Protocol Connect for TPF, program number 5799-D64, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order books through your IBM representative or the IBM branch office serving your locality. Books are not stocked at the address given below.

IBM welcomes your comments. There is a form for readers' comments at the back of this book. If the form has been removed, address your comments to:

IBM Corporation
TPF Systems Information Development
Mail Station P923
2455 South Road
Poughkeepsie, NY 12601-5400
USA

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Notices

References in this book to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service in this book is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department 830A
Mail Drop P131
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Permission Notices

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

IBM
OS/390
System/390.

IIOP is a registered trademark of Object Management Group, Inc.

Other company, product, and service names may be trademarks or service marks of others.

About This Book

This book describes Internet Inter-ORB Protocol Connect for TPF (referred to as IIOP Connect for TPF in the remainder of this book), which is ported from the Orbix IIOP Engine from IONA Technologies.

IIOP Connect for TPF provides services to TPF applications to facilitate Transmission Control Protocol/Internet Protocol (TCP/IP) interoperability with heterogeneous environments using an industry standard message protocol. This protocol, called Internet Inter-ORB Protocol (IIOP), is defined by the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA). IIOP is one component of the CORBA specification, which defines a complete distributed object platform.

IIOP Connect for TPF is a shared library that TPF applications can use to communicate, using CORBA compliant-messaging, without access to a full implementation of the CORBA specification.

Before You Begin

Before using this book, you should be familiar with both the C programming language and CORBA distributed programming. In addition, knowledge of CORBA Interface Definition Language (IDL) is fundamental to understanding documentation annotations.

Who Should Read This Book

This book is intended for:

- Application programmers or system programmers who are responsible for planning the application of or installing IIOP Connect for TPF on the TPF 4.1 system. See *TPF Migration Guide: Program Update Tapes* for more information about installation requirements for the TPF 4.1 system.
- Application programmers and designers who want to familiarize themselves with IIOP Connect for TPF. Use this book with the *Orbix IIOP Engine Programmer's Guide* to gain a more complete understanding of these protocols and how to use the interface provided by IIOP Connect for TPF.

How This Book Is Organized

This book is organized as follows:

- Chapter 1, Internet Inter-ORB Protocol Connect for TPF, provides an overview of IIOP Connect for TPF. This chapter includes information about the architecture and migration considerations. A series of tables is used to present some of the migration considerations to you. The information in each table is order sequentially or alphabetically depending on the type of information presented.

  Note: To help you to better understand the content of each table, a description of the various column headings follows in Table 0-1 on page xii.

Table 0-1. How to Read the Tables	
Column Heading	**Description**
C/C++ Language Header File	Indicates the name of the general use C/C++ language header file.
Description of Change	Provides a description of the entity.
Do You Need to Recompile?	Indicates whether you must recompile programs (Yes, No, or Not Applicable).
ISO-C	An X in this column indicates that the C/C++ language header file is for ISO-C support. A blank in this column indicates that the header file is for offline programs.
Link-Edited Module	Indicates the name of the link-edited module.
New, Changed, or No Longer Supported	Indicates whether an entity is new, changed, or no longer supported.
Book Title	Indicates the name of the book in the IIOP Connect for TPF library.
Softcopy or PDF File Name	Indicates the softcopy or Portable Document Format (PDF) file name for the book.

- Chapter 2, IIOP Connect for TPF Functions, provides a reference for the IIOP Connect for TPF application programming interface (API) functions. The description of each function in this book contains the following information:

  Format The function prototype and a description of any parameters.

  Description
  The service that the function provides.

  Normal Return
  What is returned when the requested service has been performed.

  Error Return
  What is returned when the requested service could not be performed. Specifying incorrect function parameters results in a system error with exit.

  Programming Considerations
  Remarks that help a programmer understand the correct use of the function, any side effects that may occur when the function is executed, and how the use of a particular function affects the use of another function.

  Example A code segment that shows a sample function call.

  Related Functions
  Where to find additional information pertinent to this function.

- An index helps you to locate information quickly.

Conventions Used in the IIOP Connect for TPF Library

The IIOP Connect for TPF library uses the following conventions:

Conventions	Examples of Usage
italic	Used for important words and phrases. For example: A *database* is a collection of data. Used to represent variable information. For example: Enter **ZFRST STATUS MODULE** *mod*, where *mod* is the module for which you want status.

Conventions	Examples of Usage
bold	Used to represent text that you type. For example: Enter **ZNALS HELP** to obtain help information for the ZNALS functional message. Used to represent variable information in C language. For example: **level**

monospaced	Used for messages and information that displays on a screen. For example:
	`PROCESSING COMPLETED`
	Used for C language functions. For example:
	`maskc`
	Used for examples. For example:
	`maskc(MASKC_ENABLE, MASKC_IO);`

bold italic	Used for emphasis. For example: You ***must*** type this command exactly as shown.
Bold underscore	Used to indicate the default in a list of options. For example: **Keyword=OPTION1 | DEFAULT**
Vertical bar |	Used to separate options in a list. (Also referred to as the OR symbol.) For example: **Keyword=Option1 | Option2**
CAPital LETters	Used to indicate valid abbreviations for keywords. For example: KEYWord=*option*

Scale	Used to indicate the column location of input. The scale begins at column position 1. The plus sign (+) represents increments of 5 and the numerals represent increments of 10 on the scale. The first plus sign (+) represents column position 5; numeral 1 shows column position 10; numeral 2 shows column position 20 and so on. The following example shows the required text and column position for the image clear card.
	`|...+....1....+....2....+....3....+....4....+....5....+....6....+....7...`
	`LOADER  IMAGE CLEAR`
	Notes:
	1. The word LOADER must begin in column 1.
	2. The word IMAGE must begin in column 10.
	3. The word CLEAR must begin in column 16.

Related Information

A list of related information follows. For information on how to order or access any of this information, call your IBM representative.

IBM IIOP Connect for TPF Books

- *IIOP Connect for TPF Licensed Program Specifications*, GH31-0189
- *IIOP Connect for TPF Program Directory*, GI10-0691.

IBM Transaction Processing Facility (TPF) 4.1 Books

- *TPF Application Programming*, SH31-0132
- *TPF C/C++ Language Support User's Guide*, SH31-0121
- *TPF Migration Guide: Program Update Tapes*, GH31-0187.

Non-IBM Books

- *Orbix IIOP Engine Programmer's Guide*.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other TPF information, do one of the following:

- Go to the TPF Web page at:

  http://www.s390.ibm.com/products/tpf

  Click **TPF Family Libraries** on the left menu. There you will find a link to a feedback page where you can enter and submit comments.

- Send your comments by e-mail to:

  tpfid@us.ibm.com

  Make sure you include the title and number of the book, the version of your TPF system and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

Summary of Changes

This section provides a summary of the changes made to this book. Changes to the latest edition are marked with a vertical bar (|) to the left of the change.

Major Revision SH31-0188-01

17 July 2000

This major revision, Second Edition (July 2000), contains maintenance and editorial changes.

Changed Information

The following information is changed for APAR PJ27024::

- The programming considerations and examples have been updated for the `CDRAddBuffer` function. See “CDRAddBuffer–Add a Buffer to a Common Data Representation Coder” on page 2-8 for more information.
- The programming considerations and examples have been updated for the `CDRAlloc` function. See “CDRAlloc–Register the Buffer Allocation Callback Function of a Common Data Representation Coder” on page 2-10 for more information.

Chapter 1. Internet Inter-ORB Protocol Connect for TPF

The following section discusses the migration considerations for Internet Inter-ORB Protocol Connect for TPF, referred to as IIOP Connect for TPF in the remainder of this publication.

Prerequisite APARs

APAR PJ26685 is a prerequisite for IIOP Connect for TPF. See the APEDIT for this APAR for more information.

Functional Overview

IIOP Connect for TPF, which provides a standard way to connect distributed objects across the Internet and intranets, continues to position the TPF 4.1 system for IBM's e-business initiatives. By using IIOP Connect for TPF you can leverage TPF application programs with new distributed applications. For example, Web browsers support Internet Inter-ORB Protocol (IIOP) to facilitate communication between Web-based applications and intranet enterprise applications.

The Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) specifications define a complete distributed object platform. IIOP, which is a required component of an Object Request Broker (ORB), is an implementation of the General Inter-ORB Protocol (GIOP) over Transmission Control Protocol/Internet Protocol (TCP/IP). IIOP is a key component because IIOP focuses on interoperability of distributed objects in heterogeneous environments.

IIOP Connect for TPF introduces the first phase of CORBA functionality for the TPF 4.1 system.

Note: IIOP Connect for TPF is a stand-alone IIOP, which means that it is not part of an ORB. A stand-alone IIOP provides the following advantages:

- It enables TPF applications to communicate over IIOP and is suitable for applications with strict performance constraints.
- You can leverage existing and new TPF application programs with new distributed applications.

Figure 1-1 on page 1-2 provides an overview of IIOP Connect for TPF.

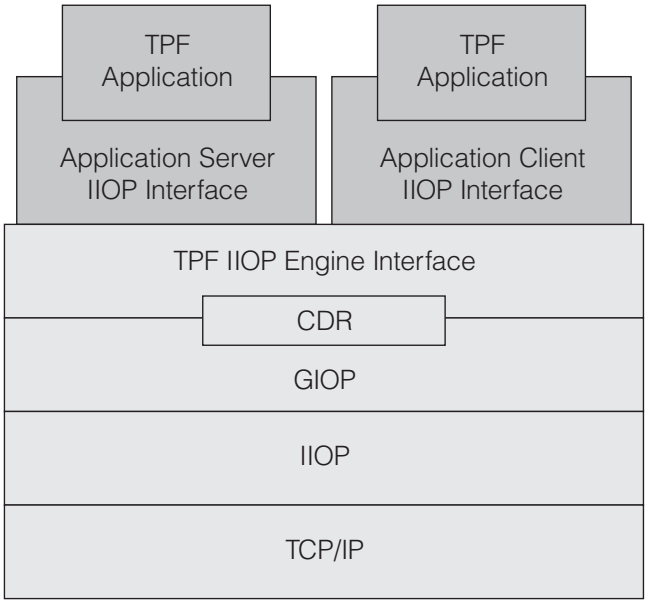


Figure 1-1. IIOP Connect for TPF

Architecture

The CORBA specification defines the GIOP for communication between independent ORB implementations. Specifically, GIOP defines a set of on-the-wire data representation and message formats that allow a client of one ORB to start operations on an object in the same ORB or a different ORB. With the IIOP, ORBs can reside on different systems or platforms.

GIOP is defined as a message-level protocol above an underlying transport layer. GIOP does not address communication issues specific to any single transport protocol, but acts as a basis for a range of interoperability protocols that map GIOP to individual transport layers. For example, the CORBA specification defines a specialization of GIOP that uses TCP/IP as the transport layer. This specialization is called the IIOP.

IIOP Connect for TPF is a set of C functions that are implemented as a single dynamic link library (DLL) or shared library that provides a complete programming interface to the GIOP using TCP/IP. IIOP and GIOP are defined by the OMG CORBA specifications.

IIOP Connect for TPF fully supports GIOP Version 1 Release 1 and GIOP Version 1 Release 0. The major difference between the two versions is that GIOP Version 1 Release 1 supports message fragmentation. The following functional areas are provided by the IIOP:

- Application programming interfaces (APIs) to create and manage the Interoperable Object Reference (IOR), which is a standard object reference format used by clients to locate objects created by a CORBA-compliant ORB or for the TPF 4.1 system, a server.
- APIs to manage connections and send GIOP messages between a client and a server:

  - Request
  - Reply
  - CancelRequest
  - LocateRequest
  - LocateReply
  - CloseConnection
  - MessageError
  - Fragment.
- APIs to encode and decode data types using Common Data Representation (CDR). These functions hide the complexity of converting data to CDR, including data alignment and byte ordering, and enable CORBA-compliant data encapsulation.
- Support to map the general APIs to the TCP/IP transport layer.
- Automatic fragmentation of reply messages.
- Interfaces with applications to provide their own memory allocations and deallocations.
- Support for standard IIOP data formats such as ASCII, big endian or little endian, and Institute of Electrical and Electronics Engineers (IEEE) floats and doubles.
- Operational support for IBM System/390 data formats on-the-wire such as EBCDIC, and IBM System/390 floating point values and double values.

See the *Orbix IIOP Engine Programmer's Guide* for more information about the IIOP. In addition, see the following Website for more information about CORBA and the IIOP:

`http://www.omg.org`

Operating Environment Requirements and Planning Information

To ensure that your TPF 4.1 system performs correctly with IIOP Connect for TPF, you must establish the required operating environment. See *TPF Migration Guide: Program Update Tapes* for more information about the minimum system configuration requirements that are necessary to operate the TPF 4.1 system.

You must have the following to use IIOP Connect for TPF:

- A TPF 4.1 system with program update tape (PUT) 10 and APAR PJ26685 installed
- TCP/IP network connectivity. See *TPF Migration Guide: Program Update Tapes* for more information about TCP/IP-based communication requirements.

Interfaces

The following section summarizes the interfaces for IIOP Connect for TPF.

C/C++ Language

The following section summarizes C/C++ language information. This information is presented in alphabetic order by the type of C/C++ language information. See the *TPF C/C++ Language Support User's Guide* and *TPF Application Programming* for more information about C/C++ language.

General Use C/C++ Language Header Files

Table 1-1 summarizes the general use C/C++ language header files. This information is presented in alphabetic order by the name of the general use C/C++ language header file.

General use means these header files are available for your use.

Table 1-1. General Use C/C++ Language Header Files for IIOP Connect for TPF

C/C++ Language Header File	ISO-C	New, Changed, or No Longer Supported?	Do You Need to Recompile?
cdr.h	X	New	No
encap.h	X	New	No
giop.h	X	New	No
iiop.h	X	New	No
ior.h	X	New	No
tcpcb.h	X	New	No
typ.h	X	New	No
ver.h	X	New	No

Link-Edited Modules

Table 1-2 summarizes the link-edited modules shipped by IBM, which should go into a data set with attributes DCB=(RECFM=U,LRECL=80,BLKSIZE=1200). This information is presented in alphabetic order by the name of the link-edited module.

Table 1-2. Link-Edited Modules for IIOP Connect for TPF

Link-Edited Module	New, Changed, or No Longer Supported?	Description of Change
CIOP	New	Load module for the IIOP dynamic link library (DLL).

Publications

Table 1-3 on page 1-5 summarizes the publications in the IIOP Connect for TPF library. This information is presented in alphabetic order by the publication title.

Table 1-3. Publications for IIOP Connect for TPF		
Publication Title	**Softcopy or PDF File Name**	**Description of Change**
IIOP Connect for TPF Licensed Program Specifications	Not Applicable	New publication that contains license and warranty information for IIOP Connect for TPF.
IIOP Connect for TPF Program Directory	Not Applicable	New publication that contains information about installing the product tape for IIOP Connect for TPF.
IIOP Connect for TPF Reference	irpref00.pdf	New publication that contains planning, installation, and application programming information for IIOP Connect for TPF. This publication is available only as a Portable Document Format (PDF) file.

Application Programming Interfaces (APIs)

IIOP Connect for TPF provides several new C functions. See Chapter 2, "IIOP Connect for TPF Functions" on page 2-1, for more information about these C functions.

Migration Scenarios

Use the following procedure to install IIOP Connect for TPF on your existing TPF 4.1 system.

1. Install program update tape (PUT) 10.
2. Apply APAR PJ26685.
3. Install the IIOP Connect for TPF product tape.
4. Load the CIOP link-edited module.
5. Compile your C or C++ applications against the IIOP Connect for TPF header files. See Table 1-1 on page 1-4 for a list of these header files.

Chapter 2. IIOP Connect for TPF Functions

This chapter contains information about the application programming interface (API) functions for IIOP Connect for TPF.

