QVS Distributed Data
Services/Controller
Services for Windows

Programming Guide

Version 3

First Edition (March 2001)

This edition applies to Version 3 of the QVS Distributed Data Services/Controller Services for Windows, and to all subsequent releases and modifications until otherwise indicated in new editions.

First Edition (November 2004)
Updated March 17, 2006

This edition applies to **Version 3** of the QVS 4690 QVS Distributed Data Services/ Controller Services Feature (**DDS/CSF**) for Windows operating systems and to all subsequent releases and modifications until otherwise indicated in new editions.

Download publications from www.qvssoftware.com. Email comments to webmaster@qvssoftware.com or address your comments to:

QVS Software, Inc.
C/o Publications
5711 Six Forks Rd. Suite 300
Raleigh, NC 27609
USA

Table of Contents

Preface

This manual explains how to use the application programming interfaces (APIs) provided with Distributed Data Services (DDS) to develop distributed applications.

Who Should Read this Manual

This manual is primarily for retail systems programmers who are programming using DDS/CSF on the Windows operating systems.

This manual assumes that readers are familiar with the Windows operating systems and are proficient in C language programming.

How This Manual Is Organized

This manual is separated into eight chapters and three appendixes:

Syntax Conventions

The syntax of DDS command line commands is shown using graphic notation consisting of a statement that is tailored to the parameter requirements of each command. To read the diagrams, follow the main path line and move from left to right and from top to bottom.

Syntax diagrams use symbols to identify the sequence of information:

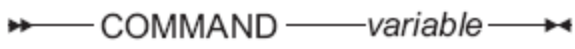
- A command statement begins with: and ends with:

- A command statement longer than one line continues to a second line with:

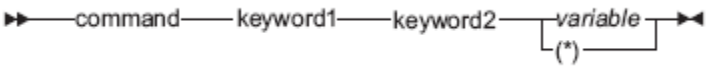
- where it resumes with:

Required Parameters

A parameter that you must include is displayed on the main path line:



If a command statement has two or more required parameters, they are shown consecutively on the main path line. A choice of required parameters is shown with one choice on the main path line and the other choices on branch lines below the first choice:

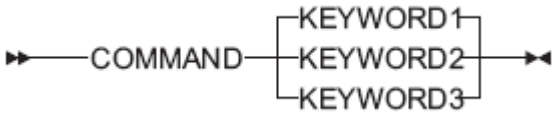


Type this command in one of two ways:

command keyword1 keyword2 (*variable*)
command keyword1 keyword2 (*)

Default Parameters

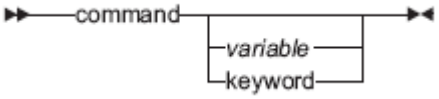
A default parameter is shown on a branch line above the main path line:



keyword1 is the default.

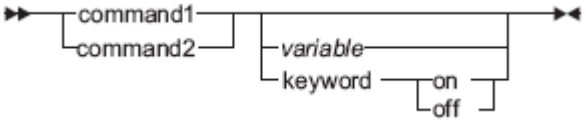
Optional Parameters

Parameters that you can include are shown on branch lines below the main path line:



Type this command in one of the following ways:
command
command *variable*
command keyword

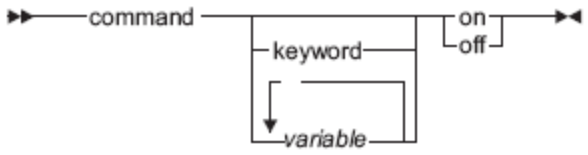
Branch lines can include branch lines of their own:



If you include the **keyword** parameter in this statement, you must also include **on** or **off**.

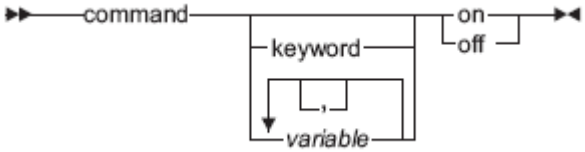
Repeating Parameters

An arrow on a line above a parameter menas that you can repeat the parameter, or enter more than one of the listed parameters:



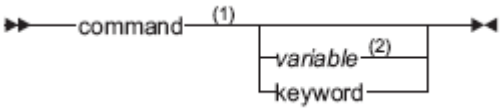
The arrow above **variable** means that you can include one or more values when you type **command**. The diagram indicates that a blank space is required between each **variable** value.

For commands that have optional separators between repeated values of a **variable**:



The arrow above **variable** means that you can include one or more values when you type **command**. This diagram indicates that a comma can optionally be placed between each **variable** value.

If a syntax diagram contains notes, the note numbers correspond to numbered elements shown in the diagram within parentheses:



where:
1. This is a syntax note that refers to the keyword **command**.
2. This syntax note describes **variable**.

Related Publications

In addition to this manual, you may want to consult the following publications.

- **QVS Distributed Data Services/Controller Services for Windows Installation and Configuration Guide.**
- *QVS Distributed Data Services/Controller Services for Windows User's* **Guide**

All of these publications may be downloaded from the QVS web site at www.qvssoftware.com. You may also call QVS at (919) 676-1991 for assistance.

Chapter 1. System Overview

The IBM Distributed Data Services/Controller Services Feature (DDS/CSF) is a distributed software platform designed as a base for the development of distributed applications for the store environment.

The main functions are:

- Interprocess communication with local and remote transparency
- Data access with local and remote transparency
- Data distribution for redundancy, performance, and availability
- Directory services
- Installation, configuration, and administration

The following sections provide an overview of the concepts that are common to all components of DDS:

- “Nodes”
- “Logical Names and Role Names”
- “Broadcast Domains”
- “Distribution Domains and Roles”
- “File Names and Queue Names”
- “Components”
- “File Distribution”
- “Disk I/O Prioritization”

Nodes

A **node** is a LAN-attached machine that is running DDS.

Nodes can be connected via LANs to form a **system.** A system is the group of nodes for which files are managed. A node can be connected to a subset of nodes via one LAN and to other subsets of nodes via different LANs. Data distribution, remote file access, and interprocess communication are supported only between nodes that are connected by a LAN. DDS does not provide a bridge or router function.

For all system topologies with more than one node, one node must be installed and configured as the configured primary distributor and another node can be installed and configured as the configured backup distributor. The backup distributor is required to add redundancy to the critical store data. Each node must be connected to the primary distributor and backup distributor by a LAN. See “Distribution Domains and Roles” for an explanation of primary distributor, backup

distributor, and subordinate.

Node IDs

Node IDs are specified for each workstation (node) during the installation of DDS, and assumed each time DDS initializes. Each node has a single node ID, regardless of how many LANs are used to connect it to other nodes.

The rules for node IDs are:

- Each workstation within a system must have a different node ID. Duplicate nodeIDs are not confirmed during installation, but they are detected when DDS initializes. If duplicate node IDs are detected, you must reinstall DDS to correct the problem.
- No two workstations on a LAN can have the same node ID even if they are members of different DDS systems.
- Node IDs can be from 1 to 8 alphanumeric characters (blanks are not allowed). Node IDs are case-sensitive.
- Do not use node IDs whose first three characters are FDS. These names are reserved.
- Greater than and less than characters (< and >), question marks (?), asterisks (*), and colons (:) are not valid characters for node IDs.
- After installation, node IDs can be changed only by reinstalling DDS.

System ID

Each system has a 4-byte system ID. The system ID is specified at each node during the installation of DDS, and defaults to 0000. The Name Services component uses the system ID as a qualifier when locating the node that has assumed a particular role. See “Logical Names and Role Names” for a description of roles. This ensures, for example, that each node finds the acting primary distributor for its system, and that files are distributed only within a system. The system ID must be set to a unique value for each system in environments where multiple systems are interconnected via bridges. A system ID cannot be changed without reinstalling DDS.

If you assign a different system ID to each system, the systems can be connected using a LAN or a gateway without causing messages intended for one system to be sent to another system.

If you assign the same system ID to each system, the LANs can be connected using a gateway; both LANs are considered the same system. If this type of system is used and you are using the Data Distribution component, there should

only be one configured primary distributor and one configured backup distributor for the entire system, even though they might be on different LANs.

Logical Names and Role Names

DDS provides a name-resolution capability, allowing applications to use logical names instead of hard-coded file names, interprocess communication (IPC) queue names, and node IDs. These logical names are dynamically resolved when the application runs.

Some names are fairly static over time. An example is the name of a configuration file. Although this name is not likely to change very often, if ever, it is still desirable to avoid using the name in an application program. The use of a logical name allows the file name to be changed without having to rebuild the application. A logical name has the following format:

<*name*>

Where **name** is 1 to 260 characters and the less than and greater than characters (< and >) are required delimiters.

Other names are more dynamic, such as the node ID of the primary distributor. This changes whenever the backup distributor takes over for the primary distributor. In this case a **role** (the primary distributor) is assumed by a particular node. A logical name can be used to identify this role and is referred to as the role name. A role name has the following format:

<*name::*>

Where **name** is 1 to 8 characters, the less than and greater than characters (< and >) are required delimiters, and double colons (::) indicate that this is a role name.

The use of a role name makes it easy for an application to open a file or IPC queue on a node that provides a particular service when the service may move from node to node as conditions change. The primary distributor is an example of a service provided by DDS. Applications can be written that provide other services and role names can be defined for them.

Note: Hard links cannot be used for distribution names.

Reserved Role Names

The following role names are reserved by DDS and are used to identify the primary distributor and backup distributor nodes. These names are dynamically maintained by DDS and cannot be modified by the user.

Role Name
Reserved For:

FDSFDXCP::
Configured primary distributor

FDSFDXCB::
Configured backup distributor

FDSFDXAP::
Acting primary distributor

FDSFDXAB::
Acting backup distributor

Broadcast Domains

A subset of nodes within a system can be grouped into a **broadcast domain**.A broadcast domain has a name, the broadcast domain name, which has the same format as a node ID: 1 to 8 bytes. However, DDS reserves all broadcast domain names that begin with the prefix FDS.

Note: DDS currently supports a maximum of one broadcast domain.

Broadcast domains are useful for maximizing performance and minimizing resource utilization when distributing messages or files to a large number of nodes. Within a broadcast domain, DDS exploits the LAN hardware's ability to broadcast a **datagram** to all nodes. The DDS Data Distribution component supports distributing files to all nodes within a broadcast domain. The Interprocess Communications component can send a message to every node within a broadcast domain.

Distribution Domains and Roles

Each node can assume a distribution role. There are three possible distribution roles:

Primary Distributor
The primary distributor controls the primary copy of all distributed files. Only the primary copy of a file can be modified directly by an application.

Backup Distributor
The backup distributor controls the backup copy of a file and can take over for the primary distributor if the primary distributor fails or is deactivated.

Subordinate
All other nodes configured with the Data Distribution component are considered subordinates and manage image copies of distributed files.

A node can be configured with a distribution role, but under certain circumstances can assume another role. In that instance, the node is said to be acting the role. Specifically, when a node that is configured as the backup distributor assumes the role of the primary distributor that has failed, the backup distributor is said to be the acting primary distributor.

A group of nodes form a distribution domain. There are two types of distribution domains:

Mirrored domain
Defined to be the primary distributor and backup distributor. There is only one mirrored domain, so it is not named.

Broadcast domain

A broadcast domain can include zero (0) or more nodes. Files that are distributed to a broadcast domain are distributed to each node in the domain, as well as to the acting backup distributor.

A given file can be distributed to only one domain.

By default, all files on a node are local files. A local file is a file that is not distributed (is not a primary copy, backup copy, or image copy). A file or subdirectory on the acting primary distributor can be made distributed using either the Data Distribution Utility or an application that calls the FdsSetDistribution() API. See the **IBM Distributed Data Services/Controller** *Services Feature for Windows User's* **Guide** for more information about using the Data Distribution Utility. See FdsSetDistribution() for more information about using the API.

File Names and Queue Names

For a specific node, you can identify each file using the operating system path where the file is located and the file name. The operating system path and file name are called the **file specification**. Similarly, you can identify a queue using the queue name.

However, with DDS you can access files and queues on any node. Therefore, you must use a retail path specification to identify the file or queue. A **retail path specification** contains a node specification or broadcast domain specification prefixed to the file specification or queue name.

Note: If you do not include a node or broadcast domain specification in the retail path specification, Distributed Data Services assumes that the file or queue resides locally.

A node specification can be in one of these forms:

- A node name followed by two colons, which specifies the ID of the node directly. For example:

  NodeID::

- A role name followed by two colons and delimited with the greater than and less than signs. The role name will resolve to the ID of the node that is acting in the role. For example:

  <RoleName::>

A broadcast domain specification is valid only to identify a queue. It must be a string, containing no blank characters, that includes a broadcast domain name followed by two colons. For example:

B_DOMAIN::

Assume that you have a file called MYFILE.DAT located in the subdirectory D:\FILES on the primary workstation with a node ID of Node1. The following retail path specifications are valid:

D:\FILES\MYFILE.DAT
Node1::D:\FILES\MYFILE.DAT
<FDSFDXAP::>D:\FILES\MYFILE.DAT

You can also use logical names for any part (or all) of a retail path specification. See “Logical Names and Role Names” for more information about logical names.

Note: If you use a logical name for a file name or queue name in a retail path specification that contains a remote node ID or role name, the logical name is resolved on the remote node.

Components

The components for DDS are:

Name Services

Provides a name-resolution capability. This allows applications to use logical names, or aliases, instead of hard-coded file names, IPC queue names, and node IDs. These logical names are dynamically resolved at run time.

See “Chapter 7. Name Services” for more information.

File Services

Allows you to access both local and remote files. It can be optionally configured on zero or more nodes to share files with other nodes.

Three types of files are supported:

- Keyed files
- Sequential files
- Binary files

See Chapter 4. File Services for more information.

File System Interface

Provides support to distribute native operating system files, referred to as byte stream files.

This component also provides a disk I/O prioritization mechanism that overrides the standard operating system prioritization scheme. DDS prioritizes disk I/O based on thread priority, allowing high priority requests, such as price lookups, to be processed ahead of lower priority requests.

Interprocess Communications

Provides a peer-to-peer messaging service that allows application programs to send and receive messages. The messaging service is provided between processes running on a single node (intranode IPC) and on different nodes (internode IPC). The internode IPC function is a configuration option. It is required on all nodes in a system unless the system consists of a single standalone node.

The following LAN media are supported:

- Ethernet
- Token ring
- IBM Wireless LAN

Note: If you install the DDS 4690 Controller Services Feature, DDS also provides store loop support.

Data Distribution

The Data Distribution component provides a distributed file capability that replicates data to multiple nodes, keeping each image synchronized during normal operations. It also performs reconciliation when failed nodes are brought back into service. See Chapter 6. Data Distribution for more information.

Node Control

This component allows you to perform administrative functions such as viewing information about the nodes within the DDS system and activating or deactivating the primary distributor. These aspects of node control are used through a utility. Node control is also responsible for synchronizing the time and date of all nodes within a system.

There is also an API that generates a list of all node IDs known to the DDS system, including nodes that DDS has detected as being active on the system and user-defined nodes that are not yet active.

See the **IBM Distributed Data Services/Controller Services Feature** *for Windows User's Guide* for more information about starting and stopping DDS and using the Node Control Utility. See "Node List" for more information about using the node list API.

Problem Determination and Analysis

The Problem Determination and Analysis component collects problem determination information. The information is presented on an interactive panel that allows you to select the system message logs, system error logs, and system dump files you wish to work with.

4690 Controller Services

This optional feature can be installed on one or two nodes. It supports 4680/4690 Operating System controller applications running under Windows NT, 2000, XP, or Server 2003 and the attachment of registers running the 4690 Operating System. Refer to the **IBM Distributed Data** *Services/Controller Services Feature for Windows User's Guide* for more information about 4690 Controller Services.

4690 Multiple Controller Feature

This optional feature can be installed on zero or more nodes. It provides support for 4690 Controller Services Feature nodes to interact with other 4690 Controller Services Feature nodes. It also provides support for 4690 terminal backup in a multi-node environment. Refer to the **IBM Distributed Data Services/Controller** *Services Feature for Windows User's Guide* for more information about 4690 Multiple Controller Feature.

File Distribution

DDS enables the distribution of files to other nodes in a distribution domain. When a file operation is directed to a controlled drive, DDS determines if the file has been defined as a distributed file or is in a distributed subdirectory, and then distributes the file operation as appropriate to other nodes.

File distribution is performed with no user intervention. Any operating system command (for example, **COPY** or **ERASE**) or application program statement that results in the modification of a distributed file causes DDS to distribute the operation to other nodes.

DDS must be running to detect whether a file is distributed. Until DDS is started, it assumes that all files on a controlled drive are distributed. Therefore, any attempt to modify a file on a controlled drive when DDS is not running results in an error.

Note: If the **DDActive** configuration keyword is set to NO, DDS does not check to see if files on controlled drives are distributed, so this error is not returned. See the Configuration Keywords chapter in the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about the **DDActive** configuration keyword.

See Chapter 6. Data Distribution for more information about distributing files. Refer to the Planning for DDS chapter in the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about controlled drives.

Disk I/O Prioritization

Processing certain file I/O operations is extremely time critical in the retail environment. For example, when scanning items at the point-of-sale terminal, the salesperson expects a consistent response time. If the processing load on the disk used to service price lookup requests increases, the response time experienced by salespeople scanning items at the point-of-sale terminals should remain relatively constant.

DDS uses a disk I/O prioritization scheme to assure that time-critical processes are given highest priority for disk access. This prioritization scheme overrides the standard operating system prioritization scheme, so that file access is granted based on the operating system priority of the process thread issuing the request.

To take advantage of disk I/O prioritization, an application program can increase the priority level of time-critical threads using the SetThreadPriority() API on Windows NT, 2000, XP, or Server 2003.

Disk I/O prioritization is most effective when all partitions on a single physical disk are controlled by DDS. This should be considered when configuring partitions as controlled drives. Refer to the Planning for DDS chapter in the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about controlled drives.

Chapter 2. Introduction to the API

The DDS application programming interface (API) is a collection of individual functions (also referred to as APIs) that you can use to enable your applications to interact with DDS.

Your application can be one of many applications that concurrently uses DDS. This chapter describes the general interaction between your application and DDS.

C Language Header Files

DDS provides the C-language header files required to compile your program using the API. The default location of the header files is in the **target_install_directory**\fds directory of the disk on which DDS was installed.

The complete list of header files is:

- config.h
- defs.h
- dist.h
- errno.h
- fds.h
- file.h
- ipc.h
- names.h
- nodes.h

Building Your Application

To build your application:

- Include the C-language header files in your application.
- Link your application executable to the DDS import libraries, which are located in

target_install_directory\lib.

The import libraries for Windows are as follows:

- fdsbase.lib
- fdscfg.lib
- fdsfile.lib
- fdsipc.lib
- fdsnames.lib
- fdsnodes.lib

The DDS API functions use the same parameter passing conventions as operating-system API functions. Any compiler that supports the calling of

operating system APIs can be used to call DDS API functions.

DDS directly supports the following compilers:

- IBM VisualAge C++ for Windows, Version 3.5 or higher
- Microsoft® Visual C++®, Version 2.0 or higher

In addition to the above compilers, DDS can be used with other compatible 32-bit compilers. The C header files included with DDS are standard C code, and the import libraries are standard import libraries (generated using implib.exe).

Note: If you use the Microsoft Visual C++ compiler, you must compile with Optimization set to Off.

If you intend to use a compiler with a default calling convention that differs from the operating-system linkage convention, you must modify the DDS header file (defs.h). You must set the constant FDS_SYSLINK in the header file to the reserved keyword used by the compiler to indicate operating-system linkage conventions. To specify the calling convention for a given function, that keyword must be used when the function is declared. Refer to your compiler documentation for more information.

Calling DDS API functions from languages other than C or C++ is possible, but no header files are included. The only requirement is that the language must be able to call functions using the operating-system linkage conventions, including passing pointers to variables and null-terminated strings.

