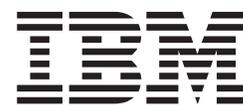


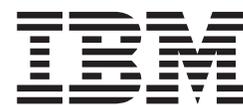
z/OS Communications Server



APPC Application Suite Programming

Version 1 Release 2

z/OS Communications Server



APPC Application Suite Programming

Version 1 Release 2

Note:

Before using this information and the product it supports, be sure to read the general information under "Appendix E. Notices" on page 171.

First Edition (October 2001)

This edition applies to Version 1 Release 2 of z/OS (program number 5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

Publications are not stocked at the address given below. If you want more IBM publications, ask your IBM representative or write to the IBM branch office serving your locality.

A form for your comments is provided at the back of this document. If the form has been removed, you may address comments to:

IBM Corporation
Software Reengineering
Department G71A/ Bldg 503
Research Triangle Park, North Carolina 27709-9990
U.S.A.

If you prefer to send comments electronically, use one of the following methods:

Fax (USA and Canada):

1-800-227-5088

Internet e-mail:

usib2hpd@vnet.ibm.com

World Wide Web:

<http://www.ibm.com/servers/eserver/zseries/zos>

IBMLink:

CIBMORCF at RALVM17

IBM Mail Exchange:

tkinlaw@us.ibm.com

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.

© Copyright International Business Machines Corporation 1994, 2001. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables	vii
About This Book	ix
Where to Find More Information	ix
Where to Find Related Information on the Internet	ix
Licensed Documents	x
LookAt, an Online Message Help Facility	x
How to Contact IBM® Service	xi
z/OS Communications Server Information	xi
Summary of Changes	xix

Part 1. APPC File Transfer Protocol (AFTP) Programming Interface 1

Chapter 1. API for APPC File Transfer Protocol	3
AFTP Defined Constants, Standard Types and Conventions	3
Defined Constants	3
Standard Types	3
Conventions	4
Compiling the AFTP Application	5
MVS	5
VM	5
Overview of API Calls	5
Create or Destroy an AFTP Connection Object	6
Establish a Connection to the AFTP Server Computer	6
Query Connection Characteristics	7
Transfer Files	7
Specify File Transfer Characteristics	7
Query File Transfer Characteristics	8
List Files on the AFTP Server Computer	8
List Files on the AFTP Client Computer	9
Perform Directory Manipulation	9
Perform File Manipulation	10
Query System Information	10
Generate Message Strings	10
Control Trace Information	10
Miscellaneous	11
Chapter 2. AFTP API Call Reference	13
aftp_change_dir	14
aftp_close	16
aftp_connect	17
aftp_create	18
aftp_create_dir	19
aftp_delete	21
aftp_destroy	23
aftp_dir_close	24
aftp_dir_open	25
aftp_dir_read	27
aftp_extract_allocation_size	30
aftp_extract_block_size	31
aftp_extract_data_type	32
aftp_extract_date_mode	33

aftp_extract_destination	34
aftp_extract_mode_name	36
aftp_extract_partner_LU_name	38
aftp_extract_password	40
aftp_extract_record_format	42
aftp_extract_record_length	44
aftp_extract_security_type	45
aftp_extract_tp_name	47
aftp_extract_trace_level	49
aftp_extract_userid	50
aftp_extract_write_mode	52
aftp_format_error	53
aftp_get_data_type_string	55
aftp_get_date_mode_string	57
aftp_get_record_format_string	59
aftp_get_write_mode_string	61
aftp_load_ini_file	63
aftp_local_change_dir	65
aftp_local_dir_close	67
aftp_local_dir_open	68
aftp_local_dir_read	70
aftp_local_query_current_dir	73
aftp_query_bytes_transferred	75
aftp_query_current_dir	76
aftp_query_local_system_info	78
aftp_query_local_version	80
aftp_query_system_info	81
aftp_receive_file	83
aftp_remove_dir	85
aftp_rename	87
aftp_send_file	89
aftp_set_allocation_size	91
aftp_set_block_size	92
aftp_set_data_type	93
aftp_set_date_mode	95
aftp_set_destination	96
aftp_set_mode_name	98
aftp_set_password	99
aftp_set_record_format	101
aftp_set_record_length	103
aftp_set_security_type	104
aftp_set_tp_name	106
aftp_set_trace_filename	108
aftp_set_trace_level	109
aftp_set_userid	110
aftp_set_write_mode	112

Chapter 3. AFTP Return Codes 113

Part 2. APPC Name Server (ANAME) Programming Interface 115

Chapter 4. API for the APPC NameServer	117
How the ANAME API Works	117
ANAME Defined Constants, Standard Types, and Conventions	117
Defined Constants	117
Standard Types	118

Conventions	118
Compiling the ANAME Application	119
MVS	119
VM	119
Overview of API Calls	120
Create or Destroy an ANAME Connection Object	120
Set Values in the Connection Object	120
Set Values in the Data Object	120
Add a Record to the Database	121
Remove Records from the Database	121
Obtain Records from the Database	121
Access Values in Returned Records	122
Obtain Error Information	122
Turn Tracing On and Off	122
Use System Administrator Functions	122
Chapter 5. ANAME API Call Reference	123
aname_create	124
aname_delete	125
aname_destroy	126
aname_extract_fqlu_name	127
aname_extract_group_name	129
aname_extract_tp_name	131
aname_extract_user_name	133
aname_format_error	135
aname_query	137
aname_receive	138
aname_register	139
aname_set_destination	140
aname_set_duplicate_register	141
aname_set_fqlu_name	143
aname_set_group_name	145
aname_set_tp_name	146
aname_set_trace_filename	147
aname_set_trace_level	148
aname_set_user_name	149
Chapter 6. ANAME Return Codes	151

Part 3. Appendixes	153
Appendix A. Entry Point Mappings	155
Appendix B. Sample Program for AFTP API	157
Appendix C. Sample Program for ANAME API	161
Appendix D. Information Apars	169
IP Information Apars	169
SNA Information Apars	170
Appendix E. Notices	171
Trademarks	174
Index	177

Tables

1.	AFTP Size Constants	3
2.	AFTP Standard Types	4
3.	ANAME Size Constants	118
4.	ANAME Standard Types.	118
5.	AFTP API Call Mappings	155
6.	ANAME API Call Mappings	156
7.	IP Information Apars	169
8.	SNA Information Apars	170

About This Book

The APPC Application Suite provides application program interfaces (APIs) to its APPC File Transfer Protocol (AFTP) functions and its APPC Name Server (ANAME) functions. This book provides the information you will need to write an application program to implement AFTP and ANAME client functions.

This book is written for application programmers. It assumes that you are familiar with writing C language applications.

Although the APPC Application Suite API is based on common programming interface for communications (CPI-C) and advanced program-to-program communications (APPC), you do not necessarily need to know these concepts to successfully write applications based on this API.

Use this book as a reference to identify and explain the various AFTP and ANAME API calls, call parameters, and line flows.

Where to Find More Information

This section contains:

- Pointers to information available on the Internet
- Information about licensed documentation
- Information about LookAt, the online message tool
- A set of tables that describes the books in the z/OS Communications Server (z/OS CS) library, along with related publications

Where to Find Related Information on the Internet

Home Page	Web address
z/OS	http://www.ibm.com/servers/eserver/zseries/zos/
z/OS Internet Library	http://www.ibm.com/servers/eserver/zseries/zos/bkserv/
IBM Communications Server product	http://www.software.ibm.com/network/commserver/
IBM Communications Server support	http://www.software.ibm.com/network/commserver/support/
IBM Systems Center publications	http://www.redbooks.ibm.com/
IBM Systems Center flashes	http://www-1.ibm.com/support/techdocs/atmastr.nsf
VTAM and TCP/IP	http://www.software.ibm.com/network/commserver/about/csos390.html
IBM	http://www.ibm.com
RFC	http://www.ietf.org/rfc.html

Information about Web addresses can also be found in informational APAR II11334.

DNS Web Sites

For information about DNS, see the following Web sites:

USENET news groups:
comp.protocols.dns.bind

For BIND mailing lists, see:

- <http://www.isc.org/ml-archives/>
 - BIND Users
 - Subscribe by sending mail to bind-users-request@isc.org
 - Submit questions or answers to this forum by sending mail to bind-users@isc.org
 - BIND 9 Users (Note: This list may not be maintained indefinitely.)
 - Subscribe by sending mail to bind9-users-request@isc.org
 - Submit questions or answers to this forum by sending mail to bind9-users@isc.org

For definitions of the terms and abbreviations used in this book, you can view or download the latest *IBM Glossary of Computing Terms* at the following Web address:

<http://www.ibm.com/ibm/terminology>

Note: Any pointers in this publication to Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Licensed Documents

z/OS Communications Server licensed documentation in PDF format is available on the Internet at the IBM Resource Link Web site at <http://www.ibm.com/servers/resourcelink>. Licensed books are available only to customers with a z/OS Communications Server license. Access to these books requires an IBM Resource Link Web user ID and password, and a key code. With your z/OS Communications Server order, you received a memo that includes this key code. To obtain your IBM Resource Link Web user ID and password, log on to <http://www.ibm.com/servers/resourcelink>. To register for access to the z/OS licensed books perform the following steps:

1. Log on to Resource Link using your Resource Link user ID and password.
2. Click on **User Profiles** located on the left-hand navigation bar.
3. Click on **Access Profile**.
4. Click on **Request Access to Licensed books**.
5. Supply your key code where requested and click on the **Submit** button.

If you supplied the correct key code, you will receive confirmation that your request is being processed. After your request is processed, you will receive an e-mail confirmation.

You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed. To access the licensed books:

1. Log on to Resource Link using your Resource Link user ID and password.
2. Click on **Library**.
3. Click on **zSeries**.
4. Click on **Software**.
5. Click on **z/OS Communications Server**.
6. Access the licensed book by selecting the appropriate element.

LookAt, an Online Message Help Facility

LookAt is an online facility that allows you to look up explanations for z/OS CS messages and system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on **News and Help** or from the z/OS V1R2 Collection, SK3T-4269.

To find a message explanation from a TSO command line, simply enter **lookat+message ID**, as in the following example:

```
lookat ezz8477i
```

This results in direct access to the message explanation for message EZZ8477I.

You can use LookAt on the Internet at the following Web site:
www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html

To find a message explanation from the LookAt Web site, simply enter the message ID. You can select the release, if applicable.

How to Contact IBM® Service

For telephone assistance in problem diagnosis and resolution (in the United States or Puerto Rico), call the IBM Software Support Center anytime (1-800-237-5511). You will receive a return call within 8 business hours (Monday – Friday, 8:00 a.m. – 5:00 p.m., local customer time).

Outside of the United States or Puerto Rico, contact your local IBM representative or your authorized IBM supplier.

z/OS Communications Server Information

This section contains descriptions of the books in the z/OS Communications Server library.

z/OS Communications Server publications are available:

- Online at the z/OS Internet Library web page at <http://www.ibm.com/servers/eserver/zseries/zos/>
- In hardcopy and softcopy
- In softcopy only

Softcopy Information

Softcopy publications are available in the following collections:

Titles	Order Number	Description
<i>z/OS V1R2 Collection</i>	SK3T-4269	This is the CD collection shipped with the z/OS product. It includes the libraries for z/OS V1R2, in both BookManager and PDF formats.
<i>z/OS Software Products Collection</i>	SK3T-4270	This CD includes, in both BookManager and PDF formats, the libraries of z/OS software products that run on z/OS but are not elements and features, as well as the <i>Getting Started with Parallel Sysplex</i> bookshelf.

Titles	Order Number	Description
<i>z/OS V1R2 and Software Products DVD Collection</i>	SK3T-4271	This collection includes the libraries of z/OS (the element and feature libraries) and the libraries for z/OS software products in both BookManager and PDF format. This collection combines SK3T-4269 and SK3T-4270.
<i>z/OS Licensed Product Library</i>	SK3T-4307	This CD includes the licensed books in both BookManager and PDF format.
<i>System Center Publication IBM S/390 Redbooks Collection</i>	SK2T-2177	This collection contains over 300 ITSO redbooks that apply to the S/390 platform and to host networking arranged into subject bookshelves.

z/OS Communications Server Library

The following abbreviations follow each order number in the tables below.

HC/SC — Both hardcopy and softcopy are available.

SC — Only softcopy is available. These books are available on the CD Rom accompanying z/OS (SK3T-4269 or SK3T-4307). Unlicensed books can be viewed at the z/OS Internet library site.

Updates to books are available on RETAIN and in the document called *OS/390 DOC APARs and ++HOLD DOC data* which can be found at http://www.s390.ibm.com/os390/bkserv/new_tech_info.html. See “Appendix D. Information Apars” on page 169 for a list of the books and the informational apars (info apars) associated with them.

Planning and Migration:

Title	Number	Format	Description
<i>z/OS Communications Server: SNA Migration</i>	GC31-8774	HC/SC	This book is intended to help you plan for SNA, whether you are migrating from a previous version or installing SNA for the first time. This book also identifies the optional and required modifications needed to enable you to use the enhanced functions provided with SNA.
<i>z/OS Communications Server: IP Migration</i>	GC31-8773	HC/SC	This book is intended to help you plan for TCP/IP Services, whether you are migrating from a previous version or installing IP for the first time. This book also identifies the optional and required modifications needed to enable you to use the enhanced functions provided with TCP/IP Services.

Resource Definition, Configuration, and Tuning:

Title	Number	Format	Description
<i>z/OS Communications Server: IP Configuration Guide</i>	SC31-8775	HC/SC	This book describes the major concepts involved in understanding and configuring an IP network. Familiarity with the z/OS operating system, IP protocols, z/OS UNIX System Services, and IBM Time Sharing Option (TSO) is recommended. Use this book in conjunction with the <i>z/OS Communications Server: IP Configuration Reference</i> .

Title	Number	Format	Description
<i>z/OS Communications Server: IP Configuration Reference</i>	SC31-8776	HC/SC	This book presents information for people who want to administer and maintain IP. Use this book in conjunction with the <i>z/OS Communications Server: IP Configuration Guide</i> . The information in this book includes: <ul style="list-style-type: none"> • TCP/IP configuration data sets • Configuration statements • Translation tables • SMF records • Protocol number and port assignments
<i>z/OS Communications Server: SNA Network Implementation Guide</i>	SC31-8777	HC/SC	This book presents the major concepts involved in implementing an SNA network. Use this book in conjunction with the <i>z/OS Communications Server: SNA Resource Definition Reference</i> .
<i>z/OS Communications Server: SNA Resource Definition Reference</i>	SC31-8778	HC/SC	This book describes each SNA definition statement, start option, and macroinstruction for user tables. It also describes NCP definition statements that affect SNA. Use this book in conjunction with the <i>z/OS Communications Server: SNA Network Implementation Guide</i> .
<i>z/OS Communications Server: SNA Resource Definition Samples</i>	SC31-8836	SC	This book contains sample definitions to help you implement SNA functions in your networks, and includes sample major node definitions.
<i>z/OS Communications Server: AnyNet SNA over TCP/IP</i>	SC31-8832	SC	This guide provides information to help you install, configure, use, and diagnose SNA over TCP/IP.
<i>z/OS Communications: Server AnyNet Sockets over SNA</i>	SC31-8831	SC	This guide provides information to help you install, configure, use, and diagnose sockets over SNA. It also provides information to help you prepare application programs to use sockets over SNA.

Operation:

Title	Number	Format	Description
<i>z/OS Communications Server: IP User's Guide and Commands</i>	SC31-8780	HC/SC	This book describes how to use TCP/IP applications. It contains requests that allow a user to: log on to a remote host using Telnet, transfer data sets using FTP, send and receive electronic mail, print on remote printers, and authenticate network users.
<i>z/OS Communications Server: IP System Administrator's Commands</i>	SC31-8781	HC/SC	This book describes the functions and commands helpful in configuring or monitoring your system. It contains system administrator's commands, such as NETSTAT, PING, TRACERTE and their UNIX counterparts. It also includes TSO and MVS commands commonly used during the IP configuration process.
<i>z/OS Communications Server: SNA Operation</i>	SC31-8779	HC/SC	This book serves as a reference for programmers and operators requiring detailed information about specific operator commands.
<i>z/OS Communications Server: Operations Quick Reference</i>	SX75-0124	HC/SC	This book contains essential information about SNA and IP commands.

Customization:

Title	Number	Format	Description
<i>z/OS Communications Server: SNA Customization</i>	LY43-0092	SC	This book enables you to customize SNA, and includes the following: <ul style="list-style-type: none"> • Communication network management (CNM) routing table • Logon-interpret routine requirements • Logon manager installation-wide exit routine for the CLU search exit • TSO/SNA installation-wide exit routines • SNA installation-wide exit routines
<i>z/OS Communications Server: IP Network Print Facility</i>	SC31-8833	SC	This book is for system programmers and network administrators who need to prepare their network to route SNA, JES2, or JES3 printer output to remote printers using TCP/IP Services.

Writing Application Programs:

Title	Number	Format	Description
<i>z/OS Communications Server: IP Application Programming Interface Guide</i>	SC31-8788	SC	This book describes the syntax and semantics of program source code necessary to write your own application programming interface (API) into TCP/IP. You can use this interface as the communication base for writing your own client or server application. You can also use this book to adapt your existing applications to communicate with each other using sockets over TCP/IP.
<i>z/OS Communications Server: IP CICS Sockets Guide</i>	SC31-8807	SC	This book is for people who want to set up, write application programs for, and diagnose problems with the socket interface for CICS using z/OS TCP/IP.
<i>z/OS Communications Server: IP IMS Sockets Guide</i>	SC31-8830	SC	This book is for programmers who want application programs that use the IMS TCP/IP application development services provided by IBM's TCP/IP Services.
<i>z/OS Communications Server: IP Programmer's Reference</i>	SC31-8787	SC	This book describes the syntax and semantics of a set of high-level application functions that you can use to program your own applications in a TCP/IP environment. These functions provide support for application facilities, such as user authentication, distributed databases, distributed processing, network management, and device sharing. Familiarity with the z/OS operating system, TCP/IP protocols, and IBM Time Sharing Option (TSO) is recommended.
<i>z/OS Communications Server: SNA Programming</i>	SC31-8829	SC	This book describes how to use SNA macroinstructions to send data to and receive data from (1) a terminal in either the same or a different domain, or (2) another application program in either the same or a different domain.
<i>z/OS Communications Server: SNA Programmers LU 6.2 Guide</i>	SC31-8811	SC	This book describes how to use the SNA LU 6.2 application programming interface for host application programs. This book applies to programs that use only LU 6.2 sessions or that use LU 6.2 sessions along with other session types. (Only LU 6.2 sessions are covered in this book.)

Title	Number	Format	Description
<i>z/OS Communications Server: SNA Programmers LU 6.2 Reference</i>	SC31-8810	SC	This book provides reference material for the SNA LU 6.2 programming interface for host application programs.
<i>z/OS Communications Server: CSM Guide</i>	SC31-8808	SC	This book describes how applications use the communications storage manager.
<i>z/OS Communications Server: CMIP Services and Topology Agent Guide</i>	SC31-8828	SC	This book describes the Common Management Information Protocol (CMIP) programming interface for application programmers to use in coding CMIP application programs. The book provides guide and reference information about CMIP services and the SNA topology agent.

Diagnosis:

Title	Number	Format	Description
<i>z/OS Communications Server: IP Diagnosis</i>	GC31-8782	HC/SC	This book explains how to diagnose TCP/IP problems and how to determine whether a specific problem is in the TCP/IP product code. It explains how to gather information for and describe problems to the IBM Software Support Center.
<i>z/OS Communications Server: SNA Diagnosis V1 Techniques and Procedures</i> and <i>z/OS Communications Server: SNA Diagnosis V2 FFST Dumps and the VIT</i>	LY43-0088 LY43-0089	HC/SC	These books help you identify an SNA problem, classify it, and collect information about it before you call the IBM Support Center. The information collected includes traces, dumps, and other problem documentation.
<i>z/OS Communications Server: SNA Data Areas Volume 1</i> and <i>z/OS Communications Server: SNA Data Areas Volume 2</i>	LY43-0090 LY43-0091	SC	These books describe SNA data areas and can be used to read an SNA dump. They are intended for IBM programming service representatives and customer personnel who are diagnosing problems with SNA.

Messages and Codes:

Title	Number	Format	Description
<i>z/OS Communications Server: SNA Messages</i>	SC31-8790	HC/SC	This book describes the ELM, IKT, IST, ISU, IUT, IVT, and USS messages. Other information in this book includes: <ul style="list-style-type: none"> • Command and RU types in SNA messages • Node and ID types in SNA messages • Supplemental message-related information
<i>z/OS Communications Server: IP Messages Volume 1 (EZA)</i>	SC31-8783	HC/SC	This volume contains TCP/IP messages beginning with EZA.
<i>z/OS Communications Server: IP Messages Volume 2 (EZB)</i>	SC31-8784	HC/SC	This volume contains TCP/IP messages beginning with EZB.
<i>z/OS Communications Server: IP Messages Volume 3 (EZY)</i>	SC31-8785	HC/SC	This volume contains TCP/IP messages beginning with EZY.

Title	Number	Format	Description
<i>z/OS Communications Server: IP Messages Volume 4 (EZZ-SNM)</i>	SC31-8786	HC/SC	This volume contains TCP/IP messages beginning with EZZ and SNM.
<i>z/OS Communications Server: IP and SNA Codes</i>	SC31-8791	HC/SC	This book describes codes and other information that appear in z/OS Communications Server messages.

APPC Application Suite:

Title	Number	Format	Description
<i>z/OS Communications Server: APPC Application Suite User's Guide</i>	GC31-8809	SC	This book documents the end-user interface (concepts, commands, and messages) for the AFTP, ANAME, and APING facilities of the APPC application suite. Although its primary audience is the end user, administrators and application programmers may also find it useful.
<i>z/OS Communications Server APPC Application Suite Administration</i>	SC31-8835	SC	This book contains the information that administrators need to configure the APPC application suite and to manage the APING, ANAME, AFTP, and A3270 servers.
<i>z/OS Communications Server: APPC Application Suite Programming</i>	SC31-8834	SC	This book provides the information application programmers need to add the functions of the AFTP and ANAME APIs to their application programs.

Redbooks

The following Redbooks may help you as you implement z/OS Communications Server.

Title	Number
<i>TCP/IP Tutorial and Technical Overview</i>	GG24-3376
<i>SNA and TCP/IP Integration</i>	SG24-5291
<i>IBM Communication Server for OS/390 V2R10 TCP/IP Implementation Guide: Volume 1: Configuration and Routing</i>	SG24-5227
<i>IBM Communication Server for OS/390 V2R10 TCP/IP Implementation Guide: Volume 2: UNIX Applications</i>	SG24-5228
<i>IBM Communication Server for OS/390 V2R10 TCP/IP Implementation Guide: Volume 3: MVS Applications</i>	SG24-5229
<i>OS/390 Secureway Communication Server V2R8 TCP/IP Guide to Enhancements</i>	SG24-5631
<i>TCP/IP in a Sysplex</i>	SG24-5235
<i>Managing OS/390 TCP/IP with SNMP</i>	SG24-5866
<i>Security in OS/390-based TCP/IP Networks</i>	SG24-5383
<i>IP Network Design Guide</i>	SG24-2580

Related Information

For information about z/OS products, refer to *z/OS Information Roadmap* (SA22-7500). The Roadmap describes what level of documents are supplied with each release of z/OS Communications Server, as well as describing each z/OS publication.