Include the Correct Header Files

Code the following statement in your IIOP application programs:

```
#include <giop.h>
```

This statement causes all the necessary header files to be included in the application program. The following is a list of these header files:

- `giop.h`
- `typ.h`
- `cdr.h`
- `ior.h`
- `encap.h`
- `tcpcb.h`
- `ver.h`
- `iiop.h`

Type Definitions

The following describes the type definitions used with IIOP Connect for TPF.

General Type Definitions

Table 2-1 lists the general type definitions for IIOP Connect for TPF.

Table 2-1. General Type Definitions

Type Definition	Description
OctetT	The octet data type is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by the communication system.
BooleanT	The Boolean data type is used to denote a data item that can take only one of the following values: • TRUE • FALSE
GIOPAllocFpT	Memory allocation function type signature: `void *(*GIOPAllocFpT) (size_t_size)`
GIOPDeallocFpT	Memory deallocation function type signature: `void *(*GIOPDeallocFpT) (void *data_p)`

Common Data Representation Type Definitions

Table 2-2 lists the type definitions for the Common Data Representation (CDR) functions.

Table 2-2. CDR Type Definitions

Type Definition	Description
CDRStatusT	The return status type for the CDR functions.
CDRModeT	The CDR coder mode.
CDRCharsetT	The type of character coding for this machine.
CDRBufferT	A single coder buffer. This contains a memory block for the buffer and various settings.
CDRAllocFpT	The function pointer type for automatic buffer allocation.
CDRDeallocFpT	The function pointer type for automatic buffer deallocation.
CDRCoderT	The main coder object. This structure is used to manage the buffer list for the coder and contains callbacks for buffer allocation or deallocation. This structure also is used to maintain total buffer usage and the byte order setting.

General Inter-ORB Protocol Type Definitions

Table 2-3 lists the type definitions for the General Inter-ORB Protocol (GIOP) functions.

Table 2-3 (Page 1 of 2). GIOP Type Definitions

Type Definition	Description
GIOPStatusT	The return status type for GIOP functions.
GIOPMsgType	The type of GIOP message.
GIOPMessageHeader_1_0T	The GIOP Version 1.0 header structure.
GIOPMessageHeader_1_1T	The GIOP Version 1.1 header structure.
GIOPRequestHeader_1_0T	The GIOP Version 1.0 request header structure.
GIOPequestHeader_1_1T	The GIOP Version 1.1 request header structure.
GIOPReplyStatusType	The reply status in the reply header.
GIOPReplyHeaderT	The GIOP reply header structure. This is the same for versions 1.0 and 1.1.
GIOPCancelRequestHeaderT	The GIOP cancel request header. This is the same for versions 1.0 and 1.1.
GIOPLocateRequestHeaderT	The GIOP locate request header. This is the same for versions 1.0 and 1.1.
GIOPLocateStatusType	The result of a locate request, returned in the LocateReply message.
GIOPLocateReplyHeaderT	The GIOP locate reply header. This is the same for versions 1.0 and 1.1.
GIOPMsgInfoT	Information about the most recently processed incoming and outgoing messages.

Table 2-3 (Page 2 of 2). GIOP Type Definitions	
Type Definition	**Description**
GIOPCtrlProfileT	A transport-specific control profile.
GIOPCtrlBlkT	The GIOP control block holds the Interoperable Object Reference (IOR), object key, and a transport-specific IIOP control block.
GIOPStateT	The main GIOP state structure contains all information for an agent (for example, a client or server application) to connect and pass messages to and from another agent. An instance of this structure is defined in the calling application and passed to each GIOP API function call.
GIOPConnectFpT	A pointer to the transport-specific connect function.
GIOPAcceptFpT	A pointer to the transport-specific accept function.
GIOPRejectFpT	A pointer to the transport-specific reject function.
GIOPListenFpT	A pointer to the transport-specific listen function.
GIOPCloseFpT	A pointer to the transport-specific close function.
GIOPRecvFpT	A pointer to the transport-specific receive function.
GIOPSendFpT	A pointer to the transport-specific send function.
GIOPProtCallsT	A pointer to the transport-specific functions for handling conversations. These functions implement the generic GIOP transport API.

Interoperable Object Reference Type Definitions

Table 2-4 lists the type definitions for the Interoperable Object Reference (IOR) functions.

Table 2-4 (Page 1 of 2). IOR Type Definitions	
Type Definition	**Description**
IORStatusT	The return status type for the IOR functions.
IORProfileIdT	The profile identifier (ID) component tag.
IORComponentIdT	The component ID of a multi-component profile.
IORServiceIdT	The object service-specific ID.
IORTaggedProfileT	The structure representing the IIOP protocol in an IOR.
IORTaggedComponentT	Version 1.1 of the **TAG_INTERNET_IOP** profile includes a **sequence**<**Tagged Component>** that can contain additional information supporting optional IIOP features and ORB services (such as security). A *tagged component* is a structural representation of the parametric values unique to the protocol.
IORMultipleComponentProfileT	A structure that contains profile components.
IORT	The IOR structure, which contains a type ID and a sequence of tagged profiles.
IORServiceContextT	A structure to hold service context data.

Table 2-4 (Page 2 of 2). IOR Type Definitions	
Type Definition	**Description**
IORServiceContextListT	A sequence of service context structures.
IIOPBody_1_0T	A structure that contains the IIOP 1.0 profile body.
IIOPBody_1_1T	A structure that contains the IIOP 1.1 profile body.

TCP/IP Control Block

The TCP/IP control block is stored in the tagged profile of the GIOP control block and is index TAG_INTERNET_IOP in the profile array. The TCP/IP control block contains the transport level information associated with a connection. This information enables applications to monitor and manage a connection. Table 2-5 lists the type definitions in the TCP/IP control block.

Table 2-5. TCP/IP Control Block Type Definitions	
Type Definition	**Description**
TCPCtrlBlkT	The TCP/IP control block.
IIOPStatusT	The return status type for the IIOP functions.

Return Values

The following describes the return values for the functions provided with IIOP Connect for TPF.

Common Data Representation Return Values

Table 2-6 lists the return values for the CDR functions.

Table 2-6. CDR Function Return Values	
CDRStatusT Return Value	**Description**
CDR_OK	The function call was completed successfully.
CDR_FAIL	The function failed with an unspecified error.
CDR_EFLOAT_RANGE	The function was unable to fit the source floating point number in the range allowed by the target.
CDR_EINV_LEN	The function received a buffer parameter and a coded length parameter, but the buffer was less than the coded length value.
CDR_EINV_MODE	The current CDR coder mode is cdr_unknown.
CDR_ENOSPACE	The function failed to allocate memory.
CDR_ENULL_CODER	The function received a null CDR coder parameter.
CDR_ENULL_DATA	The function received a null profile data parameter.
CDR_ENULL_RETURN	The caller failed to allocate memory for an outgoing parameter.
CDR_EAUTO_FRAG	Automatic fragmentation failed.
CDR_ESTR_FRAG	A partial string (fragment) was found.

General Inter-ORB Protocol Return Values

Table 2-7 lists the return values for the GIOP functions.

Table 2-7. GIOP Function Return Values

GIOPStatusT Return Value	**Description**
GIOP_OK	The function call was completed successfully.
GIOP_ENULL_STATE	The function received a null GIOP connection state structure parameter.
GIOP_EBADMAGIC	The GIOP identifier in the GIOP header of an incoming message was incorrect.
GIOP_ECLOSED	The function call failed because of a closed GIOP connection.
GIOP_ECONNECT	The function failed to connect to the server.
GIOP_EEXCEPT	An exception was returned in the reply.
GIOP_EINV_AGENT	The function call is not valid for the current application. For example, a client called a function that is valid only for a server.
GIOP_EINV_CALL	The function call is not valid at the current point in the processing.
GIOP_EINV_FRAGSZ	The function received a fragment size parameter that is not valid.
GIOP_EINV_MSGSZ	The message size is not valid for the type of message specified.
GIOP_EINV_MSGTYP	The message type specified in the GIOP header is not valid.
GIOP_EINV_TAG	The **tag** parameter passed to the function is not valid.
GIOP_ENOPROFILE	The requested tagged profile does not exist.
GIOP_ENOSPACE	The function failed to allocate memory.
GIOP_ENULL_CB	The function received a null callback function pointer.
GIOP_ENULL_CODER	The function received a null CDR coder structure parameter.
GIOP_ENULL_CTRLBLK	The transport control block is null or not valid.
GIOP_ENULL_IOR	The function received a null IOR structure parameter.
GIOP_ENULL_PARAM	The function received a null pointer parameter.
GIOP_ENULL_PROFILE	No profile exists in the specified IOR with the specified tag type.
GIOP_ENULL_TYP	The function received a null pointer parameter for an outgoing message parameter.
GIOP_EREVISION	The GIOP version number for an incoming message is not supported.
GIOP_ETRANSPORT	The call to the transport layer returned an error.

Internet Inter-ORB Protocol Return Values

The return values listed in Table 2-8 are returned in the TCP control block when a GIOP function call returns the value GIOP_ETRANSPORT, which indicates a TCP/IP (or transport) error. If a TCP/IP error occurs, use the `iiop_error_code` function to determine the specific type of error. See "iiop_error_code–Get Error Code for Last Transmission Control Protocol/Internet Protocol Error" on page 2-100 for more information about the `iiop_error_code` function.

Table 2-8. IIOP Return Values

IIOPStatusT Return Value	Description
IIOP_OK	TCP/IP returned no error.
IIOP_EINV_FD	A TCP/IP error occurred. The file descriptor is not valid.
IIOP_EINV_HOST	A TCP/IP error occurred. The host is not valid.
IIOP_EINV_IPADDR	A TCP/IP error occurred. The IP address is not valid.
IIOP_EBAD_SOCK	A TCP/IP error occurred. The socket is not valid.
IIOP_EBAD_SOCKTYPE	A TCP/IP error occurred. The socket type is not valid.
IIOP_EBAD_SOCKADDR	A TCP/IP error occurred. The socket address is not valid.
IIOP_ESOCK_INUSE	A TCP/IP error occurred. The socket is already in use.
IIOP_EADDR_INUSE	A TCP/IP error occurred. The port address is already in use.
IIOP_EBAD_INPUT	A TCP/IP error occurred. An internal error occurred related to the calling program.
IIOP_ECONN_REFUSED	A TCP/IP error occurred. The connection was refused.
IIOP_EIS_CONN	A TCP/IP error occurred. The connection is already established.
IIOP_ECONN_CLOSED	A TCP/IP error occurred. The connection is already closed or reset.
IIOP_ENO_BUFFER	A TCP/IP error occurred. There is not enough buffer space or there is no buffer space available.
IIOP_EMSG_TOO_BIG	A TCP/IP error occurred. The message is too large to send.
IIOP_EZERO_READ	A TCP/IP error occurred. TCP/IP read zero bytes of data.
IIOP_ENOPERM	A TCP/IP error occurred. There is no permission to perform the operation.
IIOP_ETIMEDOUT	A TCP/IP error occurred. The operation timed out.
IIOP_ENETWORK_ERROR	A TCP/IP error occurred. There was a network error.
IIOP_EIO_ERROR	A TCP/IP error occurred. There was an input/output (I/O) error.
IIOP_EINTR	A TCP/IP error occurred. An interrupt occurred during a TCP/IP operation.
IIOP_ENO_BLOCK	A TCP/IP error occurred. A nonblocking socket call was issued, but no data was available.
IIOP_EUNKNOWN	TCP/IP returned an unknown error.

Interoperable Object Reference Return Values

Table 2-9 lists the return values for the IOR functions.

Table 2-9. IOR Function Return Values

IORStatusT Return Value	Description
IOR_OK	The function call was completed successfully.
IOR_EINV_TAG	An unknown tag type was specified.
IOR_EINV_IORSTR	The function received an IOR string with the wrong format; for example, the string did not begin with IOR.
IOR_ENOSPACE	The function failed to allocate memory.
IOR_ENULL_CODER	The function received a null CDR coder parameter.
IOR_ENULL_DATA	The function received a null profile data parameter.
IOR_ENULL_IOR	The function received a null IOR parameter.
IOR_ENULL_PROFILE	The function received a null profile parameter.
IOR_ENULL_RETURN	The caller failed to allocate memory for an outgoing parameter.
IOR_ENULL_TYPEID	The function received a null type identifier parameter unexpectedly.
IOR_EZERO_PROFILE	The target IOR contains no profiles.

CDRAddBuffer–Add a Buffer to a Common Data Representation Coder

This function explicitly adds a buffer to a Common Data Representation (CDR) coder. If the coder already contains buffers, the new buffer is inserted after the current buffer.

Format

```
#include <cdr.h>
CDRStatusT CDRAddBuffer(CDRCoderT  *cod_p,
                        CDRBufferT *buf_p);
```

cod_p
A pointer to the CDR coder.

buf_p
A pointer to the new buffer; if the **buf_p** parameter is a null pointer, the new buffer is allocated using the buffer allocation callback routine.

Normal Return

If successful, the `CDRAddBuffer` function returns CDR_OK.

Error Return

If there is an error, the `CDRAddBuffer` function returns CDR_FAIL.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- Buffers are reset before they are used for either decoding or encoding data. For decoding buffers, this implies that newly added buffers are unmarshalled beginning with the first octet in the buffer. For encoding buffers, this implies that the complete buffer is available for marshalling. *Marshalling*is the act of copying data into an RPC packet. For decoding and encoding, the `CDRAddBuffer` function assumes that `cdrb_len` octets are available and that `cdrb_used` is initially zero.
- Only a single CDR buffer can be added to a coder with each explicit `CDRAddBuffer` function call or implicit allocation callback made. Specifically, a buffer that contains valid next and previous references to other buffers should not be provided because these references will be lost when the buffer is added to the coder during the reset operation, which is performed with each newly added buffer.

Examples

The following example adds a buffer to the CDR coder and obtains that buffer to initialize the control fields.

```
#include <cdr.h>

CDRCoderT            *cdr_coder;
CDRBufferT           *cdr_buffer;

unsigned long  len=16;
cdr_buffer     = (CDRBufferT *)malloc(sizeof(CDRBUFFERT));
cdrcoder       = (CDRCoderT  *)malloc(sizeof(CDRCoderT));

cdr_buffer->cdrb_buffer_p = (unsigned char *)mallo(len);
cdr_buffer->cdrb_pos_p = cdr_buffer->cdrb_buffer_p;
cdr_buffer->cdrb_len = len;
cdr_buffer->cdrb_next_p = 0;
cdr_buffer->cdrb_prev_p = 0;
buf_p->cdrb_used = 0;

if (CDRAddBuffer(cdr_coder, cdr_buffer) != CDR_OK)
   {
     printf("CDRAddBuffer failed \n");
   }
```

Related Functions

- “CDRAlloc–Register the Buffer Allocation Callback Function of a Common Data Representation Coder” on page 2-10
- “CDRDealloc–Buffer Deallocation Callback Function of a Common Data Representation Coder” on page 2-40.

CDRAlloc–Register the Buffer Allocation Callback Function of a Common Data Representation Coder

This function registers the buffer allocation callback function for use by a Common Data Representation (CDR) coder whenever IIOP Connect for TPF needs to allocate a new buffer.

Format

```
#include <cdr.h>
void CDRAlloc(CDRCoderT   *cod_p,
              CDRAllocFpT  alloc_fp);
```

cod_p
A pointer to the CDR coder.

alloc_fp
The buffer allocation callback function.

Normal Return

Void.

Error Return

Not applicable.

Programming Considerations

- The buffer allocation callback function must have the following prototype:

  ```
  CDRBufferT *alloc_callback(size_t min_bytes);
  ```

- The CDRAlloc function must do the following:
  - Allocate a CDRBufferT structure and an array of at least min_bytes characters
  - Initialize all the fields in the CDRBufferT structure
  - Set the cdrb_buffer_p member of the CDRBufferT structure to the address of the array
  - Set cdrb_len member of the CDRBufferT structure to the size of the array
  - Set the cdrb_pos_p member of the CDRBufferT structure to the start of the buffer (for example, the same value as cdrb_buffer_p)
  - Set the cdrb_used, which is the number of bytes used so far, to zero
  - Set cdrb_next_p, which is the pointer to the next buffer, to zero
  - Set cdrb_prev_p, which is the pointer to the previous buffer, to zero.
- Only a single buffer can be added to a CDR coder with each call.