Optimizing Application Performance

Read this section before you write applications to learn how to optimize the rate at which the applications function in conjunction with DDS. Some performance tuning can be done with little modification to the application, but some must be designed into the program. The following sections discuss several ways to optimize the performance of an application that is using DDS.

Memory Considerations

There is a direct correlation between the maximum queue size specified on the FdsCreateQ() API and the amount of memory required by DDS. Therefore, you should not create queues larger than necessary.

Each thread that uses DDS requires an amount of memory that is slightly greater than the size (in bytes) of all parameters being passed to and returned by a particular call to the API plus sufficient memory for an additional stack.

There is also a process-related memory overhead approximately the size of a single thread’s overhead.

Multiple Threads and Processes

DDS supports multiple applications on the same node simultaneously utilizing DDS. Additionally, multiple threads of a process can run within DDS concurrently (referred to as **multi-threaded**).

The following restrictions apply to each thread in a multi-threaded application:

- A single thread can open an IPC queue, keyed file, or sequential file multiple times without intervening close operations. The FileAccess attributes specified on each open of the same keyed or sequential file need not be identical. (The FileAccess attributes are described in "FdsOpenKeyedFile()".)

- DDS file handles and queue handles are not inherited by child processes and cannot be shared between processes.

- There are no differences between the restrictions on access to a single keyed or sequential file for two processes on the same node and the restrictions on access to a single file for two processes on different nodes.

See "Memory Considerations" for information about memory requirements.

Designing Your Application

This section describes attributes of DDS that apply to any of its functions. Consider these attributes when designing your application.

Accessing the Prime Copy of a File

The prime copy of a distributed file exists on the acting primary distributor. The application can access this version of the file from any node in the system via the File Services APIs by using the acting primary role as part of the retail path specification (the acting primary role is <FDSFDXAP::>).

The prime copy of the file becomes unavailable when the acting primary role is deactivated or the acting primary distributor becomes unavailable. If this happens, the File Services APIs will return error code -530 FDSERR_ROLE_CHANGE or -350 FDSERR_NODE_NOT_FOUND respectively.

Your application should perform the following steps:

1. Close the file handle using the appropriate File Services API.
2. Save any updates to the prime copy in a local file to be applied when the prime copy becomes available.
3. Attempt to reopen the prime copy.
   Note: The File Services APIs have no time-outs. Therefore, it is up to the application to continually attempt to open the file, with some delay between each attempt, to avoid overutilizing system resources.
4. If the prime copy cannot be reopened, indicate to store personnel that the primary distributor is unavailable and that the backup distributor should be activated as the primary distributor.

Note: Accessing the prime copy of a file is a remote file access and, therefore, slower than a local file access. If the application is simply reading data from a distributed file, it should access the image copy of the file using the File Services APIs. DDS will distribute any changes made to the prime to all image copies of the file.

Argument Formats

Some common types of arguments are of a standardized format throughout DDS. Follow these rules for your application when you use one of these types:

All pointer arguments passed to a DDS function must point to valid memory on input. DDS does not allocate application memory.

DDS validates pointer arguments. If pointer arguments are not valid, the error -10 FDSERR_ACCESS is returned.

All (char *) arguments passed to a DDS function must be null terminated. All (char *) arguments returned by a DDS function are also null terminated.

All length parameters in the API associated with (char *) parameters include the null terminator (\0).

Each file name passed to an API must be either a fully qualified file specification or a logical name that resolves to a fully qualified file specification.

Most of the DDS APIs require one or more of these types of parameters:

Input

Input parameters are those for which the application must provide valid data when the API is called.

For example, FileName is an input parameter for “FdsGetFileAttributes()”. When FdsGetFileAttributes() is called, FileName must be a valid name of an existing file.

Input/Output

Input/output parameters are those for which the application must provide valid data when the API is called. When the API has completed (successfully or unsuccessfully, depending on the API), DDS replaces the data passed in.

Note: When pointers are passed to an API, DDS replaces the value in the location pointed to by the pointer. It does not modify the pointer itself.

For example, RecordSizePtr is an input/output parameter for “FdsReadKeyedRecord()”. When FdsReadKeyedRecord() is called, the value in the location pointed to by RecordSizePtr is the maximum size of the record to read.

When the API has completed successfully, the value in the location pointed to by RecordSizePtr is replaced with the actual size of the record read.

If the API does not complete successfully and the error is -490 FDSERR_REC_SIZE,the value in the location pointed to by RecordSizePtr is replaced with the size of the record that could not be read.

Output

Output parameters are those for which the API returns a value to the application. However, the application must provide a valid data location for the API to provide

a value when the API is called.

For example, **CurrentSize** is an output parameter for “FdsQueryBinFileSize()”. When FdsQueryBinFileSize() has completed successfully, the value in the location pointed to by **CurrentSize** is the current size of the binary file in bytes.

CurrentSize must be a valid pointer of type unsigned long when the API is called. The API does not return a pointer; it replaces the value stored in the location pointed to by **CurrentSize**.

Error Codes

The following list contains the error codes that could be returned from any API call. The correct application recovery for each error is specified in Appendix B. Error Codes.

- -20 FDSERR_ADDRESS
- -150 FDSERR_DOWN
- -230 FDSERR_INIT
- -240 FDSERR_INTERNAL
- -250 FDSERR_INTERRUPT
- -320 FDSERR_MEMORY
- -400 FDSERR_OS
- -520 FDSERR_RESOURCE

Initializing Your Application

The first thing your application must do to use DDS APIs is to register by calling either the FdsInit() API or the FdsInit2() API.

FdsInit()

Purpose

Initializes DDS for use by the application.

Syntax

```
#include <stdlib.h>
#include <stdio.h>
#include <fds/fds.h>


long FdsInit( );
```

Remarks

After any thread of a process calls this API, all threads of that process are initialized. Therefore, FdsInit() must be called exactly once for each process.

Either FdsInit() or FdsInit2() must be called successfully before any other DDS APIs are used.

Note: To share memory among processes, DDS uses the address hex 40000000 as its base memory address. If a call to FdsInit() returns the error "-400 FDSERR_OS", and logs the following message in the event logs: **The operating system returned error 487 at location 50303**, then the memory used by your application is conflicting with the DDS memory address.

To change the DDS base memory address, add the environment variable FDS_SHARED_MEMORY to the system configuration in the Registry, specifying another memory address. The memory address must be within the range of hex 01000000 and hex 75000000. Otherwise, DDS will not initialize and an error will be logged.

Error Conditions

FdsInit() returns the following values:
-170 FDSERR_EXISTS
-200 FDSERR_FILE_NOT_FOUND
-560 FDSERR_SEQUENCE

Examples

```
#include <stdlib.h>
#include <stdio.h>
#include <fds/fds.h>
#include <fds/errno.h>

long          rc = 0;
/* Before using APIs, you must initialize Distributed Data Services */
rc = FdsInit( );
if (rc != FDS_SUCCESS)
{
   printf ("Initialization failed (rc = %X). \n", rc);
   exit (0);
}
```

FdsInit2()

Purpose

Initializes DDS for use by the application. The difference between FdsInit() and FdsInit2() is that FdsInit2() will not complete until DDS has been started and initialized completely if you specify **FDS_INIT_WAIT_FOR_DDS**.

Syntax

```
#include <stdlib.h>
#include <stdio.h>
#include <fds/fds.h>


long FdsInit2( unsigned long InitFlags );
```

Parameters

InitFlags — input

Determines whether the API will complete (returning control to the application) as soon as DDS has starting initializing or will wait until

DDS has completed initialization. Valid values are:

FDS_INIT_DEFAULT
The API will complete as soon as it starts DDS without waiting for DDS to initialize completely. This value is the default.

FDS_INIT_WAIT_FOR_DDS
The API will not complete until DDS has been started and initialized completely.

Remarks

After any thread of a process calls this API, all threads of that process are initialized. Therefore, FdsInit2() must be called exactly once for each process.

Either FdsInit() or FdsInit2() must be called successfully before any other DDS APIs are used.

Note: To share memory among processes, DDS uses the address hex 40000000 as its base memory address. If a call to FdsInit2() returns the error "-400 FDSERR_OS", and logs the message The operating system returned error 487 at location 50303 in the event logs, the memory used by your application is conflicting with the DDS memory address.

To change the DDS base memory address, add the environment variable FDS_SHARED_MEMORY to the system configuration in the Registry, specifying another memory address. The memory address must be within the range of hex 1000000 and hex 75000000. Otherwise, DDS will not initialize and an error will be logged.

Error Conditions

FdsInit2() returns the following values:
-170 FDSERR_EXISTS
-200 FDSERR_FILE_NOT_FOUND
-210 FDSERR_FLAG
-560 FDSERR_SEQUENCE

Examples

```
#include <stdlib.h>
#include <stdio.h>
#include <fds/fds.h>
#include <fds/errno.h>
long rc = 0;
long int InitFlags = 0;
/* Initialize Distributed Data Services and wait for init to complete */
InitFlags = FDS_INIT_WAIT_FOR_DDS;
rc = FdsInit2(InitFlags);
if (rc != FDS_SUCCESS)
   {
     printf ("Initialization failed (rc = %X). \n", rc);
     exit (0);
   }
```

Chapter 3. Installation and Configuration

DDS provides an API that you can use from your program to obtain information about the installation and configuration of the product on the local node. This data cannot change while DDS is running, so you must issue the API call only one time when each application is started.

An FDS_CFG structure is provided in the DDS header file, CONFIG.H, and it should be used to declare a variable. During the initialization of your application program, you can issue the API call to request that DDS store the current configuration information in this variable. This variable can be referred to while your application is running without having to make subsequent calls to DDS.

FdsQueryConfig()

Purpose

Obtain configuration data.

Syntax

```
#include <fds/config.h>

long FdsQueryConfig( FDS_CFG *ConfigInfo, unsigned int *BufferSize );
```

Parameters

ConfigInfo — input/output

Input A pointer to an FDS_CFG structure in which the configuration data is placed.

Output
When this API completes successfully, the data in the structure pointed to by **ConfigInfo** is replaced by the current configuration data. See “Appendix A. Data Types” for more information about the FDS_CFG data structure.

BufferSize — input/output

Input A pointer to the size of the structure to be returned. This value must specify the length of memory pointed to by **ConfigInfo**.

Output
When this API completes successfully, the value pointed to by **BufferSize** is replaced with the size of the configuration structure that was copied to the input buffer pointed to by the **ConfigInfo** parameter. If this API returns the error -40 FDSERR_BUFFER_SIZE,this parameter is set to the required buffer size.

Remarks

This API is used to obtain the current installation and configuration data for DDS. This API does not provide logical-names configuration data. See Chapter 7. Name Services for more information about how to query logical-names

configuration data. Your applications are required to call this API only once each time an application runs because the installation and configuration data used by DDS does not change while it is running.

Error Conditions

FdsQueryConfig() returns the following values:

- -20 FDSERR_ADDRESS
- -40 FDSERR_BUFFER_SIZE

Examples

The example below declares a variable of type FDS_CFG, which is used to hold the configuration data. This API is called to update the configuration structure with the current configuration data.

```
#include <fds/config.h>
#include <fds/fds.h>
#include <fds/errno.h>

FDS_CFG        ConfigData;
long           rc;
unsigned int   ConfigDataSize;
ConfigDataSize = sizeof(ConfigData);

rc = FdsInit();

// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   rc = FdsQueryConfig(&ConfigData,&ConfigDataSize);
   if (rc != FDS_SUCCESS)
   {
     /* perform error processing */
   }

}
```

Chapter 4. File Services

The File Services component allows you to manipulate files in a more structured manner than that provided by standard byte-stream files.

File Services defines three types of files:

- Keyed files

- Sequential files
- Binary files

The File Services component provides APIs for manipulating files of all three types as well as APIs for general functions such as deleting a file.

The File Services APIs for general functions are:

- **FdsCreateDir()** — Create a directory
- **FdsDeleteFile()** — Delete a file
- **FdsExistFile()** — Test for the existence of a file
- **FdsGetFileAttributes()** — Return the date and time, and the read and write attributes of a file
- **FdsGetFileNames()** — Return a list of the files in a directory
- **FdsQueryFileSystemInfo()** — Query the size of a disk and the amount of available space
- **FdsRemoveDir()** — Remove a directory
- **FdsRenameFile()** — Rename a file
- **FdsRestrictFile()** — Restrict access to a file
- **FdsSetFileAttributes()** — Set the date and time, and the read and write attributes of a file
- F**dsUnrestrictFile()** — Remove access restrictions for a file

The File Services APIs for keyed files are:

- **FdsCloseKeyedFile()** — Close a keyed file or write the contents to disk
- **FdsCreateKeyedFile()** — Create a new keyed file
- **FdsDeleteKeyedRecord()** — Delete a record from a keyed file
- **FdsOpenKeyedFile()** — Open an existing keyed file
- **FdsReadKeyedRecord()** — Read a record from a keyed file
- **FdsReleaseKeyedRecord()** — Release a lock on a record in a keyed file
- **FdsWriteKeyedRecord()** — Write a record to a keyed file

The File Services APIs for sequential files are:

- **FdsCloseSeqFile()** — Close a sequential file
- **FdsFindNextSeqRecord()** — Move the file pointer to the next valid record in a sequential file
- **FdsOpenSeqFile()** — Open or create a sequential file
- **FdsReadSeqRecord()** — Read a record from a sequential file
- **FdsReturnSeqFilePos()** — Return the file-position indicator for a sequential file
- **FdsSeekSeqFilePos()** — Seek to a point in a sequential file
- **FdsWriteSeqRecord()** — Append a record to a sequential file

The File Services APIs for binary files are:

- **FdsCloseBinFile()** — Close a binary file
- **FdsFlushBinFile()** — Flush any data buffered for a binary file
- **FdsOpenBinFile()** — Open or create a binary file
- **FdsQueryBinFileSize()** — Query the size of a binary file
- **FdsReadBinFile()** — Read from a binary file
- **FdsSeekBinFilePos()** — Move the file pointer in a binary file
- **FdsSetBinFileLocks()** — Lock or unlock a range in a binary file
- **FdsSetBinFileSize()** — Set the size of a binary file
- **FdsWriteBinFile()** — Write to a binary file

Services and Operation

The File Services component uses other components of DDS to provide services. This section describes those services and provides general operations information.

File Distribution

The File Services component uses the Data Distribution component to distribute file updates to other nodes. See Chapter 6. Data Distribution for more information about file distribution.

File-Location Transparency

The File Services component provides file location transparency. **File-location transparency** means that an application is not required to explicitly indicate the location of a file to access that file; instead, an application can

use logical names and role names to indirectly specify the location of a file. See Chapter 7. Name Services for more information about logical names and role names. The same API is used regardless of whether the file resides locally or remotely.

Priority

The thread priority of the calling application is preserved when accessing remote files. This, in conjunction with the prioritization provided by the File System Interface, ensures that disk access is prioritized across all applications within the system.

Data Integrity

The File Services component sets the write-through bit in the (CreatFile() on Windows for all File Services functions (except FdsOpenBinFile(), which allows the write-through bit to be optionally specified). All data written to such files using File Services APIs is written to disk before returning to the application; this step protects the integrity of the data written by the File Services component.

File Content

Although the File Services component places no restrictions on the data placed in File Services sequential files and keyed files, these files do contain File Services processing information. Therefore, the File Services component should always be used for processing File Services sequential and keyed files.

File Names

The string FDS is allowed within file names.

Operating System and File System Restrictions

The File Services component is implemented by API calls to the underlying operating system.

These calls can manage file access with a variety of file systems, such as FAT, FAT32, and NTFS. The file system might also vary between nodes, DASD devices, or DASD partitions.

There might be differences in the available services, based on differences like file naming conventions in the operating system or file system. The File Services component does not attempt to alter or mask the properties of the operating system or the file system.

The File Services component attempts to call the operating system. If an error occurs because of the properties of the operating system or file system, the File Services component returns an error.

Specifically, the File Services component does ***not*** :

- Attempt to detect parameters that are not valid, such as incorrect file names, that are sent to the operating system or file system.
- Attempt to detect calls to the operating system or file system that are not consistent with an API definition. For example, if an API expects a file name and a directory name is used instead, the File Services component does not detect the error. The behavior of the call is dependent on the operating system and file system.

- Circumvent any security implemented by the operating system or file system.
- Implement additional security.

FdsCreateDir()

Purpose

Create a new directory.

Syntax

```
#include <fds/file.h>

long FdsCreateDir(const char *DirName);
```

Parameters

DirName — input

A string containing the name of the directory to be created. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

Remarks

The directory specified by **DirName** is created.

Error Conditions

FdsCreateDir() returns the following values:
-10 FDSERR_ACCESS
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example creates a new directory.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>

long        rc;                                     // Return from API call
char        DirName[50] = "d:\\mydir";   // Directory name
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
```

```
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------
   // Call FdsCreateDir API to create the directory "d:\mydir"
   //-------------------------------------------------------------------------
   rc = FdsCreateDir( DirName );
   printf( "FdsCreateDir completed with return code = (%d).\n", rc );
} // end if
else
{
   // else process errors
}
```

FdsDeleteFile()

Purpose

Delete a file.

Syntax

```
#include <fds/file.h>

long FdsDeleteFile(const char *FileName);
```

Parameters

FileName — input

A string containing the name of the file to delete. The string can contain logical names, but must resolve to a retail path specification. See "**File Names and Queue Names**" on page 15 for more information.

Remarks

The file specified by **FileName** is deleted from disk.

Error Conditions

FdsDeleteFile() returns the following values:
-10 FDSERR_ACCESS
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND

-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example deletes a file.

```
#include <stdio.h>
#include <fds/file.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/errno.h>

long        rc;                                                    // Return from API Call
char        FileName[50] = "d:\\mydir\\myfile";     // File name
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------
   // Call FdsDeleteFile API to delete "d:\mydir\myfile"
   //-------------------------------------------------------------------------------------
   rc = FdsDeleteFile( FileName );
   printf( "FdsDeleteFile completed with return code = (%d).\n", rc );
} // end if
else
{
   // else process errors
}
```

FdsExistFile()

Purpose

Test for the existence of a file.

Syntax

```
#include <fds/file.h>

long FdsExistFile(const char *FileName);
```

Parameters

FileName — input

A string containing the name of the file to locate. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” on page 15 for more information.

Remarks

If the file specified by **FileName** exists, FDS_SUCCESS is returned. If it does not exist, -200 FDSERR_FILE_NOT_FOUND is returned.

Error Conditions

FdsExistFile() returns the following values:
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND

-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example verifies that a file exists.

```
#include <stdio.h>
#include <fds/file.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/errno.h>

long        rc;                                                    // Return from API Call
char        FileName[50] = "d:\\mydir\\myfile";     // File name
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------------------
   // Call FdsExistFile API to see if "d:\mydir\myfile" exists
   //---------------------------------------------------------------------
   rc = FdsExistFile( FileName );
   printf( "FdsExistFile completed with return code = (%d).\n", rc );
} // end if
else
{
// else process errors
}
```

FdsGetFileAttributes()

Purpose

Return the date and time of the last file modification and the read/write attribute of a file.

Syntax

```
#include <fds/file.h>

long FdsGetFileAttributes(const char *FileName,

                          FDS_DATE_TIME *DateTime, int *Flags);
```

Parameters

FileName — input

The name of the file for which the attributes should be obtained. The string can contain logical names, but must resolve to a retail path specification. See "File Names and Queue Names" on page 15 for more

information.

DateTime — output

Pointer to the location where the date and time of the last modification to this file is stored.

Flags — output

Pointer to the location where the read/write attribute is stored. The valid values are:

FDS_FILE_ATTRIBUTE_READ_ONLY

The file can be read but cannot be modified.

FDS_FILE_ATTRIBUTE_READ_WRITE

The file can be read and modified.

FDS_FILE_ATTRIBUTE_DIRECTORY

The name specified for **FileName** is a directory.

Remarks

Because file attributes can be changed at any time, you should always issue this API call for the latest attribute information.

Error Conditions

FdsGetFileAttributes() returns the following values:

-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example retrieves the file attributes for D:\MYFILE.DAT.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>

long          rc;                                                    // Return from API Call
const char *          FileName = "d:\\myfile.dat";  // File Name
FDS_DATE_TIME      DateTime;                                    // Date and time attributes
int           Flags = -1;                                  // R/W and DIR indicator attribute
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------
   // Call FdsGetFileAttributes API to get the file's attributes
```

```
   //-------------------------------------------------------------------------------------------
   rc = FdsGetFileAttributes( FileName,
                                        &DateTime,
                                        &Flags );
                                        printf("FdsGetFileAttributes completed with
                                                return code = (%d) \n",
                                        rc );
   //-------------------------------------------------------------------------------------------
   // Output the file's attributes returned from the API call
   //-------------------------------------------------------------------------------------------
   printf( " File (%s) has attributes : \n"
            " ---> Flags = (%d) \n"
            " ---> Last Modified on (%d/%d/%u) at (%d:%d:%d) \n",
            FileName,
            Flags,
            DateTime.Month,
            DateTime.Day,
            DateTime.Year,
            DateTime.Hour,
            DateTime.Minute,
            DateTime.Second );
} // end if
else
{
// else process errors
}
```

FdsGetFileNames()

Purpose

Return a list of file names contained in the specified directory.