The table below lists books that may be helpful to readers.

Title	Number
<i>z/OS SecureWay Security Server Firewall Technologies</i>	SC24-5922
<i>S/390: OSA-Express Customer's Guide and Reference</i>	SA22-7403
<i>z/OS MVS Diagnosis: Procedures</i>	GA22-7587
<i>z/OS MVS Diagnosis: Reference</i>	GA22-7588
<i>z/OS MVS Diagnosis: Tools and Service Aids</i>	GA22-7589

Determining If a Publication Is Current

As needed, IBM updates its publications with new and changed information. For a given publication, updates to the hardcopy and associated BookManager softcopy are usually available at the same time. Sometimes, however, the updates to hardcopy and softcopy are available at different times. Here is how to determine if you are looking at the most current copy of a publication:

1. At the end of a publication's order number there is a dash followed by two digits, often referred to as the dash level. A publication with a higher dash level is more current than one with a lower dash level. For example, in the publication order number GC28-1747-07, the dash level 07 means that the publication is more current than previous levels, such as 05 or 04.
2. If a hardcopy publication and a softcopy publication have the same dash level, it is possible that the softcopy publication is more current than the hardcopy publication. Check the dates shown in the Summary of Changes. The softcopy publication might have a more recently dated Summary of Changes than the hardcopy publication.
3. To compare softcopy publications, you can check the last two characters of the publication's filename (also called the book name). The higher the number, the more recent the publication. Also, next to the publication titles in the CD-ROM booklet and the readme files, there is an asterisk (*) that indicates whether a publication is new or changed.

Summary of Changes

| **Summary of Changes**
| **for SC31-8834-00**
| **z/OS Version 1 Release 2**

| This book contains information previously presented in *OS/390 V2R5 eNetwork*
| *Communications Server: APPC Applications Suite Programming*, SC31-8621.

| This book contains minor terminology, maintenance, and editorial changes.
| Technical changes or additions to the text and illustrations are indicated by a
| vertical line to the left of the change.

Part 1. APPC File Transfer Protocol (AFTP) Programming Interface

Chapter 1. API for APPC File Transfer Protocol

The APPC File Transfer Protocol (AFTP) application programming interface (API) is a set of C routines that provides APPC file transfer capabilities. This API makes file transfer programming easier by allowing you to access routines that will interact with any AFTP server.

The AFTP API adheres to the AFTP line flow standards. All AFTP client applications will send the same set of line flows over the network to the AFTP server.

AFTP Defined Constants, Standard Types and Conventions

This section describes constants, types and conventions for use in the AFTP API that are not standard to C programming.

Defined Constants

AFTP constant definitions have been created for the sizes of the buffers that are passed across the API. All buffers of the specified type must be at least the size of the defined constant to guarantee the requested call will not fail with a buffer size error.

The constants are defined in the header file for the AFTP API.

Table 1. AFTP Size Constants

Constant	Minimum Buffer Size For	Value
AFTP_DATA_TYPE_SIZE	data_type_size	64
AFTP_DATE_MODE_SIZE	date_mode_size	64
AFTP_FILE_NAME_SIZE	dir_entry_size or dir_entry_size or path_buffer_length	512
AFTP_FQLU_NAME_SIZE	destination_size or partner_LU_name_size	64
AFTP_MESSAGE_SIZE	error_str_size	2048
AFTP_MODE_NAME_SIZE	mode_name_size	8
AFTP_PASSWORD_SIZE	password_size	10
AFTP_RECORD_FORMAT_SIZE	record_format_size	64
AFTP_SYSTEM_INFO_SIZE	system_info_size	512
AFTP_TP_NAME_SIZE	tp_name_size	64
AFTP_USERID_SIZE	userid_size	10
AFTP_WRITE_MODE_SIZE	write_mode_size	64

Standard Types

Type definitions are available for many parameters to the AFTP API calls. For example, the AFTP type AFTP_LENGTH_TYPE is an alias for the C type unsigned long.

Use the AFTP types instead of the corresponding C types. Doing so will protect you from changes to the parameters in future releases. If you have used the AFTP

types, you will only need to recompile your code to use the new API definitions. If you have used the C types, you will need to modify your program source to reflect changes to the new C types.

The AFTP API avoids complex structures and pointers to structures for type definitions. These complex structures might not be supported in all languages. The only exception is the string construct which is found in many languages.

Table 2. AFTP Standard Types

Type	Description
AFTP_HANDLE_TYPE	The AFTP connection object id
AFTP_ALLOCATION_SIZE_TYPE	The file allocation size
AFTP_BLOCK_SIZE_TYPE	The file block size
AFTP_BOOLEAN_TYPE	A boolean (FALSE=0, TRUE=1) type
AFTP_DATA_TYPE_TYPE	The file data types that can be transferred
AFTP_DATE_MODE_TYPE	The date mode used for transferred files
AFTP_DETAIL_LEVEL_TYPE	Amount of information to be output when AFTP generates error messages
AFTP_FILE_TYPE_TYPE	The kind of a file (directory/file) listed by AFTP
AFTP_INFO_LEVEL_TYPE	The amount of information listed for a file by AFTP
AFTP_LENGTH_TYPE	The size of input buffers, and the actual returned length of buffers in AFTP
AFTP_RETURN_CODE_TYPE	The return codes output by AFTP
AFTP_RECORD_FORMAT_TYPE	The file record formats
AFTP_RECORD_LENGTH_TYPE	The file record length
AFTP_SECURITY_TYPE	The APPC security types
AFTP_TRACE_LEVEL_TYPE	The levels of tracing information output by AFTP
AFTP_VERSION_TYPE	The AFTP program version numbers
AFTP_WRITE_MODE_TYPE	The different ways that AFTP can write to a file

Conventions

Null-Terminated Strings

The AFTP API does not require input strings to be null-terminated. The AFTP API also does not guarantee that the output strings are null terminated. The return size does not include the null terminator in the size, if there is one.

The C programmer should be aware of the fact that strings are handled differently within AFTP than they are in the C standard library. All API calls receiving strings as input require both the string itself, and the length of the string. The `strlen()` function can be used for this. The null terminator must not be counted as part of the string length. API calls which output strings require three string related parameters:

- The string.
- The length of the string buffer that has been allocated by the calling program. Both of these are input to the API, with the string being modified.

- The actual length of the string that is output. AFTP output strings are not null terminated. In order for the C programmer to use them as standard C strings, a null character must be added to the end of the string.

AFTP_ENTRY

The AFTP API calls do not return a value. Rather, the return code parameter is set to indicate the success or failure of the call. The programmer should check the return code parameter after each call and handle error values appropriately.

The C keyword void is not used for entry points in the AFTP API. Instead, AFTP_ENTRY has been defined. AFTP_ENTRY is defined differently depending on the operating system the AFTP client will be created on.

AFTP_PTR

The C pointer indicator '*' is not used in the AFTP API. Instead, AFTP_PTR has been defined. AFTP_PTR is defined differently depending on the operating system the AFTP client will be created on.

Compiling the AFTP Application

The following sections describe compiling an AFTP application on MVS and VM.

MVS

To develop an application that uses the AFTP API on MVS, follow these steps:

Note: This process assumes all AFTP program files have been successfully installed with the provided JCL for installing AFTP.

1. Include this header file in your source modules: APPFFTP.
2. Define CM_MVS when you compile your source.
3. Edit the APPFAPIJ JCL file and make the changes indicated in the prolog comments at the top of the file.
4. Submit the APPFAPIJ JCL.

VM

To access the AFTP API calls from your application you must:

1. Include this header file in your source modules: APPFFTP H.
2. Define CM_VM when you compile your source.
3. Add APPFAPIL TXTLIB to your GLOBAL TXTLIB statement. All textdecks are packaged into this textlib.

Overview of API Calls

The calls of the AFTP API can be organized into the following categories:

- Create or destroy an AFTP connection object
- Establish a connection to the AFTP server
- Query connection characteristics
- Transfer files
- Specify file transfer characteristics
- Query file transfer characteristics
- List files on the AFTP server
- List files on the AFTP client
- Perform directory manipulation
- Perform file manipulation
- Query system information

- Generate message strings
- Control trace information
- Miscellaneous

Create or Destroy an AFTP Connection Object

The connection object represents an object-oriented approach to AFTP. An AFTP connection object represents a connection (not necessarily active) to a partner computer. Many of the other AFTP API calls require an AFTP connection object as input. When the program has finished using AFTP API calls, it should destroy the connection object.

aftp_create()

Creates an AFTP connection object and assigns a unique identifier to it. The connection object is accessed by its connection id. The connection object is never automatically destroyed. This allows you to connect to an AFTP server once, or reconnect numerous times using the same AFTP connection object.

aftp_destroy()

Destroys the AFTP connection object and recovers all resources associated with it. Once the AFTP connection object has been destroyed, that object must not be used again. If you need a connection object again, create another one.

Establish a Connection to the AFTP Server Computer

In order for the AFTP client to communicate with an AFTP server, certain communications parameters must be set. Most of the communications parameters have default values. The destination does not have a default value and must be set explicitly. Once the parameters are set as desired, the connection to the server can take place.

These are the API calls you use to manage your connection to the AFTP server:

aftp_close()

Closes a connection to the AFTP server once processing is complete.

aftp_connect()

Establishes the connection to the AFTP server for file transfer.

aftp_set_destination()

Identifies the server computer name to which the AFTP connection will be established. This server computer will run the AFTP server program.

aftp_set_mode_name()

Sets the mode name to be used for this connection. The default mode name is #BATCH.

aftp_set_password()

Sets the password used for APPC security type PROGRAM. Using this call automatically sets the security type to PROGRAM.

aftp_set_security_type()

Sets the APPC security used for the AFTP connection to the AFTP server.

aftp_set_tp_name()

Sets the transaction program name to be used for this connection. The default transaction program name is AFTPD.

aftp_set_userid()

Sets the user ID used for APPC security type PROGRAM. Using this call automatically sets the security type to PROGRAM.

Query Connection Characteristics

Use these API calls to query the characteristics of the connection to the AFTP server.

aftp_extract_destination()

Extracts the identity of the server computer on which the AFTP server runs.

aftp_extract_mode_name()

Extracts the mode name used for this connection.

aftp_extract_partner_LU_name()

Extracts the fully qualified LU name of the AFTP server computer.

aftp_extract_password()

Extracts the password used for this connection.

aftp_extract_security_type()

Extracts the security type used for this connection.

aftp_extract_tp_name()

Extracts the transaction program name used for this connection.

aftp_extract_userid()

Extracts the user ID used for this connection.

Transfer Files

The primary purpose of the file transfer protocol is to exchange files between the AFTP client and AFTP server programs. Through the API, the AFTP client program can send a file to the AFTP server, and receive a file from the AFTP server.

Use these API calls to transfer files between the client and server.

aftp_query_bytes_transferred()

Outputs the number of bytes transferred by either the `aftp_send_file()` or `aftp_receive_file()` calls.

aftp_receive_file()

Receives a single file from the AFTP server.

aftp_send_file()

Sends a single file to the AFTP server.

Specify File Transfer Characteristics

AFTP supports a variety of file transfer attributes. Both text and binary files can be transferred. The data structure of the files can be set by the programmer for mainframe applications. This allows several variations of record-based files to be transferred.

Use these API calls to specify file transfer characteristics.

aftp_set_allocation_size()

Sets the amount of space allocated for the file which is being written.

aftp_set_block_size()

Sets the size of a data block for the file which is being written.

aftp_set_data_type()

Sets the way in which the data transmitted is interpreted.

aftp_set_date_mode()

Sets how the date will be represented when the file is written (either received or sent). The new file can use the current date/time stamp or the date/time stamp of the original file.

aftp_set_record_format()

Sets the record format of the file transmitted.

aftp_set_record_length()

Sets the length of the file record transmitted.

aftp_set_write_mode()

Sets the type of write operation which will occur when the transmitted file is written (either received or sent).

Query File Transfer Characteristics

Use these API calls to query the file transfer attributes.

aftp_extract_allocation_size()

Extracts the amount of space allocated for the file which is being transmitted.

aftp_extract_block_size()

Extracts the size of a data block for the file which is being transmitted.

aftp_extract_data_type()

Extracts the way in which the transmitted data is interpreted.

aftp_extract_date_mode()

Extracts how the date will be represented when the file is written.

aftp_extract_record_format()

Extracts the record format of the file transmitted.

aftp_extract_record_length()

Extracts the length of the file record transmitted.

aftp_extract_write_mode()

Extracts the type of write operation which will occur when the transmitted file is written.

List Files on the AFTP Server Computer

File list facilities can be used to support wildcard transfers from the AFTP server. The wildcard processing is kept off the send and receive calls to make the calls as portable as possible. Obtaining a complete directory listing requires three calls: open, read, and close.

aftp_dir_close()

Closes an active directory listing on the AFTP server.

aftp_dir_open()

Begins a directory listing operation on the AFTP server. The directory open call sets up the search specifications:

- File specification which is to be matched
- Whether directories, files, or both should be included in the search
- The type of information desired (file names only or file names with attributes)

aftp_dir_read()

Gets the next file from the directory listing on the AFTP server. A text string describing the file will be returned. The format of the information returned depends on the parameters specified on the `aftp_dir_open()` call.

List Files on the AFTP Client Computer

File list facilities can be used to support wildcard transfers to the AFTP server. The wildcard processing is kept off the send and receive primitives to make the primitives as portable as possible. Obtaining a complete directory listing requires three calls: open, read, and close.

aftp_local_dir_close()

Closes an active directory listing on the AFTP client.

aftp_local_dir_open()

Begins a directory listing operation on the AFTP client. The directory open call sets up the search specifications:

- File specification to be matched
- Whether directories, files, or both should be included in the search
- The type of information desired (file names only or file names with attributes)

aftp_local_dir_read()

Gets the next file from the directory listing on the AFTP client. A text string describing the file will be returned. The format of the information returned depends upon the parameters specified on the `aftp_local_dir_open()` call.

Perform Directory Manipulation

AFTP provides methods of traversing and modifying the directory structure on the AFTP server computer. It is possible to build recursive copy routines for entire directory trees using these calls. AFTP also maintains the current directory on the AFTP server. This provides the user with a method of specifying a file name without specifying the entire directory path to that file on the AFTP server.

Use these API calls to manage directories on the client and server.

aftp_change_dir()

Changes the current working directory on the AFTP server.

aftp_create_dir()

Makes a new directory on the AFTP server.

aftp_query_current_dir()

Outputs the current working directory on the AFTP server.

aftp_remove_dir()

Removes an existing directory on the AFTP server.

AFTP provides methods to query and traverse the directory structure on the AFTP client computer. AFTP maintains the current directory on the AFTP client. This provides the user with a method of specifying a file name without specifying the entire directory path to that file on the AFTP client.

aftp_local_change_dir()

Changes the current working directory on the AFTP client.

aftp_local_query_current_dir()

Outputs the current working directory on the AFTP client.

Perform File Manipulation

The following two calls provide additional file functions which allow modifications to files on the AFTP server without using the `aftp_send_file()` call. It is possible to rename a file on the AFTP server computer as long as the rename does not cross device boundaries. It is also possible to delete files on the AFTP server computer.

aftp_delete()

Deletes a file on the AFTP server.

aftp_rename()

Renames a file on the AFTP server.

Query System Information

The query system calls can be used to learn more information about the AFTP client and AFTP server computers.

aftp_query_local_system_info()

Outputs a string describing the AFTP client operating system.

aftp_query_local_version()

Outputs the major and minor AFTP client version numbers.

aftp_query_system_info()

Outputs a string describing the AFTP server operating system.

Generate Message Strings

AFTP allows the caller to use consistent strings for AFTP transfer characteristics. AFTP will output the string to use when queried. It is also possible to output standard text messages for AFTP errors. The other API calls return an AFTP return code which can be queried to determine if an error message should be output.

Use these API calls to get text strings to use in messages issued by your application.

aftp_format_error()

Generates text output for the current AFTP error. This should be used to output error information to the user when an AFTP call returns a bad return code.

aftp_get_data_type_string()

Outputs the string corresponding to an input data type value.

aftp_get_date_mode_string()

Outputs the string corresponding to an input date mode value.

aftp_get_record_format_string()

Outputs the string corresponding to an input record format value.

aftp_get_write_mode_string()

Outputs the string corresponding to an input write mode value.

Control Trace Information

Use these API calls to control tracing of AFTP activity.

aftp_extract_trace_level()

Extracts the current trace level.

aftp_set_trace_filename()

Sets the name of the file to be used for trace output.

aftp_set_trace_level()

Sets the amount of trace data to be captured.

Miscellaneous

AFTP provides a method of loading the AFTP initialization file which contains user permission and file mapping information.

aftp_load_ini_file()

Loads the AFTP initialization file into memory.

Chapter 2. AFTP API Call Reference

This chapter provides an alphabetical reference for all of the API calls for the APPC File Transfer Protocol (AFTP). Program examples are provided for each call to illustrate its use in a program.

aftp_change_dir

Use this call to change the current working directory on the AFTP server. A connection to the AFTP server must be established before using this call.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_change_dir(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    directory,
    AFTP_LENGTH_TYPE          length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

directory

(input) The new current working directory name. The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The directory specified can be either an absolute or a relative path name.

length

(input) The length of the directory parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    /* The value used will vary based on platform:
    *   VM common naming:      directory = "/d"
    *   VM native naming:     directory = "/d"
    *   MVS PDS common naming: directory = "/user.clist/"
    *   MVS PDS native naming: directory = "'user.clist'"
    *   MVS data set prefix common: directory = "/user.qual.a."
    *   MVS data set prefix native: directory = "'user.qual.a.'"
    *   OS/2* common naming:  directory = "/c:/os2"
    *   OS/2 native naming:   directory = "c:\\os2"
    */
    static unsigned char AFTP_PTR directory = "/user.clist/"; /* MVS */

    /*
    * Before issuing the example call, you must have:
    *   a connection_id, use:      aftp_create()
    *   a connection to server, use: aftp_connect()
    */

    /*
    * Specify the new current working directory name
    * using the COMMON name format.
    */
}
```

```
    aftp_change_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)strlen(directory),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error changing AFTP directory.\n");
    }
}
```

Line Flows

The directory name is sent to the AFTP server, and the call waits for a response indicating the success or failure of the change directory operation.

aftp_close

Use this call to close an active connection. A connection to the AFTP server must be established before using this call.

Format

```
AFTP_ENTRY
aftp_close(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP object connection originally created with `aftp_create()`.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use:      aftp_create()
     *   a connection to server, use: aftp_connect()
     */

    aftp_close(connection_id, &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(
            stderr,
            "Error on aftp_close(): %s\n",
            aftp_rc);
    }
}
```

Line Flows

The close operation forces a Deallocate(FLUSH) to flow to the server.

aftp_connect

Use this call to establish a connection to the AFTP server. You must identify the destination of the AFTP server with the `aftp_set_destination()` call prior to issuing this call.

Format

```
AFTP_ENTRY
aftp_connect(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use:  aftp_create()
     *   a destination, use:   aftp_set_destination()
     */

    /*
     * Establish a connection to the server
     */

    aftp_connect(connection_id, &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(
            stderr,
            "Error on aftp_connect(): %s\n",
            aftp_rc);
    }
}
```

Line Flows

- The `aftp_connect()` call will cause an Allocate to flow to the server.
- Once a conversation is established, an exchange of version numbers and capabilities occurs between the client and the server. Therefore, this call does not return until AFTP verifies that the server program is running correctly on the remote system or an error occurs.

aftp_create

Use this call to create an AFTP connection object which can be used to connect to an AFTP server.

Format

```
AFTP_ENTRY
aftp_create(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(output) Handle of the AFTP connection object that was created by this call. All subsequent AFTP calls must use a previously created AFTP connection object.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_HANDLE_TYPE      connection_id;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Create the connection object that we will use for AFTP.
     */

    aftp_create(connection_id, &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error creating an AFTP object.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_create_dir

Use this call to create a new directory on the AFTP server. A connection to the AFTP server must be established before using this call.

Platform Differences:

1. On VM, this call is not supported. If issued, the call fails with return code AFTP_RC_FAIL_NO_RETRY.
2. On MVS, partitioned data sets act as the directory structure. This call creates a partitioned data set with the name specified.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_create_dir(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    directory,
    AFTP_LENGTH_TYPE          length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR    return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

directory

(input) The name of the directory name to be created. The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The directory specified can be either an absolute or relative path name.

length

(input) The length of the directory parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /* The value used for filespec will vary based on platform:
     * VM not supported
     * MVS PDS common naming: directory="/user.clist/"
     * MVS PDS native naming: directory="'user.clist'"
     * OS/2 native naming:    directory="d:\newdir"
     * OS/2 common naming:    directory="/d:/newdir"
     */
    static unsigned char AFTP_PTR directory = "/user.clist/";

    /*
     * Before issuing the example call, you must have:
     * a connection_id, use:      aftp_create()
     * a connection to server, use: aftp_connect()
     */
}
```

```
*/  
  
aftp_create_dir(  
    connection_id,  
    directory,  
    (AFTP_LENGTH_TYPE)strlen(directory),  
    &aftp_rc);  
  
if (aftp_rc != AFTP_RC_OK) {  
    fprintf(stderr, "Error creating directory.\n");  
}  
}
```

Line Flows

The directory name is sent to the AFTP server and the call waits for a response indicating the success or failure of the create directory operation.

aftp_delete

Use this call to delete a single file on the AFTP server. A connection to the AFTP server must be established before using this call.

Format

```
AFTP_ENTRY
aftp_delete(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    filename,
    AFTP_LENGTH_TYPE          length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

filename

(input) The name of the file to be removed. The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can contain either an absolute or relative path name.

length

(input) The length of the filename parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /* The value used for filespec will vary based on platform:
    *   VM common naming:      filespec="/a/foo*"
    *   VM native naming:     filespec="foo*.*.a"
    *   MVS PDS common naming: filespec="/user.clist/foo*"
    *   MVS PDS native naming: filespec="'user.clist(foo*)'"
    *   MVS sequential common: filespec="/user.qual*.a*.*"
    *   MVS sequential native: filespec="'user.qual*.a*.*'"
    */
    static unsigned char AFTP_PTR filespec = "/user.clist/foo*";

    /*
    * Before issuing the example call, you must have:
    *   a connection_id, use:      aftp_create()
    *   a connection to server, use: aftp_connect()
    */

    /*
    * Delete a file
    */

    aftp_delete(
        connection_id,
        filespec,
        (AFTP_LENGTH_TYPE)strlen(filespec),
```

```
        &aftp_rc);  
  
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error deleting AFTP file.\n");  
    }  
  
}
```

Line Flows

The file name is sent to the AFTP server and the call waits for a response indicating the success or failure of the delete file operation.

aftp_destroy

Use this call to destroy an AFTP connection object. Once an AFTP connection object is destroyed, it cannot be used again.

You should issue the `aftp_close()` call to end the connection before you issue this call.

Format

```
AFTP_ENTRY
aftp_destroy(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object to be destroyed. This object was originally created with `aftp_create()`.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_HANDLE_TYPE      connection_id;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     *
     * If you have opened a connection with aftp_connect()
     * you must also issue an aftp_close()
     */

    aftp_destroy(connection_id, &aftp_rc);
}
```

Line Flows

There are no line flows for this call.

aftp_dir_close

Use this call to cancel a directory listing on the AFTP server that is in progress or end a directory listing on the AFTP server after NO_MORE_ENTRIES has been returned from an aftp_dir_read() call. A connection to the AFTP server must be established before using this call. A directory listing on the AFTP server must be started by calling aftp_dir_open() prior to calling this call.

Format

```
AFTP_ENTRY
aftp_dir_close(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

See “aftp_dir_read” on page 27 for a complete example showing the related calls: aftp_dir_open, aftp_dir_read, and aftp_dir_close.