Examples

The following example initializes a CDR coder and registers a buffer allocation callback function.

```
#include <cdr.h>

CDRCoderT coder;
CDRBuffer *newHeapBuf(size_t);

CDRInit(&coder, CDR_BYTE_ORDER, 512);
CDRAlloc(&coder, newHeapBuf);
```

The following shows an example of a buffer allocation callback function that uses heap storage.

```
CDRBufferT *newHeapBuf(size_t min_bytes)
{
   CDRBufferT *buf_p = malloc(sizeof (CDRBufferT));
   if (!buf_p)
   {
      return NULL;
   }
   if (min_bytes < CDR_MIN_BUFZ)
   {
      min_bytes = CDR_MIN_BUFZ;
   }
   buf_p->cdrb_buffer_p = malloc(min_bytes);
   if (!buf_p->cdrb_buffer_p)
   {
      free(buf_p);
      return NULL;
   }
   buf_p->cdrb_len = min_bytes;
   buf_p->cdrb_pos_p = cdr_buffer->cdrb_buffer_p;
   buf_p->cdrb_used = 0;
   buf_p->cdrb_next_p = 0;
   buf_p->cdrb_prev_p = 0;
   return buf_p;
}
```

Related Functions

- “CDRDealloc–Buffer Deallocation Callback Function of a Common Data Representation Coder” on page 2-40
- “CDRInit–Initialize a Common Data Representation Coder Structure” on page 2-51.

CDRBuflen–Return Total Buffer Length in Use

This function returns the total number of bytes that are in use by buffers connected to a Common Data Representation (CDR) coder and can also reset the in-use count so that the buffers can be reused.

Format

```
#include <cdr.h>
unsigned long CDRBuflen(CDRCoderT     *cod_p,
                        unsigned char  do_reset);
```

cod_p
A pointer to the CDR coder.

do_reset
A flag indicating whether or not to reset the in-use count. Use one of the following values:

false
Do not reset.

true
Reset, which allows the buffers to be reused.

Normal Return

The number of bytes that were in use by buffers connected to the CDR coder.

Error Return

Not applicable.

Programming Considerations

None.

Examples

The following example initializes a CDR coder structure and, at some later time, gets the number of bytes in use by buffers connected to the CDR coder.

```
#include <cdr.h>
CDRCoderT coder;
CDRInit(&coder, CDR_BYTE_ORDER, 512);
  ⋮
unsigned long totalInUse = CDRBuflen(&coder, false);
```

Related Functions

- “CDRReset–Reset the Current Buffer of a Common Data Representation Coder Structure” on page 2-56
- “CDRRewind–Return to the Start of a Common Data Representation Coder Buffer” on page 2-57.

CDRByteSex–Set the Byte Order Flag of a Common Data Representation Coder Structure

This function sets the byte order flag for the Common Data Representation (CDR) coder to indicate big endian or little endian ordering.

Format

```
#include <cdr.h>
void CDRByteSex(CDRCoderT     *cod_p,
                unsigned char  order);
```

cod_p
A pointer to the CDR coder.

order
The required byte order. Use one of the following values:

0 Big endian.

1 Little endian.

Normal Return

Void.

Error Return

Not applicable.

Programming Considerations

This function is useful mainly when code is run on a machine architecture that is different from the one on which it was completed.

Examples

The following example initializes a CDR coder and sets its byte order flag to indicate big endian ordering.

```
#include <cdr.h>

CDRCoderT coder;
CDRInit(&coder, 1, 512); /* little endian */
CDRByteSex(&coder, 0);   /* big endian */
```

Related Functions

None.

CDRCodeBool–Encode or Decode a Boolean Value

This function encodes or decodes an 8-bit Boolean value, which can be false (0) or true (1).

Format

```
#include <cdr.h>
CDRStatusT CDRCodeBool(CDRCoderT     *cod_p,
                       unsigned char *bool_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

bool_p
When the coder is in encoding mode, this is a pointer to the 8-bit value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 8-bit Boolean value is stored.

Normal Return

If successful, the `CDRCodeBool` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeBool` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example encodes or decodes a Boolean value.

```
#include <cdr.h>
unsigned char boolean_value;
CDRCoderT coder;
  ⋮
CDRCodeBool(&coder, &boolean_value);
```

Related Functions

- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34

- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeChar–Encode or Decode a Char Value

This function encodes or decodes an 8-bit char value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeChar(CDRCoderT *cod_p,
                       char      *ch_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

ch_p
A pointer to the 8-bit char value to be encoded or decoded.

Normal Return

If successful, the `CDRCodeChar` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeChar` function returns CDR_FAIL.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- Under normal conditions in encoding mode, translations from EBCDIC to the ASCII character set take place. Under normal conditions in decoding mode, translations from the ASCII to EBCDIC character set take place. You can suppress translation from EBCDIC to ASCII during encoding or decoding by first calling the CDR_NOCHARSET_CONV macro.

Examples

The following example encodes or decodes a char value.

```
#include <cdr.h>
char ebcdic_value;
CDRCoderT coder;
  ⋮
CDRCodeChar(&coder, &ebcdic_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30

- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeDouble–Encode or Decode a Double Value

This function encodes an IBM 32-bit extended floating point value to a 64-bit Institute of Electrical and Electronics Engineers (IEEE) double value, or decodes an IEEE 64-bit double value to an IBM 64-bit extended floating point value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeDouble(CDRCoderT *cod_p,
                         double    *dbl_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

dbl_p
When the coder is in encoding mode, this is a pointer to the 64-bit IBM extended floating point value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 64-bit double value is stored as an IBM 64-bit extended floating point value.

Normal Return

If successful, the `CDRCodeDouble` function returns CDR_OK

Error Return

If there is an error, the `CDRCodeDouble` function returns CDR_EFLOAT_RANGE.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example encodes or decodes a double value.

```
#include <cdr.h>
double ibm_double_value;
CDRCoderT coder;
  ⋮
CDRCodeDouble(&coder, &ibm_double_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30

- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeEnum–Encode or Decode an Enumeration Value

This function encodes or decodes a 32-bit enumeration value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeEnum(CDRCoderT     *cod_p,
                       unsigned long *enum_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

enum_p
When the coder is in encoding mode, this is a pointer to the 32-bit enumeration value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 32-bit enumeration value is stored.

Normal Return

If successful, the `CDRCodeEnum` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeEnum` function returns CDR_FAIL.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- All the IBM System/390 C and C++ compilers store enumeration values in the smallest integral type that can contain all of the enumeration constants. For example, the enumerated type:

  ```
  enum Colors {RED, GREEN, BLUE};
  ```

  has a size of 1 byte. Any enumerated type that is encoded or decoded by the `CDRCodeEnum` function must be either assigned or typecast to an unsigned long value. If you use a typecast, you must ensure that the enumerated type is 32 bits by including an enumeration constant that requires more than 16 bits; for example:

  ```
  enum Colors { RED, GREEN, BLUE, COLOR_32BIT = INT_MAX };
  ```

Examples

The following example encodes and decodes a 32-bit enumeration value.

```
#include <cdr.h>
unsigned long enum_value;
CDRCoderT coder;
  ⋮
CDRCodeEnum(&coder, &enum_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeFloat–Encode or Decode a Float Value

This function encodes an IBM 32-bit floating point value to a 32-bit Institute of Electrical and Electronics Engineers (IEEE) floating point value, or decodes an IEEE 32-bit floating point value to an IBM 32-bit floating point value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeFloat(CDRCoderT *cod_p,
                        float     *flt_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

flt_p
When the coder is in encoding mode, this is a pointer to the 32-bit IBM floating value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 32-bit float value is stored as an IBM 32-bit floating point value.

Normal Return

If successful, the `CDRCodeFloat` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeFloat` function returns CDR_EFLOAT_RANGE.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example encodes or decodes a 32-bit enumeration value.

```
#include <cdr.h>
float ibm_float_value;
CDRCoderT coder;
  ⋮
CDRCodeFloat(&coder, &ibm_float_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value”
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28

- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeLong–Encode or Decode a Long Value

This function encodes or decodes a 32-bit long value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeLong(CDRCoderT *cod_p,
                       long      *long_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

long_p
When the coder is in encoding mode, this is a pointer to the 32-bit value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 32-bit long value is stored.

Normal Return

If successful, the `CDRCodeLong` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeLong` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example encodes or decodes a long value.

```
#include <cdr.h>
long long_p;
CDRCoderT coder;
  ⋮
CDRCodeLong(&coder, &long_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34

- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeNOctet–Encode or Decode Octet Values

This function encodes or decodes an array of octets. For a sequence of octets, the current length value must be encoded or decoded separately by first calling the CDRCodeULong function.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeNOctet(CDRCoderT     *cod_p,
                         OctetT       **oct_pp,
                         unsigned long *oct_len_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

oct_pp
When the coder is in encoding mode, this is a pointer to the address of the first 8-bit value to be encoded. When the coder is in decoding mode, this is a pointer to a pointer that will be set to point to the octet array or sequence contained in the decoder buffer.

oct_len_p
A pointer to the length of the array of octets to be encoded or decoded. In decoding mode, when a message fragment contains less than the entire sequence or array, the actual length of the array or sequence fragment is returned.

Normal Return

- If successful in encoding mode, the CDRCodeNOctet function returns CDR_OK.
- In decoding mode, if the entire sequence or array was received, the CDRCodeNOctet function returns CDR_OK; if a fragment of the array or sequence was received, the CDRCodeNOctet function returns CDR_ESTR_FRAG.

Error Return

If there is an error when automatically fragmenting the array or sequence of octets, the CDRCodeNOctet function returns CDR_EAUTO_FRAG. If there is another kind of error, the CDRCodeNOctet function returns CDR_FAIL.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- In encoding mode, the **oct_pp** parameter must point to a pointer to the array of octets to be encoded; there, octets are moved into the buffer of the coder. When automatic fragmentation is active, the fragment size defined for automatic fragmentation is ignored and the fragment size is changed to contain the entire array of octets.
- In decoding mode, the **oct_pp** parameter is a pointer to a pointer to OctetT; on return, the pointer to OctetT points into the buffer of the coder. If you need to modify the OctetT values, first copy them into other storage.

Examples

The following example first encodes and then decodes an octet sequence.

```
#include <cdr.h>
CDRCoderT coder;
  ⋮
/* encode */
OctetT octets[] = {0, 1, 2, 3 };
OctetT *encode_octet_ptr = octets;
unsigned long encode_sequence_length = sizeof octets;
CDRCodeUlong(&coder, &encode_sequence_length);
CDRCodeNOctet(&coder, &encode_octet_ptr, &encode_sequence_length);
  ⋮
/* decode */
unsigned long decode_sequence_length;
OctetT *decode_octet_ptr;
if (CDRCodeULong(&coder, &decode_sequence_length) != CDR_OK)
{
   /* error decoding sequence length */
}
else
{
    switch (CDRCodeULong(&coder, &decode_octet+ptr, &decode_sequence_length))
    {
    case CDR_OK:
       /* got the entire sequence */
      break;
    case CDR_ESTR_FRAG:
       /* got the first fragment, decode_sequence_length contains the
        *length of the fragment.
        */
    default:
       /* error decoding the sequence of octets */
    }
}
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeNString–Encode or Decode a String Value

This function encodes a null ('\0') terminated string of EBCDIC 8-bit characters to a null-terminated string of ASCII 8-bit characters, or decodes a null-terminated string of ASCII 8-bit characters to a null-terminated string of EBCDIC 8-bit characters.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeNString(CDRCoderT      *cod_p,
                          char          **str_pp,
                          unsigned long  *strlen_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

str_pp
When the coder is in encoding mode, this is a pointer to the address of the first 8-bit EBCDIC character to be encoded. When the coder is in decoding mode, this is a pointer to a pointer that will be set to point to the string contained in the decoder buffer and translated to EBCDIC.

strlen_p
In encoding mode, this is a pointer to the length of the string, including the terminating null byte ('\0'). In decoding mode, this is the address where the length of the string is stored. If a message fragment contains less than the entire string, the actual length of the string fragment is returned.

Normal Return

If successful in encoding mode, the `CDRCodeNString` function returns CDR_OK. In decoding mode, if the entire string was received, the `CDRCodeNString` function returns CDR_OK and sets the ***strlen_p** parameter to the length of the string, including the terminating null byte ('\0'); if a fragment of the string was received, the `CDRCodeNString` function returns CDR_ESTR_FRAG and sets the ***strlen_p** parameter to the length of the string fragment.

Error Return

If there is an error when automatically fragmenting the string, the `CDRCodeNString` function returns CDR_EAUTO_FRAG; if there is another kind of error, the `CDRCodeNString` function returns CDR_FAIL.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- In encoding mode, the **str_pp** parameter must point to a pointer to the first element of the array of EBCDIC characters to be encoded; this string is moved into the buffers of the coder. When automatic fragmentation is active, the fragment size defined for automatic fragmentation is ignored and the fragment size is changed to contain the entire string.
- In decoding mode, the **str_pp** parameter is a pointer to a pointer to char; on return, the pointer to char points into the buffer of the coder. If you need to modify the string, you must first copy it into other storage.

- The length parameter includes the terminating null byte ('\0') for the string. A length of zero indicates a null string; a length of one indicates an empty string ("").
- Under normal conditions in encoding mode, translations from EBCDIC to the ASCII character set take place. Under normal conditions in decoding mode, translations from the ASCII to EBCDIC character set take place. You can suppress translation from EBCDIC to ASCII during encoding or decoding by first calling the CDR_NOCHARSET_CONV macro.

Examples

The following example encodes and then decodes a string.

```
#include <cdr.h>
CDRCoderT coder;
  ⋮
/* encode */
char ebcdic_str[] = "I'm a string.";
char *encode_ebcdic_str_ptr = ebcdic_str;
unsigned long encode_length = sizeof ebcdic_str;
CDRCodeNString(&coder, &encode_ebcdic_str_ptr, &encode_length);
  ⋮
/* decode */
unsigned long decode_length;
char *decode_ebcdic_str_ptr;
switch (CDRCodeNString(&coder, &decode_ebcdic_str_ptr, &decode_length))
{
case CDR_OK:
     /* got the entire string */
     break;
case CDR_ESTR_FRAG:
     /*got the first fragment, decode_sequence_length contains the
      *length of the fragment.
      */
default:
     /* error decoding the sequence of octets */
}
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeOctet–Encode or Decode an Octet Value

This function encodes or decodes an octet value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeOctet(CDRCoderT *cod_p,
                        OctetT    *oct_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

oct_p
When the coder is in encoding mode, this is a pointer to the 8-bit value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the octet value is stored.

Normal Return

If successful, the `CDRCodeOctet` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeOctet` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example encodes or decodes an octet value.

```
#include <cdr.h>
OctetT octet_value;
CDRCoderT coder;
  ⋮
CDRCodeOctet(&coder, &octet_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32
- “CDRCodeString–Encode or Decode a String Value” on page 2-34

- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeShort–Encode or Decode a Short Value

This function encodes or decodes an 16-bit short value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeShort(CDRCoderT *cod_p,
                        short     *short_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

short_p
When the coder is in encoding mode, this is a pointer to the 16-bit value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 16-bit value is stored.

Normal Return

If successful, the `CDRCodeShort` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeShort` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example encodes or decodes a short value.

```
#include <cdr.h>
short short_value
CDRCoderT coder;
  ⋮
CDRCodeShort(&coder, &short_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeString–Encode or Decode a String Value” on page 2-34

- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeString–Encode or Decode a String Value

This function encodes a null ('\0') terminated string of EBCDIC 8-bit characters to a null-terminated string of ASCII 8-bit characters, or decodes a null-terminated string of ASCII 8-bit characters to a null-terminated string of EBCDIC 8-bit characters.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeString(CDRCoderT  *cod_p,
                         char      **str_pp);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

str_pp
When the coder is in encoding mode, this is a pointer to the address of the first 8-bit EBCDIC character to be encoded. When the coder is in decoding mode, this is a pointer to a pointer that is set to point to the string contained in the decoder buffer, translated to EBCDIC.