Syntax

```
#include <fds/file.h>

long FdsGetFileNames(const char *DirNamePtr, void *BufferPtr, unsigned
                                int *NBytesPtr unsigned int Flag);
```

Parameters

DirNamePtr— input

Pointer to the name of the directory. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

BufferPtr — output

Pointer to the buffer where the file names are stored. The file names will be stored as a series of null-terminated strings.

If this API fails with the error -40 FDSERR_BUFFER_SIZE, the buffer size is too small and **BufferPtr** points to a null string.

NBytesPtr— input/output

Input Pointer to the length of the buffer where the names are stored. This value must be less than or equal to 60,000.

Output

When this API has completed successfully, the data in the location pointed to by **NBytesPtr** is replaced by the actual length of the data returned.

If this API fails with the error -40 FDSERR_BUFFER_SIZE, the buffer size is too small; **NBytesPtr** specifies the correct size of **BufferPtr** required to hold all of the names returned.

Flag — input

Used to specify which names are returned from the specified directory. Valid values are:

FDS_FILE_FILE_NAMES

Return file names only. This value is the default.

FDS_FILE_DIRECTORY_NAMES

Return directory names only.

Remarks

The file names are returned in **BufferPtr** as a series of null-terminated strings. The file names are not fully qualified. The sort order of the returned file names is determined by the underlying operating system. Specifying **FDS_FILE_FILE_NAMES** and **FDS_FILE_DIRECTORY_NAMES** returns all names contained in the directory. The special directory names '.' (current directory) and '...' (previous directory) are never returned.

The last file name is terminated by two null characters, indicating the end of the last file name string and the end of the list of file names.

Because files can be created, deleted, renamed, and copied within a directory at any time, subsequent calls to FdsGetFileNames() can return different results.

Error Conditions

FdsGetFileNames() returns the following values:

-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW

Examples

This example obtains the list of file names that exist in directory D:\MYDIR.

```
#include <stdio.h>
#include <string.h>
#include <fds/fds.h>
#include <fds/defs.h>
```

```
#include <fds/file.h>
#include <fds/errno.h>

long                rc;                                 // Return from API Call
const char *        DirNamePtr = "d:\\mydir";   // Pointer to directory
char                Buffer[500]                         // Buffer for File Names
unsigned int        NBytes = sizeof(Buffer);    // Size of buffer
unsigned int        Flag = 0;                   // File names or directories
int                 entry_start = 0;            // byte entry starts
int                 entry_length = 0;           // length of current entry
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------
   // Set flag for API to get a list of file names in directory
   //-------------------------------------------------------------------------------------------
   Flag = FDS_FILE_FILE_NAMES;
   //-------------------------------------------------------------------------
   // Call FdsGetFileNames API to get a list of files
   //-------------------------------------------------------------------------
   rc = FdsGetFileNames( DirNamePtr,
                         Buffer,
                         &NBytes,
                         Flag );
                         printf("FdsGetFileNames completed with return
                                 code = (%d) \n",
                         rc );
   //-----------------------------------------------------------------
   // Find and output each entry
   //-----------------------------------------------------------------
   for (;;)
   {
      //-----------------------------------------------------------------
      // How long is the next entry to output
      // (entry_start is initially 0)
      //-----------------------------------------------------------------
      entry_length = 1 + strlen( &Buffer[entry_start] );
      // If the length of the entry is less than 2, exit this loop
      if (2 > entry_length)
           break;
      //-----------------------------------------------------------------
      // Output the entry
      //-----------------------------------------------------------------
      printf(" --> (%s)\n", &Buffer[entry_start]);
      //-------------------------------------------------------------------------------------------
      // Increment entry_start to the beginning of the next entry
      //-------------------------------------------------------------------------------------------
      entry_start += entry_length;
   }
} // end if
else
{
   // else process errors
}
```

FdsQueryFileSystemInfo()

Purpose

Query the size of a disk and the amount of space available on the disk.

Syntax

long FdsQueryFileSystemInfo(const char *FileSystemID*, unsigned long **TotalUnits*, unsigned long **AvailUnits*, unsigned long **UnitSize*);

Parameters

FileSystemID — input Pointer to the name of the disk. **FileSystemID** is a drive specification, such as C:. The string can contain a role name or node ID.

TotalUnits — output
Pointer to the location where the total units of space on the disk are stored.

AvailUnits — output
Pointer to the location where the total available units of space on the disk are stored.

UnitSize — output
Pointer to the location where the size of a unit (in bytes) is stored.

Remarks

This API returns the size of a disk and the amount of available space in units. It also returns **UnitSize**, which is the size of each unit in bytes.

Error Conditions

FdsQueryFileSystemInfo() returns the following values:
-10 FDSERR_ACCESS
-90 FDSERR_DISK
-190 FDSERR_FILE_NAME
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example queries the D drive, and returns the total space and the available space on that drive.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
```

```
#include <fds/errno.h>
long rc;          // Return from API call

char FileSystemID[3] = "d:"; // File system ID
unsigned long TotalUnits; // Total number of units
unsigned long AvailUnits; // Number of available units
unsigned long UnitSize; // Size of a unit (in bytes)

// Initialize DDS. Could use FdsInit2() instead of FdsInit()
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
      //-------------------------------------------------------------------------------------------------------
      // Call FdsQueryFileSystemInfo to get the size of the D: drive
      // and the amount of space that is currently available on the // disk
      //-------------------------------------------------------------------------------------------------------
      rc = FdsQueryFileSystemInfo( FileSystemID, &TotalUnits,
                                                &AvailUnits, &UnitSize );
      printf( "FdsQueryFileSystemInfo completed with
                  return code   = (%d).\n",
      rc );
      //-----------------------------------------------------------------
      // Output the disk characteristics
      //-----------------------------------------------------------------
      printf( "Disk (%s)                        \n"
                  " --> Total Space = %dK                  \n"
                  " --> Available Space = %dK              \n"
                  " --> %% Available Space = %d%%          \n",
                  FileSystemID,
                  TotalUnits*UnitSize/1024,
                  AvailUnits*UnitSize/1024,
                  AvailUnits*100/TotalUnits );
      } // end if
else
{
      // else process errors
}
```

FdsRemoveDir()

Purpose

Remove a directory.

Syntax

```
#include <fds/file.h>

long FdsRemoveDir(const char *DirName);
```

Parameters

DirName — input

A string containing the name of the directory to be deleted. The string can contain logical names, but must resolve to a retail path specification.

See “File Names and Queue Names” for more information.

Remarks

The directory specified by **DirName** is deleted. A directory must be empty before it can be deleted.

Error Conditions

FdsRemoveDir() returns the following values:

-10 FDSERR_ACCESS
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example removes a directory from the D drive.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long        rc;                                          // Return from API call
char        DirName[50] = "d:\\mydir";   // Directory name

// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();

// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------------------------------------------
  // Call FdsRemoveDir API to remove the directory "d:\mydir"
  //-------------------------------------------------------------------------------------------------------
  rc = FdsRemoveDir( DirName );
  printf( "FdsRemoveDir completed with return code = (%d).\n", rc );
} // end if
else
{
  // else process errors
}
```

FdsRenameFile()

Purpose

Rename a file.

Syntax

```
#include <fds/file.h>

long FdsRenameFile(const char *FileName, const char *NewFileName);
```

Parameters

FileName — input

A string containing the name of the file to rename. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

NewFileName — input

A string containing the new name of the file. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

Note: The value specified for **NewFileName** cannot be a name that was used for a distributed directory or a directory that contained distributed files, even if that directory no longer exists.

Remarks

The name of the file specified by **FileName** is changed to **NewFileName**. If you are renaming a file from one drive to another drive, the file is localized (if distributed) regardless of the value specified by **DistRenamedFile** keyword.

The rename operation itself is managed by the underlying operating system. You cannot rename a file to a different node. For example, you cannot rename a file that exists on the primary distributor to a new name on a subordinate node.

Error Conditions

FdsRenameFile() returns the following values:

-10 FDSERR_ACCESS
-170 FDSERR_EXISTS
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME

-550 FDSERR_ROLE_NOT_FOUND

Examples

This example renames a file.

```
#include <stdio.h>
#include <fds/file.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/errno.h>
long        rc;           // Return from API Call
char        OldFileName[50] = "d:\\mydir\\myoldfile";         / Old File Name
char        NewFileName[50] = "d:\\mydir\\mynewfile";  // New File Name
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
     //-------------------------------------------------------------------------
     // Call FdsRenameFile API to rename "d:\mydir\myoldfile" to
     // "d:\mydir\mynewfile"
     //-------------------------------------------------------------------------
     rc = FdsRenameFile( OldFileName, NewFileName );
     printf( "FdsRenameFile completed with return code = (%d).\n",
                  rc );
} // end if
else
{
   // else process errors
}
```

FdsRestrictFile()

Purpose

Restrict access to a file.

Syntax

```
#include <fds/file.h>

long FdsRestrictFile(const char *FileName);
```

Parameters

FileName — input

A pointer to a string containing the name of the file for which access is to be restricted. The string can contain a logical name, but it must resolve to a retail path specification. See “File Names and Queue Names” for more information.

Remarks

This API closes all open instances of the specified file that were opened through DDS APIs. It does not close open instances of the specified file that were opened directly by calls to the operating system.

Any attempt to use an existing file handle for the specified file will return -222

FDSERR_HANDLE_FORCED_CLOSED.Any new attempts to open the file will return -10 FDSERR_ACCESS. However, you can still rename the file using FdsRenameFile() or delete the file using FdsDeleteFile().

Error Conditions

FdsRestrictFile() returns the following values:

-10 FDSERR_ACCESS
-190 FDSERR_FILE_NAME
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example restricts access to a file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long          rc;                                                    // Return from API call
char          FileName[50] = "d:\\mydir\\myfile";     // File name
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
     //-------------------------------------------------------------------------------------------------------
     // Call FdsRestrictFile API to restrict access to "d:\mydir\myfile"
     //-------------------------------------------------------------------------------------------------------
     rc = FdsRestrictFile( FileName );
     printf( "FdsRestrictFile completed with return code = (%d).\n", rc );
} // end if
else
{
  // else process errors
}
```

FdsSetFileAttributes()

Purpose

Set the date and time attribute and the read/write attribute of a file.

Syntax

```
#include <fds/file.h>
```

```
long FdsSetFileAttributes(const char *FileName,
                          FDS_DATE_TIME *DateTime,
                          int Flags);
```

Parameters

FileName— input

Specifies the name of the file for which the attributes should be set. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

DateTime – input

Changes the last modification date and time of the file.

Flags— input

Specifies the read/write attribute and whether you want to change the last modification date/time for the specified file. Also indicates whether the attributes are being set for a file or a directory.

The valid values are:

FDS_FILE_ATTRIBUTE_READ_ONLY

Specifies that the file can be read but cannot be modified.

FDS_FILE_ATTRIBUTE_READ_WRITE

Specifies that the file can be read and modified.

FDS_FILE_ATTRIBUTE_DATE

Specifies that the last modified date for the file should be set.

FDS_FILE_ATTRIBUTE_TIME

Specifies the last modified time for the file should be set.

FDS_FILE_ATTRIBUTE_DIRECTORY

Specifies that the attributes are being set for a directory. The name specified by **FileName** must be a valid directory. If **FDS_FILE_ATTRIBUTE_DIRECTORY** is not specified, the attributes are being set for a file.

Remarks

This API sets the read/write attribute, the last modified date and time attribute, or both attributes of the file.

The set-file-attributes operation itself is managed by the underlying operating system.

Flags can be specified in any combination, except that **FDS_FILE_ATTRIBUTE_READ_ONLY** and **FDS_FILE_ATTRIBUTE_READ_WRITE** cannot both be set in the same API call.

If **FDS_FILE_ATTRIBUTE_DATE** is specified, the file date is changed to the date provided in **DateTime**.If **FDS_FILE_ATTRIBUTE_TIME** is specified, the file time is changed to the time provided in **DateTime**. You may specify both **FDS_FILE_ATTRIBUTE_DATE** and **FDS_FILE_ATTRIBUTE_TIME** in the same API call.

Error Conditions

FdsSetFileAttributes() returns the following values:

-10 FDSERR_ACCESS
-75 FDSERR_DATE_TIME
-80 FDSERR_DIR_INDICATOR
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-210 FDSERR_FLAG
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example sets the file attributes of D:\myfile.dat to read-only and sets the last modification date and time.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long          rc;                                          // Return from API Call
const         char * FileName = "d:\\myfile.dat";      // File Name
FDS_DATE_TIME DateTime[1];                                 // Date and time attributes
int Flags = 0;                                             // ReadWrite attribute
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------
   // Set read write attributes to READ ONLY and set date and time
   //-------------------------------------------------------------------------------------------------------------
   Flags = FDS_FILE_ATTRIBUTE_READ_ONLY |
   FDS_FILE_ATTRIBUTE_DATE |
   FDS_FILE_ATTRIBUTE_TIME;
   //-----------------------------------------------------------------
   // Set date for the file to 9/9/1997
   //-----------------------------------------------------------------
   DateTime.Year = 1997;
   DateTime.Month = 9;
   DateTime.Day = 9;
   //-----------------------------------------------------------------
   // Set time for the file 09:19:19
   //-----------------------------------------------------------------
   DateTime.Hour = 9;
   DateTime.Minute = 19;
   DateTime.Second = 19;
   //-----------------------------------------------------------------
```

```
   // Call FdsSetFileAttributes API to set the file's attributes
   //-----------------------------------------------------------------
   rc = FdsSetFileAttributes( FileName,
   &DateTime,
   Flags );
   printf("FdsSetFileAttributes completed with return code = (%d) \n",
                                                                    rc );
} // end if
else
{
  // else process errors
}
```

FdsUnrestrictFile()

Purpose

Remove access restrictions for a file.

Syntax

```
#include <fds/file.h>

long FdsUnrestrictFile(const char *FileName);
```

Parameters

FileName — input

A pointer to a string containing the name of the file for which you want to remove access restrictions. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names”
for more information.

Remarks

This API removes access restrictions that were imposed when FdsRestrictFile() was invoked.

Error Conditions

FdsUnrestrictFile() returns the following values:

-10 FDSERR_ACCESS
-190 FDSERR_FILE_NAME
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example removes file access restrictions.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long          rc;                                                    // Return from API call
char FileName[50] = "d:\\mydir\\myfile";              // File name
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------------------
   // Call FdsUnrestrictFile API to allow access to "d:\mydir\myfile"
   //---------------------------------------------------------------------
   rc = FdsUnrestrictFile( FileName );
   printf( "FdsUnrestrictFile completed with return code = (%d).\n",
                                                                                                  rc );
} // end if
else
{
  // else process errors
}
```

Keyed-File Services

Keyed files are permanent files, stored on DASD, that can be either local or remote to the application. Access to keyed files is based on a key field situated at the front of each keyed-file record. All records within a keyed file must have the same length. An example of a keyed file in the retail industry is the item price-lookup file.

All keys in a file must be unique and cannot be 0 (zero). If a record is added with a key field identical to an existing record, the existing record is overlaid by the new record.

Most DASD devices write physical sectors of 512 bytes. Because DDS has no protection from system interruptions, such as power line disturbances, a partial keyed-file record write can occur if the block size is greater than 512 bytes and if all sectors are not contiguous on disk. Ensure that the block size you use is 512 bytes to eliminate any possibility of this occurrence.

DDS supports keyed files created on a 4690 system. Keyed files created by DDS can be moved to an IBM 4690 system if the block size is 512 bytes. In some cases, file attributes used in the 4690 might not be compatible with the operating system underlying DDS. Generally, transporting the keyed file across a network connecting the 4690 system and DDS will correct attribute bit irregularities.

The APIs provided by File Services for keyed-file manipulation are:

FdsCloseKeyedFile() — Close a keyed file or write contents to disk
FdsCreateKeyedFile() — Create a new keyed file **FdsDeleteKeyedRecord()**

— Delete a record from a keyed file **FdsOpenKeyedFile()** — Open an existing keyed file **FdsReadKeyedRecord()** — Read a record from a keyed file **FdsReleaseKeyedRecord()** — Release a lock on a keyed file record **FdsWriteKeyedRecord()** — Write a record to a keyed file

Capabilities and Restrictions

These capabilities and restrictions apply to the keyed-file APIs:

- An application can add records to or delete records from a keyed file, but an existing keyed file cannot be extended. If you need to increase the size of a keyed file, the keyed file must be erased and created again. You can copy the data from the keyed file into a flat file to be reused.
- You can specify block sizes from 512 to 4,096. The block size must be a multiple of 512. Block sizes larger than 512 are not protected from partial writes.
- Record sizes can range from 1 to 4,092 bytes, but must be at least 4 bytes less than the block size.
- You can lock keyed files at the record level using FdsReadKeyedRecord().
- You can lock keyed files at the file level using FdsCreateKeyedFile() or FdsOpenKeyedFile().
- Individual records can be locked for update. Locking a record for update does not block another process from locking another record in the same block.
- Keyed-file services maintains statistics for each keyed file. These statistics are maintained individually for each instance of a distributed keyed file. The image copy of a keyed file is initialized with the statistics from the prime copy whenever a full reconciliation of the file is performed. Refer to the IBM Distributed Data Services/Controller Services Feature for Windows User's Guide for more information about keyed-file statistics. See Chapter 6. Data Distribution for more information about image copies, prime copies, and full reconciliation. See "Distributed Files" for more information about instances.

FdsCloseKeyedFile()

Purpose

Close a keyed file or write the contents of a keyed file to disk.

Syntax

```
#include <fds/file.h>

long FdsCloseKeyedFile(long FileHandle, int Flag)
```

Parameters

FileHandle— input

The file handle obtained from FdsOpenKeyedFile() or FdsCreateKeyedFile().

Flag — input

A flag consisting of the following attributes:

CloseType indicates the type of close request. Valid values are:

FDS_FILE_CLOSE_TYPE_FULL
Close the file. The file handle becomes invalid and all locks on the file are released. This is the default value.

FDS_FILE_CLOSE_TYPE_FLUSH
Write the contents of the file buffers to disk.

NullDataArea indicates whether to reset the file before closing it. The reset of the file will fill all of the data blocks to zeroes. The valid values are:

FDS_FILE_RESET_NO
Do not reset the file. This is the default value.

FDS_FILE_RESET_YES
Reset the data blocks of the file to zeroes before closing it. The file header remains intact. This value is valid only in combination with **FDS_FILE_CLOSE_TYPE_FULL**.

Remarks

If **CloseType** is **FDS_FILE_CLOSE_TYPE_FULL**, **FileHandle** becomes invalid and all locks associated with it are released.

If **CloseType** is **FDS_FILE_CLOSE_TYPE_FLUSH** and the file is distributed with a frequency of distribute on close, the distribution sequence is initiated as if the file were closed, though the file is not closed. **FileHandle** and all locks associated with it remain valid.

If **CloseType** is **FDS_FILE_CLOSE_TYPE_FLUSH** and the file is not distributed or has a distribution frequency of distribute-on-update, no action is taken.

If specified, **FDS_FILE_RESET_YES** is effective only if all of the following conditions apply:

- FDS_FILE_CLOSE_TYPE_FULL is specified. If FDS_FILE_CLOSE_TYPE_FLUSH is specified, -210 FDSERR_FLAG is returned.
- FileHandle is the only active, open instance of the keyed file.
- FDS_FILE_ACCESS_READ_WRITE was specified when the file was opened.

Except as indicated above, the failure to complete a reset request does not prevent the file from being closed. FDS_SUCCESS is returned and the reset failure is logged.

Error Conditions

FdsCloseKeyedFile() returns the following values:
-210 FDSERR_FLAG

-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND

Examples

This example flushes the contents of a keyed file to disk and then closes the file.