Line Flows

There are no line flows for this call.

aftp_dir_open

Use this call to begin a directory listing and specify the file search parameters on the AFTP server. The `aftp_dir_read()` call is used to read individual directory entries. The `aftp_dir_close()` call is used to end the directory listing. A connection to the AFTP server must be established before using this call.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_dir_open(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    filespec,
    AFTP_LENGTH_TYPE          length,
    AFTP_FILE_TYPE_TYPE       file_type,
    AFTP_INFO_LEVEL_TYPE      info_level,
    unsigned char AFTP_PTR    path,
    AFTP_LENGTH_TYPE          path_buffer_length,
    AFTP_LENGTH_TYPE AFTP_PTR path_returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

filespec

(input) The search string which the server uses to generate the directory listing. The files in the listing must match the search string. The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can either be an absolute or relative path name and can contain wildcard characters.

length

(input) The length of the `filespec` parameter in bytes.

file_type

(input) The type of information (directory names or file names) to be returned.

AFTP_FILE

Only file entries

AFTP_DIRECTORY

Only directory entries

AFTP_ALL_FILES

Both file and directory entries

info_level

(input) The level and format of information to be returned about each file or directory entry.

AFTP_NATIVE_NAMES

Native names without attributes.

AFTP_NATIVE_ATTRIBUTES

Native names and native file attributes.

path

(output) The fully qualified directory name in which all of the directory entries exist. The actual directory entries will be returned when the `aftp_dir_read()` call

is used. The path can be used along with the returned directory entry filename to create a fully qualified pathname to use on another AFTP file call.

Use the `AFTP_FILE_NAME_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

path_buffer_length

(input) The size in bytes of the buffer pointed to by the path parameter.

path_returned_length

(output) The number of bytes returned in the path parameter.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

See “`aftp_dir_read`” on page 27 for a complete example showing the related calls: `aftp_dir_open`, `aftp_dir_read`, and `aftp_dir_close`.

Line Flows

Sends a request for a directory listing to the AFTP server and waits for a response which includes the fully specified path of the directory listing or an error indicator. If the path of the directory listing is received, the AFTP server sends all of the directory entries as well. Once the list is complete, a special end-of-list indicator is sent to the AFTP client.

aftp_dir_read

Use this call to get an individual directory entry, based upon the search specified on the `aftp_dir_open()` call. A connection to the AFTP server must be established before using this call. The `aftp_dir_open()` call must be called prior to listing the directory entries.

Format

```
AFTP_ENTRY
aftp_dir_read(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    dir_entry,
    AFTP_LENGTH_TYPE          dir_entry_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_BOOLEAN_TYPE AFTP_PTR no_more_entries,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code),
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

dir_entry

(input) Pointer to a buffer into which the procedure will write the directory entry.

Use the `AFTP_FILE_NAME_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

dir_entry_size

(input) The size in bytes of the `dir_entry` buffer.

returned_length

(output) The number of bytes returned in the `dir_entry` parameter.

no_more_entries

(output) Whether or not an entry was returned on this call.

A value of zero indicates that there are more directory entries and that an entry was returned on this call.

A nonzero value indicates that there are no more directory entries and no entry was returned on this call. The returned length parameter will be set to zero. Subsequent calls to `aftp_dir_read()` will also result in `no_more_entries` being nonzero. To end the directory listing, your next call should be `aftp_dir_close()`.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

This example shows how to use the `aftp_dir_open`, `aftp_dir_read`, and `aftp_dir_close` calls together.

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             dir_entry[AFTP_FILE_NAME_SIZE +1];
    AFTP_LENGTH_TYPE          dir_entry_length;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:      filespec="/a/foo*"
     */
```

```

* VM native naming:      filespec="foo*.*.a"
* MVS PDS common naming: filespec="/user.clist/foo*"
* MVS PDS native naming: filespec="'user.clist(foo*)'"
* MVS sequential common: filespec="/user.qual*.a*.*"
* MVS sequential native: filespec="'user.qual*.a*.*'"
*/
static unsigned char AFTP_PTR filespec = "/user.clist/foo*";

unsigned char          path[AFTP_FILE_NAME_SIZE+1];
AFTP_LENGTH_TYPE      path_length;
AFTP_BOOLEAN_TYPE     no_more_entries;

/*
* Before issuing the example call, you must have:
*   a connection_id, use:      aftp_create()
*   a connection to server, use: aftp_connect()
*/

/*
* Open a new directory listing on the AFTP server. Both files and
* directory names will be listed along with their attributes.
*/

aftp_dir_open(
    connection_id,
    filespec,
    (AFTP_LENGTH_TYPE)strlen(filespec),
    AFTP_DIRECTORY | AFTP_FILE,
    AFTP_NATIVE_ATTRIBUTES,
    path,
    (AFTP_LENGTH_TYPE)sizeof(path)-1,
    &path_length,
    &aftp_rc);

if (aftp_rc == AFTP_RC_OK) {
    path[path_length] = '\0';

    printf("Directory listing of %s.", path);

    do {
        /*
        * Read one directory entry from the AFTP server
        */

        aftp_dir_read(
            connection_id,
            dir_entry,
            (AFTP_LENGTH_TYPE)sizeof(dir_entry)-1,
            &dir_entry_length,
            &no_more_entries,
            &aftp_rc);

        if (aftp_rc == AFTP_RC_OK && no_more_entries == 0) {
            dir_entry[dir_entry_length] = '\0';
            printf("File: %s\n", dir_entry);
        }
        /*
        * Loop until we either run out of directory
        * entries or an error occurs.
        */
    } while (aftp_rc == AFTP_RC_OK && no_more_entries == 0);

    /*
    * Terminate the directory listing by executing
    * a close.
    */
}

```

```
    aftp_dir_close(connection_id, &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(
            stderr,
            "Error closing AFTP directory.\n");
    }
}
else {
    fprintf(stderr, "Error opening AFTP directory.\n");
}
}
```

Line Flows

There are no line flows for this call.

aftp_extract_allocation_size

Use this call to extract the AFTP file allocation size. If the `aftp_set_allocation_size()` call has not been invoked, the AFTP default allocation size value is returned.

Format

```
AFTP_ENTRY
aftp_extract_allocation_size(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_ALLOCATION_SIZE_TYPE AFTP_PTR allocation_size,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

allocation_size

(output) The allocation size in bytes that had been set for the AFTP file transfer operation.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    AFTP_ALLOCATION_SIZE_TYPE allocation_size;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    aftp_extract_allocation_size(
        connection_id,
        &allocation_size,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP allocation size.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_block_size

Use this call to extract the file block size. If the `aftp_set_block_size()` call has not been invoked, the AFTP default block size value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_block_size(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_BLOCK_SIZE_TYPE AFTP_PTR  block_size,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

block_size

(output) The AFTP file block size in bytes.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_BLOCK_SIZE_TYPE block_size;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    aftp_extract_block_size(
        connection_id,
        &block_size,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP block size.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_data_type

Use this call to extract the data type for file transfers. If the aftp_set_data_type() call has not been invoked, the AFTP default data type value is returned.

Format

```
AFTP_ENTRY
aftp_extract_data_type(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_DATA_TYPE_TYPE AFTP_PTR  data_type,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

data_type

(output) The data type to be used for data transfers.

AFTP_ASCII

Transfer files as text files in ASCII.

AFTP_BINARY

Transfer files as a binary sequence of bytes without translation.

AFTP_DEFAULT_DATA_TYPE

Use the data transfer type set in the INI file. If no type is set in the INI file, use AFTP_ASCII.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_DATA_TYPE_TYPE   data_type;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    aftp_extract_data_type(
        connection_id,
        &data_type,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP data type.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_date_mode

Use this call to extract the way file dates are handled for data transfer. If the `aftp_set_date_mode()` call has not been invoked, the AFTP default date mode value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_date_mode(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_DATE_MODE_TYPE AFTP_PTR date_mode,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

date_mode

(output) The type of date given to the new file after transfer.

AFTP_NEWDATE

Assign the time/date stamp of the time of transfer.

AFTP_OLDDATE

Assign the time/date stamp of the source file.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE     aftp_rc;
    AFTP_DATE_MODE_TYPE       date_mode;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    aftp_extract_date_mode(
        connection_id,
        &date_mode;
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP date mode.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_destination

Use this call to extract the destination of the AFTP server. If the `aftp_set_destination()` call has not been invoked, the AFTP default destination value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_destination(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    destination,
    AFTP_LENGTH_TYPE          destination_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

destination

(output) Buffer into which the name of the AFTP server is written. This parameter can be either a symbolic destination name or a partner LU name.

See the *z/OS Communications Server: APPC Application Suite User's Guide* for information about specifying destinations in the APPC Application Suite.

destination_size

(input) The size of the buffer in which the destination will be stored.

Use the `AFTP_FQLU_NAME_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

returned_length

(output) The actual length of the destination parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             destname[AFTP_FQLU_NAME_SIZE+1];
    AFTP_LENGTH_TYPE          returned_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the destination name for AFTP.
     */

    aftp_extract_destination(
        connection_id,
        destname,
        (AFTP_LENGTH_TYPE)sizeof(destname)-1,
        &returned_length;
```

```
    &aftp_rc);  
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error extracting AFTP destination.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_extract_mode_name

Use this call to extract the mode name specified for the connection to the AFTP server. If the aftp_set_mode_name() call has not been invoked, the AFTP default mode name value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_mode_name(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    mode_name,
    AFTP_LENGTH_TYPE          mode_name_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

mode_name

(output) The buffer in which the mode name is to be stored.

Use the AFTP_MODE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

mode_name_size

(input) The size of buffer in which the mode name will be stored.

returned_length

(output) The actual length of the mode_name parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char AFTP_PTR    mode_name[AFTP_MODE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE          returned_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the APPC mode name for AFTP.
     */

    aftp_extract_mode_name(
        connection_id,
        mode_name,
        (AFTP_LENGTH_TYPE)sizeof(mode_name)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
```

```
        fprintf(stderr, "Error extracting AFTP mode name.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_extract_partner_LU_name

Use this call to extract the fully qualified LU name of the server. A connection to the AFTP server must occur before this call can be invoked.

Format

```
AFTP_ENTRY
aftp_extract_partner_LU_name(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    partner_LU_name,
    AFTP_LENGTH_TYPE          partner_LU_name_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

partner_LU_name

(output) Buffer to which the fully qualified LU name is written.

Use the AFTP_FQLU_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

partner_LU_name_size

(input) The size of the buffer that the partner_LU_name will be written to.

returned_length

(output) The actual length of the partner_LU_name parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             partner[AFTP_FQLU_NAME_SIZE+1];
    AFTP_LENGTH_TYPE          returned_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use:      aftp_create()
     *   a connection to server, use: aftp_connect()
     */

    /*
     * Extract the partner LU name for AFTP.
     */

    aftp_extract_partner_LU_name(
        connection_id,
        partner,
        (AFTP_LENGTH_TYPE)sizeof(partner)-1,
        &returned_length,;
        &aftp_rc);
```

```
if (aftp_rc != AFTP_RC_OK) {  
    fprintf(stderr, "Error extracting AFTP destination.\n");  
}  
}
```

Line Flows

There are no line flows for this call.

aftp_extract_password

Use this call to extract the password specified for the connection to the AFTP server. If the `aftp_set_password()` call has not been invoked, the AFTP default password value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_password(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    password,
    AFTP_LENGTH_TYPE          password_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

password

(output) The buffer in which the password used on the connection is written.

Use the `AFTP_PASSWORD_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

password_size

(input) The size of the buffer in which the password is to be written.

returned_length

(output) The actual length of the password parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             password[AFTP_PASSWORD_SIZE+1];
    AFTP_LENGTH_TYPE         returned_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the password for AFTP.
     */

    aftp_extract_password(
        connection_id,
        password,
        (AFTP_LENGTH_TYPE)sizeof(password)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
```

```
        fprintf(stderr, "Error extracting AFTP password.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_extract_record_format

Use this call to extract the record format for the data transfer. If the `aftp_set_record_format()` call has not been invoked, the AFTP default record format value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_record_format(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_RECORD_FORMAT_TYPE AFTP_PTR record_format,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

record_format

(output) The record format used for file transfer.

AFTP_DEFAULT_RECORD_FORMAT

Specifies that the system on which the file will be written should use its own default setting for record format. This is the initial setting.

AFTP_V

Variable length record, unblocked.

AFTP_VA

Variable length record, unblocked, ASA print-control characters.

AFTP_VB

Variable length record, blocked.

AFTP_VM

Variable length record, unblocked, machine print-control codes.

AFTP_VS

Variable length record, unblocked, spanned.

AFTP_VBA

Variable length record, blocked, ASA print-control characters.

AFTP_VBM

Variable length record, blocked, machine print-control codes.

AFTP_VBS

Variable length record, blocked, spanned.

AFTP_VSA

Variable length record, unblocked, spanned, ASA print-control characters.

AFTP_VSM

Variable length record, unblocked, spanned, machine print-control codes.

AFTP_VBSA

Variable length record, blocked, spanned, ASA print-control characters.

AFTP_VBSM

Variable length record, blocked, spanned, machine print-control codes.

AFTP_F

Fixed length record, unblocked.

AFTP_FA

Fixed length record, unblocked, ASA print-control characters.

AFTP_FB

Fixed length record, blocked.

AFTP_FM
Fixed length record, unblocked, machine print-control codes.

AFTP_FBA
Fixed length record, blocked, ASA print-control characters.

AFTP_FBM
Fixed length record, blocked, machine print-control codes.

AFTP_FBS
Fixed length record, blocked, standard.

AFTP_FBSM
Fixed length record, blocked, machine print-control codes, standard.

AFTP_FBSA
Fixed length record, blocked, ASA print-control characters, standard.

AFTP_U
Undefined length record.

AFTP_UA
Undefined length record, ASA print control characters.

AFTP_UM
Undefined length record, machine print control codes.

return_code
(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_RECORD_FORMAT_TYPE record_format;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the file record format for AFTP.
     */

    aftp_extract_record_format(
        connection_id,
        &record_format,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP record format.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_record_length

Use this call to extract the record length for fixed length records, or the maximum possible record length for variable length records used for data transfer. If the `aftp_set_record_length()` call has not been invoked, the AFTP default record length value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_record_length(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_RECORD_LENGTH_TYPE AFTP_PTR record_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

record_length

(output) The record length for the data transfer specified in bytes.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    AFTP_RECORD_LENGTH_TYPE  record_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the file record length for AFTP.
     */

    aftp_extract_record_length(
        connection_id,
        &record_length,;
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP record length.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_security_type

Use this call to extract the type of APPC conversation security used. If the `aftp_set_security_type()` call has not been invoked, the AFTP default security type value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_security_type(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_SECURITY_TYPE AFTP_PTR security_type,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

security_type

(output) The security to be used when connecting to the AFTP server.

AFTP_SECURITY_NONE

No APPC conversation security is used.

AFTP_SECURITY_SAME

The local security information determined at login time will be transferred to the AFTP server.

AFTP_SECURITY_PROGRAM

A user ID and password will be sent to be verified by the AFTP server.

This security type requires the use of the `aftp_set_userid()` and `aftp_set_password()` calls, or the connection attempt will fail.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    AFTP_SECURITY_TYPE        sec_type;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the APPC security type for AFTP.
     */

    aftp_extract_security_type(
        connection_id,
        &sec_type,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr,
            "Error extracting AFTP security type.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_tp_name

Use this call to extract the transaction program (TP) name of the AFTP server. If the `aftp_set_tp_name()` call has not been invoked, the AFTP default transaction program name value is returned.

Format

```
AFTP_ENTRY
aftp_extract_tp_name(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    tp_name,
    AFTP_LENGTH_TYPE         tp_name_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

tp_name

(output) The buffer into which the transaction program name of the AFTP server will be written.

Use the `AFTP_TP_NAME_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

tp_name_size

(input) The size of the buffer to which the transaction program name will be written.

returned_length

(output) The actual length of the transaction program name parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            tp_name[AFTP_TP_NAME_SIZE+1];
    AFTP_LENGTH_TYPE         returned_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the TP name for AFTP.
     */

    aftp_extract_tp_name(
        connection_id,
        tp_name
        (AFTP_LENGTH_TYPE)sizeof(tp_name)-1,
        &returned_length,
        &aftp_rc);
}
```

```
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error extracting AFTP TP name.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_extract_trace_level

Use this call to extract the current trace level setting.

Format

```
AFTP_ENTRY
aftp_extract_trace_level(
    AFTP_TRACE_LEVEL_TYPE AFTP_PTR    trace_level,
    AFTP_RETURN_CODE_TYPE AFTP_PTR    return_code);
```

Parameters

trace_level

(output) The current setting of the trace level in AFTP. The constants from AFTP_LVL_NO_TRACING to AFTP_LVL_MAX_TRACE_LVL incrementally increase the amount of trace information.

AFTP_LVL_NO_TRACING

No data will be written to the trace log.

AFTP_LVL_API

Traces crossings of the API boundary.

AFTP_LVL_MAX_TRACE_LVL

Provides the maximum amount of trace information.

Other trace levels are reserved for diagnosing problems with the assistance of the IBM* Support Center.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE    rc;
    AFTP_TRACE_LEVEL_TYPE    trace_level;

    /*
     * There are no prerequisite calls for this call.
     */

    aftp_extract_trace_level(&trace_level, &rc);

    if (rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting trace level\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_extract_userid

Use this call to extract the user ID specified for the connection to the AFTP server. If the CPI-C extract userid function is not supported the AFTP default user ID value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_userid(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    userid,
    AFTP_LENGTH_TYPE          userid_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

userid

(output) The buffer into which the user ID used on the connection will be written.

Use the AFTP_USERID_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

userid_size

(input) The size of the buffer into which the user ID will be written.

returned_length

(output) The actual length of the userid parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             userid[AFTP_USERID_SIZE+1];
    AFTP_LENGTH_TYPE          returned_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the user ID for AFTP.
     */

    aftp_extract_userid(
        connection_id,
        userid,
        (AFTP_LENGTH_TYPE)sizeof(userid)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
```

```
        fprintf(stderr, "Error extracting userid.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_extract_write_mode

Use this call to extract the way that existing files are treated when a data transfer copies to them. If the `aftp_set_write_mode()` call has not been invoked, the AFTP default write mode value will be returned.

Format

```
AFTP_ENTRY
aftp_extract_write_mode(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_WRITE_MODE_TYPE AFTP_PTR  write_mode,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

write_mode

(output) The method used to write a file if a copy of the file already exists. If the file does not exist on the target, the new file is created.

AFTP_REPLACE

Transferred file will replace the existing file.

AFTP_APPEND

Transferred file will be appended to the existing file.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_WRITE_MODE_TYPE write_mode;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Extract the file write mode for AFTP.
     */

    aftp_extract_write_mode(
        connection_id,
        &write_mode,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP write mode.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_format_error

Use this call to retrieve the current AFTP error information to a text buffer. The AFTP return code for the current error must not be AFTP_RC_HANDLE_NOT_VALID. If the current status is AFTP_RC_OK, and the aftp_format_error() call is invoked, the return_code value output by this call will be AFTP_RC_STATE_CHECK. The aftp_format_error() call should only be invoked when an error has occurred.

Format

```
AFTP_ENTRY
aftp_format_error(
    AFTP_HANDLE_TYPE           connection_id,
    AFTP_DETAIL_LEVEL_TYPE     detail_level,
    unsigned char AFTP_PTR     error_str,
    AFTP_LENGTH_TYPE           error_str_size,
    AFTP_LENGTH_TYPE AFTP_PTR  returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

detail_level

(input) The detail in which the error string will describe the AFTP error. These values can be OR'ed together to retrieve specific sets of information. For example, if the primary message and the error log information should be returned, specify (AFTP_DETAIL_RC | AFTP_DETAIL_LOG).

AFTP_DETAIL_RC

The AFTP return code, error category, index and primary error message will be output.

AFTP_DETAIL_SECOND

The AFTP secondary error message will be output.

AFTP_DETAIL_LOG

The error logging information will be output.

AFTP_DETAIL_INFO

The informational message associated with the error will be output.

AFTP_DETAIL_ALL

All of the previous detail levels will be output in the error string.

error_str

(output) The buffer into which the error information string will be written.

Use the AFTP_MESSAGE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

error_str_size

(input) The size of the buffer in which the error information will be written.

returned_length

(output) The actual length of the error_str parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
  AFTP_HANDLE_TYPE      connection_id;
  AFTP_RETURN_CODE_TYPE aftp_rc;
  unsigned char         error_string[AFTP_MESSAGE_SIZE+1];
  AFTP_LENGTH_TYPE      returned_length;

  /*
   * There are no specific prerequisite calls for this call,
   * but you must issue a call that returns an error return code
   */

  if (aftp_rc != AFTP_RC_OK) {
    /*
     * We had an AFTP error - so let's get
     * the description that corresponds to
     * the error.
     */

    aftp_format_error(
      connection_id,
      AFTP_DETAIL_ALL,
      error_string,
      (AFTP_LENGTH_TYPE)sizeof(error_string)-1,
      &returned_length,
      &aftp_rc);
  }
}
```

Line Flows

There are no line flows for this call.

aftp_get_data_type_string

Use this call to get a string which corresponds to the input AFTP data type value. This string is available to allow all users of the AFTP API to have consistent strings for each data type. It is not necessary to create an AFTP connection object prior to the invocation of this call.