Normal Return

If successful in encoding mode, the `CDRCodeString` function returns CDR_OK. In decoding mode, if the entire string was received, the `CDRCodeString` function returns CDR_OK; if a fragment of the string was received, the `CDRCodeString` function returns CDR_ESTR_FRAG.

Error Return

If there is an error while automatically fragmenting the string, the `CDRCodeString` function returns CDR_EAUTO_FRAG; if there is another kind of error, the `CDRCodeString` function returns CDR_FAIL.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- In encoding mode, the **str_pp** parameter must point to a pointer to the array of EBCDIC characters to be encoded; this string is moved into the buffer of the coder. When automatic fragmentation is active, the fragment size defined for automatic fragmentation is ignored and the fragment size is changed to contain the entire string.
- In decoding mode, the **str_pp** parameter is a pointer to a pointer to a char; on return, the pointer to char points into the buffer of the coder. If you need to modify the string, you must first copy it into other storage.
- Under normal conditions in encoding mode, translations from EBCDIC to the ASCII character set take place. Under normal conditions in decoding mode, translations from the ASCII to EBCDIC character set take place. You can suppress translation from EBCDIC to ASCII during encoding or decoding by first calling the CDR_NOCHARSET_CONV macro.

Examples

The following example encodes and then decodes a string.

```
#include <cdr.h>
CDRCoderT coder;
  ⋮
/* encode */
char ebcdic_str[] = "I'm a string."
char *encode_ebcdic_str_ptr = ebcdic_str;
CDRCodeString(&coder, &encode_ebcdic_str_ptr);
  ⋮
/* decode */
char *decode_ebcdic_str_ptr;
switch (CDRCodeString(&coder, &decode_ebcdic_str_ptr))
{
case CDR_OK:
     /* got the entire string */
     break;
     default:
case CDR_ESTR_FRAG:
     /*got the first fragment */
default:
     /* error decoding the string */
}
```

Related Functions

- "CDRCodeBool–Encode or Decode a Boolean Value" on page 2-14
- "CDRCodeChar–Encode or Decode a Char Value" on page 2-16
- "CDRCodeDouble–Encode or Decode a Double Value" on page 2-18
- "CDRCodeEnum–Encode or Decode an Enumeration Value" on page 2-20
- "CDRCodeFloat–Encode or Decode a Float Value" on page 2-22
- "CDRCodeLong–Encode or Decode a Long Value" on page 2-24
- "CDRCodeNOctet–Encode or Decode Octet Values" on page 2-26
- "CDRCodeNString–Encode or Decode a String Value" on page 2-28
- "CDRCodeOctet–Encode or Decode an Octet Value" on page 2-30
- "CDRCodeShort–Encode or Decode a Short Value" on page 2-32
- "CDRCodeULong–Encode or Decode an Unsigned Long Value" on page 2-36
- "CDRCodeUShort–Encode or Decode an Unsigned Short Value" on page 2-38.

CDRCodeULong–Encode or Decode an Unsigned Long Value

This function encodes or decodes a 32-bit unsigned long value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeULong(CDRCoderT     *cod_p,
                        unsigned long *ulong_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

ulong_p
When the coder is in encoding mode, this is a pointer to the 32-bit value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 32-bit unsigned long value is stored.

Normal Return

If successful, the `CDRCodeULong` function returns CDR_OK.

Error Return

If there is an error, the `CDRCodeULong` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example encodes or decodes an unsigned long value.

```
#include <cdr.h>
unsigned long ulong_value;
CDRCoderT coder;
  ⋮
CDRCodeULong(&coder, &ulong_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32

- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeUShort–Encode or Decode an Unsigned Short Value” on page 2-38.

CDRCodeUShort–Encode or Decode an Unsigned Short Value

This function encodes or decodes a 16-bit unsigned short value.

Format

```
#include <cdr.h>
CDRStatusT CDRCodeUShort(CDRCoderT      *cod_p,
                         unsigned short *ushort_p);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

ushort_p
When the coder is in encoding mode, this is a pointer to the 16-bit value to be encoded. When the coder is in decoding mode, this is a pointer to the location where the 16-bit unsigned short value is stored.

Normal Return

If successful, the `CDRCodeUShort` function returns CDR_OK

Error Return

If there is an error, the `CDRCodeUShort` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4for more information about CDR return values.

Examples

The following example first encodes or decodes an unsigned short value.

```
#include <cdr.h>
unsigned short ushort_value;
CDRCoderT coder;
  ⋮
CDRCodeUSHort(&coder, &ushort_value);
```

Related Functions

- “CDRCodeBool–Encode or Decode a Boolean Value” on page 2-14
- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeEnum–Encode or Decode an Enumeration Value” on page 2-20
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeLong–Encode or Decode a Long Value” on page 2-24
- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeOctet–Encode or Decode an Octet Value” on page 2-30
- “CDRCodeShort–Encode or Decode a Short Value” on page 2-32

- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36.

CDRDealloc–Buffer Deallocation Callback Function of a Common Data Representation Coder

This function registers the buffer deallocation callback function for use by a Common Data Representation (CDR) coder whenever the coder needs to deallocate a buffer.

Format

```
#include <cdr.h>
void CDRDealloc(CDRCoderT     *cod_p,
                CDRDeallocFpT *dealloc_fp);
```

cod_p
A pointer to the CDR coder.

dealloc_fp
A pointer to the callback function.

Normal Return

Void.

Error Return

Not applicable.

Programming Considerations

- The buffer deallocation callback function must have the following prototype:

  ```
  void dealloc_callback(CDRBufferT *buf_p);
  ```

- The buffer deallocation callback function must deallocate the cdrb_buffer_p member of the CDRBufferT structure and deallocate the CDRBufferT structure itself.

Examples

The following example initializes a CDR coder, registers a buffer allocation callback function, and registers a buffer deallocation callback function.

```
#include <cdr.h>
CDRCoderT coder;
CDRBuffer *newHeapBuf(size_t);

CDRInit(&coder, CDR_BYTE_ORDER, 512);
CDRAlloc(&coder, newHeapBuf);
CDRDealloc(&coder, freeHeapBuf);
```

The following shows an example of a callback function that frees heap storage.

```
void freeHeapBuf(CDRBufferT *buf_p)
{
     free(buf_p->cdrb_buffer_p);
     free(buf_p);
}
```

Related Functions

- "CDRAlloc–Register the Buffer Allocation Callback Function of a Common Data Representation Coder" on page 2-10
- "CDRInit–Initialize a Common Data Representation Coder Structure" on page 2-51.

CDREbcdic_OTW, CDRS390_OTW–Override Platform-Oriented Data Conversions

These functions enable TPF conversations with applications running on OS/390 platforms to assert *on-the-wire* conventions that are not supported by Common Object Request Broker Architecture (CORBA). This allows applications to be optimized by bypassing routines that translate IBM System/390 data formats to and from on-the-wire formats.

The `CDREbcdic_OTW` function overrides the default platform-oriented character set conversion routines and allows connections to TPF to use EBCDIC on the wire.

The `CDRS390_OTW` function overrides the default platform-oriented floating point data conversion routines and allows connections to TPF to use IBM System/390 float and double formats on the wire.

Format

```
#inlcude <cdr.h>
void CDREbcdic_OTW(CDRCoderT *cod_ptr);
void CDRS390_OTW(CDRCoderT *cod_ptr);
```

cod_ptr
A pointer to a Common Data Representation (CDR) coder.

Normal Return

Void.

Error Return

Not applicable.

Programming Considerations

Call these functions before using the coder to encode or decode any message data.

Examples

The following example initializes a CDR coder and suppresses character set and floating point conversion.

```
#include <cdr.h>
CDRCoderT coder;

CDRInit(&coder, CDR_BYTE_ORDER, 512);
CDREbcdic_OTW(&coder);
CDRS370_OTW(&coder);
```

Related Functions

- “CDRCodeChar–Encode or Decode a Char Value” on page 2-16
- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDRCodeNString–Encode or Decode a String Value” on page 2-28
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRInit–Initialize a Common Data Representation Coder Structure” on page 2-51
- “d390toIEEE, dIEEEto390, f390toIEEE, fIEEEto390–Convert Floating Point Numbers between IBM System/390 and IEEE Representations” on page 2-58
- “tpf_asc2ebc, tpf_ebc2asc–Convert Characters between ISO 8859-1 (ASCII) and IBM-1047 (US EBCDIC)” on page 2-116.

CDREncapCreate–Initialize a Common Data Representation Coder to Begin Encoding an Encapsulated Data Buffer

This function initializes a Common Data Representation (CDR) coder to prepare an application to begin encoding an encapsulated data buffer.

Format

```
#include <cdr.h>
CDRStatusT CDREncapCreate(CDRCoderT *cod_p,
                          OctetT     byte_sex);
```

cod_p
A pointer to the CDR coder.

byte_sex
A flag indicating the byte ordering that is used to encode the encapsulated data buffer. Use one of the following values:

0 Big endian.

1 Little endian.

Normal Return

If successful, the `CDREncapCreate` function returns CDR_OK.

Error Return

If the **cod_p** parameter is a null pointer, the `CDREncapCreate` function returns CDR_ENULL_CODER.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- The `CDREncapCreate` function resets the buffers for the CDR coder; any data is lost.
- The `CDREncapCreate` function can be used only to initialize CDR coders for encoding encapsulated data buffers. To initialize a CDR coder for decoding encapsulated data buffers, use the `CDREncapInit` function.

Examples

The following example initializes a CDR coder to begin encoding an encapsulated data buffer.

```
#include <cdr.h>
CDRCoderT encapCoder;
CDRBufferT *newBuf(size_t);
void freeBuf(CDRBufferT *);
CDRInit(&encapCoder, CDR_BYTE_ORDER, 512);
CDRAlloc(&encapCoder, newBuf);
CDRDealloc(&encapCoder, freeBuf);
CDREncapCreate(&encapCoder, CDR_BYTE_ORDER);
```

Related Functions

- “CDREncapEnd–Complete Encoding an Encapsulated Data Buffer” on page 2-46
- “CDREncapInit–Initialize a Common Data Representation Decoder to Decode or Encode an Encapsulated Data Buffer” on page 2-48.

CDREncapEnd–Complete Encoding an Encapsulated Data Buffer

This function completes the process of encoding an encapsulated data buffer. It sets the sequence length field to the final sequence length and the actual buffer containing the resulting octet stream is then returned along with its physical length.

Format

```
#include <cdr.h>
CDRStatusT CDREncapEnd(CDRCoderT      *cod_p,
                       OctetT        **oct_pp,
                       unsigned long  *octlen_p,
                       GIOPAllocFpT   *getmem);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

oct_pp
The address of a pointer to be set to the address of the encapsulated data buffer.

octlen_p
The address of an unsigned long value to be set to the length of the encapsulated data buffer (***oct_pp**).

getmem
A pointer to a storage allocation function that is used to get storage to store the encapsulated data buffer.

Normal Return

If the encoding of the encapsulated data buffer is completed successfully, the `CDREncapEnd` function returns CDR_OK.

Error Return

If there is an error, the `CDREncapEnd` function returns one of the following return values:

- CDR_ENULL_CODER
- CDR_ENULL_RETURN
- CDR_ENOSPACE.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- This function can only be passed by a CDR encapsulation coder (a coder that was initialized for encapsulation by the `CDREncapCreate` or `CDREncapInit` function) in encoding mode after the entire encapsulated data buffer has been encoded.

Examples

The following example encapsulates two long values and adds the encapsulated data buffer to a General Inter-ORB Protocol (GIOP) message.

```
#include <cdr.h>
CDRCoderT; /* GIOP message encoder */
/* initialize the GIOP message encoder and begin encoding a message */
  ⋮
/* declare the encapsulation encoder and other required data */
CDRCoderT encapCoder; /* encapsulation encoder */
CDRBufferT *newBuf(size_t);
void freeBuf(CDRBufferT *);
long data;
OctetT *encap_ptr;
unsigned long encap_len;
/* initialize the encapsulation encoder */
CDRInit(&encapCoder, CDR_BYTE_ORDER, 16);
CDRAlloc(&encapCoder, newBuf);
CDRDealloc(&encapCoder, freeBuf);
CDREncapCreate(&encapCoder, CDR_BYTE_ORDER);

/* encode the encapsulated data buffer */
data = 0;
CDRCodeLong(&encapCoder, &data);
data = 1;
CDRCodeLong(&encapCoder, &data);

/*finish the encapsulation encoding */
CDREncapEnd(&encapCoder, &encap_ptr. &encap_len, malloc);

/*add the encapsulated data buffer to the GIOP message */
CDRCodeNOctet(&coder, &encap_ptr, &encap_len);
```

Related Functions

- “CDREncapCreate–Initialize a Common Data Representation Coder to Begin Encoding an Encapsulated Data Buffer” on page 2-44
- “CDREncapInit–Initialize a Common Data Representation Decoder to Decode or Encode an Encapsulated Data Buffer” on page 2-48.

CDREncapInit–Initialize a Common Data Representation Decoder to Decode or Encode an Encapsulated Data Buffer

This function initializes a Common Data Representation (CDR) coder with an existing buffer. The resulting coder is used to encode an encapsulated data buffer or to decode an existing encapsulated data buffer.

Format

```
#include <cdr.h>
CDRStatusT CDREncapInit(CDRCoderT     *cod_p,
                        CDRBufferT    *buf_p,
                        OctetT        *data_p,
                        unsigned long  data_len,
                        CDRModeT       mode);
```

cod_p
A pointer to an uninitialized CDR coder structure.

buf_p
A pointer to an uninitialized CDR buffer structure.

data_p
A pointer to the encapsulated data buffer.

data_len
The length of the encapsulated data buffer.

mode
One of the following values:

- CDR coder mode
- cdr_decoding
- cdr_encoding.

Normal Return

If successful, the `CDREncapInit` function returns CDR_OK.

Error Return

If there is an error, the `CDREncapInit` function returns one of the following return values:

CDR_ENULL_CODER
A null coder or buffer structure or unknown mode was specified.

CDR_ENULL_DATA
A null data (raw buffer) or data length was specified.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- The `CDREncapInit` function is useful for decoding encapsulated data buffers or for encoding encapsulated data buffers when the size of the encapsulated data buffer is known and the buffer is preallocated.
- If the CDR coder is initialized in decoding mode, the `CDREncapInit` function automatically reads the initial byte-order octet and sets the state of the coder accordingly.

Examples

The following function decodes the encapsulated data buffer that is passed to it in the **data_p** and **data_len** parameters. The structure of the encapsulated data buffer is not shown.

```
#include <cdr.h>
struct decoded_data { /* members for decoded encapsulated data buffer */};
CDRStatusT decodeEncap(OctetT *data_p, unsigned long data_len,
                       struct decoded_data *output)
{
    CDRCoderT coder;
    CDRBufferT buffer;
    CDRStatusT rc =
       CDREncapInit(&coder, &buffer, data_p, data_len,
                    cdr_decoding);

    if (rc != CDR_OK) return rc;

    /* decode the encapsulated data buffer into the output structure*/

    return CDR_OK;
}
```

Related Functions

- “CDREncapCreate–Initialize a Common Data Representation Coder to Begin Encoding an Encapsulated Data Buffer” on page 2-44
- “CDREncapEnd–Complete Encoding an Encapsulated Data Buffer” on page 2-46.

CDRFree–Free All Buffers Connected to a Common Data Representation Coder

This function deallocates all of the buffers that are connected to a Common Data Representation (CDR) coder structure. This function does not free the coder structure itself.

Format

```
#include <cdr.h>
void CDRFree(CDRCoderT *cod_p);
```

cod_p
A pointer to the CDR coder.

Normal Return

Void.

Error Return

Not applicable.

Programming Considerations

None.

Examples

The following example initializes a CDR coder, registers a buffer deallocation callback function, and at some later time frees all of the buffers connected to the coder.