```
#include <stdio.h>
#include <fds/file.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/errno.h>
long          rc;                                          // Return from API Call
char          FileName[50] = "d:\\mydir\\myfile";    // File name
long          FileHandle;                                  // Open Keyed File Handle
unsigned int          KeySize;                             // Key Size
unsigned int          RecordSize;                          // Record Size
int           Flag;                                        // Flag value
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------
  // Set flag for FdsOpenKeyedFile API call
  //-------------------------------------------------------------------
  Flag = FDS_FILE_ACCESS_READ_WRITE | FDS_FILE_LOCK_NONE;
  //-------------------------------------------------------------------
  // Open existing keyed file "d:\mydir\myfile"
  //-------------------------------------------------------------------
  rc = FdsOpenKeyedFile( &FileHandle, FileName, &KeySize,
                                    &RecordSize, Flag );
  if ( rc == FDS_SUCCESS )
     {
        //-------------------------------------------------------------------------------------------------------
        // Set flag for FdsCloseKeyedFile API call - to flush the file, but not
        // close the file
        //-------------------------------------------------------------------------------------------------------
        Flag = FDS_FILE_RESET_NO | FDS_FILE_CLOSE_TYPE_FLUSH;
        //-------------------------------------------------------------------------------------
        // Call FdsCloseKeyedFile API to flush "d:\mydir\myfile"
        //-------------------------------------------------------------------------------------
        rc = FdsCloseKeyedFile( FileHandle, Flag );
        printf( "FdsCloseKeyedFile completed with return code = (%d).\n",
                  rc );
        //-------------------------------------------------------------------------------------------------------
        // Set flag for FdsCloseKeyedFile API call to close the file and reset
        // the data blocks of the file to zeros
        //-------------------------------------------------------------------------------------------------------
        Flag = FDS_FILE_RESET_YES | FDS_FILE_CLOSE_TYPE_FULL;
        //-------------------------------------------------------------------------------------
        // Call FdsCloseKeyedFile API to close "d:\mydir\myfile"
        //-------------------------------------------------------------------------------------
        rc = FdsCloseKeyedFile( FileHandle, Flag );
        printf( "FdsCloseKeyedFile completed with
                     return code =(%d).\n", rc );
    } // end if
} // end if
```

```
else
{
  // else process errors
}
```

FdsCreateKeyedFile()

Purpose

Create a new keyed file.

Syntax

```
#include <fds/file.h>

long FdsCreateKeyedFile(long *FileHandlePtr, const char *FileName,
                        unsigned int KeySize,
                        unsigned int RecordSize,
                        unsigned int BlockSize,
                        unsigned long NumBlocks,
                        unsigned long RandDivisor,
                        unsigned int ChainThreshold,
                        int Flag);
```

Parameters

FileHandlePtr — output

Pointer to the location where the file handle is stored. This value is required for all other Keyed-file APIs. This is not the operating-system file handle.

The file handle returned has read/write access to the file. The file should be closed and reopened with FdsOpenKeyedFile() if read-only access is required.

FileName — input

A string specifying the file to open. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

KeySize — input

The key size (in bytes) for the file. This value must be greater than zero, and less than or equal to **RecordSize**.

RecordSize — input

The record size (in bytes) for the file. This value must be greater than or equal to **KeySize** and less than or equal to **BlockSize** minus 4.

BlockSize — input

The block size (in bytes) for the file. This value must be a multiple of 512, from 512 to 4,096. It must also be greater than or equal to **RecordSize** plus
4.

NumBlocks — input

The number of blocks in the file. This value must be greater than or equal to**RandDivisor**.

This value should be large enough for the maximum number of records that will be added to the keyed file plus 20 percent for free space. Calculate this value by dividing the maximum number of records by the number of records per block, and then adding 20 percent.

The smallest allowed number of blocks is 1.

RandDivisor — input

The randomizing divisor for the file. This value must be less than or equal to**NumBlocks**. If this value is 0 (zero), DDS calculates a default value.

Prime numbers are effective randomizing divisors. For example, you might choose the largest prime number that is less than or equal to the total number of blocks in the keyed file.

ChainThreshold — input

The chaining threshold for the file. This value must be less than **NumBlocks**. If a new record is added to a keyed file that causes a chain greater than this value to be created in the file, an informational message is logged to indicate this event. Specifying 0 (zero) for this value suppresses the logging of these messages.

Flag — input

A flag consisting of the following attributes:

FileExistAction indicates the action to take if **FileName** already exists. Valid values are:

FDS_FILE_EXIST_FAIL
The API fails. This is the default value.

FDS_FILE_EXIST_REPLACE
Replace the existing file.

FileLock indicates the type of lock requested for the file. Valid values are:

FDS_FILE_LOCK_EXCLUSIVE
Request exclusive access to the file. No other process can access the file for reading or writing.

FDS_FILE_LOCK_SHARED
Request shared access to the file. No other process can access the file for writing, but other processes can access the file for reading.

FDS_FILE_LOCK_NONE
Request no lock for the file. Other processes can access the file for reading and writing. This is the default value.

HashingAlgorithm indicates the hashing algorithm to be used. See the **IBM Distributed Data Services/Controller Services Feature for Windows** *User's Guide* for more information about hashing algorithms. Valid values are:

FDS_FILE_HASH_POLYNOMIAL
Polynomial algorithm. This is the default value.

FDS_FILE_HASH_XOR
XOR rotation algorithm.

FDS_FILE_HASH_FOLDING
Simple folding algorithm.

Remarks

A new keyed file with the name you specified for the **FileName** parameter is created. The file size is determined by the values of **BlockSize** and **NumBlocks**.

If a file with the same name already exists, the existing file is replaced or -170 FDSERR_EXISTS is returned, depending on the value of **FileExistAction**.

The data blocks in the file are initialized to zeros. This process can take a long time for large files.

Error Conditions

FdsCreateKeyedFile() returns the following values:
-10 FDSERR_ACCESS
-30 FDSERR_BLOCK_SIZE
-50 FDSERR_CHAIN_THRESH
-100 FDSERR_DISK_FULL
-170 FDSERR_EXISTS
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-210 FDSERR_FLAG
-260 FDSERR_IO
-290 FDSERR_KEY_SIZE
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-390 FDSERR_NUM_BLOCKS
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-480 FDSERR_RAND_DIV
-490 FDSERR_REC_SIZE
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example creates a keyed file.

```
#include <stdio.h>
#include <fds/file.h>
#include <fds/fds.h>
```

```
#include <fds/defs.h>
#include <fds/errno.h>


long          rc;                                          // Return from API Call
char          FileName[50] = "d:\\mydir\\myfile";   // File name
long          FileHandle;                                  // Open Keyed File Handle
unsigned int          KeySize = 7;                         // Key Size
unsigned int          RecordSize = 50;             // Record Size
unsigned int          BlockSize = 512;             // Block Size
unsigned long         NumBlocks = 1000;                    // Number of Blocks
unsigned long         RandDivisor = 0;             // Randomizing Divisor
unsigned long         ChainThreshold = 4;                  // Chaining Threshold
int                   Flag;                                // Flag value


// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();


// If initialization was successful
if ( rc == FDS_SUCCESS )
{

    //-------------------------------------------------------------------
    // Set flag for FdsCreateKeyedFile API call
    //-------------------------------------------------------------------Flag = FDS_FILE_EXIST_REPLACE |
                FDS_FILE_LOCK_EXCLUSIVE |
                FDS_FILE_HASH_POLYNOMIAL;

    //-----------------------------------------------------------------------------------

    // Call FdsCreateKeyedFile API to create keyed file "d:\mydir\myfile".

    //-----------------------------------------------------------------------------------

    printf( "FdsCreateKeyedFile completed with return code =
    (%d).\n", rc );
  } // end if
  else
  { // else process errors }
```

FdsDeleteKeyedRecord()

Purpose

Delete a record from a keyed file.

Syntax

```
#include <fds/file.h>

long FdsDeleteKeyedRecord(long FileHandle, void *KeyPtr, unsigned int KeySize);
```

Parameters

FileHandle — input

The file handle obtained from FdsOpenKeyedFile() or FdsCreateKeyedFile().

KeyPtr — input

A pointer to the key of the record to delete. The specified key must not be null (must not contain all zeros).

KeySize —input The size of the key pointed to by **KeyPtr**. This value must equal the key size set by FdsCreateKeyedFile() or obtained from FdsOpenKeyedFile().

Remarks

The record containing the key specified by **KeyPtr** is deleted from the file. The -10 FDSERR_ACCESS error code is returned if the record is locked.

Error Conditions

FdsDeleteKeyedRecord() returns the following values:

-10 FDSERR_ACCESS
-70 FDSERR_CORRUPT
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-270 FDSERR_KEY
-280 FDSERR_KEY_NOT_FOUND
-290 FDSERR_KEY_SIZE
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-530 FDSERR_ROLE_CHANGE

Examples

This example removes a record from a keyed file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long            rc;                                     // Return from API Call
char            FileName[50] = "d:\\mydir\\myfile";     // File name
long            FileHandle;                     // Open Keyed File Handle
unsigned int    KeySize;            // Key Size
unsigned int    RecordSize;                     // Record Size
int             Flag;                           // Flag value
void*           pRecord;            // Pointer to Record
char Buffer[100] = "Record1";       // Record to delete
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-----------------------------------------------------------------
  // Set flag for FdsOpenKeyedFile API call
  //-------------------------------------------------------------------------------------------------
  Flag = FDS_FILE_ACCESS_READ_WRITE | FDS_FILE_LOCK_EXCLUSIVE;
  //-------------------------------------------------------------------------------------------------
  // Open existing keyed file "d:\mydir\myfile"
  //-------------------------------------------------------------------------------------------------
```

```
  rc = FdsOpenKeyedFile( &FileHandle, FileName, &KeySize,
                                        &RecordSize, Flag );
  if ( rc == FDS_SUCCESS )
  {
    //---------------------------------------------------------------------
    // Store key of record to delete in pRecord
    //---------------------------------------------------------------------
    pRecord = (void *) Buffer;
    //---------------------------------------------------------------------
    // Delete record that has key = "Record1"
    //---------------------------------------------------------------------
    rc = FdsDeleteKeyedRecord( FileHandle, pRecord, KeySize );
    printf( "FdsDeleteKeyedRecord completed with return code = (%d).\n", rc );
  } // end if
} // end if
else
{
// else process errors
}
```

FdsOpenKeyedFile()

Purpose

Open an existing keyed file.

Syntax

```
#include <fds/file.h>

long  FdsOpenKeyedFile(long *FileHandlePtr, const char *FileName,       unsigned int *KeySizePtr,
                unsigned int *RecordSizePtr,       int Flag);
```

Parameters

FileHandlePtr — output

Pointer to the location where the file handle is stored. This value is required for all other Keyed-file APIs. This is not the operating-system file handle.

FileName — input

A string specifying the file to open. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

KeySizePtr — output

Pointer to the location where the key size (in bytes) for the file is stored.

RecordSizePtr — output

Pointer to the location where the record size (in bytes) for the file is stored.

Flag — input

A flag consisting of the following attributes:

FileAccess indicates whether write access to the file is requested. Valid values are:

FDS_FILE_ACCESS_READ_ONLY
Request read-only access to the file. This is the default value.

FDS_FILE_ACCESS_READ_WRITE
Request read and write access to the file.

FileLock indicates the type of lock requested for the file. Valid values are:

FDS_FILE_LOCK_EXCLUSIVE
Request exclusive access to the file. No other process can access the file for reading or writing.

FDS_FILE_LOCK_SHARED
Request shared access to the file. No other process can access the file for writing, but other processes can access the file for reading.

FDS_FILE_LOCK_NONE
Request no lock for the file. Other processes can access the file for reading and writing. This is the default value.

Remarks

The file specified by **FileName** is opened with the attributes specified by **Flag**. The system attempts to verify that the file is a valid keyed file.

Error Conditions

This example removes a record from a keyed file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long          rc;                    // Return from API Call
char          FileName[50] = "d:\\mydir\\myfile";    // File name
long          FileHandle;            // Open Keyed File Handle
unsigned int  KeySize;          // Key Size
unsigned int  RecordSize;            // Record Size
int           Flag;                  // Flag value
void*         pRecord;          // Pointer to Record
char          Buffer[100] = "Record1";      // Record to delete
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------
  // Set flag for FdsOpenKeyedFile API call
  //-------------------------------------------------------------------
  Flag = FDS_FILE_ACCESS_READ_WRITE | FDS_FILE_LOCK_EXCLUSIVE;
  //-------------------------------------------------------------------
  // Open existing keyed file "d:\mydir\myfile"
```

```
  //-------------------------------------------------------------------------
  rc = FdsOpenKeyedFile( &FileHandle, FileName, &KeySize, &RecordSize, Flag );
  if ( rc == FDS_SUCCESS )
  {
     //-------------------------------------------------------------------------
     // Store key of record to delete in pRecord
     //-------------------------------------------------------------------------
     pRecord = (void *) Buffer;
     //-------------------------------------------------------------------------
     // Delete record that has key = "Record1"
     //-------------------------------------------------------------------------
     rc = FdsDeleteKeyedRecord( FileHandle, pRecord, KeySize );
     printf( "FdsDeleteKeyedRecord completed with return code = (%d).\n", rc );
  } // end if
} // end if
else
{
   // else process errors
}
```

FdsReadKeyedRecord()

Purpose

Read a record from a keyed file.

Syntax

```
#include <fds/file.h>

long FdsReadKeyedRecord(long FileHandle, void *BufferPtr, unsigned int KeySize, unsigned int
                                    *RecordSizePtr, int Flag);
```

Parameters

FileHandle — input

The file handle obtained from FdsOpenKeyedFile() or FdsCreateKeyedFile().

BufferPtr — input/output

Input A pointer to the key of the record to read. The specified key must not be null (must not contain all zeros).

Output

A pointer to the record containing a matching key.

KeySize — input The size (in bytes) of the key pointed to by **BufferPtr**. This value must equal the key size set by FdsCreateKeyedFile() or obtained from FdsOpenKeyedFile().

RecordSizePtr — input/output

Input A pointer to the maximum size (in bytes) of the record to read.

This value must be greater than or equal to the record size set by FdsCreateKeyedFile() or obtained from FdsOpenKeyedFile(). This value must also be less than or equal to the size of the allocated space pointed to by **BufferPtr**.

Output

If the call succeeds, a pointer to the size (in bytes) of the record that was read. If the call fails and the error code is -490 FDSERR_REC_SIZE,a pointer to the size (in bytes) of the record that could not be read. The value will be equal to the record size set by FdsCreateKeyedFile() or obtained from FdsOpenKeyedFile() in both of these cases.

If the call fails and the error code is not -490 FDSERR_REC_SIZE, the output value is undefined.

Flag — input

A flag consisting of the following attribute:

RecordLock indicates whether to lock the record. Valid values are:

FDS_FILE_RECORD_LOCK_NO

Do not lock the record. This is the default value.

FDS_FILE_RECORD_LOCK_YES

Lock the record. Other processes can continue to read the record, but cannot update it.

Remarks

If **RecordLock** is **FDS_FILE_RECORD_LOCK_YES**, the record is locked until an FdsWriteKeyedRecord() request with **RecordUnlock** equal to **FDS_FILE_RECORD_UNLOCK_YES** is issued, or until the keyed record is released via an FdsReleaseKeyedRecord() request. A record can be read by other processes while it is locked, but it cannot be updated.

Error Conditions

FdsReadKeyedRecord() returns the following values:

-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-70 FDSERR_CORRUPT
-210 FDSERR_FLAG
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-270 FDSERR_KEY
-280 FDSERR_KEY_NOT_FOUND
-290 FDSERR_KEY_SIZE
-350 FDSERR_NODE_NOT_FOUND
-490 FDSERR_REC_SIZE
-530 FDSERR_ROLE_CHANGE
-560 FDSERR_SEQUENCE

Examples

This example removes a record from a keyed file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long          rc;                                          // Return from API Call
char          FileName[50] = "d:\\mydir\\myfile";    // File name
long          FileHandle;                                  // Open Keyed File Handle
unsigned int          KeySize;                             // Key Size
unsigned int          RecordSize;                                  // Record Size
int                   Flag;                                        // Flag value
void*                 pRecord;                             // Pointer to Record
char                  Buffer[100] = "Record1";       // Record to delete
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------------
  // Set flag for FdsOpenKeyedFile API call
  //-------------------------------------------------------------------------
  Flag = FDS_FILE_ACCESS_READ_WRITE | FDS_FILE_LOCK_EXCLUSIVE;
  //-------------------------------------------------------------------------
  // Open existing keyed file "d:\mydir\myfile"
  //-------------------------------------------------------------------------
  rc = FdsOpenKeyedFile( &FileHandle, FileName, &KeySize,
                                        &RecordSize, Flag );
  if ( rc == FDS_SUCCESS )
  {
     //-------------------------------------------------------------------------
     // Store key of record to delete in pRecord
     //-------------------------------------------------------------------------
     pRecord = (void *) Buffer;
     //-------------------------------------------------------------------------
     // Delete record that has key = "Record1"
     //-------------------------------------------------------------------------
     rc = FdsDeleteKeyedRecord( FileHandle, pRecord, KeySize );
     printf( "FdsDeleteKeyedRecord completed with return
                  code = (%d).\n", rc );
  } // end if
} // end if
else
{
  // else process errors
}
```

FdsReleaseKeyedRecord()

Purpose

Release a lock on a record in a keyed file.

Syntax

```
#include <fds/file.h>

long FdsReleaseKeyedRecord(long FileHandle, void *KeyPtr,          unsigned int KeySize);
```

Parameters

FileHandle — input
The file handle obtained from FdsOpenKeyedFile() or FdsCreateKeyedFile().

KeyPtr — input
A pointer to the key of the record to release. The specified key must not be null (must not contain all zeros).

KeySize — input The size of the key pointed to by **KeyPtr**. This value must equal the key size set by FdsCreateKeyedFile() or obtained from FdsOpenKeyedFile().

Remarks

The lock on the record containing the key specified by **KeyPtr** is released. The lock must have been previously established by FdsReadKeyedRecord().