Format

```
AFTP_ENTRY
aftp_get_data_type_string(
    AFTP_DATA_TYPE_TYPE           data_type,
    unsigned char AFTP_PTR        data_type_string,
    AFTP_LENGTH_TYPE             data_type_size,
    AFTP_LENGTH_TYPE AFTP_PTR    returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

data_type

(input) An AFTP data type value.

AFTP_ASCII

Transfer files as text files in ASCII.

AFTP_BINARY

Transfer files as a binary sequence of bytes without translation.

AFTP_DEFAULT_DATA_TYPE

Use the data transfer type set in the INI file. If no type is set in the INI file, use AFTP_ASCII.

data_type_string

(output) The buffer into which the data type string will be written.

Use the AFTP_DATA_TYPE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

data_type_size

(input) The size of the buffer into which the data type string will be written.

returned_length

(output) The actual length of the data_type_string parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            data_type[AFTP_DATA_TYPE_SIZE+1];
    AFTP_LENGTH_TYPE        returned_length;

    /*
     * There are no prerequisite calls for this call.
     */
    /*
     * Get the data type string.
     */

    aftp_get_data_type_string(
        AFTP_ASCII,
```

```
        data_type,  
        (AFTP_LENGTH_TYPE)sizeof(data_type)-1,  
        &returned_length,  
        &aftp_rc);  
    }
```

Line Flows

There are no line flows for this call.

aftp_get_date_mode_string

Use this call to get a string which corresponds to the input AFTP date mode value. This string is available to allow all users of the AFTP API to have consistent strings for each date mode type. It is not necessary to create an AFTP connection object prior to the invocation of this call.

Format

```
AFTP_ENTRY
aftp_get_date_mode_string(
    AFTP_DATE_MODE_TYPE          date_mode,
    unsigned char AFTP_PTR       date_mode_string,
    AFTP_LENGTH_TYPE             date_mode_size,
    AFTP_LENGTH_TYPE AFTP_PTR    returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

date_mode

(input) An AFTP date mode value.

AFTP_NEWDATE

Assign the time/date stamp of the time of transfer.

AFTP_OLDDATE

Assign the time/date stamp of the source file.

date_mode_string

(output) The buffer into which the date mode string will be written.

Use the AFTP_DATE_MODE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

date_mode_size

(input) The size of the buffer into which the date mode string will be written.

returned_length

(output) The actual length of the date_mode_string parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            date_mode[AFTP_DATE_MODE_SIZE+1];
    AFTP_LENGTH_TYPE         returned_length;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Get the date mode string.
     */

    aftp_get_date_mode_string(
        AFTP_OLDDATE,
        date_mode,
```

```
        (AFTP_LENGTH_TYPE)sizeof(date_mode)-1,  
        &returned_length,  
        &aftp_rc);  
    }
```

Line Flows

There are no line flows for this call.

aftp_get_record_format_string

Use this call to get a string which corresponds to the input AFTP record format value. This string is available to allow all users of the AFTP API to have consistent strings for each record format type. It is not necessary to create an AFTP connection object prior to issuing this call.

Format

```
AFTP_ENTRY
aftp_get_record_format_string(
    AFTP_RECORD_FORMAT_TYPE          record_format,
    unsigned char AFTP_PTR            record_format_string,
    AFTP_LENGTH_TYPE                 record_format_size,
    AFTP_LENGTH_TYPE AFTP_PTR        returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR   return_code);
```

Parameters

record_format

(input) An AFTP record format value.

AFTP_DEFAULT_RECORD_FORMAT

Specifies that the system on which the file will be written should use its own default setting for record format. This is the initial setting.

AFTP_V

Variable length record, unblocked.

AFTP_VA

Variable length record, unblocked, ASA print-control characters.

AFTP_VB

Variable length record, blocked.

AFTP_VM

Variable length record, unblocked, machine print-control codes.

AFTP_VS

Variable length record, unblocked, spanned.

AFTP_VBA

Variable length record, blocked, ASA print-control characters.

AFTP_VBM

Variable length record, blocked, machine print-control codes.

AFTP_VBS

Variable length record, blocked, spanned.

AFTP_VSA

Variable length record, unblocked, spanned, ASA print-control characters.

AFTP_VSM

Variable length record, unblocked, spanned, machine print-control codes.

AFTP_VBSA

Variable length record, blocked, spanned, ASA print-control characters.

AFTP_VBSM

Variable length record, blocked, spanned, machine print-control codes.

AFTP_F

Fixed length record, unblocked.

AFTP_FA

Fixed length record, unblocked, ASA print-control characters.

AFTP_FB

Fixed length record, blocked.

AFTP_FM

Fixed length record, unblocked, machine print-control codes.

AFTP_FBA

Fixed length record, blocked, ASA print-control characters.

AFTP_FBM

Fixed length record, blocked, machine print-control codes.

AFTP_FBS

Fixed length record, blocked, standard.

AFTP_FBSM

Fixed length record, blocked, machine print-control codes, standard.

AFTP_FBSA

Fixed length record, blocked, ASA print-control characters, standard.

AFTP_U

Undefined length record.

AFTP_UA

Undefined length record, ASA print control characters.

AFTP_UM

Undefined length record, machine print control codes.

record_format_string

(output) The buffer into which the record format string will be written.

Use the AFTP_RECORD_FORMAT_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

record_format_size

(input) The size of the buffer into which the record format string will be written.

returned_length

(output) The actual length of the data structure string parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
  AFTP_RETURN_CODE_TYPE      aftp_rc;
  unsigned char               recfm[AFTP_RECORD_FORMAT_SIZE+1];
  AFTP_LENGTH_TYPE           returned_length;

  /*
   * There are no prerequisite calls for this call.
   */

  /*
   * Get the record format string.
   */

  aftp_get_record_format_string(
    AFTP_F,
    recfm,
    (AFTP_LENGTH_TYPE)sizeof(recfm)-1,
    &returned_length,
    &aftp_rc);
}
```

Line Flows

There are no line flows for this call.

aftp_get_write_mode_string

Use this call to get a string which corresponds to the input AFTP write mode value. This string is available to allow all users of the AFTP API to have consistent strings for each write mode type. It is not necessary to create an AFTP connection object prior to the invocation of this call.

Format

```
AFTP_ENTRY
aftp_get_write_mode_string(
    AFTP_WRITE_MODE_TYPE          write_mode,
    unsigned char AFTP_PTR        write_mode_string,
    AFTP_LENGTH_TYPE              write_mode_size,
    AFTP_LENGTH_TYPE AFTP_PTR     returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

write_mode

(input) The method used to write a file if a copy of the file already exists. If the file does not exist on the target, the new file will be created.

AFTP_REPLACE

Transferred file will replace the existing file.

AFTP_APPEND

Transferred file will be appended to the existing file.

write_mode_string

(output) The buffer into which the write mode string will be written.

Use the AFTP_WRITE_MODE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

write_mode_size

(input) The size of the buffer into which the write mode string will be written.

returned_length

(output) The actual length of the write mode string parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE          aftp_rc;
    unsigned char                  write_mode[AFTP_WRITE_MODE_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Get the write mode string.
     */

    aftp_get_write_mode_string(
        AFTP_REPLACE,
        write_mode
```

```
        (AFTP_LENGTH_TYPE)sizeof(write_mode)-1,  
        &returned_length,  
        &aftp_rc);  
    }
```

Line Flows

There are no line flows for this call.

aftp_load_ini_file

Use this call to read the AFTP initialization file into memory. The AFTP and ACOPY programs both use the initialization file. This file includes information required to map filenames on the current platform. Once the AFTP initialization file is stored in memory, AFTP will automatically consult the data it contains before proceeding with any operations. It is not necessary to create an AFTP connection object prior to the invocation of this call.

The name of the initialization file varies by operating system:

- MVS: DD:APPFTPI
- VM: AFTP INI
- OS/2: AFTP.INI

Format

```
AFTP_ENTRY
aftp_load_ini_file(
    unsigned char AFTP_PTR          filename,
    AFTP_LENGTH_TYPE               filename_size,
    unsigned char AFTP_PTR          program_path,
    AFTP_LENGTH_TYPE               path_size,
    unsigned char AFTP_PTR          error_string,
    AFTP_LENGTH_TYPE               error_string_size,
    AFTP_LENGTH_TYPE AFTP_PTR      returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

filename

(input) The filename of the AFTP initialization file.

filename_size

(input) The length of the filename parameter in bytes.

program_path

(input) For OS/2, the fully qualified file specification of the program that is running. The path from the file specification will be used to locate the AFTP initialization file.

For MVS or VM where this information is not available, provide a 0-length string (not a null string) for this parameter.

path_size

(input) The length of the program_path parameter in bytes.

error_string

(output) The buffer into which any error messages will be written during loading of the initialization file.

Use the AFTP_INI_MESSAGE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

error_string_size

(input) The size of the buffer into which the error information will be written.

returned_length

(output) The actual size of the error information in bytes.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
  AFTP_RETURN_CODE_TYPE      aftp_rc;
  static unsigned char AFTP_PTR init_file_name = "DD:APPFTPI";
  static unsigned char AFTP_PTR program_name = ""
  unsigned char              error_string[AFTP_INI_MESSAGE_SIZE+1];
  AFTP_LENGTH_TYPE          returned_length;

  /*
   * There are no prerequisite calls for this call.
   */

  /*
   * Load the AFTP initialization file into memory.
   */
  aftp_load_ini_file(
    init_file_name,
    (AFTP_LENGTH_TYPE)strlen(init_file_name),
    program_name,
    (AFTP_LENGTH_TYPE)strlen(program_name),
    error_string,
    (AFTP_LENGTH_TYPE)sizeof(error_string),
    &returned_length,
    &aftp_rc);
  if (aftp_rc != AFTP_RC_OK {
    error_string[returned_length]='\0';
    printf(stderr, error_string);
  }
}
```

Line Flows

There are no line flows for this call.

aftp_local_change_dir

Use this call to change the current working directory on the AFTP client. A connection to the AFTP server is not required before using this call.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_local_change_dir(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    directory,
    AFTP_LENGTH_TYPE         length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

directory

(input) The new directory name. The format of this name can either be the native syntax on the AFTP client or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The directory specified can either be an absolute or relative path name.

length

(input) The length of the directory parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /* The value used will vary based on platform:
    *   VM common naming:      directory = "/d"
    *   VM native naming:     directory = "/d"
    *   MVS PDS common naming: directory = "/user.clist/"
    *   MVS PDS native naming: directory = "'user.clist'"
    *   MVS data set prefix common: directory = "/user.qual.a."
    *   MVS data set prefix native: directory = "'user.qual.a.'"
    *   OS/2 common naming:   directory = "/c:/os2"
    *   OS/2 native naming:   directory = "c:\\os2"
    */
    static unsigned char AFTP_PTR directory = "/user.clist/"; /* MVS */

    /*
    * Before issuing the example call, you must have:
    *   a connection_id, use: aftp_create()
    */

    /*
    * Specify the new current working directory name on the AFTP
    * client.
    */
}
```

```
    aftp_local_change_dir(  
        connection_id,  
        directory,  
        (AFTP_LENGTH_TYPE)strlen(directory),  
        &aftp_rc);  
  
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error changing AFTP directory.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_local_dir_close

Use this call to cancel a directory listing on the AFTP client that is in progress or end a directory listing on the AFTP client after `no_more_entries` has been returned from an `aftp_local_dir_read()` call. A connection to the AFTP server is not required before using this call. A directory listing on the AFTP client must be started by calling `aftp_local_dir_open()` prior to making this call.

Format

```
AFTP_ENTRY
aftp_local_dir_close(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

See “`aftp_local_dir_read`” on page 70 for a complete example showing the related calls: `aftp_local_dir_open`, `aftp_local_dir_read`, and `aftp_local_dir_close`.

Line Flows

There are no line flows for this call.

aftp_local_dir_open

Use this call to begin a directory listing and specify the file search parameters on the AFTP client. The `aftp_local_dir_read()` call is used to read individual directory entries. The `aftp_local_dir_close()` call is used to end the directory listing. A connection to the AFTP server is not required before using this call.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_local_dir_open(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    filespec,
    AFTP_LENGTH_TYPE          length,
    AFTP_FILE_TYPE_TYPE       file_type,
    AFTP_INFO_LEVEL_TYPE      info_level,
    unsigned char AFTP_PTR    path,
    AFTP_LENGTH_TYPE          path_buffer_length,
    AFTP_LENGTH_TYPE AFTP_PTR path_returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

filespec

(input) The search string which the client uses to generate the directory listing. The files in the listing must match the search string. The format of this name can either be the native syntax on the AFTP client or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can either be an absolute or relative path name and can contain wildcard characters.

length

(input) The length of the `filespec` parameter in bytes.

file_type

(input) The type of information (directory names or file names) to be returned.

AFTP_FILE

Only file entries

AFTP_DIRECTORY

Only directory entries

AFTP_ALL_FILES

Both file and directory entries

info_level

(input) The level and format of information to be returned about each file or directory entry.

AFTP_NATIVE_NAMES

Native names without attributes.

AFTP_NATIVE_ATTRIBUTES

Native names and native file attributes.

path

(output) The fully qualified directory name in which all of the directory entries exist. The actual directory entries will be returned when the

`aftp_local_dir_read()` call is used. The path can be used along with the returned directory entry filename to create a fully qualified path name to use on another AFTP file call.

Use the `AFTP_FILE_NAME_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

path_buffer_length

(input) The size in bytes of the buffer pointed to by the path parameter.

path_returned_length

(output) The number of bytes returned in the path parameter.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

See “`aftp_local_dir_read`” on page 70 for a complete example showing the related calls: `aftp_local_dir_open`, `aftp_local_dir_read`, and `aftp_local_dir_close`.

Line Flows

There are no line flows for this call.

aftp_local_dir_read

Use this call to get an individual directory entry from the AFTP client, based upon the search specified on the aftp_local_dir_open() call. A connection to the AFTP server is not required before using this call. The aftp_local_dir_open() call must be called prior to listing the directory entries.

Format

```
AFTP_ENTRY
aftp_local_dir_read(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    dir_entry,
    AFTP_LENGTH_TYPE          dir_entry_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_BOOLEAN_TYPE AFTP_PTR no_more_entries,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

dir_entry

(input) Pointer to a buffer into which the procedure will write the directory entry.

Use the AFTP_FILE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

dir_entry_size

(input) The size in bytes of the dir_entry buffer.

returned_length

(output) The number of bytes returned in the dir_entry parameter.

no_more_entries

(output) Whether or not an entry was returned on this call.

A value of zero indicates that there are more directory entries and that an entry was returned on this call.

A nonzero value indicates that there are no more directory entries and that no entry was returned on this call. The returned length parameter is set to zero. Subsequent calls to aftp_local_dir_read() will also result in no_more_entries being nonzero. To end the directory listing, your next call should be aftp_local_dir_close().

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             dir_entry[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE          dir_entry_length;

    /* The value used for filespec will vary based on platform:
     * VM common naming:      filespec="/a/foo*"
     * VM native naming:      filespec="foo*.*a"
    */
}
```

```

* MVS PDS common naming: filespec="/user.clist/foo*"
* MVS PDS native naming: filespec="'user.clist(foo*)'"
* MVS sequential common: filespec="/user.qual*.a*.**"
* MVS sequential native: filespec="'user.qual*.a*.**'"
*/
static unsigned char AFTP_PTR filespec = "/user.clist/foo*";

unsigned char          path[AFTP_FILE_NAME_SIZE+1];
AFTP_LENGTH_TYPE      path_length;
AFTP_BOOLEAN_TYPE     no_more_entries;

/*
 * Before issuing the example call, you must have:
 *   a connection_id, use: aftp_create()
 */

/*
 * Open a new directory listing on the AFTP client. Both files and
 * directory names will be listed along with their attributes.
 */

aftp_local_dir_open(
    connection_id,
    filespec,
    (AFTP_LENGTH_TYPE)strlen(filespec),
    AFTP_DIRECTORY | AFTP_FILE,
    AFTP_NATIVE_ATTRIBUTES,
    path,
    (AFTP_LENGTH_TYPE)sizeof(path)-1,
    &path_length,
    &aftp_rc);

if (aftp_rc == AFTP_RC_OK) {
    path[path_length] = '\0';

    printf("Directory listing of %s.", path);

    do {
        /*
         * Read one directory entry from the AFTP client
         */

        aftp_local_dir_read(
            connection_id,
            dir_entry,
            (AFTP_LENGTH_TYPE)sizeof(dir_entry)-1,
            &dir_entry_length,
            &no_more_entries,
            &aftp_rc);

        if (aftp_rc == AFTP_RC_OK && no_more_entries == 0) {
            dir_entry[dir_entry_length] = '\0';
            printf("Local file: %s\n", dir_entry);
        }
        /*
         * Loop until we either run out of directory
         * entries or an error occurs.
         */
    } while (aftp_rc == AFTP_RC_OK && no_more_entries == 0);

    /*
     * Terminate the directory listing by executing
     * a close.
     */

    aftp_local_dir_close(connection_id, &aftp_rc);
}

```

```
        if (aftp_rc != AFTP_RC_OK) {
            fprintf(
                stderr,
                "Error closing local AFTP directory.\n");
        }
    }
    else {
        fprintf(stderr, "Error opening local AFTP directory.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_local_query_current_dir

Use this call to query the current working directory on the AFTP client. A connection to the AFTP server is not required before using this call.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_local_query_current_dir(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    directory,
    AFTP_LENGTH_TYPE          directory_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

directory

(output) The buffer into which the current working directory on the AFTP client will be written.

Use the `AFTP_FILE_NAME_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

directory_size

(input) The size in bytes of the directory buffer.

returned_length

(output) The actual length of the directory parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             directory[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE         length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Query the current working directory on the
     * AFTP client.
     */

    aftp_local_query_current_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)sizeof(directory)-1,
```

```
        &length,  
        &aftp_rc);  
  
    directory[length] = '\\0';  
  
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error in query of local current directory.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_query_bytes_transferred

Use this call to query the total number of bytes transferred after either an `aftp_send_file()` or `aftp_receive_file()` call has completed. The number of bytes transferred is valid only after a file transfer operation has completed. A connection to the AFTP server must be established before using this call.

Format

```
AFTP_ENTRY
aftp_query_bytes_transferred(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_LENGTH_TYPE AFTP_PTR  bytes_transferred,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

bytes_transferred

(output) The number of bytes of data transferred during the last send or receive file operation.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_LENGTH_TYPE      number_bytes;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use:      aftp_create()
     *   a connection to server, use: aftp_connect()
     *   completed a send or receive, use: aftp_send() or aftp_receive()
     */

    aftp_query_bytes_transferred(connection_id, &number_bytes, &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error getting number bytes transferred.\n");
    } else {
        fprintf(stdout, "Number of bytes %d.\n", (int)number_bytes);
    }
}
```

Line Flows

There are no line flows for this call.

aftp_query_current_dir

Use this call to query the current directory on the AFTP server. A connection to the AFTP server must be established before using this call.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_query_current_dir(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    directory,
    AFTP_LENGTH_TYPE          directory_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

directory

(output) The buffer into which the current working directory on the AFTP server will be written.

Use the `AFTP_FILE_NAME_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

directory_size

(input) The size in bytes of the directory buffer.

returned_length

(output) The actual length of the directory parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             directory[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE         length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use:      aftp_create()
     *   a connection to server, use: aftp_connect()
     */

    /*
     * Query the current working directory on the AFTP server.
     */

    aftp_query_current_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)sizeof(directory)-1,
```

```
        &length,  
        &aftp_rc);  
  
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error in query of current directory.\n");  
    }  
}
```

Line Flows

The request for the directory name is sent to the AFTP server and the call waits for a response indicating the success or failure of the query current working directory operation. The directory name of the current working directory on the AFTP server will be sent as the response if the query was successful.

aftp_query_local_system_info

Use this call to get information about the AFTP client and the computer it is running on. A connection to the AFTP server is not required before using this call.

Format

```
AFTP_ENTRY
aftp_query_local_system_info(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    system_info,
    AFTP_LENGTH_TYPE          system_info_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

system_info

(output) Buffer to store a text string describing the operating system and AFTP client version.

Use the `AFTP_SYSTEM_INFO_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

system_info_size

(input) Size in bytes of the `system_info` parameter.

returned_length

(output) Number of bytes stored into the `system_info` parameter.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char             system_info[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE         system_info_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Query the AFTP client computer for more information.
     */

    aftp_query_local_system_info(
        connection_id,
        system_info,
        (AFTP_LENGTH_TYPE)sizeof(system_info)-1,
        &system_info_length,
        &aftp_rc);
}
```

```
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error querying AFTP client system.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_query_local_version

Use this call to query the AFTP version number on the AFTP client computer. A connection to the AFTP server is not required before using this call.

Format

```
AFTP_ENTRY
aftp_query_local_version(
    AFTP_VERSION_TYPE AFTP_PTR      major_version,
    AFTP_VERSION_TYPE AFTP_PTR      minor_version,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

major_version

(output) The major version number of the AFTP code on the client computer. For example **5.4**, the **5** is the major version number.

minor_version

(output) The minor version number of the AFTP code on the client computer. For example **5.4**, the **4** is the minor version number.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    AFTP_VERSION_TYPE        major_version;
    AFTP_VERSION_TYPE        minor_version;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Query the AFTP version number on the
     * AFTP client computer.
     */

    aftp_query_local_version(
        &major_version,
        &minor_version,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error in query of local version.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_query_system_info

Use this call to get information about the AFTP server and the computer it is running on. A connection to the AFTP server must be established before using this call.

Format

```
AFTP_ENTRY
aftp_query_system_info(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    system_info,
    AFTP_LENGTH_TYPE          system_info_size,
    AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

system_info

(output) Buffer to store a text string describing the operating system and AFTP server version.

Use the `AFTP_SYSTEM_INFO_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

system_info_size

(input) Size in bytes of the `system_info` parameter.

returned_length

(output) Number of bytes stored into the `system_info` parameter.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            system_info[AFTP_SYSTEM_INFO_SIZE+1];
    AFTP_LENGTH_TYPE         system_info_length;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use:      aftp_create()
     *   a connection to server, use: aftp_connect()
     */

    /*
     * Query the AFTP server computer for more information.
     */

    aftp_query_system_info(
        connection_id,
        system_info,
        (AFTP_LENGTH_TYPE)sizeof(system_info)-1,
        &system_info_length,
        &aftp_rc);
}
```

```
system_info[system_info_length] = '\0';  
if (aftp_rc != AFTP_RC_OK) {  
    fprintf(stderr, "Error querying AFTP server system.\n");  
}  
}
```

Line Flows

The request for the system information is sent to the AFTP server and the call waits for a response indicating the success or failure of the query system information operation. The system information of the AFTP server computer will be sent as the response if the query was successful.

aftp_receive_file

Use this call to receive a single file from the AFTP server. A connection to the AFTP server must be established before using this call.