```
#include <cdr.h>
void freeBuf(CDRBufferT *);
CDRCoderT coder;
CDRInit(&coder, CDR_BYTE_ORDER, 512);
CDRDealloc(&coder, freeBuf);
  ⋮
CDRFree(&coder);
```

Related Functions

- “CDRDealloc–Buffer Deallocation Callback Function of a Common Data Representation Coder” on page 2-40.

CDRInit–Initialize a Common Data Representation Coder Structure

This function initializes a Common Data Representation (CDR) coder structure.

Format

```
#include <cdr.h>
CDRStatusT CDRInit(CDRCoderT      *cod_p,
                   unsigned short  byte_sex,
                   size_t          alloc_min);
```

cod_p
A pointer to the address of the CDR coder that will be initialized.

byte_sex
The byte ordering for this platform, which is one of the following:

0 Big endian.

1 Little endian.

Normally, use CDR_BYTE_ORDER.

alloc_min
The minimum number of bytes in each CDR coder buffer. This value must be a multiple of 8.

Normal Return

If successful, the `CDRInit` function returns CDR_OK.

Error Return

If there is an error, the `CDRInit` function returns a CDR return value.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- Call this function to initialize a CDR coder.
- Use the `CDRAlloc` function to specify the storage allocation function for the coder to use and the `CDRDealloc` function to specify the storage deallocation function.

Examples

The following example initializes a CDR coder.

```
#include <cdr.h>

CDRCoderT coder;
CDRInit(&coder, CDR_BYTE_ORDER, 512);
```

Related Functions

- “CDRAlloc–Register the Buffer Allocation Callback Function of a Common Data Representation Coder” on page 2-10
- “CDRDealloc–Buffer Deallocation Callback Function of a Common Data Representation Coder” on page 2-40.

CDRMode–Set the Common Data Representation Coder Mode

This function sets the Common Data Representation (CDR) coder mode and returns the previous mode.

Format

```
#include <cdr.h>
CDRModeT CDRMode(CDRCoderT *cod_p,
                 CDRModeT   mode);
```

cod_p
A pointer to the CDR coder.

mode
The new mode for the CDR coder. Use one of the following values:

- cdr_encoding
- cdr_decoding
- cdr_unknown.

Normal Return

The previous CDR coder mode is returned.

Error Return

Not applicable.

Programming Considerations

The CDR coder mode is set automatically depending on the connection context.

Examples

The following example initializes a CDR coder; at some later time the CDR coder mode is set to cdr_encoding and the previous mode is saved.

```
#include <cdr.h>
CDRCoderT coder;CDRInit(&coder, CDR_BYTE_ORDER, 512);
  ⋮
CDRModeT previousMode = CDRMode(&coder, cdr_encoding);
```

Related Functions

None.

CDRNeedBuffer–Find a Buffer in a Common Data Representation Coder Structure

This function finds the minimum length of a single buffer in the coder. Each intermediate buffer that is not long enough is released. If a buffer is found, it is found at the start of the buffer list. If no such buffer is found, a new one is allocated at the top of the list.

Format

```
#include <cdr.h>
CDRStatusT CDRNeedBuffer(CDRCoderT *cod_p,
                         size_t     min_len,
                         BooleanT   free_trailing);
```

cod_p
A pointer to the Common Data Representation (CDR) coder.

min_len
The minimum length required for the buffer that is found.

free_trailing
A flag indicating whether or not to free all trailing buffers following the buffer that is found:

0 Indicates not to release the trailing buffers.

1 Releases the trailing buffers.

Normal Return

If a buffer of the required size is found or allocated, the `CDRNeedBuffer` function returns CDR_OK.

Error Return

If there is an error, the `CDRNeedBuffer` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example initializes the CDR coder and registers a buffer allocation routine. In this example, the buffers have not been allocated so the `CDRNeedBuffer` function allocates a buffer. As a result, there are no trailing buffers.

```
#include <cdr.h>
CDRCoderT               *cdr_coder;
unsigned long    min_buffer_size = 100;
coder = (CDRCoderT *)malloc(sizeof(CDRCoderT));
free_trailing_buffers = 1;

if (CDRNeedBuffer(cdr_coder, min_buffer, free_trailing_buffers)
                  !=CDR_OK)

{
printf("CDRNeedBuffer failed\n");
}
```

Related Functions

- “CDRAlloc–Register the Buffer Allocation Callback Function of a Common Data Representation Coder” on page 2-10
- “CDRDealloc–Buffer Deallocation Callback Function of a Common Data Representation Coder” on page 2-40.

CDRReset–Reset the Current Buffer of a Common Data Representation Coder

Structure

This function clears the current buffer of the Common Data Representation (CDR) coder for reuse.

Format

```
#include <cdr.h>
CDRStatusT CDRReset(CDRCoderT     *cod_p,
                    unsigned char  in_use);
```

cod_p
A pointer to the CDR coder.

in_use
A flag indicating whether or not the buffer is in use.

0 Indicates that the buffer is not in use.

1 Indicates that the buffer is in use.

Normal Return

If successful, the `CDRReset` function returns CDR_OK.

Error Return

If there is no current buffer to reset, the `CDRReset` function returns CDR_FAIL.

Programming Considerations

See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.

Examples

The following example initializes a CDR coder and, at some later time, rewinds and resets its buffer.

```
#include <cdr.h>
CDRCoderT coder;
CDRInit(&coder, CDR_BYTE_ORDER, 512);
  ⋮
CDRRewind(&coder, true);
CDRReset(&coder, false);
```

Related Functions

“CDRRewind–Return to the Start of a Common Data Representation Coder Buffer” on page 2-57.

CDRRewind–Return to the Start of a Common Data Representation Coder Buffer

This function returns to the beginning of the first buffer of the Common Data Representation (CDR) coder.

Format

```
#include <cdr.h>
CDRStatusT CDRRewind(CDRCoderT     *cod_p,
                     unsigned char  reset_length);
```

cod_p
A pointer to the CDR coder.

reset_length
A flag indicating whether or not to reset the buffer length.

0 Indicates not to reset the buffer length.

1 Resets the buffer length to 0.

Normal Return

If successful, the `CDRRewind` function returns CDR_OK.

Error Return

If there is no buffer, the `CDRRewind` function returns CDR_FAIL.

Programming Considerations

- See “Common Data Representation Return Values” on page 2-4 for more information about CDR return values.
- This function is useful for scanning the buffer again or for resetting the entire buffer. To scan the buffer again, code the **reset_ length** parameter with a value of true.

Examples

The following example initializes a CDR coder and, at some later time, rewinds the buffer for the coder.

```
#include <cdr.h>
CDRCoderT coder;
CDRInit(&coder, CDR_BYTE_ORDER, 512);
  ⋮
CDRRewind(&coder, true);
```

Related Functions

“CDRReset–Reset the Current Buffer of a Common Data Representation Coder Structure” on page 2-56.

d390toIEEE, dIEEEto390, f390toIEEE, fIEEEto390–Convert Floating Point Numbers between IBM System/390 and IEEE Representations

These functions convert a floating point value from IBM System/390 format to Institute of Electrical and Electronics Engineers (IEEE) format, or the reverse:

- `dIEEEto390` converts an IEEE format double value to an IBM System/390 format double value.
- `d390toIEEE` converts an IBM System/390 format double value to an IEEE format double value.
- `fIEEEto390` converts an IEEE format floating point value to an IBM System/390 format floating point value.
- `f390toIEEE` converts an IBM System/390 format floating point value to an IEEE format floating point value.

Format

```
#include <cdr.h>
int dIEEEto390(double *scr, double *dst);
int d390toIEEE(double *scr, double *dst);
int fIEEEto390(float *scr, float *dst);
int f390toIEEE(float *scr, float *dst);
```

src
A pointer to the floating point value to be converted.

dst
A pointer to the location where the converted value is stored.

Normal Return

If the conversion is successful, these functions return 0.

Error Return

If an overflow or underflow condition occurs during conversions, these functions return 1.

Programming Considerations

Floating-point instructions are used to perform calculations on operands with a wide range of magnitude and to yield results that are scaled to preserve precision. Magnitudes that underflow during conversion are converted to 0, while magnitudes that overflow during conversion are converted to the largest represented number.

Examples

The following example asserts IBM System/390 float on the wire, receives an IEEE double value, and manually converts the value to IBM System/390 format.

```
#include <cdr.h>
CDRCoderT coder;
double dS390, dIEEE;
CDRInit(&coder, CDR_BYTE_ORDER, 512);
CDRS390_OTW(&coder);
  ⋮
CDRCodeDouble(&coder, &dIEEE);
if (dIEEEto390(&dIEEE, &dS390) == 0)
{
       /* Conversion was successful. */
}
else if (dS390 == 0.0)
{
     /* underflow */
}
else
{
     /* overflow */
}
```

Related Functions

- “CDRCodeDouble–Encode or Decode a Double Value” on page 2-18
- “CDRCodeFloat–Encode or Decode a Float Value” on page 2-22
- “CDREbcdic_OTW, CDRS390_OTW–Override Platform-Oriented Data Conversions” on page 2-42.

GIOPAccept–Accept a Connection from a Client

This function accepts a connection request from a client. This follows a successful call to the GIOPListen function. The connection is opened to the client so it can begin sending requests.

Format

```
#include <giop.h>
GIOPStatusT GIOPAccept(GIOPStateT *giop);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

Normal Return

If successful, the GIOPAccept function returns GIOP_OK.

Error Return

If there is an error, the GIOPAccept function returns a GIOP error code.

Programming Considerations

See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.

Examples

The following example listens for and accepts a connection request from a client.

```
#include <giop.h>
GIOPStateT gstate;
IORT *ior_p;
  ⋮
switch (GIOPListen(&gstate, ior_p, TAG_INTERNET_IOP))
{
case GIOP_OK:
     switch (GIOPAccept(&gstate))
     {
     case GIOP_OK:
        /* ready to start receiving requests from the client */
        break;
     default:
        /* some kind of accept error */
        break;
     }
default:
     /* there was some kind of listen error */
     break;
}
```

Related Functions

- “GIOPListen–Listen for Client Requests to Connect” on page 2-80
- “GIOPReject–Reject a Connection from a Client” on page 2-89.

GIOPAutoFrag–Change the Automatic Fragmentation Behavior

This function changes the automatic fragmentation behavior of a client or server agent. *Automatic fragmentation* allows you to specify a fragment size that IIOP Connect for TPF will use to automatically convert a message to a series of fragments. *Manual fragmentation* allows you to create each Fragment message individually.

Format

```
#include <giop.h>
GIOPStatusT GIOPAutoFrag(GIOPStateT    *giop,
                         OctetT         flags,
                         unsigned long  nbyte);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a conversation with a client or server.

flags
One of the following flags:

GIOP_AFRAG_CLR
Clears the specified flags.

GIOP_AFRAG_ON
Turns on automatic fragmentation.

GIOP_AFRAG_STR
Allows strings to be fragmented.

GIOP_AFRAG_MSGHDR
Allows message headers to be fragmented.

Use the GIOP_AFRAG_ON flag and a logical OR operation to set an option; for example, to enable string fragmentation, use the following:

```
GIOP_AFRAG_ON|GIOP_AFRAG_STR
```

Use the GIOP_AFRAG_CLR flag and a logical OR operation to turn off an option; for example, to turn off automatic fragmentation, use the following:

```
GIOP_AFRAG_CLR|GIOP_AFRAG_ON
```

Note: The GIOP_AFRAG_STR and GIOP_AFRAG_MSGHDR flags are not supported in this release of IIOP Connect for TPF and are only described here for completeness. Currently, both flags are simply ignored.

nbyte
The maximum size of a fragment when automatic fragmentation is turned on.

Normal Return

If successful, the GIOPAutoFrag function returns GIOP_OK.

Error Return

If there is an error, the GIOPAutoFrag function returns a GIOP error code.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- With automatic fragmentation, Request and Reply messages are fragmented automatically based on the value of the **nbyte** parameter.

  Note: For strings and arrays of octets, this value is ignored and the fragment size is changed to include the entire string or array.
- The IIOP Connect for TPF dynamic link library (DLL) is built with automatic fragmentation enabled.
- IIOP Connect for TPF supports both GIOP Version 1 Release 1 and GIOP Version 1 Release 0. Only GIOP 1.1 supports fragmentation.

Examples

The following example turns on automatic fragmentation for a conversation that was established with a remote client or server object.

```
#include <giop.h>
GIOPStateT gstate;
  ⋮
/* establish a conversation with a remote object */
  ⋮
switch (GIOPAutoFrag(&gstate, GIOP_AFRAG_ON, 1024))
{
case GIOP_OK:
     /* automatic fragmentation is turned on for the conversation */
     break;
default:
     /* there was some kind of listen error */
     break;
}
```

Related Functions

- “GIOPAutoFragGetSize–Get the Current Maximum Automatic Fragment Size” on page 2-64
- “GIOPFragCreate–Create a Fragment Message” on page 2-70
- “GIOPFragSend–Send a Fragment Message” on page 2-72.

GIOPAutoFragGetSize–Get the Current Maximum Automatic Fragment Size

This function gets the maximum fragment size that is currently set for automatic fragmentation. *Automatic fragmentation* allows you to specify a fragment size that IIOP Connect for TPF will use to automatically convert a message to a series of fragments. *Manual fragmentation* allows you to create each Fragment message individually.

Format

```
#include <giop.h>
unsigned long GIOPAutoFragGetSize(GIOPStateT *giop);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a conversation with a client or server.

Normal Return

If successful, the `GIOPAutoFragGetSize` function returns the current maximum fragment size.

Error Return

Not applicable.

Programming Considerations

None.

Examples

The following example gets the automatic fragmentation size for a conversation that was established with a remote client or server object.

```
#include <giop.h>
GIOPStateT gstate;
unsigned long autofragsize;
  ⋮
/* establish a conversation with a remote object */
  ⋮
autofragsize=GIOPAutoFragGetSize(&gstate)
```

Related Functions

- “GIOPAutoFrag–Change the Automatic Fragmentation Behavior” on page 2-62
- “GIOPFragCreate–Create a Fragment Message” on page 2-70
- “GIOPFragSend–Send a Fragment Message” on page 2-72.

GIOPCancelRequestSend–Cancel a Previously Sent Request Message

This function requests that the server stop handling a Request message that was sent previously over the current conversation.

Format

```
#include <giop.h>
GIOPStatusT GIOPCancelRequestSend(GIOPStateT    *giop
                                  unsigned long  request_id);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a client conversation with a server.

request_id
The identifier (ID) of the Request message that was sent previously.

Normal Return

If successful, the `GIOPCancelRequestSend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPCancelRequestSend` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- Use this function when the client no longer cares about the results of a request that was sent previously to the server.
- If the server receives a cancel request, the server avoids processing the specified request if possible. Only a request that is still in progress will be canceled.

Examples

The following example creates and sends a Request message, and then decides that the result is no longer required and cancels the request.

```
#include <giop.h>
#include <stdlib.h>
#include <string.h>
#include <cdr.h>
#define OPERATION "aMethod"
GIOPStateT gstate;
char *op;
/* establish a conversation with a server */
  ⋮
op=malloc(sizeof OPERATION);
memcpy(op, OPERATION, sizeof OPERATION);
switch (GIOPRequestCreate(&gstate, op, 0, (OctetT *)0,
        (IORServiceContextListT *)0, FALSE))
{
case GIOP_OK:
    /* the request message was successfully created,
       call CDR functions to build the request body */

  ⋮
    switch (GIOPRequestSend(&gstate;, FALSE, FALSE))
    {
    case GIOP_OK:
        /* the request message was successfully sent */

  ⋮
        /* decide that the results of the previous message
           are no longer required */
        switch (GIOPCancelRequestSend(&gstate, gstate.gs_request_id))
        {
        case GIOP_OK;
            /* The cancel request was successfully sent,
               DO NOT ASSUME that the request was cancelled */
            break;
        default:
            /* there was some kind of cancel request send error */
            break;
        }
        break;
    default:
        /* there was some kind of send error */
        break;
    }
    break;

default:
    /* there was some kind of create error */
    break;
}
```

Related Functions

- “GIOPRequestCreate–Create a Request Message” on page 2-95.

GIOPCloseConnectionSend–Close an Open Connection

This function closes a connection that was established previously and is currently still valid.

Format

```
#include <giop.h>
GIOPStatusT GIOPCloseConnectionSend(GIOPStateT *giop);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a conversation with a client or server.