Error Conditions

FdsWriteKeyedRecord() returns the following values:
-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-70 FDSERR_CORRUPT
-180 FDSERR_FILE_FULL
-210 FDSERR_FLAG
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-270 FDSERR_KEY
-290 FDSERR_KEY_SIZE
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-490 FDSERR_REC_SIZE
-530 FDSERR_ROLE_CHANGE
-560 FDSERR_SEQUENCE

Examples

This example releases a locked record.

```
#include <stdio.h>
#include <fds/file.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/errno.h>
long          rc;                              // Return from API Call
char          FileName[50] = "d:\\mydir\\myfile";     // File name
long          FileHandle;                      // Open Keyed File Handle
unsigned int          KeySize;   // Key Size
unsigned int          RecordSize;          // Record Size
int                   Flag;                // Flag value
```

```
void*                   pRecord;  // Pointer to Record
char                    Buffer[100] = "Record1";       // Record to release
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //---------------------------------------------------------------------
  // Set flag for FdsOpenKeyedFile API call
  //---------------------------------------------------------------------
  Flag = FDS_FILE_ACCESS_READ_WRITE | FDS_FILE_LOCK_EXCLUSIVE;
  //-----------------------------------------------------------------------
  // Open existing keyed file "d:\mydir\myfile"
  //-----------------------------------------------------------------------
  rc = FdsOpenKeyedFile( &FileHandle, FileName, &KeySize,
                                    &RecordSize, Flag );
  if ( rc == FDS_SUCCESS )
  {
     //-------------------------------------------------------------------------------------------------------
     // Set flag for FdsReadKeyedRecord API call to lock the record
     //-------------------------------------------------------------------------------------------------------
     Flag = FDS_FILE_RECORD_LOCK_YES;
     //---------------------------------------------------------------------
     // Store key of record to lock in pRecord
     //---------------------------------------------------------------------
     pRecord = (void *) Buffer;
     //---------------------------------------------------------------------
     // Read record that has key = "Record1"
     //---------------------------------------------------------------------
     rc = FdsReadKeyedRecord( FileHandle, // File Handle from Open
     pRecord, // Pointer to Record
     KeySize, // Key size
     &RecordSize, // Record size
     Flag ); // Flag value
     if ( rc != FDS_SUCCESS )
     {
     printf("FdsReadKeyedRecord failed (%d).\n", rc );
     return(-1);
     }
     //-----------------------------------------------------------------------
     // Store key of record to unlock in pRecord
     //-----------------------------------------------------------------------
     pRecord = (void *) Buffer;
     //-----------------------------------------------------------------------
     // Unlock record that has key = "Record1"
     //-----------------------------------------------------------------------
     rc = FdsReleaseKeyedRecord( FileHandle, pRecord, KeySize );
     printf( "FdsReleaseKeyedRecord completed with return code = (%d).\n", rc );
  } // end if
} // end if
else
{
// else process errors
}
```

FdsWriteKeyedRecord()

Purpose

Write a record to a keyed file.

Syntax

```
#include <fds/file.h>

long FdsWriteKeyedRecord(long FileHandle, void *RecordPtr,
                         unsigned int KeySize,
                         unsigned int RecordSize,
                         int Flag);
```

Parameters

FileHandle — input

The file handle obtained from FdsOpenKeyedFile() or FdsCreateKeyedFile().

RecordPtr — input

A pointer to the record to write. The first **KeySize** bytes of the record must contain a non-null key (must not consist of all zeros).

KeySize — input

The size (in bytes) of the key at the front of the record pointed to by **RecordPtr**. This value must equal the key size set by FdsCreateKeyedFile() or obtained from FdsOpenKeyedFile().

RecordSize — input

The size (in bytes) of the record pointed to by **RecordPtr**. This value must equal the record size set by FdsCreateKeyedFile() or obtained from FdsOpenKeyedFile().

Flag — input

A flag consisting of the following attribute:

RecordUnlock indicates whether to unlock the record after the write. Valid values are:

FDS_FILE_RECORD_UNLOCK_NO

Do not unlock the record. This is the default value.

FDS_FILE_RECORD_UNLOCK_YES

Unlock the record after the write. A lock on the record must have been previously established using FdsReadKeyedRecord().

Remarks

If the record specified by **RecordPtr** contains a key that already exists in a record in the file, the record in the file is replaced with the new record. If the existing record is locked, **FDS_FILE_RECORD_UNLOCK_YES** must be specified. If the existing record is not locked, you do not have to specify a value (**FDS_FILE_RECORD_UNLOCK_NO** is the default).

If the record specified by **RecordPtr** contains a key that does not already exist

in a record in the file, the new record is added to the file.

Error Conditions

FdsWriteKeyedRecord() returns the following values:

-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-70 FDSERR_CORRUPT
-180 FDSERR_FILE_FULL
-210 FDSERR_FLAG
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-270 FDSERR_KEY
-290 FDSERR_KEY_SIZE
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-490 FDSERR_REC_SIZE
-530 FDSERR_ROLE_CHANGE
-560 FDSERR_SEQUENCE

Examples

This example updates a record in a keyed file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long          rc;                                          // Return from API Call
char          FileName[50] = "d:\\mydir\\myfile";    // File name
long          FileHandle;                                  // Open Keyed File Handle
unsigned int          KeySize;                             // Key Size
unsigned int          RecordSize;                                  // Record Size
int           Flag;                                        // Flag value
void*         pRecord;                                     // Pointer to Record
char          Buffer[100] = "Record1";       // Key of record to write
char          Buffer2[100] = "Record1 New Record 1 data";     // New Record
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
 {
 //-------------------------------------------------------------------------
 // Set flag for FdsOpenKeyedFile API call
 //-------------------------------------------------------------------------
 Flag = FDS_FILE_ACCESS_READ_WRITE | FDS_FILE_LOCK_SHARED;
 //-------------------------------------------------------------------------
 // Open existing keyed file "d:\mydir\myfile"
 //-------------------------------------------------------------------------
 rc = FdsOpenKeyedFile( &FileHandle, FileName, &KeySize, &RecordSize, Flag );
 if ( rc == FDS_SUCCESS )
 {
  //-------------------------------------------------------------------------
  // Set flag for FdsReadKeyedRecord API call to lock the record
  //-------------------------------------------------------------------------
  Flag = FDS_FILE_RECORD_LOCK_YES;
  //-------------------------------------------------------------------------
```

```
   // Store key of record to read in pRecord
   //-------------------------------------------------------------------
   pRecord = (void *) Buffer;
   //-------------------------------------------------------------------
   // Read record that has key = "Record1"
   //-------------------------------------------------------------------
   rc = FdsReadKeyedRecord( FileHandle, // File Handle from Open
   pRecord, // Pointer to Record
   KeySize, // Key size
   &RecordSize, // Record size
   Flag ); // Flag value
   if ( rc != FDS_SUCCESS )
   {
      printf( "FdsReadKeyedRecord failed (%d).\n", rc );
      return(-1);
   }
   //-------------------------------------------------------------------------------------------------------------------
   // Call FdsWriteKeyedRecord to write "Record1" back to the file after
   // changing the data in the record. The read was done with lock so
   // the write must be done with unlock.
   //-------------------------------------------------------------------------------------------------------------------
   Flag = FDS_FILE_RECORD_UNLOCK_YES;
   //-------------------------------------------------------------------
   // Store the new record in pRecord
   //-------------------------------------------------------------------
   pRecord = (void *) Buffer2;
   //-------------------------------------------------------------------
   // Call FdsWriteKeyedRecord API
   //-------------------------------------------------------------------
   rc = FdsWriteKeyedRecord( FileHandle, pRecord, KeySize, RecordSize, Flag );
   printf( "FdsWriteKeyedRecord completed with return code = (%d).\n", rc );
  } // end if
} // end if
else
{
  // else process errors
}
```

Sequential File Services

Sequential files are composed of a sequence of records of variable lengths. Records are read in order from the beginning of the file to the end. New records are added to the end of the file. An existing record cannot be deleted, replaced, or removed from the file.

Sequential files are stored on DASD as contiguous data with self-defining records. Each record consists of a 4-byte record header followed by user data. The first 2 bytes of the record header contain a delimiter that is used only in error-recovery situations. The hex value of the delimiter is hex BEEF. The second 2 bytes of the record header contain the length of the subsequent user data. This structure is summarized in the following table:

Table 1. Sequential-File, Record-Header Format

Description	Size	Notes
Delimiter	2 bytes	Hexidecimal value is hex BEEF.

Length of user data	2 bytes	Range is from 1 to 49 152.
User data	User-defined (must be within above range)	No content restrictions.

The APIs provided by File Services for sequential file manipulation are:

FdsCloseSeqFile() — Close a sequential file

FdsFindNextSeqRecord() — Move the file pointer to the next valid record in a sequential file

FdsOpenSeqFile() — Open or create a sequential file

FdsReadSeqRecord() — Read a record from a sequential file

FdsReturnSeqFilePos() — Return the file position indicator for a sequential file

FdsSeekSeqFilePos() — Seek to a point in a sequential file

FdsWriteSeqRecord() — Append a record to a sequential file

FdsCloseSeqFile()

Purpose

Close a sequential file.

Syntax

```
#include <fds/file.h>

long FdsCloseSeqFile(long FileHandle);
```

Parameters

FileHandle — input
The file handle obtained from FdsOpenSeqFile().

Remarks

FileHandle becomes invalid and any locks on the file are released.

Error Conditions

FdsCloseSeqFile() returns the following values:
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED

Examples

This example closes a sequential file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
```

```
#include <fds/file.h>
#include <fds/errno.h>
long        rc;                             // Return from API Call
long        FileHandle;                     // File Handle returned from Open
const       char * FileName = "d:\\itemrec.dat";    // Name of file to open
int         Flag;                           // Flag value
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //---------------------------------------------------------------------
  // Set Flag for FdsOpenSeqFile API call
  //---------------------------------------------------------------------
  Flag = FDS_FILE_EXIST_OPEN |
  FDS_FILE_ACCESS_READ_ONLY |
  FDS_FILE_LOCK_NONE;
  //---------------------------------------------------------------------
  // Open "d:\itemrec.dat"
  //---------------------------------------------------------------------
  rc = FdsOpenSeqFile( &FileHandle, FileName, Flag );
  // If Open was successful
  if ( rc == FDS_SUCCESS )
  {
     //---------------------------------------------------------------------
     // Call FdsCloseSeqFile API to close "d:\itemrec.dat"
     //---------------------------------------------------------------------
     rc = FdsCloseSeqFile( FileHandle );
     printf("FdsCloseSeqFile completed with return code = (%d) \n", rc);
  } // end if
} // end if
else
{
// else process errors
}
```

FdsFindNextSeqRecord()

Purpose

Move the file pointer to the next valid record in a sequential file

Syntax

```
#include <fds/file.h>

long FdsFindNextSeqRecord(long FileHandle);
```

Parameters

FileHandle — input
The file handle value obtained from FdsOpenSeqFile().

Remarks

The file pointer is advanced to the next valid record in the file, beginning at the current position of the file pointer. The file pointer is advanced, even if it is located on a valid record when the call is made, unless a valid record cannot be found.

Use this API for error recovery when the File Services component or the caller detects a damaged record in a file.

Error Conditions

FdsFindNextSeqRecord() returns the following values:

-10 FDSERR_ACCESS
-160 FDSERR_EOF
-220 FDSERR_HANDLE
-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND
-530 FDSERR_ROLE_CHANGE

Examples

This example moves the file pointer to the next valid record in a sequential file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long          rc;                    // Return from API Call
long          FileHandle;            // File Handle returned from Open
const         char * FileName = "d:\\itemrec.dat"; // Name of file to create
int           Flag;                  // Flag value
char          Record[500] = "";    // Record to read
unsigned int            RecordSize = sizeof(Record);  // Size of Record
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------------
  // Set Flag for FdsOpenSeqFile API call
  //-------------------------------------------------------------------------
  Flag = (unsigned int) FDS_FILE_EXIST_OPEN |
  (unsigned int) FDS_FILE_ACCESS_READ_ONLY |
   (unsigned int) FDS_FILE_LOCK_SHARED;
  //-------------------------------------------------------------------------
  // Open "d:\itemrec.dat"
  //-------------------------------------------------------------------------
  rc = FdsOpenSeqFile( &FileHandle, FileName, Flag );
  // If Open was successful
  if ( rc == FDS_SUCCESS )
  {
     //-------------------------------------------------------------------------
     // Call FdsFindNextSeqRecord API to advance the position of the
     // file pointer in the file to the next record
     //
     // (e.g. if the file pointer currently points to the first record
     // in the file, after calling the FdsFindNextSeqRecord API, the
     // file pointer will point to the second record in the file)
```

```
    //-----------------------------------------------------------------
    rc = FdsFindNextSeqRecord( FileHandle );
    printf("FdsFindNextSeqRecord completed with return code = (%d).\n", rc);
    if ( rc == FDS_SUCCESS )
    {
      // Read the second record in the file
      rc = FdsReadSeqRecord( FileHandle, (void*) Record, &RecordSize );
    } // end if
  } // end if
} // end if
else
{
  // else process errors
}
```

FdsOpenSeqFile()

Purpose

Open or create a sequential file.

Syntax

```
#include <fds/file.h>

long FdsOpenSeqFile(long *FileHandlePtr, const char *FileName, int Flag);
```

Parameters

FileHandlePtr — output

Pointer to the location where the file handle is stored. This value is required for all the other Sequential-file APIs. This handle is not the operating system file handle.

FileName — input

A string specifying the file to open. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

Flag — input

A flag consisting of the following attributes:

FileExistAction indicates the action to take if **FileName** already exists. Valid values are:

FDS_FILE_EXIST_OPEN

Open the existing file. This is the default value.

FDS_FILE_EXIST_REPLACE

Replace the existing file. **FDS_FILE_ACCESS_READ_WRITE** must also be specified if this value is specified.

FileAccess indicates whether write access to the file is requested. Valid values are:

FDS_FILE_ACCESS_READ_ONLY

Request read-only access to the file. This is the default value.

FDS_FILE_ACCESS_READ_WRITE
Request read and write access to the file.

FileLock indicates the type of lock requested for the file. Valid values are:

FDS_FILE_LOCK_EXCLUSIVE
Request exclusive access to the file. No other process can access the file for reading or writing.

FDS_FILE_LOCK_SHARED
Request shared access to the file. No other process can access the file for writing, but other processes can access the file for reading.

FDS_FILE_LOCK_NONE
Request no lock for the file. Other processes can access the file for reading and writing. This is the default value.

Remarks

A file named **FileName** is opened with the attributes specified by **Flag**. If the file exists, it is either opened or replaced, depending on the value of **FileExistAction**.If the file does not exist and **FDS_FILE_ACCESS_READ_WRITE** is specified, a new file is created. Otherwise, an error is returned if the file does not exist.

File Services does not attempt to validate the contents of an existing file if an existing file is opened.

The file pointer is placed at the first record in the file.

File Services does not implement access control for file locking and sharing. These features are implemented by the operating system and file system based on the **Flag** parameter.

Error Conditions

FdsOpenSeqFile() returns the following values:
-10 FDSERR_ACCESS
-70 FDSERR_CORRUPT
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-210 FDSERR_FLAG
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-410 FDSERR_OVERFLOW
-460 FDSERR_QUEUE_NOT_FOUND
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example opens a sequential file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long        rc;                         // Return from API Call
long        FileHandle;                 // File Handle returned from Open
const       char * FileName = "d:\\itemrec.dat"; // Name of file to create
int         Flag;                       // Flag value
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------
  // Set flag for FdsOpenSeqFile API call
  //-------------------------------------------------------------------
  Flag = FDS_FILE_EXIST_REPLACE |
  FDS_FILE_ACCESS_READ_WRITE |
  FDS_FILE_LOCK_SHARED;
  //-------------------------------------------------------------------------------------------------------
  // Call FdsOpenSeqFile API to create/replace "d:\itemrec.dat"
  //-------------------------------------------------------------------------------------------------------
  rc = FdsOpenSeqFile( &FileHandle, FileName, Flag );
  printf( "FdsOpenSeqFile completed with return code = (%d) \n", rc );
} // end if
else
{
// else process errors
}
```

FdsReadSeqRecord()

Purpose

Read a record from a sequential file.

Syntax

```
#include <fds/file.h>

long FdsReadSeqRecord(long FileHandle,
                      void *BufferPtr,
                           unsigned int *BufferSizePtr);
```

Parameters

FileHandle — input

The file handle value obtained from FdsOpenSeqFile().

BufferPtr — output

A pointer to the location where the record that was read is stored.

BufferSizePtr — input/output

Input A pointer to the maximum size (in bytes) of the record to read. This value must be less than or equal to the size of the allocated space pointed to by *BufferPtr*.

Output If the call succeeds, a pointer to the size (in bytes) of the record read. The output value is less than or equal to the input value in this case.

If the call fails and the error code is -40 FDSERR_BUFFER_SIZE, a pointer to the size (in bytes) of the record that could not be read. The output value is greater than the input value in this case.

If the call fails and the error code is not -40 FDSERR_BUFFER_SIZE, the output value is undefined.

Remarks

The record beginning at the current file pointer is read. File Services does not adjust the file pointer before processing the request.

If the input value of **BufferSize** is at least as large as the size of the user-data portion of the record, the user data is placed in the location specified by **BufferPtr**, and the output value of **BufferSize** is the actual size of the user data. See “Sequential File Services” for more information about the size of the user-data portion of the record.

If the input value of **BufferSize** is smaller than the size of the user data, -40 FDSERR_BUFFER_SIZE is returned, and the output value of **BufferSize** is the actual size (in bytes) of the user data. The contents of the location specified by **BufferPtr** are undefined in this case. You can immediately attempt to read the record again, indicating a larger value for **BufferSize**.

If the call succeeds, FDS_SUCCESS is returned and the file pointer is advanced to the next record.

Error Conditions

FdsReadSeqRecord() returns the following values:

-10 FDSERR_ACCESS

-20 FDSERR_ADDRESS

-40 FDSERR_BUFFER_SIZE

-70 FDSERR_CORRUPT

-160 FDSERR_EOF

-220 FDSERR_HANDLE

-222 FDSERR_HANDLE_FORCED_CLOSED

-260 FDSERR_IO

-350 FDSERR_NODE_NOT_FOUND

-530 FDSERR_ROLE_CHANGE

Examples

This example reads a record in a sequential file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long          rc;                    // Return from API Call
long          FileHandle;            // File Handle returned from Open
const         char * FileName = "d:\\itemrec.dat";    // Name of file to open
int           Flag;                  // Flag value
char          Record[500] = "";    // Record to read
unsigned int            RecordSize = sizeof(Record);  // Size of Record
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------
  // Set Flag for FdsOpenSeqFile API call
  //-------------------------------------------------------------------
  Flag = FDS_FILE_EXIST_OPEN |
  FDS_FILE_ACCESS_READ_ONLY |
  FDS_FILE_LOCK_NONE;
  //-------------------------------------------------------------------
  // Open "d:\itemrec.dat"
  //-------------------------------------------------------------------
  rc = FdsOpenSeqFile( &FileHandle, FileName, Flag );
  // If Open was successful
  if ( rc == FDS_SUCCESS )
  {
     //-------------------------------------------------------------------
     // Call FdsReadSeqRecord API to read the first record in the file
     //-------------------------------------------------------------------
     rc = FdsReadSeqRecord( FileHandle, (void*) Record, &RecordSize );
     printf("FdsReadSeqRecord completed with return code = (%d) \n"
               " ---> Record read = (%s)        \n", rc,Record);
  } // end if
} // end if
else
{
  // else process errors
}
```

FdsReturnSeqFilePos()

Purpose

Return the file position indicator for a sequential file.

Syntax

```
#include <fds/file.h>

long FdsReturnSeqFilePos(long FileHandle, unsigned long *PositionPtr);
```

Parameters

FileHandle — input
The file handle value obtained from FdsOpenSeqFile().

PositionPtr — output
A pointer to the location where the file position indicator will be stored.

Remarks

The current position of the file pointer is returned. This value can be used later, via a call to FdsSeekSeqFilePos(), to return the file pointer to its current position.

Error Conditions

FdsReturnSeqFilePos() returns the following values:
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-350 FDSERR_NODE_NOT_FOUND
-530 FDSERR_ROLE_CHANGE

Examples

This example saves the current file position of a sequential file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long          rc;                          // Return from API Call
long          FileHandle;                  // File Handle returned from Open
const         char * FileName = "d:\\itemrec.dat";    // Name of file to open
int           Flag;                        // Flag value
unsigned long          Position = 0;       // File position
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------------
  // Set Flag for FdsOpenSeqFile API call
  //-------------------------------------------------------------------------
  Flag = FDS_FILE_EXIST_OPEN |
  FDS_FILE_ACCESS_READ_WRITE |
  FDS_FILE_LOCK_NONE;
  //-------------------------------------------------------------------------
  // Open "d:\itemrec.dat"
  //-------------------------------------------------------------------------
  rc = FdsOpenSeqFile( &FileHandle, FileName, Flag );
```

```
  // If Open was successful
  if ( rc == FDS_SUCCESS )
  {
     //------------------------------------------------------------------------
     // Call FdsReturnSeqFilePos API to save the current position
     // of the file pointer.
     //
     // This call is used in conjunction with the FdsSeekSeqFilePos,
     // which will return the file pointer to the saved position.
     // (See FdsSeekSeqFilePos() for more information.)
     //------------------------------------------------------------------------
     rc = FdsReturnSeqFilePos( FileHandle, &Position );
     printf("FdsReturnSeqFilePos completed with return code = (%d).\n", rc);
  } // end if
} // end if
else
{
// else process errors
}
```

FdsSeekSeqFilePos()

Purpose

Seek to a previously determined point in a sequential file.