Format

```
AFTP_ENTRY
aftp_receive_file(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    local_file,
    AFTP_LENGTH_TYPE         local_file_length,
    unsigned char AFTP_PTR    remote_file,
    AFTP_LENGTH_TYPE         remote_file_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

local_file

(input) The name given to the file received on the AFTP client. The format of this name can either be the native syntax on the AFTP client or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can contain either an absolute or relative path name.

local_file_length

(input) The length of the `local_file_name` parameter in bytes.

remote_file

(input) The name of the file sent from the AFTP server. The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can contain either an absolute or relative path name.

remote_file_length

(input) The length of the `remote_file_name` parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /* The value used for filespec will vary based on platform:
     * VM common naming:      filespec="/a/myfile.dat"
     * VM native naming:     filespec="myfile.dat.a"
     * MVS PDS common naming: filespec="/user.mypds/myfile"
     * MVS PDS native naming: filespec="'user.mypds(myfile)'"
     * MVS sequential common: filespec="/user.qual.myfile"
     * MVS sequential native: filespec="'user.qual.myfile'"
     */
    static unsigned char AFTP_PTR local_file = "/user.mypos/myfile"; /* MVS */
    static unsigned char AFTP_PTR remote_file = "/a/myfile.dat";      /* VM */

    /*
```

```

* Before issuing the example call, you must have:
*   a connection_id, use:      aftp_create()
*   a connection to server, use: aftp_connect()
*/

aftp_receive_file(
    connection_id,
    local_file,
    (AFTP_LENGTH_TYPE)strlen(local_file),
    remote_file,
    (AFTP_LENGTH_TYPE)strlen(remote_file),
    &aftp_rc);

if (aftp_rc != AFTP_RC_OK) {
    fprintf(stderr, "Error receiving AFTP file.\n");
}
}

```

Line Flows

The request to receive the file(s) is sent to the AFTP server. A send file indicator will be returned to the AFTP client. All records of each file are then sent from the AFTP server to the AFTP client.

aftp_remove_dir

Use this call to remove a directory from the AFTP server. A connection to the AFTP server must be established before using this call.

Platform Differences:

1. On VM, this call is not supported. If issued, the call will fail with return code AFTP_RC_FAIL_NO_RETRY.
2. On MVS, partitioned data sets act as the directory structure. This call will delete a partitioned data set with the name specified.

Read the *z/OS Communications Server: APPC Application Suite User's Guide* for details on how the directory concept is handled for supported operating systems.

Format

```
AFTP_ENTRY
aftp_remove_dir(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    directory,
    AFTP_LENGTH_TYPE          length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR    return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

directory

(input) The directory to be removed. The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The directory specified can be either an absolute or relative path name.

length

(input) The length of the directory parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /* The value used for filespec will vary based on platform:
     * VM not supported
     * MVS PDS common naming: directory="/user.clist/"
     * MVS PDS native naming: directory="'user.clist'"
     */
    static unsigned char AFTP_PTR directory = "/user.clist/";

    /*
     * Before issuing the example call, you must have:
     * a connection_id, use: aftp_create()
     * a connection to server, use: aftp_connect()
     */

    aftp_remove_dir(
```

```
        connection_id,  
        directory,  
        (AFTP_LENGTH_TYPE)strlen(directory),  
        &aftp_rc);  
  
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error removing AFTP directory.\n");  
    }  
  
}
```

Line Flows

The remove directory request and the directory name to remove are sent to the AFTP server and the call waits for a response indicating the success or failure of the remove directory operation.

aftp_rename

Use this call to rename a file on the AFTP server. A connection to the AFTP server must be established before using this call.

Format

```
AFTP_ENTRY
aftp_rename(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    oldfile,
    AFTP_LENGTH_TYPE         oldlength,
    unsigned char AFTP_PTR    newfile,
    AFTP_LENGTH_TYPE         newlength,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

oldfile

(input) The name of the file to be renamed.

The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can either be an absolute or relative path name.

oldlength

(input) The length in bytes of the oldfile parameter.

newfile

(input) The new name of the file.

The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can either be an absolute or relative path name.

newlength

(input) The length in bytes of the newfile parameter.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /* The value used for filespec will vary based on platform:
     * VM common naming:      newfile="/a/foo.file"
     * VM native naming:      newfile="foo.file.a"
     * MVS PDS common naming: newfile="/user.clist/foo"
     * MVS PDS native naming: newfile="'user.clist(foo)'"
     * MVS sequential common: newfile="/user.qual.a.foo"
     * MVS sequential native: newfile="'user.qual.a.foo'"
     */
    static unsigned char AFTP_PTR newfile = "/user.clist/foo";
    static unsigned char AFTP_PTR oldfile = "/user.clist/abc";
```

```

/*
 * Before issuing the example call, you must have:
 *   a connection_id, use:      aftp_create()
 *   a connection to server, use: aftp_connect()
 */

aftp_rename(
    connection_id,
    oldfile,
    (AFTP_LENGTH_TYPE)strlen(oldfile),
    newfile,
    (AFTP_LENGTH_TYPE)strlen(newfile),
    &aftp_rc);

if (aftp_rc != AFTP_RC_OK) {
    fprintf(stderr, "Error renaming AFTP file.\n");
}
}

```

Line Flows

The rename request, old, and new file names are sent to the AFTP server and the call waits for a response indicating the success or failure of the rename operation.

aftp_send_file

Use this call to send a single file to the AFTP server. A connection to the AFTP server must be established before using this call.

Format

```
AFTP_ENTRY
aftp_send_file(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    local_file,
    AFTP_LENGTH_TYPE         local_file_length,
    unsigned char AFTP_PTR    remote_file,
    AFTP_LENGTH_TYPE         remote_file_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

local_file

(input) The name of the file sent from the AFTP client. The format of this name can either be the native syntax on the AFTP client or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can contain either an absolute or relative path name.

local_file_length

(input) The length of the `local_file_name` parameter in bytes.

remote_file

(input) The name given to the file received on the AFTP server. The format of this name can either be the native syntax on the AFTP server or the AFTP common naming convention (described in the *z/OS Communications Server: APPC Application Suite User's Guide*). The file specified can contain either an absolute or relative path name.

remote_file_length

(input) The length of the `remote_file_name` parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /* The value used for filespec will vary based on platform:
     * VM common naming:      filespec="/a/myfile.dat"
     * VM native naming:     filespec="myfile.dat.a"
     * MVS PDS common naming: filespec="/user.mypds/myfile"
     * MVS PDS native naming: filespec="'user.mypds(myfile)'"
     * MVS sequential common: filespec="/user.qual.myfile"
     * MVS sequential native: filespec="'user.qual.myfile'"
     */
    static unsigned char AFTP_PTR local_file = "/user.mypos/myfile"; /* MVS */
    static unsigned char AFTP_PTR remote_file = "/a/myfile.dat";      /* VM */

    /*
```

```

* Before issuing the example call, you must have:
*   a connection_id, use:      aftp_create()
*   a connection to server, use: aftp_connect()
*/

aftp_send_file(
    connection_id,
    local_file,
    (AFTP_LENGTH_TYPE)strlen(local_file),
    remote_file,
    (AFTP_LENGTH_TYPE)strlen(remote_file),
    &aftp_rc);

if (aftp_rc != AFTP_RC_OK) {
    fprintf(stderr, "Error sending AFTP file.\n");
}
}

```

Line Flows

The send file request is sent to the AFTP server, immediately followed by all records of the files. The call waits for a response indicating the success or failure of the send file operation.

aftp_set_allocation_size

Use this call to set the AFTP file allocation size. A connection to the AFTP server is not required before using this call. The file allocation size can be changed at any time.

Format

```
AFTP_ENTRY
aftp_set_allocation_size(
    AFTP_HANDLE_TYPE          connection_id,
    AFTP_ALLOCATION_SIZE_TYPE  allocation_size,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

allocation_size

(input) The allocation size in bytes to set for the AFTP file. The default allocation size value is 0.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Set the file allocation size for AFTP file
     * transfers.
     */

    aftp_set_allocation_size(
        connection_id,
        500,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP allocation size.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_block_size

Use this call to set the file block size. A connection to the AFTP server is not required before using this call. The file block size may be changed at any time.

Format

```
AFTP_ENTRY
aftp_set_block_size(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_BLOCK_SIZE       block_size,
    AFTP_RETURN_CODE_TYPE AFTP_PTR   return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

block_size

(input) The AFTP file block size in bytes. The default block size value is 0.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Set the file block size for AFTP file
     * transfers.
     */

    aftp_set_block_size(
        connection_id,
        512,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP block size.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_data_type

Use this call to set the data type for file transfers. A connection to the AFTP server is not required before using this call. The data type can be changed at any time.

Format

```
AFTP_ENTRY
aftp_set_data_type(
    AFTP_HANDLE_TYPE      connection_id
    AFTP_DATA_TYPE_TYPE   data_type,
    AFTP_RETURN_CODE_TYPE AFTP_PTR   return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

data_type

(input) The data type to be used for subsequent data transfers.

AFTP_ASCII

Transfer files as text files in ASCII.

AFTP_BINARY

Transfer files as a binary sequence of bytes without translation.

AFTP_DEFAULT_DATA_TYPE

Use the data transfer type set in the INI file. If no type is set in the INI file, use `AFTP_ASCII`.

This is the default setting.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Set the data type for AFTP file
     * transfers.
     */

    aftp_set_data_type(
        connection_id,
        AFTP_BINARY,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP data type.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_date_mode

Use this call to set the way file dates are handled during data transfer. A connection to the AFTP server is not required before using this call. The date mode can be changed at any time.

Format

```
AFTP_ENTRY
aftp_set_date_mode(
    AFTP_HANDLE_TYPE          connection_id
    AFTP_DATE_MODE_TYPE       date_mode,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

date_mode

(input) Specifies the way file dates are handled during data transfer.

AFTP_NEWDATE

Assign the time/date stamp of the time of transfer.

AFTP_OLDDATE

Assign the time/date stamp of the source file. This is the default.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE     aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Set the date mode for AFTP file
     * transfers.
     */

    aftp_set_date_mode(
        connection_id,
        AFTP_OLDDATE,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP date mode.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_destination

Use this call to specify the destination of the AFTP server. This call must be issued prior to establishing a connection to the AFTP server. After a connection is established, the destination cannot be changed.

Format

```
AFTP_ENTRY
aftp_set_destination(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    destination,
    AFTP_LENGTH_TYPE         length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR    return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

destination

(input) Identifies the location of the AFTP server. This parameter can be either a symbolic destination name or a partner LU name.

See the *z/OS Communications Server: APPC Application Suite User's Guide* for information about specifying destinations in the APPC Application Suite.

length

(input) The length of the destination parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    static unsigned char AFTP_PTR destination = "NETWORK.SERVER";
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the partner we want to communicate with - who will
     * be running the AFTP server.
     */

    aftp_set_destination(
        connection_id,
        destination,
        (AFTP_LENGTH_TYPE)strlen(destination),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP destination.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_mode_name

Use this call to specify the mode name for the connection to the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. Once a connection is open, the mode name cannot be changed.

Format

```
AFTP_ENTRY
aftp_set_mode_name(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    mode_name,
    AFTP_LENGTH_TYPE          length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR    return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

mode_name

(input) Specifies the mode name to be used on the connection. The default is **#BATCH**. The mode name must be 1–8 bytes in length.

length

(input) The length of the `mode_name` parameter in bytes.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    static unsigned char AFTP_PTR mode_name = "#INTER";
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the mode name for the AFTP connection.
     */

    aftp_set_mode_name(
        connection_id,
        mode_name,
        (AFTP_LENGTH_TYPE)strlen(mode_name),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP mode name.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_password

Use this call to specify the password for the connection to the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. Once a connection is open, the password cannot be changed. If a password is set, a user ID also must be set using `aftp_set_userid()` before connecting to the AFTP server. Use of this call sets the security type to `AFTP_SECURITY_PROGRAM`.

Format

```
AFTP_ENTRY
aftp_set_password(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    password,
    AFTP_LENGTH_TYPE         length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

password

(input) The password to be used on the connection. The password can be 1–8 bytes long.

length

(input) The length of the password parameter in bytes.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    static unsigned char AFTP_PTR password = "MYPASS";
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the password for the AFTP connection.
     */

    aftp_set_password(
        connection_id,
        password,
        (AFTP_LENGTH_TYPE)strlen(password),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP password.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_record_format

Use this call to set the record format for the data transfer. A connection to the AFTP server is not required before using this call. The record format may be changed at any time.

Format

```
AFTP_ENTRY
aftp_set_record_format(
    AFTP_HANDLE_TYPE           connection_id,
    AFTP_RECORD_FORMAT_TYPE    record_format,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

record_format

(input) The record format used for file transfer.

AFTP_DEFAULT_RECORD_FORMAT

Specifies that the system on which the file will be written should use its own default setting for record format. This is the initial setting.

AFTP_V

Variable length record, unblocked. This is the default.

AFTP_VA

Variable length record, unblocked, ASA print-control characters.

AFTP_VB

Variable length record, blocked.

AFTP_VM

Variable length record, unblocked, machine print-control codes.

AFTP_VS

Variable length record, unblocked, spanned.

AFTP_VBA

Variable length record, blocked, ASA print-control characters.

AFTP_VBM

Variable length record, blocked, machine print-control codes.

AFTP_VBS

Variable length record, blocked, spanned.

AFTP_VSA

Variable length record, unblocked, spanned, ASA print-control characters.

AFTP_VSM

Variable length record, unblocked, spanned, machine print-control codes.

AFTP_VBSA

Variable length record, blocked, spanned, ASA print-control characters.

AFTP_VBSM

Variable length record, blocked, spanned, machine print-control codes.

AFTP_F

Fixed length record, unblocked.

AFTP_FA

Fixed length record, unblocked, ASA print-control characters.

AFTP_FB

Fixed length record, blocked.

AFTP_FM

Fixed length record, unblocked, machine print-control codes.

AFTP_FBA

Fixed length record, blocked, ASA print-control characters.

AFTP_FBM

Fixed length record, blocked, machine print-control codes.

AFTP_FBS

Fixed length record, blocked, standard.

AFTP_FBSM

Fixed length record, blocked, machine print-control codes, standard.

AFTP_FBSA

Fixed length record, blocked, ASA print-control characters, standard.

AFTP_U

Undefined length record.

AFTP_UA

Undefined length record, ASA print control characters.

AFTP_UM

Undefined length record, machine print control codes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
  AFTP_HANDLE_TYPE      connection_id;
  AFTP_RETURN_CODE_TYPE aftp_rc;

  /*
   * Before issuing the example call, you must have:
   *   a connection_id, use: aftp_create()
   */

  /*
   * Set the record format value for the file transfer.
   */

  aftp_set_record_format(
    connection_id,
    AFTP_VSA,
    &aftp_rc);
  if (aftp_rc != AFTP_RC_OK) {
    fprintf(stderr, "Error setting AFTP record format.\n");
  }
}
```

Line Flows

There are no line flows for this call.

aftp_set_record_length

Use this call to set the record length for fixed length records, or the maximum possible record length for variable length records used for data transfer. A connection to the AFTP server is not required before using this call. The record length may be changed at any time.

Format

```
AFTP_ENTRY
aftp_set_record_length(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_RECORD_LENGTH_TYPE record_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR    return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

record_length

(input) The record length for the data transfer specified in bytes. The default value is 0.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Set the record length for the file transfer.
     */

    aftp_set_record_length(
        connection_id,
        64,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP record length.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_security_type

Use this call to specify the type of APPC conversation security to be used. This call can only be invoked prior to the establishment of a connection to the AFTP server. Once a connection is open, the APPC security type cannot be changed. If AFTP_SECURITY_PROGRAM is used for the security type a user ID and password also must be set using aftp_set_userid() and aftp_set_password(), before connecting to the AFTP server.

Format

```
AFTP_ENTRY
aftp_set_security_type(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_SECURITY_TYPE    security_type,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

security_type

(input) The security to be used when connecting to the AFTP server.

AFTP_SECURITY_NONE

No APPC conversation security is used. This is the default unless CPI-C side information is set otherwise.

AFTP_SECURITY_SAME

The local security information determined at login time will be transferred to the AFTP server.

AFTP_SECURITY_PROGRAM

A user ID and password will be sent to be verified by the AFTP server. This security type requires the use of the aftp_set_userid() and aftp_set_password() calls, or the connection attempt will fail.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the APPC conversation security type for the
     * AFTP connection.
     */

    aftp_set_security_type(
        connection_id,
        AFTP_SECURITY_SAME,
        &aftp_rc);
```

```
    if (aftp_rc != AFTP_RC_OK) {  
        fprintf(stderr, "Error setting AFTP security type.\n");  
    }  
}
```

Line Flows

There are no line flows for this call.

aftp_set_tp_name

Use this call to specify the transaction program (TP) name of the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. Once a connection is open, the transaction program name cannot be changed. The AFTP API defaults the transaction program name on the server to be AFTPD. This call is not necessary unless you want to experiment with a server which may not have the same behavior as AFTPD.

Format

```
AFTP_ENTRY
aftp_set_tp_name(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    tp_name,
    AFTP_LENGTH_TYPE         length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with aftp_create().

tp_name

(input) The transaction program name of the AFTP server. The transaction program name can be 1-64 bytes long. The default transaction program name is AFTPD.

length

(input) The length of the transaction program name parameter in bytes.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    static unsigned char AFTP_PTR tp_name = "AFTPME";
    AFTP_RETURN_CODE_TYPE     aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the TP name for the AFTP server.
     */

    aftp_set_tp_name(
        connection_id,
        tp_name,
        (AFTP_LENGTH_TYPE)strlen(tp_name),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP TP name.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_trace_filename

Use this call to set the name of the file to which trace information will be written. If trace is turned on by the `aftp_set_trace_level()` call and the `aftp_set_trace_filename()` call is not issued, the trace file generated will be:

- On VM: ASUITE TRC
- On MVS: DD:SYSOUT
- On OS/2: ASUITE.TRC

Format

```
AFTP_ENTRY
aftp_set_trace_filename(
    unsigned char AFTP_PTR          filename,
    AFTP_LENGTH_TYPE               filename_length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

filename

(input) The name of the file to be used for trace output.

filename_length

(input) The length of the filename parameter in bytes.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE    rc;

    /* The value used for filespec will vary based on platform:
     * VM common naming:      filename="/a/aftp.trace"
     * VM native naming:      filename="aftp.trace.a"
     * MVS PDS common naming: filename="/user.clist/aftptrac"
     * MVS PDS native naming: filename="'user.clist(aftptrac)'"
     * MVS sequential common: filename="/user.qual.a.aftptrac"
     * MVS sequential native: filename="'user.qual.a.aftptrac'"
     */
    static unsigned char AFTP_PTR filename = "/user.clist/aftptrac";

    /*
     * There are no prerequisite calls for this call.
     */

    aftp_set_trace_filename(
        filename,
        (AFTP_LENGTH_TYPE)strlen(filename),
        &rc);

    if (rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting tracing filename\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_trace_level

Use this call to set the level of tracing to use for AFTP activities. The new trace level will take effect immediately upon making this call. The trace output will be captured in the file specified in the `aftp_set_trace_filename()` call, which must be issued before this call.

Format

```
AFTP_ENTRY
aftp_set_trace_level(
    AFTP_TRACE_LEVEL_TYPE      trace_level,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

trace_level

(input) The amount of trace information to be generated. The constants from `AFTP_LVL_NO_TRACING` to `AFTP_LVL_MAX_TRACE_LVL` incrementally increase the amount of trace information.

AFTP_LVL_NO_TRACING

No data will be written to the trace log.

AFTP_LVL_API

Traces crossings of the API boundary.

AFTP_LVL_MAX_TRACE_LVL

Provides the maximum amount of trace information.

Other trace levels are reserved for diagnosing problems with the assistance of the IBM Support Center.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    /*
     * The following calls must be issued here:
     *   aftp_set_trace_filename()
     */

    /*
     * Turn on the tracing.
     */

    aftp_set_trace_level(trace_level, &rc);

    if (rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting the trace level\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_userid

Use this call to specify the user ID for the connection to the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. Once a connection is open, the user ID cannot be changed. If a user ID is set, a password also must be set using `aftp_set_password()` before connecting to the AFTP server. Use of this call sets the security type to `AFTP_SECURITY_PROGRAM`.

Format

```
AFTP_ENTRY
aftp_set_userid(
    AFTP_HANDLE_TYPE          connection_id,
    unsigned char AFTP_PTR    userid,
    AFTP_LENGTH_TYPE         length,
    AFTP_RETURN_CODE_TYPE AFTP_PTR return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

userid

(input) The user ID to be used on the connection. The user ID can be 1–8 bytes long.

length

(input) The length of the `userid` parameter in bytes.

return_code

(output) See “Chapter 3. AFTP Return Codes” on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    static unsigned char AFTP_PTR userid = "LBONANNO";
    AFTP_RETURN_CODE_TYPE    aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the user ID for the AFTP connection.
     */

    aftp_set_userid(
        connection_id,
        userid,
        (AFTP_LENGTH_TYPE)strlen(userid),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting user ID.\n");
    }
}
```

Line Flows

There are no line flows for this call.

aftp_set_write_mode

Use this call to set the way existing files will be handled during data transfer. A connection to the AFTP server is not required before using this call. The write mode may be changed at any time.

Format

```
AFTP_ENTRY
aftp_set_write_mode(
    AFTP_HANDLE_TYPE      connection_id,
    AFTP_WRITE_MODE_TYPE  write_mode,
    AFTP_RETURN_CODE_TYPE AFTP_PTR  return_code);
```

Parameters

connection_id

(input) An AFTP connection object originally created with `aftp_create()`.

write_mode

(input) The method used to write a file if a copy of the file already exists. If the file does not exist on the target, the new file will be created.

AFTP_REPLACE

Transferred file will replace the existing file. This is the default.

AFTP_APPEND

Transferred file will be appended to the existing file.

return_code

(output) See "Chapter 3. AFTP Return Codes" on page 113 for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE      connection_id;
    AFTP_RETURN_CODE_TYPE aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *   a connection_id, use: aftp_create()
     */

    /*
     * Set the write mode for the AFTP
     * file transfer.
     */

    aftp_set_write_mode(
        connection_id,
        AFTP_REPLACE,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP write mode.\n");
    }
}
```

Line Flows

There are no line flows for this call.

Chapter 3. AFTP Return Codes

These are the possible return codes that can be issued for each AFTP API call.

AFTP_RC_BUFFER_TOO_SMALL

The buffer supplied by the caller for output data was too small to hold the data.

AFTP_RC_COMM_CONFIG_LOCAL

The call failed due to a local configuration error. Communications will fail until the configuration problem is resolved.

AFTP_RC_COMM_CONFIG_REMOTE

The call failed due to a remote configuration error. Communications will fail until the configuration problem is resolved.

AFTP_RC_COMM_FAIL_NO_RETRY

The call failed due to a communications problem. The call will not successfully complete using the current parameters.

AFTP_RC_COMM_FAIL_RETRY

The call failed due to a communications problem. The call may successfully complete if tried again.

AFTP_RC_FAIL_FATAL

A serious system error has occurred; no calls can complete successfully.