Normal Return

If successful, the `GIOPCloseConnectionSend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPCloseConnectionSend` function returns a GIOP error code.

Programming Considerations

See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.

Examples

The following example closes a conversation that was established previously with a remote client or server object.

```
#include <giop.h>
GIOPStateT gstate;
  ⋮
/* establish a conversation with a remote object */
  ⋮
switch (GIOPCloseConnectionSend(&gstate))
{
case GIOP_OK:
    /* the conversation ended cleanly */
    break;
default:
    /* there was some kind of error */
    break;
}
```

Related Functions

None.

GIOPConnect–Connect a Client to a Server

This establishes an Internet Inter-ORB Protocol (IIOP) connection from a client process to a server process.

Format

```
#include <giop.h>
GIOPStatusT GIOPConnect(GIOPStateT *giop,
                        IORT       *ior_p,
                        short       policy);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a client conversation with a server.

ior_p
A pointer to an Interoperable Object Reference (IOR) object that points to the server object.

policy
The connection policy for the GIOP state. The policy determines the behavior when a connection is lost (subsequent to the `GIOPConnect` call). Use one of the following values:

AUTO_RETRY
The IIOP engine will automatically try to connect again.

TRY_ONE
The IIOP engine will exit if the initial connection is lost.

Normal Return

If successful, the `GIOPConnect` function returns GIOP_OK, and the **giop** parameter is set to idle (GIOP_SIDLE) state.

Error Return

If there is an error, the `GIOPConnect` function returns a GIOP error code.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- The first IOR entry to provide a successful connection is chosen.

Examples

The following example connects an IIOP client process with a remote server object.

```
#include <giop.h>
GIOPStateT gstate;
IORT *ior_p;
  ⋮
switch (GIOPConnect(&gstate, ior_p, AUTO_RETRY))
{
    case GIOP_OK:
        /* successful connection, prepare to send a request */
        break;
    default:
        /* some kind of error */
        break;
}
```

Related Functions

None.

GIOPFragCreate–Create a Fragment Message

This function creates a Fragment message. *Automatic fragmentation* allows you to specify a fragment size that IIOP Connect for TPF will use to automatically convert a message to a series of fragments. *Manual fragmentation* allows you to create each Fragment message individually.

Format

```
#include <giop.h>
GIOPStatusT GIOPFragCreate(GIOPStateT *giop);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a conversation with a client or server.

Normal Return

If successful, the `GIOPFragCreate` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPFragCreate` function returns a GIOP return value .

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- This function is used for manual fragmentation. With automatic fragmentation, Request and Reply messages are fragmented automatically beyond a set message size. Use the `GIOPAutoFrag` function to turn automatic fragmentation on or off and to set the message size.

  Note: For strings and arrays of octets, the set message size is ignored and the fragment size is changed to include the entire string or array.
- The IIOP Connect for TPF dynamic link library (DLL) is built with automatic fragmentation enabled.
- IIOP Connect for TPF supports both GIOP Version 1 Release 1 and GIOP Version 1 Release 0. Only GIOP 1.1 supports fragmentation.

Examples

The following example creates a Fragment message for a conversation that was established with a remote client or server object.

```
#include <giop.h>
GIOPStateT gstate;
  ⋮
/* establish a conversation with a remote object */
  ⋮
switch (GIOPFragCreate(&gstate))
{
case GIOP_OK:
    /* a fragment message is created */
    break;
default:
    /* there was some kind of error */
    break;
}
```

Related Functions

- “GIOPAutoFrag–Change the Automatic Fragmentation Behavior” on page 2-62
- “GIOPAutoFragGetSize–Get the Current Maximum Automatic Fragment Size” on page 2-64
- “GIOPFragSend–Send a Fragment Message” on page 2-72.

GIOPFragSend–Send a Fragment Message

This function sends the current Fragment message and indicates to the remote agent whether or not more fragments will follow. *Automatic fragmentation* allows you to specify a fragment size that IIOP Connect for TPF will use to automatically convert a message to a series of fragments. *Manual fragmentation* allows you to create each Fragment message individually.

Format

```
#include <giop.h>
GIOPStatusT GIOPFragSend(GIOPStateT *giop
                         BooleanT    more_fragments);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a conversation with a client or server.

more_fragments
One of the following values:

TRUE
Indicates that more fragments will follow the current Fragment message.

FALSE
Indicates that this is the last fragment of the current message.

Normal Return

If successful, the `GIOPFragSend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPFragSend` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- This function is used for manual fragmentation. With automatic fragmentation, Request and Reply messages are fragmented automatically beyond a set message size. Use the `GIOPAutoFrag` function to turn on or turn off automatic fragmentation and to set the message size.

  Note: For strings and arrays of octets, the set message size is ignored and the fragment size is changed to include the entire string or array.
- The IIOP Connect for TPF dynamic link library (DLL) is built with automatic fragmentation enabled.
- IIOP Connect for TPF supports both GIOP Version 1 Release 1 and GIOP Version 1 Release 0. Only GIOP 1.1 supports fragmentation.

Examples

The following example creates and sends two Fragment messages for a conversation that was established with a remote client or server object.

```
#include <giop.h>
GIOPStateT gstate;
  ⋮
/* establish a conversation with a remote object */
  ⋮
switch (GIOPFragCreate(&gstate))
{
case GIOP_OK:
    /* set the first fragment message contents */

  ⋮
    switch (GIOPFragSend(&gstate, TRUE))
    {
    case GIOP_OK:
        /* the first fragment was sent successfully,
           create and send the second (last) fragment */
        switch (GIOPFragCreate(&gstate))
        {
        case GIOP_OK:
            /* set the second fragment message contents */
            switch (GIOPFragSend(&gstate, FALSE))
            {
            case GIOP_OK:
                /* the second fragment was sent successfully */

            default:
                /* there was some kind of send error for the
                   second fragment */
                break;
            }
            break;

        default:
            /* there was some kind of creation error for the
               second fragment */
            break;
        }
        break;

    default:
        /* there was some kind of send error for the first fragment */
        break;
    }
    break;

default:
    /* there was some kind of creation error for the first fragment */
    break;
}
```

Related Functions

- “GIOPAutoFrag–Change the Automatic Fragmentation Behavior” on page 2-62
- “GIOPAutoFragGetSize–Get the Current Maximum Automatic Fragment Size” on page 2-64
- “GIOPFragCreate–Create a Fragment Message” on page 2-70.

GIOPGetNextMsg–Get the Next Incoming General Inter-ORB Protocol Message

This function gets the next General Inter-ORB Protocol (GIOP) message (if one is available) and prepares it for processing.

Format

```
#include <giop.h>
GIOPStatusT GIOPGetNextMsg(GIOPStateT  *giop,
                           GIOPMsgType *typ_p);
```

giop
A pointer to the GIOP state structure for a conversation with a client or server.

typ_p
A pointer to enumerated type GIOPMsgType. On return, GIOPMsgType contains the type of message received, which is one of the following:

GIOPRequest
Request message.

GIOPReply
Reply message.

GIOPCancelRequest
Cancel request message.

GIOPLocateRequest
Locate request message.

GIOPLocateReply
Locate reply message.

GIOPCloseConnection
Close connection message.

GIOPMessageError
Message error message.

GIOPFragment
Message fragment.

GIOPUnknown
Unknown message type.

Normal Return

If successful, the `GIOPGetNextMsg` function returns GIOP_OK and the **typ_p** parameter points to the message type.

Error Return

If there is an error, the `GIOPGetNextMsg` function returns a GIOP error code.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- The buffer containing the message received is maintained by the GIOP state structure.
- Incoming messages are read in stages. First, the GIOP identifier (the value GIOP) is read, followed by the remainder of the GIOP header. Finally, the message contents are read. The CDR coder is rewound for each stage.
- Incoming messages are expected to be in GIOP Version 1 Release 1 format unless the server is receiving a Version 1.0 message from a Version 1.0 based client.
- Use the Common Data Representation (CDR) functions declared in the `cdr.h` header file to decode the contents of the message received.

Examples

The following message receives the next inbound message and prepares the GIOP state structure to process it.

```
#include <giop.h>
GIOPStateT gstate;
GIOPMsgType mtype;
  ⋮
switch (GIOPGetNextMsg(&gstate, &mtype))
{
case GIOP_OK:
    switch (mtype)
    {
    case GIOPRequest:
        /* handle request message */
        break;

    case GIOPCancelRequest:
        /* handle cancel request message */
        break;

    case GIOPLocateRequest:
        /* handle locate request message */
        break;

    case GIOPMessageError:
        /* handle message error message */
        break;

    case GIOPFragment:
        /* handle message fragment */
        break;

    default:
        /* some other kind of message */
        break;
    }
    break;

default;
    /* there was an error */
    break;
}
```

Related Functions

None.

GIOPInit–Initialize a General Inter-ORB Protocol State Object

This function initializes a General Inter-ORB Protocol (GIOP) state structure, associating it with a (previously initialized) Common Data Representation (CDR) coder object, and preparing it to be used for other Internet Inter-ORB Protocol (IIOP) engine application programming interface (API) functions.

Format

```
#include <giop.h>
GIOPStatusT GIOPInit(GIOPStateT *giop,
                     CDRCoderT  *coder_p,
                     BooleanT    is_server);
```

giop
A pointer to the GIOP state structure for a conversation with a client or server.

coder_p
A pointer to a CDR coder object; the coder must be initialized already.

is_server
One of the following values:

TRUE
Initializes the GIOP state structure as a server.

FALSE
Initializes the GIOP state structure as a client.

Normal Return

If successful, the `GIOPInit` function returns GIOP_OK and the GIOP state structure is ready to connect with a remote client or server object.

Error Return

If there is an error, the `GIOPInit` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- The default is set for no automatic fragmentation.

Examples

The following example initializes a GIOP state structure to run as a server that is ready to continue processing IIOP APIs to accept and process messages from a client.

```
#include <giop.h>
GIOPStateT gstate;
CDRCoderT coder
  ⋮
switch (GIOPInit(&gstate, &coder, TRUE))
{
case GIOP_OK:
    /* wait for client to connect and begin conversation */
    break;
default:
    /* some kind of error */
    break;
}
```

Related Functions

- “GIOPConnect–Connect a Client to a Server” on page 2-68
- “GIOPListen–Listen for Client Requests to Connect” on page 2-80.

GIOPListen–Listen for Client Requests to Connect

This function listens for any requests made by clients to connect. If a request is detected, the server can accept the connection by using the `GIOPAccept` function or reject the connection by using the `GIOPReject` function.

Format

```
#include <giop.h>
GIOPStatusT GIOPListen(GIOPStateT   *giop,
                       IORT         *ior_p,
                       IORProfilIdT  tag);
```

giop
A pointer to the GIOP state structure for a server conversation with a client.

ior_p
A pointer to a published Interoperable Object Reference (IOR) object.

tag
The type of transport over which the server will listen. Use the value TAG_INTERNET_IOP.

Normal Return

If successful, the `GIOPListen` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPListen` function returns a GIOP return value.

Programming Considerations

See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.

Examples

The following example listens for a client to request a connection with the server.

```
#include <giop.h>
GIOPStateT gstate;
IORT *ior_p;
  ⋮
switch (GIOPListen(&gstate, ior_p, TAG_INTERNET_IOP))
{
case GIOP_OK:
    /* accept or reject the connection */
    break;
default:
    /* there was some kind of error */
    break;
}
```

Related Functions

- “GIOPAccept–Accept a Connection from a Client” on page 2-60
- “GIOPReject–Reject a Connection from a Client” on page 2-89
- “GIOPStopListen–Notify Clients That the Server Is No Longer Listening for New Connections” on page 2-99.

GIOPLocateReplyCreate–Create a LocateReply Message

This function creates a LocateReply message.

Format

```
#include <giop.h>
GIOPStatusT GIOPLocateReplyCreate(GIOPStateT           *giop,
                                  OctetT                major,
                                  OctetT                minor,
                                  GIOPLocateStatusType  status,
                                  unsigned long         request_id);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

major
The GIOP major release number of the reply. Use the following value:

1 Major release 1.

minor
The GIOP minor release number of the reply. Use one of the following values:

0 Minor release 0.

1 Minor release 1.

status
The result of the locate query. Use one of the following values:

GIOP_UNKNOWN_OBJECT
The server has no knowledge of the requested object.

GIOP_OBJECT_HERE
The server can access the requested object.

GIOP_OBJECT_FORWARD
The server provides an Interoperable Object Reference (IOR) to another server that may have access to the requested object.

request_id
The identifier (ID) of the LocateRequest message to which you are responding.

Normal Return

If successful, the `GIOPLocateReplyCreate` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPLocateReplyCreate` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- Use this function to reply after receiving a LocateRequest message from the client.
- When **status** is GIOP_UNKNOWN_OBJECT or GIOP_OBJECT_HERE, the LocateReply message has no message body. When **status** is GIOP_OBJECT_FORWARD, the LocateReply message body consists of an IOR for the object located on another server.

Examples

The following example creates a LocateReply message in response to a LocateRequest message that was received from a conversation with a client object.

```
#include <giop.h>
GIOPStateT gstate;
OctetT major, minor;
unsigned long rid;
  ⋮
/* receive a LocateRequest message from the remote client;
   reply using the request ID from the client */
major=gstate.gs_msg_in.mi_major;
minor=gstate.gs_msg_in.mi_minor;
CDRCodeULong(gstate.gs_coder_p, &rid);
  ⋮
switch (GIOPLocateReplyCreate(&gstate, major, minor,
        GIOP_OBJECT_HERE, rid))
{
case GIOP_OK:
    /* the LocateReply message was created successfully */
    break;
default:
    /* there was some kind of error */
    break;
}
```

Related Functions

- “GIOPLocateReplySend–Send a LocateReply Message” on page 2-84
- “GIOPLocateRequestSend–Create and Send a LocateRequest Message to a Server” on page 2-86.

GIOPLocateReplySend–Send a LocateReply Message

This function sends a LocateReply message that was created previously and, if required, has an Interoperable Object Reference (IOR) appended.

Format

```
#include <giop.h>
GIOPStatusT GIOPLocateReplySend(GIOPStateT *giop);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

Normal Return

If successful, the `GIOPLocateReplySend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPLocateReplySend` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- Use this function to reply after receiving a LocateRequest message from the client.
- The LocateReply message header must first be created by calling the `GIOPLocateReplyCreate` function.
- When **status** is GIOP_UNKNOWN_OBJECT or GIOP_OBJECT_HERE, the LocateReply message has no message body. When **status** is GIOP_OBJECT_FORWARD, the LocateReply message body consists of an IOR for the object located on another server.

Examples

The following example creates and sends a LocateReply message in response to a LocateRequest message that was received from a conversation with a client object.

```
#include <giop.h>
GIOPStateT gstate;
OctetT major, minor;
unsigned long rid;
  ⋮
/* receive a LocateRequest message from the remote client;
   reply using the request ID from the client */
major=gstate.gs_msg_in.mi_major;
minor=gstate.gs_msg_in.mi_minor;
CDRCodeULong(gstate.gs_coder_p, &rid);
  ⋮
switch (GIOPLocateReplyCreate(&gstate, major, minor,
        GIOP_OBJECT_HERE, rid))
{
case GIOP_OK:
    switch (GIOPLocateReplySend(&gstate))
    {
    case GIOP_OK:
        /* the LocateReply message was sent successfully */
        break;
    default:
        /* there was some kind of send error */
        break;
    }
    break;

default:
    /* there was some kind of create error */
    break;
}
```

Related Functions

- “GIOPLocateReplyCreate–Create a LocateReply Message” on page 2-82.

GIOPLocateRequestSend–Create and Send a LocateRequest Message to a Server

This function creates and sends a LocateRequest message to a server with which a conversation was established.

Format

```
#include <giop.h>
GIOPStatusT GIOPLocateRequestSend(GIOPStateT    *giop
                                  unsigned long  objkey_len
                                  OctetT        *objkey_p);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

objkey_len
The length of the key for the object that you want to locate.

objkey_p
A pointer to the first octet of the object key.