Syntax

```
#include <fds/file.h>

long FdsSeekSeqFilePos(long FileHandle, unsigned long Position);
```

Parameters

FileHandle — input

The file handle value obtained from FdsOpenSeqFile().

Position — input

The file position indicator obtained from FdsReturnSeqFilePos(). Position = –1 positions the pointer at the end of the file.

Context

The file pointer is moved to the location specified by **Position**. This value must have been previously obtained from a call to FdsReturnSeqFilePos() to ensure correct alignment of the file pointer.

Remarks

FdsSeekSeqFilePos() returns the following values:

-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-350 FDSERR_NODE_NOT_FOUND
-530 FDSERR_ROLE_CHANGE

Examples

This example seeks to a previously saved position in a sequential file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long          rc;                                  // Return from API Call
long          FileHandle;                          // File Handle returned from Open
const         char * FileName = "d:\\itemrec.dat";    // Name of file to open
int           Flag;                                                  // Flag value
char          Record[500] = "";                    // Record to read
unsigned int          RecordSize = sizeof(Record);  // Size of Record
unsigned long         Position = 0;                                  // File position
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------------
  // Set Flag for FdsOpenSeqFile API call
  //-------------------------------------------------------------------------
  Flag = FDS_FILE_EXIST_OPEN |
  FDS_FILE_ACCESS_READ_ONLY |
  FDS_FILE_LOCK_EXCLUSIVE;
  //-------------------------------------------------------------------------
  // Open "d:\itemrec.dat"
  //-------------------------------------------------------------------------
  rc = FdsOpenSeqFile( &FileHandle, FileName, Flag );
  // If Open was successful
  if ( rc == FDS_SUCCESS )
  {
     //-----------------------------------------------------------------------
     // Save the current file position
     //-----------------------------------------------------------------------
     rc = FdsReturnSeqFilePos( FileHandle, &Position );
     //-----------------------------------------------------------------------
     // Call FdsSeekSeqFilePos API to go to the position saved as a
     // result of calling FdsReturnSeqFilePos.
     //
     // (see FdsReturnSeqFilePos for more information)
     //-----------------------------------------------------------------------
     rc = FdsSeekSeqFilePos( FileHandle, Position );
     printf("FdsSeekSeqFilePos completed with return code = (%d).\n", rc);
     if ( rc == FDS_SUCCESS )
     {
        //--------------------------------------------------------------------
        // Read the record in the file
        //--------------------------------------------------------------------
        rc = FdsReadSeqRecord( FileHandle, (void*) Record,
                                           &RecordSize );
        printf(" Record read = (%s) \n", Record);
     } // end if
  } // end if
} // end if
else
```

```
{
  // else process errors
}
```

FdsWriteSeqRecord()

Purpose

Append a record to a sequential file.

Syntax

```
#include <fds/file.h>

long FdsWriteSeqRecord(long FileHandle, const void *BufferPtr, unsigned int BufferSize);
```

Parameters

FileHandle — input

The file handle value obtained from FdsOpenSeqFile().

BufferPtr — input

A pointer to the location at which the data to write is stored.

BufferSize — input

The size (in bytes) of the record to write. If the file is distributed in a broadcast domain, this value must be less than or equal to 4,096. If the file is distributed in the mirrored domain or if the file is not distributed, this value must be less than or equal to 49,152. In either case, this value must also be less than or equal to the size (in bytes) of the allocated space pointed to by **BufferPtr**.

Remarks

A record is appended to the file. The file pointer is advanced to the end of the file before the write operation. The first **BufferSize** bytes of the data specified by **BufferPtr** constitute the user-data portion of the record. The value for **BufferSize** must be within the range specified for the user-data portion of the record. See “Sequential File Services” for more information about the size of the user-data portion of the record.

If the call succeeds, FDS_SUCCESS is returned, the entire record is appended to the file, and the file pointer remains at the end of the file. If the call fails, the portion of the record that is written, as well as the location of the file pointer, will vary with the type of error.

Error Conditions

FdsWriteSeqRecord() returns the following values:

-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-70 FDSERR_CORRUPT
-100 FDSERR_DISK_FULL
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED

-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-490 FDSERR_REC_SIZE
-530 FDSERR_ROLE_CHANGE

Examples

This example writes a record in a sequential file.

```
#include <stdio.h>
#include <string.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long        rc;                 // Return from API Call
long        FileHandle;         // File Handle returned from Open
const       char * FileName = "d:\\itemrec.dat"; // Name of file to create
int         Flag;               // Flag value
char        Record[500];        // Record to read
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //-------------------------------------------------------------------------
  // Set Flag for FdsOpenSeqFile API call
  //-------------------------------------------------------------------------
  Flag = FDS_FILE_EXIST_OPEN |
  FDS_FILE_ACCESS_READ_WRITE |
  FDS_FILE_LOCK_EXCLUSIVE;
  //-------------------------------------------------------------------------
  // Open "d:\itemrec.dat"
  //-------------------------------------------------------------------------
  rc = FdsOpenSeqFile( &FileHandle, FileName, Flag );
  // If Open was successful
  if ( rc == FDS_SUCCESS )
  {
     //-------------------------------------------------------------------------
     // Set record to write to "d:\itemrec.dat"
     //-------------------------------------------------------------------------
     strcpy( Record, "Write this new record to the file" );
     //-------------------------------------------------------------------------
     // Call FdsWriteSeqRecord API to write record to "d:\itemrec.dat"
     // - record will be added as the last record in the file
     //-------------------------------------------------------------------------
     rc = FdsWriteSeqRecord( FileHandle, (void*) Record, sizeof(Record) );
     printf("FdsWriteSeqRecord completed with return code = (%d).\n", rc);
  } // end if
} // end if
else
{
  // else process errors
}
```

Binary File Services

Binary files are byte-stream files. In byte-stream files, data can be read from or written to any position within the file. The data within the file has no predetermined structure, and the API calls used to manipulate the data do not translate any control characters. In addition to read and write operations, the file-pointer location can be moved, ranges of the file can be locked in either a read-shared or exclusive mode, and file buffering and caching can be controlled.

Binary files provide more flexibility for the application to manage the data in a file, but they require more complexity in the application.

The APIs provided by File Services for binary file manipulation are:

- **FdsCloseBinFile()** — Close a binary file
- **FdsFlushBinFile()** — Flush any data buffered for a binary file
- **FdsOpenBinFile()** — Open or create a binary file
- **FdsReadBinFile()** — Read from a binary file
- **FdsSeekBinFilePos()** — Move the file pointer in a binary file
- **FdsSetBinFileLocks()** — Lock or unlock a range in a binary file
- **FdsWriteBinFile()** — Write to a binary file

FdsCloseBinFile()

Purpose

Close a binary file.

Syntax

```
#include <fds/file.h>

long FdsCloseBinFile(long FileHandle);
```

Parameters

FileHandle — input
The file handle obtained from FdsOpenBinFile().

Remarks

FileHandle becomes invalid, and any file pointers or locks on the file are released.

Error Conditions

FdsCloseBinFile() returns the following values:
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-350 FDSERR_NODE_NOT_FOUND

Examples

This example closes a binary file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long        rc;                                 // Return from API Call
long        FileHandle;                         // File Handle returned from Open
const       char * FileName = "d:\\binary.dat";      // Name of file to close
int         Flag;                               // Flag value
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
  //---------------------------------------------------------------------
  // Set Flag for FdsOpenBinFile API call
  //---------------------------------------------------------------------
  Flag = FDS_FILE_EXIST_OPEN
  FDS_FILE_ACCESS_READ_ONLY
  FDS_FILE_LOCK_NONE;
  //---------------------------------------------------------------------
  // Open "d:\binary.dat"
  //---------------------------------------------------------------------
  rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
  // If Open was successful
  if ( rc == FDS_SUCCESS )
  {
     //---------------------------------------------------------------------
     // Call FdsCloseBinFile API to close "d:\binary.dat"
     //---------------------------------------------------------------------
     rc = FdsCloseBinFile( FileHandle );
     printf("FdsCloseBinFile completed with return code = (%d) \n", rc);
  } // end if
} // end if
else
{
 // else process errors
}
```

FdsFlushBinFile()

Purpose

Force all updates to the binary file to be written to disk.

Syntax

```
#include <fds/file.h>

long FdsFlushBinFile(long FileHandle);
```

Parameters

FileHandle — input
The file handle value obtained from FdsOpenBinFile().

Remarks

All updates to the binary file that can be cached in buffers are written to disk.

Error Conditions

FdsFlushBinFile() returns the following values:
-10 FDSERR_ACCESS
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND
-530 FDSERR_ROLE_CHANGE

Examples

This example will flush any updates that have been cached in buffers, so that they will be written to the disk.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long        rc;                 // Return from API Call
long        FileHandle;         // File Handle returned from Open
const       char * FileName = "d:\\binary.dat"; // Name of file to flush
int         Flag;               // Flag value
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------
   // Set Flag for FdsOpenBinFile API call
   //-------------------------------------------------------------------
   Flag = FDS_FILE_EXIST_OPEN |
   FDS_FILE_ACCESS_READ_WRITE |
   FDS_FILE_LOCK_NONE |
   FDS_FILE_WRITETHRU_YES;
   //-------------------------------------------------------------------
   // Open "d:\binary.dat"
   //-------------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
   // If Open was successful
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------------
      // Call FdsFlushBinFile API to flush the file to disk
      //-------------------------------------------------------------------
      rc = FdsFlushBinFile( FileHandle );
      printf("FdsFlushBinFile completed with return code = (%d) \n", rc);
   } // end if
} // end if
else
{
```

```
    // else process errors
}
```

FdsOpenBinFile()

Purpose

Open or create a binary file.

Syntax

```
#include <fds/file.h>

long FdsOpenBinFile(long *FileHandlePtr, const char *FileName,
                    int Flag);
```

Parameters

FileHandlePtr — output

Pointer to the location where the file handle will be stored. This value is required for all the other binary-file APIs. This file handle is not the operating-system file handle.

FileName — input

A string specifying the file to open. The string can contain logical names, but must resolve to a retail path specification. See “File Names and Queue Names” for more information.

Flag — input

A flag consisting of the following attributes:

FileExistAction indicates the action to take if **FileName** already exists. Valid values are:

FDS_FILE_EXIST_OPEN

Open the existing file. This is the default value

FDS_FILE_EXIST_REPLACE

Replace the existing file. **FDS_FILE_ACCESS_READ_WRITE** must
also be specified if this value is specified.

FileNewAction indicates the action to take if **FileName** does not already exist. Valid values are:

FDS_FILE_NEW_CREATE

Create the file. This is the default.

FDS_FILE_NEW_FAIL

The API fails if the file does not exist and an error is returned.

FileAccess indicates whether write access to the file is requested. Valid values are:

FDS_FILE_ACCESS_READ_ONLY

Request only read access to the file. This is the default value.

FDS_FILE_ACCESS_READ_WRITE

Request write and read access to the file.

FileLock indicates the type of lock requested for the file. Valid values are:

FDS_FILE_LOCK_EXCLUSIVE
Request exclusive access to the file. No other file handle can access the file for reading or writing.

FDS_FILE_LOCK_SHARED
Request shared access to the file. No other file handle can access the file for writing, but other file handles can access the file for reading.

FDS_FILE_LOCK_NONE
Request no lock for the file. Other processes can access the file for reading and writing. This is the default value.

WriteThru indicates whether buffering or caching of file input and output is disabled. Valid values are:

FDS_FILE_WRITETHRU_YES
File buffering or caching is disabled. All FdsWriteBinFile() operations are immediately committed to DASD. All other DDS file services automatically set **FDS_FILE_WRITETHRU_YES**.

FDS_FILE_WRITETHRU_NO
File buffering or caching is not disabled. The file buffering or caching capabilities of the underlying operating system and file system are exploited. This is the default.

Remarks

A file named **FileName** is opened with the attributes specified by **Flag**. If the file exists, it is either opened or replaced, depending on the value of **FileExistAction**.If the file does not exist, it is either opened or the API fails, depending upon the value of **FileNewAction**.

The file pointer is placed at the first byte in the file.

File Services does not implement access control for file locking and sharing, nor does it implement file buffering or caching. These features are implemented by the operating system and file system based on the **Flag** parameter.

Error Conditions

FdsOpenBinFile() returns the following values:

-10 FDSERR_ACCESS
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-210 FDSERR_FLAG
-260 FDSERR_IO
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-410 FDSERR_OVERFLOW

-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

This example opens a binary file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long        rc;                          // Return from API Call
long        FileHandle;                  // File Handle returned from Open
const       char * FileName = "d:\\binary.dat"; // Name of file to create
int         Flag;                        // Flag value
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------
   // Set flag for FdsOpenBinFile API call. Uses the default action of
   // creating file if it does not exist (FDS_FILE_NEW_CREATE is default).
   //-------------------------------------------------------------------------------------------------------------
   Flag = FDS_FILE_EXIST_REPLACE |
   FDS_FILE_ACCESS_READ_WRITE |
   FDS_FILE_LOCK_SHARED |
   FDS_FILE_WRITETHRU_YES;
   //-------------------------------------------------------------------------------------------------
   // Call FdsOpenBinFile API to create/replace "d:\binary.dat"
   //-------------------------------------------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
   printf("FdsOpenBinFile completed with return code = (%d) \n", rc);
} // end if
else
{
  // else process errors
}
```

FdsQueryBinFileSize()

Purpose

Query the size of a binary file.

Syntax

```
#include <fds/file.h>

long FdsQueryBinFileSize(long FileHandle, unsigned long *CurrentSize);
```

Parameters

FileHandle — input

The file handle value obtained from FdsOpenBinFile()

CurrentSize — output

A pointer to the location of the current size of the binary file (in bytes). If this API has not completed successfully, this value is undefined.

Remarks

The current size of the binary file (in bytes) is returned.

Error Conditions

FdsQueryBinFileSize() returns the following values:
-220 FDSERR_HANDLE
-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND
-530 FDSERR_ROLE_CHANGE

Examples

This example returns the size of a binary file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>
long        rc;                                 // Return from API call
long         FileHandle;                        // File Handle from Open
const       char * FileName = "d:\\binary.dat"; // Name of file to read
int         Flag;                               // Flag value
unsigned long        CurrentSize;            // Current file size
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------
   // Set Flag for FdsOpenBinFile API call
   //-------------------------------------------------------------------------
   Flag = FDS_FILE_EXIST_OPEN |
   FDS_FILE_ACCESS_READ_ONLY |
   FDS_FILE_LOCK_NONE |
   FDS_FILE_WRITETHRU_NO;
   //-------------------------------------------------------------------------
   // Open "d:\binary.dat"
   //-------------------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
   if ( rc == FDS_SUCCESS )
   {
      //----------------------------------------------------------------------
      // Call FdsQueryBinFileSize to get the size (in bytes) of
      // "d:\binary.dat"
      //----------------------------------------------------------------------
      rc = FdsQueryBinFileSize( FileHandle, &CurrentSize);
      printf( "FdsQueryBinFileSize completed with return code = (%d).\n" " ---> File Size = (%d) \n",
      rc, CurrentSize );
   } // end if
} // end if
```

```
else
{
  // else process errors
}
```

FdsReadBinFile()

Purpose

Read a range of data from a binary file.

Syntax

```
#include <fds/file.h>

long FdsReadBinFile(long FileHandle, void *BufferPtr,
                    unsigned int *NBytesPtr,long Offset,
                    unsigned long Origin);
```

Parameters

FileHandle — input

The file handle value obtained from FdsOpenBinFile().

BufferPtr — input/output

A pointer to the location where the data that was read will be stored.

NBytesPtr — input/output

Input A pointer to the maximum amount (in bytes) of the data to read. This value must be less than or equal to 59 000. In either case, this value must be less than or equal to the size of the allocated space pointed to by **BufferPtr**.

Output If the call succeeds, a pointer to the amount (in bytes) of data actually read. The output value is always less than or equal to the input value.

Offset — input

The number of bytes to move the file pointer. The **Offset** parameter is used in conjunction with the **Origin** parameter to determine the new, file-pointer position. If **Offset** is greater than 0 (zero), the file pointer is moved that many bytes from the **Origin** position toward the end of the file. If **Offset** is less than 0 (zero), the file pointer is moved that many bytes from the **Origin** position towards the beginning of the file. If **Offset** is 0 (zero), the file pointer is moved in accordance with the **Origin** parameter.

Origin — input

The location from which to move the file pointer based on the value of **Offset**. The **Origin** is specified as:

FDS_FILE_START_OF_FILE

Apply the value in **Offset** from the beginning of the file (the file pointer is 0).

FDS_FILE_CURRENT_POS

Apply the value in **Offset** from the current file pointer position.

FDS_FILE_END_OF_FILE

Apply the value in **Offset** from the end of the file (the file pointer is equal to the size of the file).

Remarks

The data beginning at the current file pointer is read. File Services adjusts the file pointer before processing the request, based on the values supplied for **Offset** and **Origin**.

If the value specified by **NBytesPtr** is greater than the number of bytes remaining in the file, the actual number of bytes that were read is returned in **NBytesPtr**, the -160 FDSERR_EOF error is returned, and the file pointer is set to the end of the file.

If **Offset** is set to 0 (zero), and **Origin** is set to **FDS_FILE_CURRENT_POS**, no seek action is performed before the read.

If the call succeeds, the FDS_SUCCESS message is returned and the file pointer is advanced by the number of bytes that were read.

If the call fails, the location of the file pointer is not advanced.

Error Conditions

FdsReadBinFile() returns the following values:
-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-160 FDSERR_EOF
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND
-490 FDSERR_REC_SIZE
-530 FDSERR_ROLE_CHANGE
-558 FDSERR_SEEK_TYPE

Examples

This example reads from a binary file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long rc; // Return from API Call
long FileHandle; // File Handle from Open
const char * FileName = "d:\\binary.dat"; // Name of file to read
int Flag; // Flag value
char Buffer[500]; // Data read
unsigned int NBytes = 500; // Number of bytes to read
long Offset; // Offset
unsigned long Origin; // Origin
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
```

```
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //------------------------------------------------------------------------
   // Set Flag for FdsOpenBinFile API call
   //------------------------------------------------------------------------
   Flag = FDS_FILE_EXIST_OPEN |
   FDS_FILE_ACCESS_READ_ONLY |
   FDS_FILE_LOCK_NONE |
   FDS_FILE_WRITETHRU_NO;
   //------------------------------------------------------------------------
   // Open "d:\binary.dat"
   //------------------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
   // If Open was successful
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------------------------------------------------
      // Set Origin and Offset, to designate where in the file to read
      //-------------------------------------------------------------------------------------------------------
      Origin = FDS_FILE_END_OF_FILE;   // Go to the end of the file
      Offset = -10;                                    // Begin reading at offset -10
      //-------------------------------------------------------------------------------------------
      // Call FdsReadBinFile API to read the first record in the file
      //-------------------------------------------------------------------------------------------
      rc = FdsReadBinFile( FileHandle,
               (void *) Buffer,
               &NBytes,
               Offset,
               Origin );
      printf("FdsReadBinFile completed with return code = (%d) \n"
               " ---> Bytes read = (%s) \n"
               " ---> Number of Bytes read = (%d) \n",
               rc,
               Buffer,
               NBytes);
   } // end if
} // end if
else
{
  // else process errors
}
```

FdsSeekBinFilePos()

Purpose

Move the file pointer to a specific location within the binary file.