AFTP_RC_FAIL_INPUT_ERROR

The call may successfully complete after new input parameters are supplied.

AFTP_RC_FAIL_NO_RETRY

The call will not successfully complete using the current parameters.

AFTP_RC_FAIL_RETRY

The call may successfully complete if tried again.

AFTP_RC_HANDLE_NOT_VALID

The call failed because the AFTP connection object passed into the AFTP API was not valid.

AFTP_RC_OK

The call completed successfully.

AFTP_RC_PARAMETER_CHECK

The call failed due to an error in one of the parameters passed into the AFTP API.

AFTP_RC_PROGRAM_INTERNAL_ERROR

The call failed due to a programming error.

AFTP_RC_SECURITY_NOT_VALID

The call failed because of APPC security.

AFTP_RC_STATE_CHECK

The call failed because the current AFTP API call was made when AFTP was not in the correct state required for the current call. For example, you will get a state check error if you try to use `aftp_format_error()` when no error has occurred.

Part 2. APPC Name Server (ANAME) Programming Interface

Chapter 4. API for the APPC NameServer

The APPC NameServer (ANAME) application programming interface (API) is a set of C procedures that provide APPC name resolution. This API is referred to as the ANAME API and allows the programmer to access routines that will interact with any ANAME server.

The ANAME API adheres to the ANAME line flow standards. All ANAME client applications will send the same set of line flows over the network to the ANAME server.

How the ANAME API Works

This section is an overview of how the ANAME API works from the perspective of the application programmer. This is a logical overview instead of an actual explanation of what is sent and received to and from the ANAME server. However, this logical overview must be understood to determine the ANAME API calls that are required by an application. The actual line flows for each call are briefly described in “Chapter 5. ANAME API Call Reference” on page 123.

The ANAME API handles requests in the following manner:

1. A connection object is created on the client. This connection object contains:
 - Information that is relevant to the setup and maintenance of the connection with the ANAME server
 - A data object for this connection
2. A data object is built on the client side containing data that is specified by the client program. A data object is not accessed directly, instead it is accessed through the connection object it is associated with. Depending on the number of fields that are explicitly set, this is one full or partial record.
3. The data object is sent to the ANAME server with a code indicating the requested function.
4. The ANAME server receives the data object and processes the request based on the fields that are set in the received data object.
5. If the function requested was a query, data records must be returned to the requesting client. The ANAME server sends the records one at a time back to the client to be used by the application program. These records must be individually received by the application program.

An interface is provided to turn on and off the trace facility. The trace settings will be maintained throughout the existence of one or many connection objects.

ANAME Defined Constants, Standard Types, and Conventions

This section describes constants, types, and conventions for use in the ANAME API that are not standard to C programming.

Defined Constants

ANAME constant definitions have been created for the sizes of the buffers that are passed across the API. All buffers of the specified type must be at least the size of the defined constant to guarantee the requested call will not fail with a buffer size error.

The constants are defined in the header file for the ANAME API.

Table 3. ANAME Size Constants

Constant	Minimum Buffer Size For	Value
ANAME_FQ_SIZE	Fully qualified LU name	17
ANAME_GN_SIZE	Group name	64
ANAME_HND_SIZE	Connection handle	8
ANAME_DEST_SIZE	Destination name	17
ANAME_TP_SIZE	Transaction program name	64
ANAME_UN_SIZE	User name	64

Standard Types

Type definitions are available for many parameters to the ANAME API calls. For example, the ANAME type ANAME_RETURN_CODE_TYPE is an alias for the C type unsigned long.

Use the ANAME types instead of the corresponding C types. Doing so will protect you from changes to the parameters in future releases. If you have used the ANAME types, you will only need to recompile your code to use the new API definitions. If you have used the C types, you will need to modify your program source to reflect changes to the new C types.

The ANAME API avoids complex structures and pointers to structures for type definitions. These complex structures might not be supported in all languages. The only exception is the string construct which is found in many languages.

Table 4. ANAME Standard Types

Type Name	Type to Define	Actual Type	Size
ANAME_DATA_RECEIVED_TYPE	Data received variable	Unsigned long	Long (4 bytes)
ANAME_DESTINATION_TYPE	Destination name of the data record	Unsigned character ARRAY	[18]
ANAME_DETAIL_LEVEL_TYPE	Detail level to be passed to the aname_format_error() call	Unsigned long	Long (4 bytes)
ANAME_HANDLE_TYPE	ANAME connection object	Unsigned character array	[8]
ANAME_LENGTH_TYPE	Any buffer length	Unsigned long	Long (4 bytes)
ANAME_RETURN_CODE_TYPE	ANAME return code	Unsigned long	Long (4 bytes)
ANAME_TRACE_LEVEL_TYPE	Trace level	Unsigned long	Long (4 bytes)
ANAME_DUP_FLAG_TYPE	Duplicate flag variable	Unsigned long	Long (4 bytes)

Conventions

Null-Terminated Strings

The ANAME API does not require input strings to be null-terminated. The ANAME API also does not guarantee that the output strings are null terminated. The return size does not include the null terminator in the size, if there is any.

The C programmer should be aware of the fact that strings are handled differently within ANAME than they are in the C standard library. All API calls receiving strings as input require both the string itself, and the length of the string. The `strlen()` function can be used for this. The null terminator must not be counted as part of the string length. API calls which output strings require three string related parameters:

- The string.
- The length of the string buffer that has been allocated by the calling program. Both of these are input to the API, with the string being modified.
- The length of the string that is output. ANAME output strings are not null terminated. In order for the C programmer to use them as standard C strings, a null character must be added to the end of the string.

ANAME_ENTRY

The ANAME API calls do not return a value. Rather, the return code parameter is set to indicate the success or failure of the call. The programmer should check the return code parameter after each call and handle error values appropriately.

The C keyword `void` is not used for entry points in the ANAME API. Instead, `ANAME_ENTRY` has been defined. `ANAME_ENTRY` is defined differently depending on the operating system the ANAME client will be created on.

ANAME_PTR

The C pointer indicator `**` is not used in the ANAME API prototypes. Instead, `ANAME_PTR` has been defined. `ANAME_PTR` indicates that the address of the value should be passed, rather than the actual value. `ANAME_PTR` is defined differently depending on the operating system the ANAME client will be created on.

Compiling the ANAME Application

MVS

To develop an application that uses the ANAME API on MVS, follow these steps:

Note: This process assumes all ANAME program files have been successfully installed with the provided JCL for installing ANAME.

1. Include this header file in your source modules: `APPMAPIH`.
2. Define `CM_MVS` when you compile your source.
3. Edit the `APPMAPIJ` JCL file and make the changes indicated in the prolog comments at the top of the file.
4. Submit the `APPMAPIJ` JCL.

VM

To access the ANAME API calls from your application, follow these steps:

1. Include the following header file in your source modules: `APPMAPIH`.
2. Define `CM_VM` when you compile your source.
3. Add `APPMAPIL` `TXTLIB` to your `GLOBAL` `TXTLIB` statement. All textdecks are packaged into this `textlib`.

Overview of API Calls

The calls of the ANAME API can be organized into the following categories:

- Create or destroy an ANAME connection object
- Set values in the connection object
- Set values in the data object
- Add a record to the database
- Remove records from the database
- Obtain records from the database
- Access values in returned records
- Obtain error information
- Turn tracing on and off
- Use system administrator functions

The following sections give a high-level explanation of the calls available in each category. For information on the syntax of the calls and parameters required, see “Chapter 5. ANAME API Call Reference” on page 123.

Create or Destroy an ANAME Connection Object

The connection object represents an object-oriented approach to ANAME. All other ANAME API calls (with the exception of the trace calls) require a connection object as an input parameter. When the program has finished using ANAME API calls, it should destroy the connection object.

aname_create()

Creates an ANAME connection object and assigns a unique identifier to it. The connection object is accessed by its connection id. The connection object is never automatically destroyed. This allows you to connect to an ANAME server once, or reconnect numerous times using the same ANAME connection object.

aname_destroy()

Destroys the ANAME connection object and recovers all resources associated with it. Once the ANAME connection object has been destroyed, that object must not be used again. If you need a connection object again, create another one.

Set Values in the Connection Object

Specify which server and/or database is accessed by setting the destination name for the connection.

aname_set_destination()

Identifies the destination to be used for the connection.

Set Values in the Data Object

The ANAME client can assign values to fields of a data object accessed by the connection object. These fields correspond to the fields of the records in the database on the ANAME server. Set calls are used to assign values to the fields of the data object.

aname_set_fqlu_name()

Assigns a value to the fully qualified LU name field of the data object.

The fully qualified LU name field of the data object can be set for any query request, but its use for register and delete requests is limited. The fully qualified LU name of the user is automatically generated by ANAME during register and delete operations. This value can be overridden only by an

ANAME administrator. For example, if a user other than the ANAME administrator issues this call to set the fully qualified LU name and then issues an `aname_register()` call, the register call will fail.

aname_set_group_name()

Assigns a value to the group name field of the data object.

aname_set_tp_name()

Assigns a value to the transaction program name field of the data object.

This call does not set the transaction program name for the connection. The transaction program name in this case is simply treated as data in the record.

aname_set_user_name()

Assigns a value to the user name field of the data object.

Add a Record to the Database

You can add a record to the database using the register call. The register call causes the client data object to be sent to the ANAME server.

aname_register()

Adds the client data object to the ANAME database.

For this call to succeed, at least one field in the data object other than the fully qualified LU name must be set.

Remove Records from the Database

You can remove records from the database using the delete call. The delete call causes the client data object to be sent to the ANAME server.

aname_delete()

Removes all records from the ANAME database which contain all values specified in the client data object.

For this call to succeed, at least one field in the data object other than the fully qualified LU name must be set.

Obtain Records from the Database

A program can request records from the database with the query call. The query call causes the client data object to be sent to the ANAME server. All records containing the values set in the client data object are returned to the program.

Obtaining the records is a two-step process. The first step is to make the request. The second step is to receive the returned records so the information in the records can be used by the program.

aname_query()

Sends the client data object to the ANAME server. The ANAME server will process the request by finding all records in the ANAME database that contain all values in the client data object.

This call will not succeed unless at least one of the fields in the data object has been set.

aname_receive()

Finds the next data record which was returned from the ANAME server and makes the record accessible to the program making the query request.

This call will only succeed after a query call has been issued.

This call must be issued before values in a new record received from the server are accessed.

Access Values in Returned Records

Once a data record has been received, the values in the fields can be accessed for use in a program via the extract calls.

aname_extract_fqlu_name()

Fills a buffer with the value in the fully qualified LU name field of the returned record.

aname_extract_group_name()

Fills a buffer with the value in the group name field of the returned record.

aname_extract_tp_name()

Fills a buffer with the value in the transaction program name field of the returned record.

aname_extract_user_name()

Fills a buffer with the value in the user name field of the returned record.

Obtain Error Information

A call is provided for a programmer to return error information to the application user. The error information returned in a user-supplied buffer is formatted for displaying or logging.

aname_format_error()

Fills a buffer with a printable text string for displaying or logging.

Turn Tracing On and Off

The ANAME program provides a tracing facility to monitor the transaction below the API. You can control the amount of trace information generated and specify the name of the file in which to put the trace output.

aname_set_trace_filename()

Specifies the name of the file to which trace output should be written. If this call is not made, the default trace filename will be used.

If this call is not made, no trace output is generated.

Use System Administrator Functions

There are certain functions that will only succeed if the request is made by a system administrator LU. An LU is determined to be the LU of the system administrator if it is configured on the ANAME server. See the *z/OS Communications Server APPC Application Suite Administration* for information on configuring the ANAME server.

aname_set_duplicate_register()

Specifies whether duplicate user names can be added to the database. The system administrator has authority to add records to the database with a user name that matches an existing user name. This allows name services to be used by multiple LU systems.

Chapter 5. ANAME API Call Reference

This chapter provides an alphabetical reference for all of the API calls for the APPC NameServer (ANAME). Program examples are provided for each call to illustrate its use in a program.

aname_create

Use this call to create an ANAME connection object which can be used to connect to an ANAME server. This call will allocate memory to store the function information and returns a handle used to access that memory space.

Format

```
ANAME_ENTRY
aname_create(
    ANAME_HANDLE_TYPE          connection_id,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code);
```

Parameters

connection_id

(output) Handle of the ANAME connection object that was created by this call. Most subsequent ANAME calls require use a previously-created ANAME connection object.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE          handle;
    ANAME_RETURN_CODE_TYPE    rc;

    aname_create( handle, &rc );
}
```

Line Flows

There are no line flows for this call.

aname_delete

Use this call to remove records from the ANAME database. At least one field in the client data object other than the fully qualified LU name must be set with a valid value before this call is issued.

Format

```
ANAME_ENTRY
aname_delete(
    ANAME_HANDLE_TYPE          connection_id,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code);
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE          handle;
    ANAME_RETURN_CODE_TYPE    rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example aname_set_user_name())
     */

    if ( rc == ANAME_RC_OK ) {
        aname_delete( handle, &rc );
    }
}
```

Line Flows

The data object with values set by the ANAME client is sent to the ANAME server. The call waits for a response indicating the success or failure of the delete request.

aname_destroy

Use this call to destroy an ANAME connection object which was created with the `aname_create()` call. This call will free memory that was used to store the function information. The connection id is not valid after this call completes successfully.

Format

```
ANAME_ENTRY
aname_destroy(
    ANAME_HANDLE_TYPE      connection_id,
    ANAME_RETURN_CODE_TYPE ANAME_PTR  return_code);
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call. Following this call, the connection id will be invalid until another `aname_create()` is issued.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE      handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   some processing
     */

    aname_destroy( handle, &rc );
}
```

Line Flows

There are no line flows for this call.

aname_extract_fqlu_name

Use this call to obtain the value in the fully qualified LU name field of a record returned from the ANAME server and received at the ANAME client. Previous `aname_query()` and `aname_receive()` calls must have been issued.

Format

```
ANAME_ENTRY
aname_extract_fqlu_name(
    ANAME_HANDLE_TYPE          connection_id,
    unsigned char ANAME_PTR    fqlu_name,
    ANAME_LENGTH_TYPE         buffer_size,
    ANAME_LENGTH_TYPE ANAME_PTR fqlu_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

fqlu_name

(output) Value in the fully qualified LU name field of the current data record received from the ANAME server.

Use the `ANAME_FQ_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

buffer_size

(input) Size of the buffer supplied.

fqlu_name_length

(output) Length of the fully qualified LU name value returned.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    unsigned char          fqlu_name[ ANAME_FQ_SIZE + 1 ];
    ANAME_LENGTH_TYPE     fqlu_name_length;
    ANAME_HANDLE_TYPE     handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example, aname_set_user_name())
     *   aname_query()
     *   aname_receive()
     */

    aname_extract_fqlu_name(
        handle,
        fqlu_name,
        ANAME_FQ_SIZE,
        &fqlu_name_length,
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_extract_group_name

Use this call to obtain the value in the group name field of a record returned from the ANAME server and received at the ANAME client. Previous `aname_query()` and `aname_receive()` calls must have been issued.

Format

```
ANAME_ENTRY
aname_extract_group_name(
    ANAME_HANDLE_TYPE          connection_id,
    unsigned char ANAME_PTR    group_name,
    ANAME_LENGTH_TYPE          buffer_size,
    ANAME_LENGTH_TYPE ANAME_PTR group_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

group_name

(output) Value in the group name field of the current data record received from the ANAME server.

Use the `ANAME_GN_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

buffer_size

(input) Size of the buffer supplied.

group_name_length

(output) Length of the group name value returned.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    unsigned char          group_name[ ANAME_GN_SIZE + 1 ];
    ANAME_LENGTH_TYPE     group_name_length;
    ANAME_HANDLE_TYPE     handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example, aname_set_user_name())
     *   aname_query()
     *   aname_receive()
     */

    aname_extract_group_name(
        handle,
        group_name,
        ANAME_GN_SIZE,
        &group_name_length,
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_extract_tp_name

Use this call to obtain the value in the transaction program name field of a record returned from the ANAME server and received at the ANAME client. Previous `aname_query()` and `aname_receive()` calls must have been issued.

Format

```
ANAME_ENTRY
aname_extract_tp_name(
    ANAME_HANDLE_TYPE          connection_id,
    unsigned char ANAME_PTR    tp_name,
    ANAME_LENGTH_TYPE         buffer_size,
    ANAME_LENGTH_TYPE ANAME_PTR tp_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

tp_name

(output) Value in the transaction program name field of the current data record received from the ANAME server.

Use the `ANAME_TP_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

buffer_size

(input) Size of the buffer supplied.

tp_name_length

(output) Length of the transaction program name value returned.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    unsigned char          tp_name[ ANAME_TP_SIZE + 1 ];
    ANAME_LENGTH_TYPE     tp_name_length;
    ANAME_HANDLE_TYPE     handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example, aname_set_user_name())
     *   aname_query()
     *   aname_receive()
     */

    aname_extract_tp_name(
        handle,
        tp_name,
        ANAME_TP_SIZE,
        &tp_name_length,
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_extract_user_name

Use this call to obtain the value in the user name field of a record returned from the ANAME server and received at the ANAME client. Previous `aname_query()` and `aname_receive()` calls must have been issued.

Format

```
ANAME_ENTRY
aname_extract_user_name(
    ANAME_HANDLE_TYPE          connection_id,
    unsigned char ANAME_PTR    user_name,
    ANAME_LENGTH_TYPE         buffer_size,
    ANAME_LENGTH_TYPE ANAME_PTR user_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

user_name

(output) Value in the user name field of the current data record received from the ANAME server.

Use the `ANAME_UN_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

buffer_size

(input) Size of the buffer supplied.

user_name_length

(output) Length of the user name value returned.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    unsigned char          user_name[ ANAME_UN_SIZE + 1 ];
    ANAME_LENGTH_TYPE     user_name_length;
    ANAME_HANDLE_TYPE     handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example, aname_set_fqlu_name())
     *   aname_query()
     *   aname_receive()
     */

    aname_extract_user_name(
        handle,
        user_name,
        ANAME_UN_SIZE,
        &user_name_length,
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_format_error

Use this call after an error return code to obtain additional error information. This call returns the error information in a string that is formatted for displaying or logging.

Format

```
ANAME_ENTRY
aname_show_error(
    ANAME_HANDLE_TYPE           connection_id,
    ANAME_DETAIL_LEVEL_TYPE     detail_level,
    unsigned char ANAME_PTR     error_string,
    ANAME_LENGTH_TYPE           error_string_size,
    ANAME_LENGTH_TYPE ANAME_PTR returned_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

detail_level

(input) Numeric value defining the amount of error information that should be returned in the supplied buffer. These values can be OR'ed together to retrieve specific sets of information. For example, if the primary message and the error log information should be returned, specify `(ANAME_DETAIL_RC | ANAME_DETAIL_LOG)`.

ANAME_DETAIL_RC

Return the return code value, location of the error, primary message number and primary message text.

ANAME_DETAIL_SECOND

Return the secondary message number and message text if one exists.

ANAME_DETAIL_LOG

Return the error log specific text with message number if applicable.

ANAME_DETAIL_INFO

Return the informational message and message number if applicable.

ANAME_DETAIL_ALL

Return all information.

error_string

(output) String containing error information which is formatted for display or logging.

The size of the error string will vary, but will not be more than 2 Kbytes.

error_string_size

(input) Size of the `error_string` buffer supplied.

returned_length

(output) Size of the error string returned in the `error_string` buffer.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE          handle;
    ANAME_DETAIL_LEVEL_TYPE    detail_level = ANAME_DETAIL_LOG;
    unsigned char              error_string[ 2048 + 1 ];
    ANAME_LENGTH_TYPE          error_string_length_out;
    ANAME_RETURN_CODE_TYPE     rc;

    /*
     * At least one ANAME call that returned with an error return code
     * must have been issued here.
     */

    aname_format_error(
        handle,
        detail_level,
        error_string,
        1024,
        error_string_size_out,
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_query

Use this call to send the client data object to the ANAME server requesting records in return. Records will be returned which contain the values specified in the supplied data object. At least one field in the data object must have been set with a valid value before this call is issued.

Format

```
ANAME_ENTRY
aname_query(
    ANAME_HANDLE_TYPE      connection_id,
    ANAME_RETURN_CODE_TYPE ANAME_PTR  return_code);
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE      handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   aname_set_xxx call (for example, aname_set_fqlu_name())
     */

    aname_query( handle, &rc );
}
```

Line Flows

The data object containing values set by the client is sent to the ANAME server. The call waits for:

1. A response from the ANAME server indicating whether the query request succeeded. If the response indicates an error occurred, the call returns to the calling program.
2. One data record for each matching record in the ANAME database.
3. A response indicating the last data record has been sent.

All data is stored at the ANAME client before the query call returns to the program.

aname_receive

Use this call to increment the current record pointer to the next available record. After a query call is issued, each record must be individually received before the contents in the record are available to the application. A successful query call must have been issued before the receive call will succeed.

Format

```
ANAME_ENTRY
aname_receive(
    ANAME_HANDLE_TYPE          connection_id,
    ANAME_DATA_RECEIVED_TYPE ANAME_PTR data_received,
    ANAME_RETURN_CODE_TYPE ANAME_PTR  return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

data_received

(output) Value indicating if a new record is available for the application.

ANAME_DR_DATA_RECEIVED_OK

There was another record and the values in that record can now be extracted and used by the program.

ANAME_DR_NO_MORE_DATA

There was not another record to receive.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE          handle;
    ANAME_RETURN_CODE_TYPE     rc;
    ANAME_DATA_RECEIVED_TYPE   data_received;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example, aname_set_fqlu_name())
     *   aname_query()
     */

    aname_receive( handle, &data_received, &rc );
}
```

Line Flows

There are no line flows for this call.

aname_register

Use this call to add a record containing the values in the client data object to the ANAME database. At least one field in the data object other than the fully qualified LU name must be set with a valid value before this call is issued.

Format

```
ANAME_ENTRY
aname_register(
    ANAME_HANDLE_TYPE      connection_id,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code);
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE      handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example, aname_set_fqlu_name())
     */

    aname_register( handle, &rc );
}
```

Line Flows

The data object with values set by the ANAME client is sent to the ANAME server. The call waits for a response indicating the success or failure of the register request.

aname_set_destination

Use this call to assign a value to the destination field of the client data object which will be sent to the ANAME server.