Normal Return

If successful, the `GIOPLocateRequestSend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPLocateRequestSend` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- You can get the length and address of an object key from an Interoperable Object Reference (IOR) structure or directly from the GIOPStateT structure for the connection.

Examples

The following example creates and sends a LocateRequest message to a server.

```
#include <giop.h>
GIOPStateT gstate;
unsigned long keylen
OctetT *key_p;
  ⋮
/* establish a conversation with a server */
  ⋮
keylen=giop.gs_ctrl_blk.cb_objkey_len;
key_p=giop.gs_ctrl_blk.cb_objkey_p;
switch (GIOPLocateRequestSend(&gstate, keylen, key_p))
{
case GIOP_OK:
    /* the locate request message was successfully created and sent */
    break;

default:
    /* there was some kind of create error */
    break;
}
```

Related Functions

- “GIOPLocateReplyCreate–Create a LocateReply Message” on page 2-82
- “GIOPLocateReplySend–Send a LocateReply Message” on page 2-84.

GIOPMessageErrorSend–Create and Send a MessageError Message

This function creates and sends a MessageError message.

Format

```
#include <giop.h>
GIOPStatusT GIOPMessageErrorSend(GIOPStateT *giop);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a conversation with a client or server.

Normal Return

If successful, the `GIOPMessageErrorSend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPMessageErrorSend` function returns a GIOP return value.

Programming Considerations

See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.

Examples

The following example receives the next inbound message, finds that it is not valid, replies with a MessageError message, and closes the connection.

```
#include <giop.h>
GIOPStateT gstate;
GIOPMsgType mtype;
  ⋮
switch (GIOPGetNextMsg(&gstate, &mtype))
{
case GIOP_EBADMAGIC:
case GIOP_EREVISION:
case GIOP_EINV_MSGTYP:
case GIOP_EINV_MSGSZ:
    GIOPMessageErrorSend(&gstate);
    GIOPCloseConnectionSend(&gstate);
    /* Other error handling code */
    break;
/* and so on */
}
```

Related Functions

None.

GIOPReject–Reject a Connection from a Client

This function rejects a connection request from a client. This follows a successful call to the GIOPListen function.

Format

```
#include <giop.h>
GIOPStatusT GIOPReject(GIOPStateT *giop,);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

Normal Return

If successful, the GIOPReject function returns GIOP_OK.

Error Return

If there is an error, the GIOPReject function returns a GIOP return value.

Programming Considerations

See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.

Examples

The following example listens for and then rejects a connection request from a client.

```
#include <giop.h>
GIOPStateT gstate;
IORT *ior_p;
  ⋮
switch (GIOPListen(&gstate, ior_p, TAG_INTERNET_IOP))
{
case GIOP_OK:
    switch (GIOPReject(&gstate))
    {
    case GIOP_OK:
        /* clean up and finish */
        break;
    default:
        /* some kind of reject error */
        break;
    }
default:
    /* there was some kind of listen error */
    break;
}
```

Related Functions

- “GIOPAccept–Accept a Connection from a Client” on page 2-60
- “GIOPListen–Listen for Client Requests to Connect” on page 2-80.

GIOPReplyCreate–Create a Reply Message

This function creates a Reply message in response to a Request message from a client object.

Format

```
#include <giop.h>
GIOPStatusT GIOPReplyCreate(GIOPStateT             *giop,
                            OctetT                  major,
                            OctetT                  minor,
                            GIOPReplyStatusType     status,
                            unsigned long           request_id
                            IORServiceContextListT *scxt_p);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

major
The GIOP major release number of the reply. Use the following value:

1 Major release 1.

minor
The GIOP minor release number of the reply. Use one of the following values:

0 Minor release 0.

1 Minor release 1.

status
The result of answering the request. Use one of the following values:

GIOP_NO_EXCEPTION
The request is completed successfully.

GIOP_USER_EXCEPTION
The request resulted in a user exception, which is contained in the reply.

GIOP_SYSTEM_EXCEPTION
The request resulted in a system exception, which is contained in the reply.

GIOP_LOCATION_FORWARD
The reply contains an Interoperable Object Reference (IOR) for another server that may be able to satisfy the request.

request_id
The identifier (ID) of the LocateRequest message to which you are responding.

scxt_p
A pointer to the first element of an array of service contexts.

Normal Return

If successful, the GIOPLocateReplySend function returns GIOP_OK.

Error Return

If there is an error, the GIOPLocateReplySend function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- Use this function to reply after receiving a Request message from the client.
- After this function returns successfully, use the Common Data Representation (CDR) functions to code the Reply message body, and then the GIOPReplySend function to send the reply to the client.

Examples

The following example creates a Reply message in response to a Request message that was received from a conversation with a client object.

```
#include <giop.h>
GIOPStateT gstate;
OctetT major, minor;
unsigned long rid;
  ⋮
/* receive a Request message from the remote client;
   reply using the request ID from the client */
major=gstate.gs_msg_in.mi_major;
minor=gstate.gs_msg_in.mi_minor;
CDRCodeULong(&gstate, &rid);
  ⋮
switch (GIOPReplyCreate(&gstate, major, minor,
        GIOP_NO_EXCEPTION, rid, (IORServiceContextListT *)0))
{
case GIOP_OK:
    /* the Reply message was created successfully */
    break;
default:
    /* there was some kind of error */
    break;
}
```

Related Functions

- “GIOPReplySend–Send a Reply Message” on page 2-93.

GIOPReplySend–Send a Reply Message

This function sends a Reply message to a client.

Format

```
#include <giop.h>
GIOPStatusT GIOPReplySend(GIOPStateT *giop
                          BooleanT    more_fragments);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

more_fragments
One of the following:

TRUE
Indicates that this message is a fragment, and more fragments will follow to complete the message.

FALSE
Indicates that this is a complete message.

Normal Return

If successful, the `GIOPReplySend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPReplySend` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- After receiving a Request message from the client, do the following:
  1. Use the `GIOPReplyCreate` function to create the Reply message.
  2. Use the CDR functions to build the message body.
  3. Use the `GIOPReplySend` function to send the Reply message.

Examples

The following example creates and sends a Reply message in response to a Request message that was received from a conversation with a client object.

```
#include <giop.h>
GIOPStateT gstate;
OctetT major, minor;
unsigned long rid;
  ⋮
/* receive a Request message from the remote client;
   reply using the request ID from the client */
major=gstate.gs_msg_in.mi_major;
minor=gstate.gs_msg_in.mi_minor;
CDRCodeULong(&gstate, &rid);
  ⋮
switch (GIOPReplyCreate(&gstate, major, minor,
        GIOP_NO_EXCEPTION, rid, (IORServiceContextListT *)0))
{
case GIOP_OK:
    /* add the reply message body by calling the CDR functions */
    switch (GIOPReplySend(&gstate, FALSE))
    {
    case GIOP_OK:
        /* the Reply message was sent successfully */
        break;
    default:
        /* there was some kind of send error */
        break;
    }
    break;

default:
    /* there was some kind of create error */
    break;
}
```

Related Functions

- “GIOPReplyCreate–Create a Reply Message” on page 2-91.

GIOPRequestCreate–Create a Request Message

This function creates a Request message to be sent from a client to a server over a previously established conversation.

Format

```
#include <giop.h>
GIOPStatusT GIOPRequestCreate(GIOPStateT             *giop
                              char                   *operation_p,
                              unsigned long           principal_len,
                              OctetT                 *principal_p,
                              IORServiceContextListT *scxt_p,
                              BooleanT                no_response);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a client conversation with a server.

operation_p
A pointer to a string that contains the name of the operation (member function) to be called on the server object.

principal_len
The size of the principal array.

principal_p
A pointer to the first octet in the principal array.

scxt_p
A pointer to the first element of an array of service contexts.

no_response
A flag that indicates the expected response. Use one of the following values:

FALSE
A response is expected from the server.

TRUE
A response is not expected from the server.

Normal Return

If successful, the `GIOPRequestCreate` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPRequestCreate` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- The principal identifies, in an ORB-dependent way, the client sending the request. In Common Data Representation (CDR), the principal is encoded as a sequence of octets.

- After this function returns successfully, call the CDR functions to build the request body and then call the GIOPRequestSend function to send the completed Request message to the server.
- The GIOP message version (major or minor) depends on the version of the Internet Inter-ORB Protocol (IIOP) profile used to open the connection; that is, the version is determined by the server.

Examples

The following example creates a Request message.

```
#include <giop.h>
#include <stdlib.h>
#include <string.h>
#define OPERATION "aMethod"
GIOPStateT gstate;
char *op;
/* establish a conversation with a server */
  :
op=malloc(sizeof OPERATION);
memcpy(op, OPERATION, sizeof OPERATION);
switch (GIOPRequestCreate(&gstate, op, 0, (OctetT *)0,
        (IORServiceContextListT *)0, FALSE))
{
case GIOP_OK:
    /* the request message was successfully created */
    break;

default:
    /* there was some kind of create error */
    break;
}
```

Related Functions

- “GIOPRequestSend–Send a Request Message” on page 2-97.

GIOPRequestSend–Send a Request Message

This function sends a previously created Request message from a client to a server over a previously established conversation.

Format

```
#include <giop.h>
GIOPStatusT GIOPRequestSend(GIOPStateT *giop
                            BooleanT    no_response,
                            BooleanT    more_fragments);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a client conversation with a server.

no_response
A flag that indicates the expected response. Use one of the following values:

FALSE
A response is expected from the server.

TRUE
A response is not expected from the server.

more_fragments
One of the following:

TRUE
Indicates that this message is a fragment, and more fragments will follow to complete the message.

FALSE
Indicates that this is a complete message.

Normal Return

If successful, the `GIOPRequestSend` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPRequestSend` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- Call the `GIOPRequestCreate` function to create a Request message and call the Common Data Representation (CDR) functions to build the request body before calling the `GIOPRequestSend` function to send the completed Request message to the server.

Examples

The following example creates and sends a Request message.

```
#include <giop.h>
#include <stdlib.h>
#include <string.h>
#include <cdr.h>
#define OPERATION "aMethod"
GIOPStateT gstate;
char *op;
/* establish a conversation with a server */
  ⋮
op=malloc(sizeof OPERATION);
memcpy(op, OPERATION, sizeof OPERATION);
switch (GIOPRequestCreate(&gstate, op, 0, (OctetT *)0,
        (IORServiceContextListT *)0, FALSE))
{
case GIOP_OK:
    /* the request message was successfully created,
       call CDR functions to build the request body */

  ⋮
    switch(GIOPRequestSend(&gstate;, FALSE, FALSE))
    {
    case GIOP_OK:
        /* the request message was successfully sent */
        break;
    default:
        /* there was some kind of send error */
        break;
    }
    break;
default:
    /* there was some kind of create error */
    break;
}
```

Related Functions

- “GIOPRequestCreate–Create a Request Message” on page 2-95.

GIOPStopListen–Notify Clients That the Server Is No Longer Listening for New Connections

This function notifies connected clients that the server will not be listening for new connections in the future.

Format

```
#include <giop.h>
GIOPStatusT GIOPStopListen(GIOPStateT *giop);
```

giop
A pointer to the General Inter-ORB Protocol (GIOP) state structure for a server conversation with a client.

Normal Return

If successful, the `GIOPStopListen` function returns GIOP_OK.

Error Return

If there is an error, the `GIOPStopListen` function returns a GIOP return value.

Programming Considerations

- See “General Inter-ORB Protocol Return Values” on page 2-5 for more information about GIOP return values.
- This function does not affect conversations that are currently established.
- Call this function only after the `GIOPListen` function has already been called.

Examples

The following example notifies a connected client that the server will not accept future connection requests.

```
#include <giop.h>
GIOPStateT gstate;
  ⋮
switch (GIOPStopListen(&gstate))
{
case GIOP_OK:
    /* the client knows that the server will not listen for future connections */
    break;
default:
    /* there was some kind of error */
    break;
}
```

Related Functions

“GIOPListen–Listen for Client Requests to Connect” on page 2-80.

iiop_error_code–Get Error Code for Last Transmission Control Protocol/Internet Protocol Error

This function gets the error code for the last Transmission Control Protocol/Internet Protocol (TCP/IP) error. Use this function to determine the type of error when a function receives a TCP/IP (or transport) error.

Format

```
#include <tcpcb.h>
IIOPStatusT iiop_error_code(void);
```

Normal Return

The `iiop_error_code` function returns an IIOPStatusT value that describes the last TCP/IP error.

Error Return

Not applicable.

Programming Considerations

- See “Internet Inter-ORB Protocol Return Values” on page 2-6 for more information about the IIOP return values.
- This function maps the TCP/IP error code to an Internet Inter-ORB Protocol (IIOP) error code.
- The General Inter-ORB Protocol (GIOP) functions store the IIOP status in the TCP state structure. This status can then be accessed directly from the structure. This function allows event handlers to report standard TCP/IP errors and maintain IIOP status in the TCP control block.

Examples

The following example receives the next inbound message, determines that there is a TCP/IP error, and gets the TCP/IP error code.

```
#include <giop.h>
GIOPStateT gstate;
GIOPMsgType mtype;
  ⋮
switch (GIOPGetNextMsg(&gstate, &mtype))
{
case GIOP_ECLOSED:
case GIOP_ETRANSPORT:
    switch (iiop_error_code))
    {
    case IIOP_EZERO_READ:
        /* handle error */
        break;
    case IIOP_ETIMEDOUT:
        /* handle error */
        break;
    case IIOP_ENETWORK_ERROR:
        /* handle error */
        break;
    case IIOP_EIO_ERROR:
        /* handle error */
        break;
    case IIOP_EUNKNOWN:
        /* handle error */
        break;
    /* and so on */
    }
    break;
/* and so on */
}
```

Related Functions

- “GIOPAccept–Accept a Connection from a Client” on page 2-60
- “GIOPCancelRequestSend–Cancel a Previously Sent Request Message” on page 2-65
- “GIOPCloseConnectionSend–Close an Open Connection” on page 2-67
- “GIOPConnect–Connect a Client to a Server” on page 2-68
- “GIOPFragSend–Send a Fragment Message” on page 2-72
- “GIOPGetNextMsg–Get the Next Incoming General Inter-ORB Protocol Message” on page 2-75
- “GIOPListen–Listen for Client Requests to Connect” on page 2-80
- “GIOPLocateReplyCreate–Create a LocateReply Message” on page 2-82
- “GIOPLocateReplySend–Send a LocateReply Message” on page 2-84
- “GIOPLocateRequestSend–Create and Send a LocateRequest Message to a Server” on page 2-86
- “GIOPMessageErrorSend–Create and Send a MessageError Message” on page 2-88
- “GIOPReject–Reject a Connection from a Client” on page 2-89
- “GIOPReplyCreate–Create a Reply Message” on page 2-91

- “GIOPRequestCreate–Create a Request Message” on page 2-95
- “GIOPRequestSend–Send a Request Message” on page 2-97.

IORAddTaggedProfile–Add a Tagged Profile to an Interoperable Object Reference Structure

This function adds a tagged profile to an Interoperable Object Reference (IOR) structure.

Format

```
#include <ior.h>
IORStatusT IORAddTaggedProfile(IORT          *ior_p,
                               IORProfileIdT  tag,
                               unsigned long  data_len,
                               OctetT        *data_p,
                               GIOPAllocFpT   getmem)
```

ior_p
A pointer to the target IOR structure.

tag
The tag type identifier of the profile to be added to the IOR.

data_len
The length of the profile data to be added to the IOR.

data_p
A pointer to the profile data to be added to the IOR. The data is an array of octets that hold encapsulated information.

getmem
A pointer to the memory allocation function of the calling application. This function is called to allocate a memory block. The `IORAddTaggedProfile` function copies the profile data in the allocated memory block. If the value of the **getmem** parameter is NULL, the passed data is used directly (data pointed to by the **data_p** parameter).