Syntax

```
#include <fds/file.h>

long FdsSeekBinFilePos(long FileHandle,
                       long Offset,
                       unsigned long Origin,
```

unsigned long *_NewPosPtr_);

Parameters

FileHandle — input

The file-handle value obtained from FdsOpenBinFile().

Offset — input

The number of bytes to move the file pointer. The **Offset** parameter is used in conjunction with the **Origin** parameter to determine the new, file-pointer position. If **Offset** is greater than 0 (zero), the file pointer is moved that many bytes from the **Origin** position toward the end of the file. If **Offset** is less than 0 (zero), the file pointer is moved that many bytes from the **Origin** position toward the beginning of the file. If **Offset** is 0 (zero), the file pointer is moved in accordance with the **Origin** parameter.

Origin — input

The location from which to move the file pointer based on the value of **Offset**. The **Origin** is specified as:

FDS_FILE_START_OF_FILE

Apply the value in **Offset** from the beginning of the file (the file pointer is 0).

FDS_FILE_CURRENT_POS

Apply the value in **Offset** from the current file pointer position.

FDS_FILE_END_OF_FILE

Apply the value in **Offset** from the end of the file (the file pointer is equal to the size of the file).

NewPosPtr — output

A pointer to the location where the new file position is stored.

Remarks

The file pointer is moved to the location specified by **Offset** and **Origin**.

It is not an error to seek past the end of the file, and the file size is not affected by seeking past the end of the file. It is an error to specify a negative, file-pointer position.

If FDS_SUCCESS is returned, NewPosPtr indicates the current position of the file pointer.

Error Conditions

FdsSeekBinFilePos() returns the following values:
-10 FDSERR_ACCESS
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-350 FDSERR_NODE_NOT_FOUND
-530 FDSERR_ROLE_CHANGE
-558 FDSERR_SEEK_TYPE

Examples

This example will move the file pointer to a specified location in a binary file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>
long        rc;                         // Return from API Call
long        FileHandle;                 // File Handle returned from Open
const       char * FileName = "d:\\binary.dat";      // Name of file to use
int         Flag;                       // Flag value
long        Offset;                     // Offset
unsigned long       Origin;             // Origin
unsigned long       NewPos = 0;         // Current file position
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------------
   // Set Flag for FdsOpenBinFile API call
   //---------------------------------------------------------------
   Flag = FDS_FILE_EXIST_OPEN |
   FDS_FILE_ACCESS_READ_ONLY |
   FDS_FILE_LOCK_NONE |
   FDS_FILE_WRITETHRU_NO;
   //---------------------------------------------------------------
   // Open "d:\binary.dat"
   //---------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
   // If Open was successful
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------------------------------------------------
      // Set Origin and Offset, to determine byte count in the file
      //-------------------------------------------------------------------------------------------------------
      Origin = FDS_FILE_END_OF_FILE; // Go to the end of the file
      Offset = 0; // Offset 0
      //-------------------------------------------------------------------------------------------------
      // Call FdsSeekBinFilePos API to go to the last byte in the file
      //-------------------------------------------------------------------------------------------------
      rc = FdsSeekBinFilePos( FileHandle,
                              Offset,
                              Origin,
                              &NewPos );
      printf("FdsSeekBinFilePos completed with return code = (%d).\n"
             " ---> The size of the file = (%d) bytes. \n",
             rc,
             NewPos);
   } // end if
} // end if
else
{
  // else process errors
}
```

FdsSetBinFileLocks()

Purpose

Lock or unlock a range of bytes within a binary file.

Syntax

```
#include <fds/file.h>

long FdsSetBinFileLocks(long FileHandle, long Offset, unsigned
                        int NBytes, int Flag);
```

Parameters

FileHandle — input
The file-handle value obtained from FdsOpenBinFile().

Offset — input
The offset (in bytes) from the beginning of the file to the starting position of the range to lock or unlock.

NBytes — input
The length of the range to lock or unlock. **NBytes** must be a positive non-zero integer.

Flag — input
Used to control the specific lock or unlock action. It consists of the following attributes:

RangeLockAction indicates the lock or unlock action for the range specified. Valid values are:

FDS_FILE_LOCK
Lock the specified region. This is the default.

FDS_FILE_UNLOCK
Unlock the specified region.

RangeFileLock indicates the type of lock requested for the range specified. These flags are valid only if **FDS_FILE_LOCK** is also specified. Valid values are:

FDS_FILE_LOCK_RANGE_SHARED
Lock the region in shared mode. All programs can read the data in the specified region, but cannot change the data. This includes the program that issues this API call. This is the default.

FDS_FILE_LOCK_RANGE_EXCLUSIVE
Lock the region in exclusive mode. Only the program that acquires this lock can read or change the data in the specified region.

Remarks

Regions of the binary file are unlocked or locked.

The locking operation itself is managed by the underlying operating system, so the results of this API may differ among operating systems.

Error Conditions

This example will lock a specified number of bytes in a binary file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>

long        rc;                             // Return from API Call
long        FileHandle;                     // File Handle returned from Open
const       char * FileName = "d:\\binary.dat";      // Name of file to lock
int         Flag;                           // Flag value
long        Offset;                         // Offset
unsigned long       NBytes = 0;             // Number of bytes tolock
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------
   // Set Flag for FdsOpenBinFile API call
   //-------------------------------------------------------------------
   Flag = FDS_FILE_EXIST_OPEN |
   FDS_FILE_ACCESS_READ_ONLY |
   FDS_FILE_LOCK_NONE |
   FDS_FILE_WRITETHRU_NO;
   //-------------------------------------------------------------------
   // Open "d:\binary.dat"
   //-------------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
   // If Open was successful
   if ( rc == FDS_SUCCESS )
   {
   //-------------------------------------------------------------------------------------------
   // Set Offset and the Number of bytes in the file to lock
   //-------------------------------------------------------------------------------------------
   Offset = 0;                                  // Offset 0
      NBytes = 4;                   // Number of bytes to lock

      //-------------------------------------------------------------------------------------------------------
      // Set the Flag for the FdsSetBinFileLocks API, to lock the bytes
      //
      // Everyone will be allowed to read these bytes, but no one //(including this process) will be
      allowed to write to these bytes
      //-------------------------------------------------------------------------------------------------------
      Flag = FDS_FILE_LOCK | FDS_FILE_LOCK_RANGE_SHARED;
      //-------------------------------------------------------------------------------------------------------
      // Call FdsSetBinFileLocks API to lock the first 4 bytes in the file
      //-------------------------------------------------------------------------------------------------------
      rc = FdsSetBinFileLocks( FileHandle,
      Offset,
      NBytes,
      Flag );
      printf("FdsSetBinFileLocks completed with return code = (%d).\n"
               " ---> (%d) bytes were locked. \n",
               rc,
               NBytes);
```

```
   } // end if
} // end if
else
{
   // else process errors
}
```

FdsSetBinFileSize()

Purpose

Set the size of a binary file.

Syntax

```
#include <fds/file.h>

long FdsSetBinFileSize(long FileHandle, unsigned long NewSize);
```

Parameters

FileHandle — input
The file-handle value obtained from FdsOpenBinFile().

NewSize — input
The new size of the binary file in bytes.

Remarks

The size of the binary file is set to the size specified by **NewSize**.

Error Conditions

FdsSetBinFileSize() returns the following values:
-10 FDSERR_ACCESS
-100 FDSERR_DISK_FULL
-220 FDSERR_HANDLE
-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-530 FDSERR_ROLE_CHANGE

Examples

This example returns the size of a binary file.

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/defs.h>
#include <fds/errno.h>

long        rc;                             // Return from API call
long        FileHandle;                     // File Handle from Open
const       char * FileName = "d:\\binary.dat";      // Name of file to read
int         Flag;                           // Flag value
```

```
unsigned long        NewSize; // New size of file

// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------
   // Set Flag for FdsOpenBinFile API call
   //-------------------------------------------------------------------
   Flag = FDS_FILE_EXIST_OPEN |
   FDS_FILE_ACCESS_READ_WRITE |
   FDS_FILE_LOCK_NONE |
   FDS_FILE_WRITETHRU_NO;
   //-------------------------------------------------------------------
   // Open "d:\binary.dat"
   //-------------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );
   if ( rc == FDS_SUCCESS )
   {
      //-----------------------------------------------------------------------------------
      // Set the new size (in bytes) of "d:\binary.dat"
      //-----------------------------------------------------------------------------------
      NewSize = 1024;
      //-----------------------------------------------------------------------------------
      // Call FdsSetBinFileSize to set the size (in bytes) of
      // "d:\binary.dat"
      //-----------------------------------------------------------------------------------
      rc = FdsSetBinFileSize( FileHandle, NewSize);
      printf( "FdsSetBinFileSize completed with return code = (%d).\n",
                  rc );
   } // end if
} // end if
else
{
// else process errors
}
```

FdsWriteBinFile()

Purpose

Write a range of data to a binary file.

Syntax

```
#include <fds/file.h>

long FdsWriteBinFile(long FileHandle, const void *BufferPtr, unsigned int *NBytesPtr, long Offset,
                                 unsigned long Origin);
```

Parameters

FileHandle — input

The file-handle value obtained from FdsOpenBinFile().

BufferPtr — input

A pointer to the location at which the data to write is stored.

NBytesPtr — input/output

Input Pointer to the location where the size (in bytes) of the data to write is stored. This value must be less than or equal to 59 000. In either case, this value must also be less than or equal to the size (in bytes) of the allocated space pointed to by **BufferPtr**.

If the current file pointer plus the size specified in **NBytesPtr** is greater than the current size of the file, File Services attempts to extend the end of the file.

Output

When this API has completed successfully, the data stored in the location pointed to by **NBytesPtr** is replaced with the actual number of bytes written, which could be less than the requested number of bytes in error situations.

Offset — input

The number of bytes to move the file pointer. The **Offset** parameter is used in conjunction with the **Origin** parameter to determine the new file pointer position. If **Offset** is greater than 0 (zero), the file pointer is moved that many bytes from the **Origin** position towards the end of the file. If **Offset** is less than 0 (zero), the file pointer is moved that many bytes from the **Origin** position toward the beginning of the file. If **Offset** is 0 (zero), the file pointer is moved in accordance with the **Origin** parameter.

Origin — input

The location from which to move the file pointer based on the value of **Offset**. The **Origin** is specified as:

FDS_FILE_START_OF_FILE

Apply the value in **Offset** from the beginning of the file (the file pointer is 0 (zero)).

FDS_FILE_CURRENT_POS

Apply the value in **Offset** from the current file-pointer position.

FDS_FILE_END_OF_FILE

Apply the value in **Offset** from the end of the file (the file pointer is equal to the size of the file).

Remarks

The file-pointer position is moved before a write operation, based on the values provided for **Offset** and **Origin**. The data is written starting at the new file-pointer position.

If **Offset** is set to 0 (zero) and **Origin** is set to **FDS_FILE_CURRENT_POS**, no seek action is performed before the write.

If the call succeeds, the FDS_SUCCESS message is returned and the entire range of data is written.

If the call fails, **NBytesPtr** contains the actual number of bytes written, if any.

In all situations, the file pointer is advanced by the actual number of bytes written.

Error Conditions

FdsWriteBinFile() returns the following values:
-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-100 FDSERR_DISK_FULL
-220 FDSERR_HANDLE
-222 FDSERR_HANDLE_FORCED_CLOSED
-260 FDSERR_IO
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-490 FDSERR_REC_SIZE
-530 FDSERR_ROLE_CHANGE
-558 FDSERR_SEEK_TYPE

Examples

This example writes to a binary file.

```
#include <stdio.h>
#include <string.h>
#include <fds/fds.h>
#include <fds/defs.h>
#include <fds/file.h>
#include <fds/errno.h>

long        rc;                                 // Return from API Call
long        FileHandle;                         // File Handle returned from Open
const       char * FileName = "d:\\binary.dat";      // Name of file to write
int         Flag;                               // Flag value
char        Buffer[500]; // Data to write
unsigned int        NBytes = 500;       // Number of bytes to write
long                Offset;             // Offset
unsigned long       Origin;             // Origin

// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();

// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------------------
   // Set Flag for FdsOpenBinFile API call
   //---------------------------------------------------------------------
   Flag = FDS_FILE_EXIST_OPEN |
   FDS_FILE_ACCESS_READ_WRITE |
   FDS_FILE_LOCK_NONE |
   FDS_FILE_WRITETHRU_NO;

   //---------------------------------------------------------------------
   // Open "d:\binary.dat"
   //---------------------------------------------------------------------
   rc = FdsOpenBinFile( &FileHandle, FileName, Flag );

   // If Open was successful
   if ( rc == FDS_SUCCESS )
   {
      //---------------------------------------------------------------------------------------------------
```

```
        // Set data to write to "d:\binary.dat"
        //-------------------------------------------------------------------------------------------------------
        strcpy( Buffer, "Write this to the file" );
        NBytes = strlen( Buffer );
        //-------------------------------------------------------------------------------------------------------
        // Set Origin and Offset, to designate where in the file to write
        //-------------------------------------------------------------------------------------------------------
        Origin = FDS_FILE_START_OF_FILE;          // Go to the beginning of the file
        Offset = 0;                  // Begin writing at offset 0
        //-------------------------------------------------------------------------------------------------------
        // Call FdsWriteBinFile API to write record to "d:\binary.dat"
        // - buffer will be written beginning at byte 0
        //-------------------------------------------------------------------------------------------------------
        rc = FdsWriteBinFile( FileHandle,
                              Buffer,
                              &NBytes,
                              Offset,
                              Origin );

        printf("FdsWriteBinFile completed with return code = (%d).\n",
                rc);
    } // end if
} // end if
else
{
// else process errors
}
```

Chapter 5. Node Control

The Node Control APIs allow you to view a list of all nodes known to the DDS system as well as to obtain the status of the acting primary distributor.

Node List

DDS maintains a list of all node IDs known to the DDS system, and each node's communication status with the acting primary distributor. The list includes nodes that DDS has detected as being active on the system as well as user-defined nodes that are not yet active.

An API is provided for applications to obtain this list. The list is maintained on the acting primary distributor, but it can be obtained by calling the API from any node in the system.

Before calling the API, an array of FDS_NODE_INFO structures must be declared by your application. The FDS_NODE_INFO structure is defined in the DDS header file NODES.H. This structure consists of a node ID and a status flag that will be set DDS to either FDS_ACTIVE or FDS_INACTIVE, as defined in defs.h. See Appendix A. Data Types for a definition of the FDS_NODE_INFO structure. An unsigned, integer variable must also be declared that contains the size of the memory buffer allocated for the array of FDS_NODE_INFO structures.

A void pointer to the array and a pointer to the size of the array buffer are

passed as variables to the API, which updates the array with the node list. The array size must be large enough to contain all of the node IDs and node status information contained in the node list. To determine the array size, multiply the size of the FDS_NODE_INFO structure by the number of node IDs.

The list of known node IDs and status might change while DDS is running, so this API should be called by an application each time a current list of node IDs is required.

This API returns successfully only when DDS is running on both the node that calls the API and the acting primary distributor node, and when communication is established between the two nodes.

See “FdsGetNodes()” for information about how to use this API

FdsGetNodes()

Purpose

Obtain a list of all node IDs known to the DDS system and each node’s communication status with the primary node.

Syntax

```
#include <fds/defs.h>
#include <fds/nodes.h>


long FdsGetNodes( void *NodeList, unsigned int *BufferSize );
```

Parameters

NodeList — input/output

Input A void pointer to the allocated memory in which to store an array of FDS_NODE_INFO structures.

Output
When this API completes successfully, an array of FDS_NODE_INFO structures is copied into the memory pointed to by this parameter. Each FDS_NODE_INFO structure contains a node ID and a node status flag. See Appendix A. Data Types for more information about the FDS_NODE_INFO structure.

BufferSize — input/output

Input When this API is called, this parameter must point to an integer that specifies the length of the **NodeList** buffer.

Output
When this API returns successfully, the length in bytes of the data returned in the **NodeList** buffer is stored in the integer pointed to by this parameter. If this API returns the error -40 FDSERR_BUFFER_SIZE,the required buffer size is stored in the integer pointed to by this parameter, and the list of node IDs is not returned.

Remarks

FdsGetNodes() is used to obtain a list of all node IDs known to the DDS system and each node's communication status with the primary node. This API should be called every time a current list of node IDs or node status information is required, because the node list or the node status might change while DDS is running.

When FdsGetNodes() is called on a node that is not communicating with the acting primary distributor, it returns an FDSERR_ROLE_NOT_FOUND return code, indicating that the current node is not online and therefore cannot obtain status information for any other node.

Error Conditions

FdsGetNodes() returns the following values:

-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-550 FDSERR_ROLE_NOT_FOUND
-580 FDSERR_TIMEOUT

Examples

This example declares an array of FDS_NODE_INFO variables that holds the node ID and status for 100 nodes. The FdsGetNodes() API is called to update the array with the current node list and status.

```
#include <fds/fds.h>
#include <fds/nodes.h>
#include <fds/defs.h>
#include <fds/errno.h>


long rc;
FDS_NODE_INFO NodeList[100]
unsigned int BufferLength;


// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful


if ( rc == FDS_SUCCESS )

{

  BufferLength = sizeof(FDS_NODE_INFO) * 100;

  rc = FdsGetNodes((void*) NodeList, &BufferLength);

  if (rc != FDS_SUCCESS)
  {
    /* perform error processing */
  }

}
```

Obtaining the Status of the Acting Primary Distributor

The Node Control component opens a queue on a node whenever it is activated as the acting primary distributor, and closes it when it is deactivated as the acting primary distributor. This queue provides a method for applications to receive notification messages when the acting primary distributor is no longer online.

In order for an application to receive notification messages, it must create a queue of its own to receive those messages. Then it must open the queue defined by the constant FDS_ONLINE_Q on the primary using the FdsOpenQ() function, specifying the handle to its own opened queue for the NotificationQHandle parameter.

The queue that is specified by the NotificationQHandle parameter will receive a message whenever the queue is closed on the acting primary distributor. The application can then attempt to open the queue on the node that assumes the acting primary distributor role.

The message that is received by the application's queue is defined as FDS_IPC_MSG, and an FDS_IPC_MSG_STRUCT data structure is received with the message. The FDS_IPC_MSG_STRUCT data structure contains the handle of the closed queue and a reason code for the message. Applications should compare this handle with the handle received from the FdsOpenQ() function when fdsOnlineQ was opened. Both the FDS_IPC_MSG message and the FDS_IPC_MSG_STRUCT data structure are defined in the ipc.h include file.

See "FdsOpenQ()" for more information about the FdsOpenQ() function.

The fdsOnlineQ queue does not accept any messages written to its queue. Its sole purpose is to provide a method for determining when the role changes on the acting primary distributor.

The return codes:

-350 FDSERR_NODE_NOT_FOUND,
-460 FDSERR_QUEUE_NOT_FOUND,
and -550 FDSERR_ROLE_NOT_FOUND

are normal error codes returned by the FdsOpenQ() function when the acting primary distributor is not online or when a role change is in progress.

This example opens a queue called MyApplQ for receiving FDS_IPC_MSG messages.