Format

```
ANAME_ENTRY
aname_set_destination(
    ANAME_HANDLE_TYPE          connection_id,
    unsigned char ANAME_PTR    destination,
    ANAME_LENGTH_TYPE         destination_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

destination

(input) Value to be assigned to the destination field of the data record to be sent to the ANAME server. The destination can be a symbolic destination name, a fully qualified LU name, or a locally-defined LU alias. If the value is a symbolic destination name, it must be defined in the ANAME client's CPI-C side information. The definition in the CPI-C side information specifies the partner name and the transaction program to be invoked on the server.

Use the `ANAME_DEST_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

destination_length

(input) Length of the destination parameter that was supplied on this call.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

Example

```
{
    unsigned char          destination[ ANAME_DEST_SIZE + 1 ];
    ANAME_HANDLE_TYPE     handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following calls must be issued here:
     *   aname_create()
     */

    strcpy( destination, "MYANAMED" );
    aname_set_destination(
        handle,
        destination,
        strlen( destination ),
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_set_duplicate_register

Use this call to specify whether the record being registered may contain a user name that currently exists in the database and that the matching user name in the database may contain a different value in the fully qualified LU name field than the value in the record being registered. This call must be made before the register call is made for the record that may have a duplicate user name.

The duplicate register setting remains until it is explicitly changed with another call or until the connection object is destroyed.

The setting ANAME_ALLOW_DUPLICATES is only valid if the call is issued from the ANAME administrator's LU. See *z/OS Communications Server APPC Application Suite Administration* for information on configuring the ANAME administrator.

Format

```
ANAME_ENTRY
aname_set_duplicate_register(
    ANAME_HANDLE_TYPE           connection_id,
    ANAME_DUP_FLAG_TYPE         duplicate_flag
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code);
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

duplicate_flag

(input) Flag which is set to one of the following values to indicate whether a duplicate user name may or may not be registered.

ANAME_ALLOW_DUPLICATES

A duplicate user name may be registered. This can only be set from the system administrator's LU.

ANAME_DISALLOW_DUPLICATES

A duplicate user name may not be registered. This is the default.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

Example

```
{
    ANAME_HANDLE_TYPE           handle;
    ANAME_RETURN_CODE_TYPE      rc;
    ANAME_DUP_FLAG_TYPE         duplicate_flag = ANAME_ALLOW_DUPLICATES;

    /*
     * The following calls must be issued here:
     *   aname_create()
     *   an aname_set_xxx call (for example, aname_set_fqlu_name())
     */

    aname_set_duplicate_register(
```

```
        handle,  
        duplicate_flag,  
        &rc );  
    }
```

Line Flows

There are no line flows for this call.

aname_set_fqlu_name

Use this call to assign a value to the fully qualified LU name field of a data object which will be sent to the ANAME server.

The fully qualified LU name field of the data object can be set for any query request, but its use for register and delete requests is limited. The fully qualified LU name of the user is automatically generated by ANAME during register and delete operations. This value can be overridden only by an ANAME administrator. For example, if a user other than the ANAME administrator issues this call to set the fully qualified LU name and then issues an `aname_register()` call, the register call will fail.

Format

```
ANAME_ENTRY
aname_set_fqlu_name(
    ANAME_HANDLE_TYPE      connection_id,
    unsigned char ANAME_PTR fqlu_name,
    ANAME_LENGTH_TYPE      fqlu_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

fqlu_name

(input) Value to be assigned to the fully qualified LU name field of the data record to be sent to the ANAME server.

Use the `ANAME_FQ_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

fqlu_name_length

(input) Length of the `fqlu_name` parameter that was supplied on this call.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

Example

```
{
    unsigned char      fqlu_name[ ANAME_FQ_SIZE + 1 ];
    ANAME_HANDLE_TYPE handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following call must be issued here:
     *   aname_create()
     */

    strcpy( fqlu_name, "NET.LUA );
    aname_set_fqlu_name(
        handle,
        fqlu_name,
        strlen( fqlu_name ),
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_set_group_name

Use this call to assign a value to the group name field of the client data object which will be sent to the ANAME server.

Format

```
ANAME_ENTRY
aname_set_group_name(
    ANAME_HANDLE_TYPE          connection_id,
    unsigned char ANAME_PTR    group_name,
    ANAME_LENGTH_TYPE         group_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

group_name

(input) Value to be assigned to the group name field of the data record to be sent to the ANAME server.

Use the `ANAME_GN_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

group_name_length

(input) Length of the `group_name` parameter that was supplied on this call.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

Example

```
{
    unsigned char          group_name[ ANAME_GN_SIZE + 1 ];
    ANAME_HANDLE_TYPE     handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following call must be issued here:
     *   aname_create()
     */

    strcpy( group_name, "GROUPA");
    aname_set_group_name(
        handle,
        group_name,
        strlen( group_name ),
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_set_tp_name

Use this call to assign a value to the transaction program name field of the client data object which will be sent to the ANAME server.

Note: This call does not set the TP name for the conversation. To set the TP name for the conversation use the `aname_set_destination()` call.

Format

```
ANAME_ENTRY
aname_set_tp_name(
    ANAME_HANDLE_TYPE          connection_id,
    unsigned char ANAME_PTR    tp_name,
    ANAME_LENGTH_TYPE         tp_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

tp_name

(input) Value to be assigned to the transaction program name field of the data record to be sent to the ANAME server.

Use the `ANAME_TP_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

tp_name_length

(input) Length of the `tp_name` parameter that was supplied on this call.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

Example

```
{
    unsigned char          tp_name[ ANAME_TP_SIZE + 1 ];
    ANAME_HANDLE_TYPE     handle;
    ANAME_RETURN_CODE_TYPE rc;

    /*
     * The following call must be issued here:
     *   aname_create()
     */

    strcpy( tp_name, "TPA");
    aname_set_tp_name(
        handle,
        tp_name,
        strlen( tp_name ),
        &rc );
}
```

Line Flows

There are no line flows for this call.

aname_set_trace_filename

Use this call to specify the name of the file to which trace output should be written. The trace filename will remain until the application ends or until the trace filename is reset to a different value with another `aname_set_trace_filename()` call. If trace is turned on by the `aname_set_trace_level()` call and the `aname_set_trace_filename()` call is not issued, the trace file generated will be:

- On VM: NUL TRC
- On MVS: DD:SYSOUT
- On OS/2: NUL.TRC

If the file does not exist, it will be created. If the file does exist, new data will be appended to the end of the file.

Format

```
ANAME_ENTRY
aname_set_trace_filename(
    unsigned char ANAME_PTR      trace_filename,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

trace_filename

(input) Name of the file to which trace information should be written.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

```
{
    ANAME_RETURN_CODE_TYPE rc;
    unsigned char trace_filename[ 256 + 1 ];

    /*
     * The following call should be issued here:
     *   aname_set_trace_level()
     */

    /*
     * The file name to use for the trace log will vary by operating
     * system, the following example is for OS/2
     */
    strcpy( trace_filename, "TRACE.LOG" );
    aname_set_trace_filename( trace_filename, &rc );
}
```

Line Flows

There are no line flows for this call.

aname_set_trace_level

Use this call to define the amount of trace information that should be generated while the program is running. This call can be issued at any time and does not depend on any previous calls. The trace level will remain until the application ends or until the trace level is reset to a different value with another `aname_set_trace_level()` call.

Use the `aname_set_trace_filename()` call to identify the file to use to capture the trace data.

Format

```
ANAME_ENTRY
aname_set_trace_level(
    ANAME_TRACE_LEVEL_TYPE      trace_level,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

trace_level

(input) The amount of trace information to be generated. The constants from `ANAME_LVL_NO_TRACING` to `ANAME_LVL_MAX_TRACE_LVL` incrementally increase the amount of trace information.

ANAME_LVL_NO_TRACING

No data will be written to the trace log.

ANAME_LVL_API

Traces crossings of the API boundary.

ANAME_LVL_MAX_TRACE_LVL

Provides the maximum amount of trace information.

Other trace levels are reserved for diagnosing problems with the assistance of the IBM Support Center.

return_code

(output) See “Chapter 6. ANAME Return Codes” on page 151 for possible return codes.

Example

```
{
    ANAME_TRACE_LEVEL_TYPE      trace_level = ANAME_LVL_API;
    aname_set_trace_level( trace_level, &rc );
}
```

Line Flows

There are no line flows for this call.

aname_set_user_name

Use this call to assign a value to the user name field of the client data object which will be sent to the ANAME server.

Format

```
ANAME_ENTRY
aname_set_user_name(
    ANAME_HANDLE_TYPE      connection_id,
    unsigned char ANAME_PTR user_name,
    ANAME_LENGTH_TYPE      user_name_length,
    ANAME_RETURN_CODE_TYPE ANAME_PTR return_code );
```

Parameters

connection_id

(input) An ANAME connection object that was created by a previous `aname_create()` call.

user_name

(input) Value to be assigned to the user name field of the data record to be sent to the ANAME server.

Use the `ANAME_UN_SIZE` constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

user_name_length

(input) Length of the `user_name` parameter that was supplied on this call.

return_code

(output) See "Chapter 6. ANAME Return Codes" on page 151 for possible return codes.

Example

```
{
    unsigned char      user_name[ ANAME_UN_SIZE + 1 ];
    ANAME_HANDLE_TYPE handle;
    ANAME_RETURN_CODE_TYPE rc;
    boolean           error_flag = FALSE;

    /*
     * The following call must be issued here:
     *   aname_create()
     */

    strcpy( user_name, "MYNAME" );
    aname_set_user_name(
        handle,
        user_name,
        strlen( user_name ),
        &rc );
}
```

Line Flows

There are no line flows for this call.

Chapter 6. ANAME Return Codes

These are the possible return codes that can be issued for each of the ANAME API calls.

ANAME_RC_BUFFER_TOO_SMALL

The buffer supplied by the caller for output data was too small to hold the data.

ANAME_RC_COMM_CONFIG_LOCAL

The call failed due to a local configuration error. Communications will fail until the configuration problem is resolved.

ANAME_RC_COMM_CONFIG_REMOTE

The call failed due to a remote configuration error. Communications will fail until the configuration problem is resolved.

ANAME_RC_COMM_FAIL_NO_RETRY

The call failed due to a communications problem. The call will not successfully complete using the current parameters.

ANAME_RC_COMM_FAIL_RETRY

The call failed due to a communications problem. The call may successfully complete if tried again.

ANAME_RC_FAIL_FATAL

A serious system error has occurred; no calls can complete successfully.

ANAME_RC_FAIL_INPUT_ERROR

The call may successfully complete after new input parameters are supplied.

ANAME_RC_FAIL_NO_RETRY

The call will not successfully complete using the current parameters.

ANAME_RC_FAIL_RETRY

The call may successfully complete if tried again.

ANAME_RC_HANDLE_NOT_VALID

The call failed because the ANAME connection object passed into the ANAME API was not valid.

ANAME_RC_OK

The call completed successfully.

ANAME_RC_PARAMETER_CHECK

The call failed due to an error in one of the parameters passed into the ANAME API.

ANAME_RC_PROGRAM_INTERNAL_ERROR

The call failed due to a programming error.

ANAME_RC_RECORD_ALREADY_OWNED

The call failed because the user name supplied already exists in the database in a record with a different LU name. The call might complete successfully if another user name is supplied.

ANAME_RC_SECURITY_NOT_VALID

The call failed because of APPC security.

ANAME_RC_STATE_CHECK

The call failed because the current ANAME API call was made when ANAME was not in the correct state required for the current call. For

example, you will get a state check error if you try to use `aname_format_error()` when no error has occurred.

Part 3. Appendixes

Appendix A. Entry Point Mappings

Table 5. AFTP API Call Mappings

Call Name	Entry Point
aftp_change_dir	ftcd
aftp_close	ftclose
aftp_connect	ftconn
aftp_create	ftcreate
aftp_create_dir	ftcrtdir
aftp_delete	ftdel
aftp_destroy	ftdestroy
aftp_dir_close	ftdircls
aftp_dir_open	ftdiropn
aftp_dir_read	ftdirrd
aftp_extract_allocation_size	fteas
aftp_extract_block_size	ftebs
aftp_extract_data_type	ftedt
aftp_extract_date_mode	ftedm
aftp_extract_destination	ftedst
aftp_extract_mode_name	ftemn
aftp_extract_partner_LU_name	fteplu
aftp_extract_password	ftepw
aftp_extract_record_format	fterf
aftp_extract_record_length	fterl
aftp_extract_security_type	ftest
aftp_extract_tp_name	ftetpn
aftp_extract_trace_level	ftetl
aftp_extract_userid	fteui
aftp_extract_write_mode	ftewm
aftp_format_error	ffe
aftp_get_data_type_string	ftgds
aftp_get_date_mode_string	ftgdms
aftp_get_record_format_string	ftgrfs
aftp_get_write_mode_string	ftgwms
aftp_load_ini_file	ftlif
aftp_local_change_dir	ftlcl
aftp_local_dir_close	ftlcl
aftp_local_dir_open	ftldo
aftp_local_dir_read	ftldr
aftp_local_query_current_dir	ftlqcd
aftp_query_bytes_transferred	ftqbt
aftp_query_current_dir	ftqcd
aftp_query_local_system_info	ftqlsi
aftp_query_local_version	ftqlv
aftp_query_system_info	ftqsys
aftp_receive_file	ftrecv
aftp_remove_dir	ftrd
aftp_rename	ftren
aftp_send_file	ftsend
aftp_set_allocation_size	ftsas
aftp_set_block_size	ftsbs
aftp_set_data_type	ftsdt
aftp_set_date_mode	ftsdm
aftp_set_destination	ftsdest
aftp_set_mode_name	ftsmn

Table 5. AFTP API Call Mappings (continued)

Call Name	Entry Point
aftp_set_password	ftsp
aftp_set_record_format	ftsrp
aftp_set_record_length	ftslr
aftp_set_security_type	ftsst
aftp_set_tp_name	ftstp
aftp_set_trace_filename	ftstf
aftp_set_trace_level	ftstl
aftp_set_userid	ftsud
aftp_set_write_mode	ftswm

Table 6. ANAME API Call Mappings

Function Name	Entry Point
aname_create	ancrt
aname_delete	andel
aname_destroy	andest
aname_extract_fqlu_name	anefq
aname_extract_group_name	anegn
aname_extract_tp_name	anetp
aname_extract_user_name	aneun
aname_format_error	anferr
aname_query	anqry
aname_receive	anrcv
aname_register	anreg
aname_set_destination	ansdest
aname_set_duplicate_register	ansdr
aname_set_fqlu_name	ansfq
aname_set_group_name	ansgn
aname_set_tp_name	anstp
aname_set_trace_filename	anstf
aname_set_trace_level	anstl
aname_set_user_name	ansun

Appendix B. Sample Program for AFTP API

```
/******  
*  
* PROGRAM:          AFTP Sample Code  
*  
* COPYRIGHTS:  
*   This module contains code made available by IBM  
*   Corporation on an AS IS basis. Any one receiving the  
*   module is considered to be licensed under IBM copyrights  
*   to use the IBM-provided source code in any way he or she  
*   deems fit, including copying it, compiling it, modifying  
*   it, and redistributing it, with or without  
*   modifications. No license under any IBM patents or  
*   patent applications is to be implied from this copyright  
*   license.  
*  
*   A user of the module should understand that IBM cannot  
*   provide technical support for the module and will not be  
*   responsible for any consequences of use of the program.  
*  
* Purpose : An example program to show a simple exercise of  
*           using the AFTP programming interface  
*  
* Function: Gets a single file from a remote machine. The user  
*           must know the machine name and the file name.  
*  
*  
*****/  
  
/******  
* System include files  
*****/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
/******  
* AFTP API include file  
*****/  
  
/*  
* NOTE: If you want to use the header file as it was shipped, change  
* this include statement to appfftp.h. Otherwise, rename the member  
* appfftp.h to aftpapi.h for consistency with other platforms.  
*/  
  
#include "aftpapi.h"  
  
int  
main( int argc, char *argv[] ) {  
  
    AFTP_HANDLE_TYPE connection_id;    /* connection id          */  
    AFTP_RETURN_CODE_TYPE aftp_rc;    /* return code            */  
    AFTP_SECURITY_TYPE sec_type;      /* security type          */  
    unsigned char * LU_name;          /* partner LU name       */  
    unsigned char * srcfilename;      /* source file name      */  
    unsigned char * destfilename;     /* destination file name */  
  
    printf( "\n" );  
  
    if( argc != 4 ) {  
  
        printf( "Usage : aget <LU name> <source filename> \" \  
                \" <destination filename> \n");  
  
    }  
}
```

```

    exit( -1 );
}

/*
 * NOTE: C/370 runtime treats backslashes as escape characters so
 * you would specify a backslash character by typing double
 * backslash characters.  Alternatively, you could define the
 * NO_ARGPARSE constant, use the runopts pragma, and parse the
 * arguments yourself.
 */

LU_name = argv[1];
srcfilename = argv[2];
destfilename = argv[3];

/*
 * Create the connection object
 */

aftp_create ( connection_id, &aftp_rc );

if ( aftp_rc != AFTP_RC_OK ) {

    printf ( "Error creating connection object.\n" );
    exit ( -1 );
}

/*
 * Set the partner LU name as the destination.
 */

aftp_set_destination (
    connection_id,
    (unsigned char AFTP_PTR)LU_name,
    (AFTP_LENGTH_TYPE)strlen ( LU_name ),
    &aftp_rc );

if( aftp_rc != AFTP_RC_OK ) {

    printf ( "Error setting the destination.\n" );
    exit ( -1 );
}

/*
 * Set the security to NONE unless you need security.
 */

aftp_set_security_type (
    connection_id,
    AFTP_SECURITY_NONE,
    &aftp_rc );

if ( aftp_rc == AFTP_RC_OK ) {

    printf ( "Setting security type to NONE.\n" );
}
else {

    printf ( "Error setting security type.\n" );
}

/*
 * Establish a connection with AFTP server.
 */

aftp_connect ( connection_id, &aftp_rc );

```

```

if ( aftp_rc != AFTP_RC_OK ) {
    printf ( "Error establishing the connection.\n" );
    exit ( -1 );
}

/*
 * Set up file transfer mode.
 */

aftp_set_write_mode (
    connection_id,
    AFTP_REPLACE,
    &aftp_rc );

if ( aftp_rc != AFTP_RC_OK ) {
    printf ( "Error setting write mode.\n" );
}

/*
 * Extract the security type and display it.
 */

aftp_extract_security_type (
    connection_id,
    &sec_type,
    &aftp_rc );

if ( aftp_rc == AFTP_RC_OK ) {
    printf ( "Security type is : %lu\n", sec_type );
}
else {
    printf ( "Error extracting security type.\n" );
}

/*
 * Transfer the file from the server to the client.
 */

aftp_receive_file (
    connection_id,
    (unsigned char AFTP_PTR)destfilename,
    (AFTP_LENGTH_TYPE)strlen ( destfilename ),
    (unsigned char AFTP_PTR)srcfilename,
    (AFTP_LENGTH_TYPE)strlen ( srcfilename ),
    &aftp_rc );

if ( aftp_rc == AFTP_RC_OK ) {
    printf ( "File successfully transfered.\n" );
}
else {
    /*
     * This is an example of how to show error reporting.
     */

    AFTP_LENGTH_TYPE return_length;
    char error_string[ AFTP_MESSAGE_SIZE ];

    printf ( "Error %lu transferring the file.\n", aftp_rc );
}

```

```

/*
 * Specify a detail level according to how much information you
 * want returned. In this case, return code information is
 * requested.
 */

aftp_format_error (
    connection_id,
    (AFTP_DETAIL_LEVEL_TYPE)AFTP_DETAIL_RC,
    (unsigned char AFTP_PTR)error_string,
    (AFTP_LENGTH_TYPE)( sizeof ( error_string )-1 ),
    &return_length,
    &aftp_rc );

/*
 * Add a null terminator.
 */

error_string[ return_length ] = '\0';
printf ( "%s", error_string );
}

/*
 * Close a connection with AFTP server.
 */

aftp_close ( connection_id, &aftp_rc );

if ( aftp_rc != AFTP_RC_OK ) {

    printf ( "Error closing the connection.\n" );
    exit ( -1 );
}

/*
 * Destroy the connection id.
 */

aftp_destroy ( connection_id, &aftp_rc );

if ( aftp_rc != AFTP_RC_OK ) {

    printf ( "Error destroying the connection id.\n" );
    exit ( -1 );
}

return( 0 );
} /* END SAMPLE PROGRAM */

```

Appendix C. Sample Program for ANAME API

```
/******  
*  
* PROGRAM:          APPC NameServer Sample code  
*  
* COPYRIGHTS:  
*   This module contains code made available by IBM  
*   Corporation on an AS IS basis. Any one receiving the  
*   module is considered to be licensed under IBM copyrights  
*   to use the IBM-provided source code in any way he or she  
*   deems fit, including copying it, compiling it, modifying  
*   it, and redistributing it, with or without  
*   modifications. No license under any IBM patents or  
*   patent applications is to be implied from this copyright  
*   license.  
*  
*   A user of the module should understand that IBM cannot  
*   provide technical support for the module and will not be  
*   responsible for any consequences of use of the program.  
*  
* Purpose : An example program to show a simple exercise of  
*           using the ANAME programming interface  
*  
* Function: Registers, Queries and Deletes a record using the ANAME  
*           API.  
*  
* Notes: The FQLU_NAME #define variable will work only if you are a  
*        system administrator. You MUST change this to be equal to  
*        your fully qualified LU name before you try running this or  
*        you will get an error message when you try running this  
*        program.  
*  
*        The user should also replace "SampleName", "SampleGroup",  
*        and "SampleTP" in the local type and constant definitions  
*        section below with a user name, group name and TP name.  
*  
*****/  
  
/******  
* System include files  
*****/  
  
#include <stdio.h>  
#include <string.h>  
  
/******  
* ANAME API include file  
*****/  
  
/*  
* NOTE: If you want to use the header file as it was shipped, change  
* this include statement to appmapi.h. Otherwise, rename the member  
* appmapi.h to anameapi.h for consistency with other platforms.  
*/  
  
#include "anameapi.h"  
  
/******  
* Local type and constant definitions  
*****/  
  
typedef unsigned short boolean;  
  
#ifndef TRUE  
#define TRUE ((boolean) (1))
```

```

#endif

#ifndef FALSE
#define FALSE ((boolean) (0))
#endif

/*****
* ### WARNING ### - You MUST change SAMPLE.LU to your fully-qualified
* LU name for this to work!
*****/

#define FQLU_NAME    "SAMPLE.LU"

#define USER_NAME    "SampleName" /* Set your User Name here      */
#define GROUP_NAME   "SampleGroup" /* Set your Group Name here   */
#define TP_NAME      "SampleTP"    /* Set your TP Name here     */

void
main(void) {

    /*
     * Buffers storing values to query
     */

    unsigned char          user_name[ ANAME_UN_SIZE + 1 ];
    unsigned char          fqlu_name[ ANAME_FQ_SIZE + 1 ];
    unsigned char          group[ ANAME_GN_SIZE + 1 ];
    unsigned char          tpname[ ANAME_TP_SIZE + 1 ];

    /*
     * Length of values entered by user
     */

    ANAME_LENGTH_TYPE     user_name_length = 0;
    ANAME_LENGTH_TYPE     fqlu_name_length = 0;
    ANAME_LENGTH_TYPE     group_length    = 0;
    ANAME_LENGTH_TYPE     tpname_length   = 0;

    /*
     * Buffer to store error string returned from aname_format_error
     */

    unsigned char          message[ 2048 ];
    ANAME_LENGTH_TYPE     message_len_in =
        (ANAME_LENGTH_TYPE)sizeof(message);
    ANAME_LENGTH_TYPE     message_len_out = 0;

    /*
     * Type of information requested from aname_format_error
     */

    ANAME_DETAIL_LEVEL_TYPE detail_level = ANAME_DETAIL_ALL;

    /*
     * Parameter required for all ANAME calls
     */

    ANAME_HANDLE_TYPE      handle;

    /*
     * Return Code from ANAME calls
     */

    ANAME_RETURN_CODE_TYPE rc;

    /*
     * Data indicator for aname_receive call.