Normal Return

If successful, the `IORAddTaggedProfile` function returns IOR_OK.

Error Return

If there is an error, the `IORAddTaggedProfile` function returns an IOR return value. See “Interoperable Object Reference Return Values” on page 2-7.

Programming Considerations

- If the calling application supplies a memory allocation function (**getmem** parameter), the calling application is responsible for deallocating the memory.
- The data pointed to by the **data_p** parameter is an encapsulation octet stream of either an Internet Inter-ORB Protocol (IIOP) profile body or a multiple component profile. The caller encapsulates the data using the `IOREncapXX` functions before calling the `IORAddTaggedProfile` function.

Examples

The following example adds a tagged profile to an IOR structure.

```
#include <ior.h>
IORT   MyIOR;
char   MyTypeId_p[] = "MessageDisplay";
IORCreateIor(&MyIOR, MyTypeId_p, MyAllocationFunction);
  ⋮
/* Create an object key */
/* Encapsulate the object key, if necessary */
/* Create IIOP profile body and assign the IIOP version number, TCP/IP
   address and object key fields to the profile body structure. */
/* Encapsulate the profile body */
  ⋮
IORAddTaggedProfile(&MYIOR, TAG_INTERNET_IOP, DataLength, DataPtr, 0);
```

Related Functions

- “CDRAlloc–Register the Buffer Allocation Callback Function of a Common Data Representation Coder” on page 2-10
- “CDRDealloc–Buffer Deallocation Callback Function of a Common Data Representation Coder” on page 2-40.

IORCreateIor–Initialize an Interoperable Object Reference Structure

This function initializes an Interoperable Object Reference (IOR) structure.

Format

```
#include <ior.h>
IORStatusT IORCreateIor(IORT          *ior_p,
                        char          *typid_p,
                        unsigned long  max_profiles,
                        GIOPAllocFpT   getmem)
```

ior_p
A pointer to the IOR structure to be initialized.

typid_p
A pointer to the type identifier of the IOR to be initialized.

max_profiles
The number of tagged profiles to be maintained by this IOR.

getmem
A pointer to the memory allocation function of the calling application.

Normal Return

If successful, the `IORCreateIor` function returns IOR_OK.

Error Return

If there is an error, the `IORCreateIor` function returns an IOR return value. See “Interoperable Object Reference Return Values” on page 2-7.

Programming Considerations

Use the `IORFree` function to free the memory associated with the contents of the IOR structure before freeing the structure itself.

Examples

The following example initializes an IOR structure.

```
#include <ior.h>
IORT  MyIOR;
char  MyTypeId_p[] = "MessageDisplay";
IORCreateIor(&MyIOR, MyTypeId_p, MyAllocationFunction);
/* process IOR */
IORFree(&MyIOR, MyDeallocationFunction);
return;
/* My Memory Allocation function */
void *MyAllocationFunction(unsigned long Size)
{
void *m;
m = malloc ((unsigned int) Size);
return m;
}
/* My Memory Deallocation Function */
void MyDeallocationFunction(void *Data_p)
{
if (Data_p)
free (Data_p);
}
```

Related Functions

- “IORAddTaggedProfile–Add a Tagged Profile to an Interoperable Object Reference Structure” on page 2-103
- “IORFree–Free Resources Allocated to an Interoperable Object Reference Structure” on page 2-110
- User-written function of type GIOPAllocFpT
- User-written function of type GIOPDeallocFpT.

IOREncapIIOP–Encapsulate Internet Inter-ORB Protocol Profile Body

This function encapsulates the generic Internet Inter-ORB Protocol (IIOP) profile body, common to both Versions 1.0 and 1.1 of IIOP. The Version 1.1 specifics are coded manually by the application, following this call.

Format

```
#include <encap.h>
IORStatusT IOREncapIIOP(CDRCoderT       *cod_p,
                        OctetT          *major_p,
                        OctetT          *minor_p,
                        char           **host_pp,
                        unsigned short  *port_p,
                        unsigned long   *objkey_len_p,
                        Octet          **objkey_pp);
```

cod_p
A pointer to the Common Data Representation (CDR) coder that is used to encode the Interoperable Object Reference (IOR).

major_p
A pointer to the major IIOP version number.

minor_p
A pointer to the minor IIOP version number.

host_pp
A pointer to the address of the host name.

port_p
A pointer to the port number.

objkey_len_p
A pointer to the length of the object key.

objkey_pp
A pointer to the address of the object key.

Normal Return

If the `IOREncapIIOP` function is successful, it returns IOR_OK.

Error Return

If a null coder is specified, the `IOREncapIIOP` function returns IOR_ENULL_CODER.

Programming Considerations

- See “Interoperable Object Reference Return Values” on page 2-7 for more information about IOR return values.
- Under normal conditions in encoding mode, translations from EBCDIC to the ASCII character set take place. Under normal conditions in decoding mode, translations from the ASCII to EBCDIC character set take place. If the host name is coded in EBCDIC, it will automatically be converted to ASCII. You can suppress translation from EBCDIC to ASCII during encoding or decoding by first calling the CDR_NOCHARSET_CONV macro.

Examples

The example that follows creates an IIOP IOR by using the following algorithm:

1. Call the `IORCreateIor` function to initialize an IOR structure.
2. Create an object key that identifies the object in the server.
3. Optionally, encapsulate this object key in a sequence of octets by using the CDR functions.
4. Create an IIOP profile body of type IIOPBody_1_1T and assign the IIOP version number, Transmission Control Protocol/Internet Protocol (TCP/IP) address, and object key fields of the profile body structure.
5. Encapsulate this profile body in a sequence of octets by using the CDR functions.
6. Call the `IORAddTaggedProfile` function to add the encapsulated profile body to the IOR.
7. Repeat steps 4 to 6 to add additional IIOP profile bodies to the IOR, if required.

```
#include <giop.h>
CDRCoderT coder;
IORT ior;                       /* IOR. */
char *typeid_p;                 /* IORtype id. */
IIOPBody_1_1T iiop_body;        /* IIOP profile body. */
const char *okey_p;             /* Object key. */
unsigned long okey_len;
OctetT *pdata_p;                /* Encapsulated profile body */
unsigned long plen;
unsigned long num_tc = 0;       /* Number of tagged components. */

/* Initialize CDR coder */
  ⋮
/* Step 1: Create an initial IOR structure. */
typeid_p="IDL:MessageDisplay:1.0";
IORCreateIOR(&ior, typeid_p, 1, malloc);

/* Step 2: Create an object key. */
okey_p = "my object";
okey_len = strlen(okay_p);

/* Step 3: Encapsulate the object key.                      */
/* (The object key is coded in EBCDIC and is automatically */
/* converted to ASCII by the CDRCodeNString() function.)   */
CDREncapCreate(&coder, CDR_BYTE_ORDER);
CDRCodeNString(&coder, &okey_p, &okey_len);
CDREncapEnd(&coder, &iiop_body.ib_objkey_p, &iiop_body.ib_objkey_len, malloc);

/* Step 4: Assign version number, address and               */
/* encapsulated object key to profile body.                 */
/* (The assignment of the encapsulated object key           */
/* to iiop_body.ib.objkey_p is actually implicit in step 3.) */
iiop_body.ib_major = iiop_body.ib_minor = 1; /*IIOP 1.1 */
iiop_body.ib_host_p = malloc(strlen ("my_host") + 1);
strcpy(iiop_body.ib_host_p, "my host");
iiop_body.ib_port = 5678;
```

```
/* Step 5: Encapsulate the profile body. */
CDREncapCreate(&coder, CDR_BYTE_ORDER);
IOREncapIIOP(&coder, &iiop_body.ib_major, &iiop_body.ib_minor,
             &iiop_body.ib_host_p, &iiop_body.ib_port,
             &iiop_body.ib_objkey_len, &iiop_body.ib_objkey_p);
CDRCodeULong(&coder, &num_tc); /*IIOP 1.1 only */
CDREncapEnd(&coder, &pdate_p, &plen, malloc);
free (iiop_body.ib_host_p);

/* Step 6: Add encapsulated profile body to IOR. */
IORAddTaggedProfile(&ior, TAG_INTERNET_IOP, plen, pdata_p, getmem);
```

Related Functions

- “CDREncapCreate–Initialize a Common Data Representation Coder to Begin Encoding an Encapsulated Data Buffer” on page 2-44
- “CDREncapEnd–Complete Encoding an Encapsulated Data Buffer” on page 2-46
- “IORAddTaggedProfile–Add a Tagged Profile to an Interoperable Object Reference Structure” on page 2-103
- “IORCreateIor–Initialize an Interoperable Object Reference Structure” on page 2-105.

IORFree–Free Resources Allocated to an Interoperable Object Reference Structure

This function frees all resources that are allocated to an Interoperable Object Reference (IOR) structure.

Format

```
#include <ior.h>
void IORFree(IORT           *ior_p,
             GIOPDeallocFpT  delmem);
```

ior_p
A pointer to the IOR structure to be initialized.

delmem
A pointer to the memory deallocation function of the calling application.

Normal Return

Void.

Error Return

Not applicable.

Programming Considerations

Call this function to free the memory associated with the contents of the IOR structure before freeing the structure itself.

Examples

The following example frees resources associated with an IOR structure.

```
#include <ior.h>
IORT MyIOR;
IORCreateIor(&MyIOR, MyTypeId_p, MyAllocationFunction);
  ⋮
/* process IOR, add Tagged Profile, etc  */
  ⋮
IORFree(&MyIOR, MyDeallocationFunction);
return;
/* My Memory Allocation function */
void *MyAllocationFunction(void *Data_p)
{
if (Data_p)
free (Data_p);
}
```

Related Functions

- "IORCreateIor–Initialize an Interoperable Object Reference Structure" on page 2-105
- User-written function of type GIOPAllocFpT
- User-written function of type GIOPDeallocFpT.

IORFromString–Convert an Interoperable Object Reference String to an IOR Structure

This function converts an Interoperable Object Reference (IOR) string to an IOR structure.

Format

```
#include <ior.h>
IORStatusT IORFromString(IORT          *ior_p,
                         unsigned char *iorstr_p,
                         GIOPAllocFpT   getmem,
                         GIOPAllocFpT   delmem)
```

ior_p
A pointer to the target IOR structure.

iorstr_p
A pointer to the address of the IOR string to be converted.

getmem
A pointer to the memory allocation function of the calling application.

delmem
A pointer to the memory deallocation function of the calling application.

Normal Return

If successful, the `IORFromString` function returns IOR_OK.

Error Return

If there is an error, the `IORFromString` function returns an IOR return value. See “Interoperable Object Reference Return Values” on page 2-7.

Programming Considerations

The format of the string must be "IOR: *string*".

Examples

The following example converts an IOR string to an IOR structure.

```
#include <ior.h>
IORT  MyIOR;
char *MyIORString[]="IOR: Data containing the ID type and Tagged Profiles";

IORFromString(&MyIOR, MyIORString, MyAllocationFunction, MyDeallocationFunction);
```

Related Functions

- “IORFree–Free Resources Allocated to an Interoperable Object Reference Structure” on page 2-110
- “IORToString–Convert an Interoperable Object Reference Structure to a String” on page 2-114
- User-written function of type GIOPAllocFpT

- User-written function of type GIOPDeAllocFpT.

IORToString–Convert an Interoperable Object Reference Structure to a String

This function converts an Interoperable Object Reference (IOR) structure to a string.

Format

```
#include <ior.h>
IORToString(IORT           *ior_p,
            unsigned char **iorstr_pp,
            GIOPAllocFpT    getmem);
```

ior_p
A pointer to the target IOR structure.

iorstr_p
A pointer to the address obtained from the **getmem** parameter, which will hold the resulting IOR string.

getmem
A pointer to the memory allocation function of the calling application. This function allocates the memory to hold the resulting IOR string.

Normal Return

If successful, the `IORToString` function returns IOR_OK.

Error Return

If there is an error, the `IORToString` function returns an IOR return value. See “Interoperable Object Reference Return Values” on page 2-7.

Programming Considerations

- The resulting string is in the form "IOR: *string*".
- The resulting string can be used by the server application to publish an IOR.

Examples

The following example converts an IOR structure to a string.

```
#include <ior.h>
IORT            MYIOR;
unsigned char  *MyIORString;
IORFromString(&MyIOR, MyIORString, MyAllocationFunction);
```

Related Functions

- “CDRCodeNOctet–Encode or Decode Octet Values” on page 2-26
- “CDRCodeString–Encode or Decode a String Value” on page 2-34
- “CDRCodeULong–Encode or Decode an Unsigned Long Value” on page 2-36
- “CDREncapInit–Initialize a Common Data Representation Decoder to Decode or Encode an Encapsulated Data Buffer” on page 2-48
- “IORFromString–Convert an Interoperable Object Reference String to an IOR Structure” on page 2-112

- User-written function of type GIOPAllocFpT.

tpf_asc2ebc, tpf_ebc2asc–Convert Characters between ISO 8859-1 (ASCII) and IBM-1047 (US EBCDIC)

These functions do the following respectively:

- `tpf_asc2ebc` converts an ISO 8859-1 coded array of characters to IBM-1047 coding.
- `tpf_ebc2asc` converts an IBM-1047 coded array of characters to ISO 8859-1 coding.

Format

```
#include <cdr.h>
int tpf_asc2ebc(char *first_char, int char_count);
int tpf_ebc2asc(char *first_char, int char_count);
```

first_char
A pointer to the address of the first character to be converted.

char_count
The number of characters to be converted.

Normal Return

These functions return the number of bytes that were translated.

Error Return

Not applicable.

Programming Considerations

You can modify the conversion tables to convert to a different single-byte-coded code page. See *TPF Application Programming* for more information.

Examples

The following example asserts EBCDIC on the wire and then receives an ASCII coded string and manually converts it to EBCDIC.

```
#include <cdr.h>
CDRCoderT coder;
char buffer[80];
char *bptr = buffer;
unsigned long len = sizeof buffer;
CDRInit(&coder, CDR_BYTE_ORDER, 512);
CDREbcdic_OTW(&coder);
  ⋮
CDRCodeNString(&coder, &bptr, &len);
tpf_asc2ebc(buffer, sizeof buffer);
```

Related Functions

- "CDRCodeChar–Encode or Decode a Char Value" on page 2-16
- "CDRCodeNString–Encode or Decode a String Value" on page 2-28
- "CDRCodeString–Encode or Decode a String Value" on page 2-34
- "CDREbcdic_OTW, CDRS390_OTW–Override Platform-Oriented Data Conversions" on page 2-42.

Index

A

C

D

E

F

G

H

I

Communicating Your Comments to IBM

Internet Inter-ORB Protocol Connect for TPF
Reference
Release 1

Publication No. SH31-0188-01

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 1 + 845 + 432 + 9788
  - Other countries: (international code) + 845 + 432 + 9788
- If you prefer to send comments electronically, use this network ID:
  - Internet e-mail: tpfid@us.ibm.com
  - IBMLINK (and DialIBM in Europe): ETR function of ServiceLink

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies.

Readers' Comments — We'd Like to Hear from You

Internet Inter-ORB Protocol Connect for TPF
Reference
Release 1

Publication No. SH31-0188-01

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

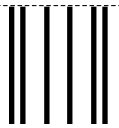
Company or Organization _____

Phone No. _____

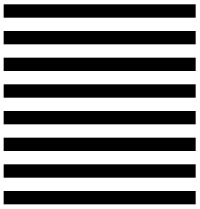
Readers' Comments — We'd Like to Hear from You
SH31-0188-01

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
TPF Systems Information Development
Mail Station P923
2455 SOUTH ROAD
POUGHKEEPSIE NY 12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

SH31-0188-01

Cut or Fold
Along Line

File Number: S370/30XX-40
Program Number: 5799-D64

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SH31-0188-01