```

```

    */
ANAME_DATA_RECEIVED_TYPE    data_received = 0;

/*
 * Flag indicating error condition.
 */

boolean                      error_flag = FALSE;

/*
 * Create ANAME handle.  If error, get error information
 * display to user.  Program ends on error.
 */

aname_create( handle, &rc );

if ( rc == ANAME_RC_OK ) {

    /*
     * Let's do a Register CALL, First well set up some fields
     */

    strcpy( user_name, USER_NAME);    /* Set User name          */
    strcpy( fqlu_name, FQLU_NAME);    /* Set FQLU name         */
    strcpy( group, GROUP_NAME);       /* Set Group name        */
    strcpy( tpname, TP_NAME);         /* Set TP name           */

    /*
     * Set the User_name, the Group name and the transaction
     * program name. Check the return code each step of the way.
     */

    aname_set_user_name(
        handle,
        user_name,
        strlen(user_name),
        &rc );

    if ( rc == ANAME_RC_OK ) {

        aname_set_group_name(
            handle,
            group,
            strlen(group),
            &rc );

        if ( rc == ANAME_RC_OK ) {

            aname_set_tp_name(
                handle,
                tpname,
                strlen(tpname),
                &rc );

            if ( rc == ANAME_RC_OK ) {

                aname_register( handle, &rc );

                if ( rc == ANAME_RC_OK ) {

                    /*
                     * Show what we have just registered
                     * successfully.
                     */

                    printf("\nRegister was successful for :\n");

```

```

        printf("\n User Name:  %s", user_name );
        printf("\n Group:      %s", group );
        printf("\n TP Name:    %s", tpname );
        printf("\n\n");

    } /* end if register succeeded */

} /* end if set tp name succeeded */

} /* end if set group name succeeded */

} /* end if set user name succeeded */

if ( rc != ANAME_RC_OK ) {

    error_flag = TRUE;

    aname_format_error(
        handle,
        detail_level,
        message,
        message_len_in,
        &message_len_out,
        &rc );

    if ( rc == ANAME_RC_OK ) { /* Success on format_error */

        printf("\n***** ");
        printf("ERROR MESSAGE");
        printf(" *****\n");
        printf("%s\n",message);
        printf( "\n*****");
        printf("*****\n\n");
    }
    else {

        printf("\naname_format_error failed. RC = %d\n",rc);
    }

    aname_destroy( handle, &rc );

} /* end if something failed after the create */

else {

    /*
    * IF the Register all WORKED Now lets do a Query to see
    * what was put into the Data Base. We DON'T have to do
    * any SET calls since we will be using the same "handle"
    * that was used in the register call and all the fields
    * are set the way we want them.
    */

    aname_query( handle, &rc );

    if ( rc == ANAME_RC_OK ) {

        /*
        * The Query was GOOD! Receive the data returned from
        * the NameServer. Extract the User_name, the FQLU_name,
        * the Group name and the TP_names so that you may
        * display them.
        */

        printf("\n***** Query was successful *****\n");

        do {

```

```

/*
 * Receive a record of data.
 */

aname_receive( handle, &data_received, &rc );

if ((rc == ANAME_RC_OK) &&
    (data_received == ANAME_DR_DATA_RECEIVED_OK)) {

    memset(user_name, 0, ANAME_UN_SIZE + 1);
    memset(fqlu_name, 0, ANAME_FQ_SIZE + 1);
    memset(group, 0, ANAME_GN_SIZE + 1);
    memset(tpname, 0, ANAME_TP_SIZE + 1);

    user_name_length = 0;
    fqlu_name_length = 0;
    group_length = 0;
    tpname_length = 0;

/*
 * Extract the user name.
 */

aname_extract_user_name(
    handle,
    user_name,
    ANAME_UN_SIZE,
    &user_name_length,
    &rc );

if ( rc == ANAME_RC_OK ) {

/*
 * Extract the fully-qualified LU name.
 */

aname_extract_fqlu_name(
    handle,
    fqlu_name,
    ANAME_FQ_SIZE,
    &fqlu_name_length,
    &rc );

if ( rc == ANAME_RC_OK ) {

/*
 * Extract the group name.
 */

aname_extract_group_name(
    handle,
    group,
    ANAME_GN_SIZE,
    &group_length,
    &rc );

if ( rc == ANAME_RC_OK ) {

/*
 * Extract the tp name.
 */

aname_extract_tp_name(
    handle,
    tpname,
    ANAME_TP_SIZE,

```

```

        &tpname_length,
        &rc );

if ( rc == ANAME_RC_OK ) {
    /*
     * Append nuls for printing
     */
    user_name[ user_name_length ] =
        '\0';
    fqlu_name[ fqlu_name_length ] =
        '\0';
    group[group_length] =
        '\0';
    tpname[ tpname_length ] =
        '\0';

    printf("\n");
    printf(
        "\n  User Name:   %s",
        user_name );
    printf(
        "\n  FQ LU Name:  %s",
        fqlu_name );
    printf(
        "\n  Group:       %s",
        group );
    printf(
        "\n  TP Name:    %s",
        tpname );

    } /* end if ext tp name succeeded */

    } /* end if ext group name succeeded */

    } /* end if extract fqlu name succeeded */

    } /* end if extract user name succeeded */

    } /* end if receive succeeded */

} while (((rc == ANAME_RC_OK) &&
        (data_received == ANAME_DR_DATA_RECEIVED_OK)) &&
        (!error_flag));

/*
 * Make sure we ended the LOOP OK with no more data
 * from the QUERY.
 */

if ( data_received == ANAME_DR_NO_MORE_DATA ) {

    printf( "\n*****");
    printf("*****\n\n");
    }
} /* End IF ok to receive data */

} /* end if query succeeded */

/*
 * Let's delete the record that we just have created if no
 * errors occurred previously. All the values are still
 * set in the connection object so we don't have to reset them.
 */

if ( !error_flag && rc == ANAME_RC_OK ) {

    aname_delete( handle, &rc );

```

```

if ( rc == ANAME_RC_OK ) {

    printf("\n***** Delete was successful for :");
    printf(" *****\n");
    printf("\n  User Name:  %s", user_name );
    printf("\n  FQ LU Name: %s", fqlu_name );
    printf("\n  Group:      %s", group );
    printf("\n  TP Name:    %s", tpname );
    printf( "\n*****\n");
    printf("*****\n\n");

}
else {

    printf("\naname_delete failed. ");
    printf("RC = %d\n",rc);
    error_flag = TRUE;
}
}

/*
 * Before leaving: If there was an error display error info
 * to the user.
 */

if ( error_flag ) {

    aname_format_error(
        handle,
        detail_level,
        message,
        message_len_in,
        &message_len_out,
        &rc );

    if ( rc == ANAME_RC_OK ) {
        printf("\n***** ERROR MESSAGE ");
        printf("*****\n");
        printf("%s\n",message);
        printf( "\n*****\n");
        printf("*****\n\n");
    }
    else {

        printf("\naname_format_error failed with an RC = %d\n",
            rc);
    }
}

/* End If: Everything is still OK */

/*
 * Destroy the handle created. If error, display error info
 * to the user. Program ends on error.
 */

aname_destroy( handle, &rc );

if ( rc != ANAME_RC_OK ) {

    printf("\naname_destroy failed. RC = %d\n",rc );

    aname_format_error(
        handle,
        detail_level,
        message,
        message_len_in,
        &message_len_out,

```

```

        &rc );

    if ( rc == ANAME_RC_OK ) {
        printf("\n***** ERROR MESSAGE ");
        printf("*****\n");
        printf("%s\n",message);
        printf( "*****");
        printf("*****\n\n");
    }
    else {

        printf("\naname_format_error failed. RC = %d\n",rc);
    }
}

} /* End if: aname_create worked */

return;
} /* END SAMPLE PROGRAM */

```

Appendix D. Information Apars

This appendix lists information apars for IP and SNA books.

Notes:

1. Information apars contain updates to previous editions of the manuals listed below. Books updated for V1R2 are complete except for the updates contained in the information apars that may be issued after V1R2 books went to press.
2. Information apars are predefined for z/OS V1R2 Communications Server and may not contain updates.

IP Information Apars

Table 7 lists information apars for IP books.

Table 7. IP Information Apars

Title	z/OS CS V1R2	CS for OS/390 2.10 and z/OS CS V1R1	CS for OS/390 2.8	CS for OS/990 2.7	CS for OS/390 2.6	CS for OS/390 2.5
IP API Guide	ii12861	ii12371	ii11635	ii11558	ii11405	ii11144
IP CICS Sockets Guide	ii12862		ii11626	ii11559	ii11406	ii11145
IP Configuration			ii11620 ii12068 ii12353 ii12649	ii11555 ii11637 ii11995 ii12325	ii11402 ii11619 ii12066 ii12455	ii11159 ii11979 ii12315
IP Configuration Guide	ii12498	ii12362 ii12493				
IP Configuration Reference	ii12499	ii12363 ii12494 ii12712				
IP Diagnosis	ii12503	ii12366 ii12495	ii11628	ii11565	ii11411	ii11160 ii11414
IP Messages Volume 1	ii12857	ii12367	ii11630	ii11562	ii11408	ii11636
IP Messages Volume 2	ii12858	ii12368	ii11631	ii11563	ii11409	ii11281
IP Messages Volume 3	ii12859	ii12369	ii11632 ii12883	ii11564 ii12884	ii11410 ii12885	ii11158
IP Messages Volume 4	ii12860					
IP Migration	ii12497	ii12361	ii11618	ii11554	ii11401	ii11204
IP Network Print Facility	ii12864		ii11627	ii11561	ii11407	ii11150
IP Programmer's Reference	ii12505		ii11634	ii11557	ii11404	ii12496

Table 7. IP Information Apars (continued)

Title	z/OS CS V1R2	CS for OS/390 2.10 and z/OS CS V1R1	CS for OS/390 2.8	CS for OS/990 2.7	CS for OS/390 2.6	CS for OS/390 2.5
IP and SNA Codes	ii12504	ii12370	ii11917	Added TCP/IP codes to VTAM codes V2R6 ii11611	ii11361	ii11146 ii11097
IP User's Guide		ii12365	ii11625	ii11556	ii11403	ii11143
IP User's Guide and Commands	ii12501					
IP System Admin Guide	ii12502					
Quick Reference	ii12500	ii12364				

SNA Information Apars

Table 8 lists information apars for SNA books.

Table 8. SNA Information Apars

Title	z/OS CS V1R2	CS for OS/390 2.10 and z/OS CS V1R1	CS for OS/390 2.8	CS for OS/390 2.7	CS for OS/390 2.6	CS for OS/390 2.5
Anynet SNA over TCP/IP			ii11922	ii11633	ii11624	ii11623
Anynet Sockets over SNA			ii11921	ii11622	ii11519	ii11518
CSM Guide						
IP and SNA Codes		ii12370	ii11917	ii11611	ii11361	ii11097
SNA Customization	ii12872	ii12388	ii11923	ii11925 ii12008	ii11924 ii12007	ii11092 ii11621 ii12006
SNA Diagnosis	ii12490	ii12389	ii11915	ii11615	ii11357	ii11585
SNA Messages	ii12491	ii12382	ii11916	ii11610	ii11358	ii11096
SNA Network Implementation Guide	ii12487	ii12381	ii11911	ii11609 ii12683	ii11353 ii11493	ii11095
SNA Operation	ii12489	ii12384	ii11914	ii11612	ii11355	ii11098
SNA Migration	ii12486	ii12386	ii11910	ii11614	ii11359	ii11100
SNA Programming		ii12385	ii11920	ii11613	ii11360	ii11099
Quick Reference	ii12500	ii12364	ii11913	ii11616	ii11356	
SNA Resource Definition Reference	ii12488	ii12380 ii12567	ii11912 ii12568	ii11608 ii12569	ii11354 ii12259 ii12570	ii11094 ii11151 ii12260 ii12571
SNA Resource Definition Samples						

Appendix E. Notices

IBM may not offer all of the products, services, or features discussed in this document. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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:

Site Counsel
IBM Corporation
P.O.Box 12195
3039 Cornwallis Road
Research Triangle Park, North Carolina 27709-2195
U.S.A

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly

tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

This product includes cryptographic software written by Eric Young.

If you are viewing this information softcopy, photographs and color illustrations may not appear.

You can obtain softcopy from the z/OS Collection (SK3T-4269), which contains BookManager and PDF formats of unlicensed books and the z/OS Licensed Product Library (LK3T-4307), which contains BookManager and PDF formats of licensed books.

Trademarks

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

ACF/VTAM	Micro Channel
Advanced Peer-to-Peer Networking	MVS
AFP	MVS/DFP
AD/Cycle	MVS/ESA
AIX	MVS/SP
AIX/ESA	MVS/XA
AnyNet	MQ
APL2	Natural
APPN	NetView
AS/400	Network Station
AT	Nways
BookManager	Notes
BookMaster	NTune
CBPDO	NTuneNCP
C/370	OfficeVision/MVS
CICS	OfficeVision/VM
CICS/ESA	Open Class
C/MVS	OpenEdition
Common User Access	OS/2
C Set ++	OS/390
CT	OS/400
CUA	Parallel Sysplex
DATABASE 2	Personal System/2
DatagLANce	PR/SM
DB2	PROFS
DFSMS	PS/2
DFSMSdfp	RACF
DFSMSHsm	Resource Link
DFSMS/MVS	Resource Measurement Facility
DPI	RETAIN
Domino	RFM
DRDA	RISC System/6000
eNetwork	RMF
Enterprise Systems Architecture/370	RS/6000
ESA/390	S/370
ESCON	S/390
@server	SAA
ES/3090	SecureWay
ES/9000	Slate
ES/9370	SP
EtherStreamer	SP2
Extended Services	SQL/DS
FAA	System/360

FFST	System/370
FFST/2	System/390
FFST/MVS	SystemView
First Failure Support Technology	Tivoli
GDDM	TURBOWAYS
Hardware Configuration Definition	UNIX System Services
IBM	Virtual Machine/Extended Architecture
IBMLink	VM/ESA
IBMLINK	VM/XA
IMS	VSE/ESA
IMS/ESA	VTAM
InfoPrint	WebSphere
Language Environment	XT
LANStreamer	z/Architecture
Library Reader	z/OS
LPDA	zSeries
MCS	400
	3090
	3890

Lotus, Freelance, and Word Pro are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Tivoli and NetView are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

DB2 and NetView are registered trademarks of International Business Machines Corporation or Tivoli Systems Inc. in the U.S., other countries, or both.

The following terms are trademarks of other companies:

ATM is a trademark of Adobe Systems, Incorporated.

BSC is a trademark of BusiSoft Corporation.

CSA is a trademark of Canadian Standards Association.

DCE is a trademark of The Open Software Foundation.

HYPERchannel is a trademark of Network Systems Corporation.

UNIX is a registered trademark in the United States, other countries, or both and is licensed exclusively through X/Open Company Limited.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ActionMedia, LANDesk, MMX, Pentium, and ProShare are trademarks of Intel Corporation in the United States, other countries, or both. For a complete list of Intel trademarks, see <http://www.intel.com/tradmarx.htm>.

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

Index

A

AFTP API

- compiling
 - MVS 5, 119
 - VM 5, 119
- conventions 4
- defined constants 3
- description 3
- sample program 157
- standard types 3

AFTP API Calls

- aftp_change_dir 14
- aftp_close 16
- aftp_connect 17
- aftp_create 18
- aftp_create_dir 19
- aftp_delete 21
- aftp_destroy 23
- aftp_dir_close 24
- aftp_dir_open 25
- aftp_dir_read 27
- aftp_extract_allocation_size 30
- aftp_extract_block_size 31
- aftp_extract_data_type 32
- aftp_extract_date_mode 33
- aftp_extract_destination 34
- aftp_extract_mode_name 36
- aftp_extract_partner_LU_name 38
- aftp_extract_password 40
- aftp_extract_record_format 42
- aftp_extract_record_length 44
- aftp_extract_security_type 45
- aftp_extract_tp_name 47
- aftp_extract_trace_level 49
- aftp_extract_userid 50
- aftp_extract_write_mode 52
- aftp_format_error 53, 55
- aftp_get_date_mode_string 57
- aftp_get_record_format_string 59
- aftp_get_write_mode_string 61
- aftp_load_ini_file 63
- aftp_local_change_dir 65
- aftp_local_dir_close 67
- aftp_local_dir_open 68
- aftp_local_dir_read 70
- aftp_local_query_current_dir 73
- aftp_query_bytes_transferred 75
- aftp_query_current_dir 76
- aftp_query_local_system_info 78
- aftp_query_local_version 80
- aftp_query_system_info 81
- aftp_receive_file 83
- aftp_remove_dir 85
- aftp_rename 87
- aftp_send_file 89
- aftp_set_allocation_size 91
- aftp_set_block_size 92

AFTP API Calls (*continued*)

- aftp_set_data_type 93
- aftp_set_date_mode 95
- aftp_set_destination 96
- aftp_set_mode_name 98
- aftp_set_password 99
- aftp_set_record_format 101
- aftp_set_record_length 103
- aftp_set_security_type 104
- aftp_set_tp_name 106
- aftp_set_trace_filename 108
- aftp_set_trace_level 109
- aftp_set_userid 110
- aftp_set_write_mode 112
- connection object 6
- controlling trace information 10
- entry point mappings 155
- generating message strings 10
- listing files
 - on client 9
 - on server 8
- loading initialization file 11
- managing connections 6
- managing directories 9
- managing files 10
- overview 5
- query connections 7
- query file transfer characteristics 8
- query system 10
- Return Codes 113
- specifying file transfer characteristics
 - binary 7
 - text 7
- transferring files 7
- aftp_change_dir 14
- aftp_close 16
- aftp_connect 17
- aftp_create 18
- aftp_create_dir 19
- aftp_delete 21
- aftp_destroy 23
- aftp_dir_close 24
- aftp_dir_open 25
- aftp_dir_read 27
- aftp_entry 5
- aftp_extract_allocation_size 30
- aftp_extract_block_size 31
- aftp_extract_data_type 32
- aftp_extract_date_mode 33
- aftp_extract_destination 34
- aftp_extract_mode_name 36
- aftp_extract_partner_LU_name 38
- aftp_extract_password 40
- aftp_extract_record_format 42
- aftp_extract_record_length 44
- aftp_extract_security_type 45
- aftp_extract_tp_name 47
- aftp_extract_trace_level 49

- aftp_extract_userid 50
- aftp_extract_write_mode 52
- aftp_format_error 53, 55
- aftp_get_date_mode_string 57
- aftp_get_record_format_string 59
- aftp_get_write_mode_string 61
- aftp_load_ini_file 63
- aftp_local_change_dir 65
- aftp_local_dir_close 67
- aftp_local_dir_open 68
- aftp_local_dir_read 70
- aftp_local_query_current_dir 73
- aftp_ptr 5
- aftp_query_bytes_transferred 75
- aftp_query_current_dir 76
- aftp_query_local_system_info 78
- aftp_query_local_version 80
- aftp_query_system_info 81
- aftp_receive_file 83
- aftp_remove_dir 85
- aftp_rename 87
- aftp_send_file 89
- aftp_set_allocation_size 91
- aftp_set_block_size 92
- aftp_set_data_type 93
- aftp_set_date_mode 95
- aftp_set_destination 96
- aftp_set_mode_name 98
- aftp_set_password 99
- aftp_set_record_format 101
- aftp_set_record_length 103
- aftp_set_security_type 104
- aftp_set_tp_name 106
- aftp_set_trace_filename 108
- aftp_set_trace_level 109
- aftp_set_userid 110
- aftp_set_write_mode 112
- ANAME API
 - conventions 118
 - defined constants 117
 - overview 117
 - sample program 161
 - standard types 118
- ANAME API Calls
 - accessing returned records 122
 - aname_create 124
 - aname_delete 125
 - aname_destroy 126
 - aname_extract_fqlu_name 127
 - aname_extract_group_name 129
 - aname_extract_tp_name 131
 - aname_extract_user_name 133
 - aname_format_error 135
 - aname_query 137
 - aname_register 139
 - aname_set_destination 140
 - aname_set_duplicate_register 141
 - aname_set_fqlu_name 143
 - aname_set_group_name 145
 - aname_set_tp_name 146
 - aname_set_trace_filename 147

- ANAME API Calls (*continued*)
 - aname_set_trace_level 148
 - aname_set_user_name 149
 - connection object 120
 - database records
 - adding 121
 - obtaining 121
 - removing 121
 - entry point mappings 156
 - obtaining error information 122
 - overview 120
 - Return Codes 151
 - setting data object values 120
 - setting destination name 120
 - system administrator functions 122
 - tracing 122
- aname_create 124
- aname_delete 125
- aname_destroy 126
- aname_extract_fqlu_name 127
- aname_extract_group_name 129
- aname_extract_tp_name 131
- aname_extract_user_name 133
- aname_format_error 135
- aname_query 137
- aname_register 139
- aname_set_destination 140
- aname_set_duplicate_register 141
- aname_set_fqlu_name 143
- aname_set_group_name 145
- aname_set_tp_name 146
- aname_set_trace_filename 147
- aname_set_trace_level 148
- aname_set_user_name 149
- APPC File Transfer Protocol 3

C

- Call Reference, API 123

N

- null-terminated strings 4

R

- Return Codes 151

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

**z/OS Communications Server
APPC Application Suite Programming
Version 1 Release 2**

Publication No. SC31-8834-00

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

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>				

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

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>				
Complete	<input type="checkbox"/>				
Easy to find	<input type="checkbox"/>				
Easy to understand	<input type="checkbox"/>				
Well organized	<input type="checkbox"/>				
Applicable to your tasks	<input type="checkbox"/>				

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.



Fold and Tape

Please do not staple

Fold and Tape



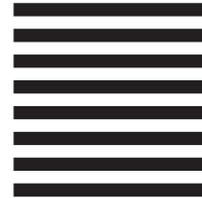
NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

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

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Software Reengineering
Department G71A/ Bldg 503
Research Triangle Park, NC
27709-9990



Fold and Tape

Please do not staple

Fold and Tape



Program Number: 5694-A01



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

SC31-8834-00



Spine information:



z/OS Communications Server

z/OS VIR2.0 CS: APPC Application Suite
Programming

Version 1
Release 2