```
#include <fds/fds.h>
#include <fds/ipc.h>
#include <fds/defs.h>
#include <fds/errno.h>
#define MAX_Q_SIZE 4096
```

```
long MyReadQFn(void)
{
  long        MyApplQHandle;
  long        Timeout = -1;                    // wait forever
  long        PrimaryQHandle;
  FDS_IPC_MSG_STRUCT        Buffer;
  unsigned int          BufferSize;
  int         MessageType;
  long        rc;
  char        OnlineQueueName(23);

  // Initialize DDS. Could use FdsInit2() instead of FdsInit()
  rc = FdsInit();
  // If initialization was successful
  if ( rc == FDS_SUCCESS )
  {
     rc = FdsCreateQ("MyApplQ",                    // name of local queue
     MAX_Q_SIZE,                                   // amount of data the queue can hold
     &MyApplQHandle);                              // handle to this queue
     if (rc == FDS_SUCCESS )
     {
        strcpy(OnlineQueueName, "<FDSFDXAP::>");
        strcat(OnlineQueueName, FDS.ONLINE_Q; //<FDSFDXAP::>fdsOnlineQ
        rc = FdsOpenQ(OnlineQueueName,             // primary queue name
                              MyApplQHandle,                  // notification queue
                               Timeout,        // time to wait until queue is open
                               &PrimaryQHandle);              // queue handle
        if (rc == FDS_SUCCESS )
        {
        BufferSize = sizeof(Buffer);
              rc = FdsReadQ(MyApplQHandle,          // handle to local queue
                                    &BufferSize,          // max size of single message
                                                // and size of message returned
                                    (void *) &Buffer,     // message data
                                    Timeout, // time to wait for a message
                                    &MessageType);   // type of message received.
              else
              {
                    if (MessageType == FDS_IPC_MSG)
                    {
                       if (Buffer.ClosedQHndl == PrimaryQHandle)
                       {
                          // do whatever the application
                          // needs to do when the
                          // primary is not online.
                       }
                    }
              }
           }
     }
     if (rc != FDS_SUCCESS )
     {
        // handle errors
     }
     return rc;
  }
}
```

Chapter 6. Data Distribution

This chapter describes the Data Distribution component of DDS. It also describes the following APIs, which are available for use with the Data Distribution component:

- **FdsActivateAsPrimary()** — Activate the local node as the acting primary distributor
- **FdsAddDomainNode()** — Add a node to a broadcast domain
- **FdsCreateBcastDomain()** — Create a broadcast domain
- **FdsCreateSyncID()** — Create a synchronization ID
- **FdsDeactivatePrimary()** — Change the role at the local node from acting
- primary distributor to acting backup distributor
- **FdsDeleteBcastDomain()** — Delete a broadcast domain
- **FdsDeleteDomainNode()** — Remove a node from a broadcast domain
- **FdsGetDomainList()** — Get the list of all the domains in the system
- **FdsGetDomainNodes()** — Get the list of nodes that are in a broadcast domain
- **FdsQueryBackupState()** — Query the state of the backup distributor
- **FdsQueryDistribution()** — Query a distribution directory entry for a file or subdirectory
- **FdsSetDistribution()** — Modify the distribution characteristics of files and subdirectories
- **FdsSetupDistMonitor()** — Prepare to receive notification of data-distribution events
- **FdsSetupSyncIDNotify()** — Prepare to receive notification of file or directory synchronization

The Data Distribution component provides a file-distribution service that replicates data on multiple nodes. The component synchronizes each image during normal operations and performs file reconciliation when failed nodes are brought back into service.

The Data Distribution component is optional on any particular node, though at least one node in a multi-node system must be installed and configured as the primary distributor. The prime copy of all distributed files resides on the primary distributor. As the prime copy of a distributed file is updated, renamed, or deleted, the changes are distributed to all nodes that have an image of that file. The prime copy of a distributed file is the only copy of the file that can be changed or updated. The changes are distributed according to the distribution frequency of the file (see “Distribution Frequency”)

In addition to file distribution, the Data Distribution component provides a reconciliation service. This service ensures that if a node misses updates for any reason, each distributed file on the node is resynchronized with the prime copy of the file when LAN communication is established with the primary distributor.

Two different methods are used for distributing updates and performing file reconciliation. Connection-oriented messages are exchanged between the node being updated (the primary distributor) and the backup node. Broadcast is used for distributing updates to subordinate nodes to maximize performance.

Note: The Data Distribution component does not interoperate with IBM 4680 Operating System data distribution or with IBM 4690 Operating System data distribution.

Distributed Files

Files are distributed across multiple nodes. An **instance** is a copy of a distributed file on a given node within a distribution domain. There are three types of instances:

- Prime copy: resides on the acting primary distributor node
- Backup copy: resides on the acting backup distributor node
- Image copy: all instances other than the prime copy

File Types

All types of files can be distributed. All operations that result in updates to a distributed file, including deletions for files and subdirectories, are distributed except for file-attribute operations (the read-only, system, hidden, archive flags and file security attributes). A copy operation is treated by the Data Distribution component as an update to the copied-to file. If the copied-to file exists before the copy operation, and if it is a distributed file, the update is distributed. If the copied-to file does not exist before the copy, the update is distributed (that is, the copied-to file is created on image nodes) if and only if the newly created file is created in a subdirectory that is distributed.

DDS can only distribute files to which DDS has access. DDS runs under the System account, so any files that are to be distributed must be accessible to the System (Administrator) account.

Note: DDS supports the distribution of files up to a maximum size of 4 GB.

If the **DistRenamedFile** configuration keyword is set to NO, the net effect of renaming a file or subdirectory is the same as that of a copy followed by a deletion of the original file or subdirectory. The renamed-to file or subdirectory will remain distributed after the rename if and only if the new file or subdirectory is in a subdirectory that is distributed. However, if the **DistRenamedFile** configuration keyword is set to YES (the default), the effect is the same in all cases, except that distributed files not in a distributed directory will remain distributed, though with a new name.

Your applications do not have to use any special APIs to cause data distribution to occur once a file or subdirectory has been specified as distributed. For example, file updates that result from native, operating-system file operations to byte-stream files (such as WriteFile()) are distributed. (See the note under “Reconciliation” for information about Data Distribution’s use of distributed files.)

Your applications can access a keyed file as a byte-stream file, using your native, operating-system file operations instead of the keyed-file APIs available with DDS. When native, operating-system file operations are used to modify a distributed keyed file, the Data Distribution component distributes the file as if it were a byte-stream file. When keyed-file APIs are used, keyed-file updates are distributed. To ensure correct distribution, an application should not have a keyed file open with write access via both native operating system and keyed-file APIs concurrently, and should open a distributed keyed file in a way that prevents other processes

from having write access to it when native, operating-system calls will be used to modify it. You can provide this protection by specifying the **FDS_FILE_LOCK_EXCLUSIVE** or **FDS_FILE_LOCK_SHARED** flag on FdsOpenBinFile().

Distribution Directory

Data distribution allows distribution to be managed at both the file level and the subdirectory level. You can specify the distribution of entire subdirectories without having to be aware of the specific files that exist in them. A distribution directory provides this capability.

The distribution directory determines which files are distributed, the nodes to which the files are distributed, and the distribution frequency for each file. Each entry in the directory represents either a single file or a subdirectory, and stores the following information:

Name Data distribution supports a hierarchical name space. Files with arbitrarily long path specifications can be distributed.

Subdirectory indicator
The subdirectory indicator specifies whether the directory entry represents a file or a subdirectory.

Domain type
The distribution domain type specifies either mirrored domain or broadcast domain.

Distribution domain name
The domain name is a broadcast domain name if the type is broadcast domain. There is only one mirrored domain, so no name is required in this case. Note that a file or subdirectory can be distributed to one distribution domain at most.

Distribution frequency
The distribution frequency is one of the following:

- Distribute on close
- Distribute on update

See “Distribution Frequency” for an explanation of distribution frequency.

Scope qualifier
The scope qualifier is applicable only to directory entries that represent subdirectories. The two possible values and their meanings are:

FILE Only the files in the subdirectory are distributed.
TREE All files and subdirectories are distributed.

The following APIs are provided to manipulate the distribution directory:

- **FdsAddDomainNode()** -Add a node to a broadcast domain

- **FdsCreateBcastDomain()** -Create a broadcast domain
- **FdsDeleteBcastDomain()** -Delete a broadcast domain
- **FdsDeleteDomainNode()** -Remove a node from a broadcast domain
- **FdsGetDomainNodes()** -Get the list of nodes that are in a broadcast domain
- **FdsQueryDistribution()** -Query a distribution directory entry for a file or subdirectory
- **FdsSetDistribution()** -Modify the distribution characteristics of files and subdirectories

Directory Management

The distribution directory resides on the primary distributor node. Because the backup distributor must be prepared to assume the role of primary distributor at any time, it maintains a duplicate copy of the directory. In addition, each subordinate node maintains locally that portion of the directory with entries relevant to it.

Logical Names

Logical names can be used to cause the instances of a given distributed file or subdirectory to have different path names on the nodes to which it is distributed. This function provides flexibility when using a system of nodes where every node is not configured identically in terms of applications and disks.

To use logical names, define a logical name for a distribution directory entry that resolves to a different file or subdirectory operating-system path name on each node within the distribution domain. The logical name is the operating-system path name from the distribution directory entry, prefixed with the percent character (%).

Note: Because Windows files systems are not case-sensitive, and the Name Services component is case-sensitive, logical names that contain the percent character must be uppercase.

The percent character (%), although not reserved, has a special meaning for data distribution. It identifies a logical name on the acting primary distributor that itself resolves to another logical name. This second logical name must exist on each node within the distribution directory. The second logical name can be used by applications and the Data Distribution component to access the prime copy or an image copy of the distributed file, even though the operating system path name might differ on each node. All of the logical names mentioned above must be active on each node within the distribution domain before the creation of the associated distribution-directory entry.

For example, assume that you want to distribute a subdirectory within a broadcast domain that contains three nodes:

- The primary distributor
- The backup distributor

- A node with a node ID of OTIS

The operating-system path name of the subdirectory on the configured primary distributor is c:\otis_stuff\config_files\. The operating-system path name of the subdirectory on the configured backup distributor is d:\otis_stuff\config_files\. The operating-system path name of the subdirectory on node OTIS is c:\config_files\.

To distribute the subdirectory, follow these steps:

1. Define the following logical names on the configured primary distributor:

Logical Name	Resolved Name
<%C:\OTIS_STUFF\CONFIG_FILES\>	otis_stuff
<otis_stuff>	c:\otis_stuff\config_files\

2. Define the following logical names on the configured backup distributor:

Logical Name	Resolved Name
<%D:\OTIS_STUFF\CONFIG_FILES\>	otis_stuff
<otis_stuff>	c:\otis_stuff\config_files\

3. Define the following logical names on node OTIS:

Logical Name	Resolved Name
<%C:\CONFIG_FILES\>	otis_stuff
<otis_stuff>	c:\config_files\

See Chapter 7. Name Services and the refer to **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for information about how to create logical names.

The Data Distribution component uses the logical names that begin with the percent character (%) to detect when a logical name must be used when accessing files and subdirectories on different nodes within the distribution domain. This logical name resolves to the logical name that must be used by the Data Distribution component to access the file or subdirectory on different nodes within the distribution domain.

Distribution Frequency

The effective distribution frequency of a file is specified in the distribution directory as either distribute-on-close or distribute-on-update. Changes are distributed when the file is closed or when the contents are flushed if the distribution frequency is distribute on close. (You can flush the contents of a file using the FlushFileBuffer() API on Window.)

The File System Interface component forces the write-through option for all opens of files that have a distribution frequency of distribute-on-update. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation**

and Configuration Guide for more information about the File System Interface component.

Each separate update to byte-stream files distributed to broadcast domains with an effective distribution frequency of distribute-on-update must be limited in size to a maximum of 4 KB. Updates larger than 4 KB to such files are rejected by the Data Distribution component. The application detects this condition as an error returned by the update API (for example, WriteFile() on Windows.

Reconciliation

Reconciliation is the process of making an image copy of a file identical to the prime copy. There are two forms of reconciliation:

Full reconciliation

Copies the prime copy of the file to the backup distributor or to a subordinate node.

Partial reconciliation

Achieves the same end result by applying a saved list of updates to the down-level image. This method is used for keyed files. See "Keyed-File Services" for more information about keyed files.

Byte stream files are reconciled in a similar manner, by copying to the down-level image only those portions of the prime copy that have been modified since the last time the two instances were known to be identical.

Partial reconciliation is used only for keyed files larger than 32 KB and byte stream files that have an effective distribution frequency of distribute on update. Even then, certain error conditions can cause full reconciliation to occur. Full reconciliation is used in all other circumstances. Partial reconciliation is not used for small keyed files, because full reconciliation of small files is more efficient than applying multiple updates.
Reconciliation occurs, if required, at an image node each time it establishes a connection with the primary distributor, including during IPLs. There are two exceptions to this rule:
Distribute-on-close files in the mirrored domain are not reconciled immediately.
A distribute-on-close file in a broadcast domain is not reconciled immediately if it has been modified by an application since the last open, close, or flush operation.

In both cases, the file is effectively reconciled the next time it is closed or flushed at the primary distributor.

Note: The Data Distribution component must open the prime copy of a file on the primary distributor to reconcile it to the acting backup distributor or a subordinate node. An open file cannot be:

- Deleted. A user could be performing an erase function at the command line or a program calling the DeleteFile() on Windows.
- Renamed. A user could be performing a rename function at the command line or a program calling the MoveFile() on Windows.
- The target of the CopyFile() on Windows.

Because reconciliation of a file can occur at any time (for example, as the result of power being turned on at a node), an attempt to delete or rename a distributed file, or a program's call to DosCopy() or CopyFile() could fail unexpectedly. The user or program that encounters such a situation should respond by retrying the operation at a later time.

The reconciliation subsystem requires an additional, free working area. This free working area must be available on a controlled drive of the subordinate workstation and backup distributor. The amount of free working area must be equal to the size of the largest file that will be reconciled after applications have been started on these workstations. Any file defined as distribute-on-close is normally distributed via full reconciliation. Of course, there must also be enough disk space for the distributed files, as well as for the free working area.

Data Integrity and Availability

Reconciliation recovers from single failures by making all instances of a file identical after the failure is corrected. However, the order in which files are reconciled to a node is not specified and is unpredictable. This fact has two ramifications:

1. While a node is being reconciled, files, including individual files and files relative to other files, pass through inconsistent states.
2. There are some combinations of primary-distributor and backup-distributor failures that could result in an acting primary distributor with an inconsistent set of files. These are double failures from which the Data Distribution component cannot automatically recover. For example:
    - The acting primary distributor fails.
    - The acting backup distributor is made the acting primary distributor.
    - The old primary distributor is repaired and begins to reconcile from the current acting primary distributor.
    - The old acting backup distributor (current primary distributor) fails.
    - The current acting backup distributor is made the acting primary distributor again, and now has inconsistent files.

Role changes can not only result in data loss (from distribute on close files), but can also lead to double failures that result in inconsistent files.

In the context of data integrity and availability, failures are:

- Abnormal termination of a node's operating system
- Abnormal termination of DDS on a node
- Hard-disk failures CPU or memory failures
- LAN failure
- Power-line disturbances (PLDs)

See the note under "Reconciliation" for information about the Data Distribution component's use of distributed files.

Distribute-on-Update Files: DDS prevents the loss of data from a distribute-on-update file due to a single failure. When an application receives a successful

return code from a file-update operation, the update has been performed on the prime copy of the file, and has been recorded in a way such that:

1. The update will not be lost even if the primary distributor fails. This fact implies that the update has been saved by the backup distributor
2. All nodes to which the file is distributed will eventually receive the update.

Distribute-on-Close Files: Distribute-on-close files do not provide the same degree of data integrity as distribute-on-update files. Distribute-on-close files are appropriate when performance is more important than ensuring that no updates are lost.

After a single failure, DDS prevents the loss of data applied to a distribute-on-close file up to the time of the last successful close of the file or up to the last flush operation. However, Data Distribution distributes distribute-on-close files asynchronously, so that a close operation or a flush operation will be completed before the distribution is completed.

Updates to distribute-on-close files are blocked while the file is being distributed. Such a distribution might be the result of a FlushFileBuffers() on Windows, or due to the reconciliation of a node. This restriction allows a consistent version of the file to be distributed.

Activating and Deactivating the Acting Primary Distributor

The primary task of the configured backup distributor is to assume control when the configured primary distributor becomes disabled or is deactivated. The first time the system is started, the configured primary distributor assumes the acting primary-distributor role. It remains the acting primary distributor until it becomes disabled or is deactivated.

Two options are available for activating the acting backup distributor as the acting primary distributor. You can issue a DDS command using the Node Control Utility to manually activate the acting backup distributor as the acting primary distributor. This option gives you control over the timing the activation and is useful in preparing for scheduled machine outages. DDS also provides an automatic switch-over option that results in the acting backup distributor automatically assuming the role of primary distributor. Each of these functions is described in the following sections.

User-Initiated Activation of the Primary Distributor

The default behavior of DDS results in the acting backup distributor **not** assuming the acting primary-distributor role automatically. In this case, two steps are required before the acting backup distributor can be activated as the primary distributor.

1. Deactivating the acting primary distributor
2. Activating the configured backup distributor as the acting primary distributor

You must deactivate the current, acting primary distributor before you can activate a new primary distributor, unless the current, acting primary distributor is not running or is not communicating with other workstations. Refer to **IBM Distributed Data Services/Controller Services Feature f***or Windows User's Guide* for more information about using the Node Control Utility to perform these deactivation and activation steps.

Note: If the configured primary distributor will be disabled for only a short period of time, you may not need to activate the configured backup distributor as the acting primary distributor. Such might be the case if the applications running on other nodes use image copies of input files and asynchronously write output data to the acting primary distributor.

When the configured primary distributor resumes normal operation (it is powered ON and connected to the LAN) it does not automatically resume its role as the acting primary distributor. If the configured backup distributor was activated by the operator as the acting primary distributor, the configured primary distributor assumes the acting backup-distributor role.

To return the system to its normal state, you must first deactivate the acting primary distributor (the configured backup distributor) and then reactivate the configured primary distributor as the acting primary distributor.

More complex scenarios are possible. For example, the acting primary distributor could fail, the operator could activate the acting backup distributor as the acting primary distributor, and then the new, acting primary distributor could fail. At this point, there are several possibilities:

- The new acting primary distributor could resume normal operation. In this case, no data would be lost.
- The original acting primary distributor could resume normal operation before the new acting primary distributor. In this case the original acting primary distributor assumes its role again as the acting primary distributor. Data will be lost that was collected (updated) on the new acting primary distributor after the original acting primary distributor failed.
- Both the original acting primary distributor and the new acting primary distributor could resume normal operation simultaneously. In this case, the new acting primary distributor detects that it assumed the role of acting primary distributor more recently than the original acting primary distributor, and again assumed the acting primary distributor role. No data would be lost in this case.

The following APIs control the activation and deactivation of the primary distributor:

- FdsActivateAsPrimary()
- FdsDeactivatePrimary()

The deactivation of the acting primary distributor will fail if the backup distributor is not online and fully reconciled. Use the FdsQueryBackupState() API to determine if the backup distributor is ready to be activated as the primary distributor.

Applications that run on the acting primary distributor should be stopped before deactivating the acting primary distributor.

The FdsSetupDistMonitor() API can be used by an application to detect the activation and deactivation of the primary distributor.

Automatic Switch-Over

The DDS Automatic Switch-Over feature provides the capability for automatically activating the acting backup distributor as the acting primary distributor. The activation occurs when the workstation that was performing the acting backup distributor role has lost communication with the acting primary distributor. Lost communications can result from a hardware failure, shutdown of the operating system, or any other event that results in the termination of DDS on the acting primary distributor.

To enable automatic switch-over, you must install the DDS Automatic Switch-Over feature and change your DDS configuration to enable it. Refer to**IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for information about:

- configuring automatic switch-over using the AutoSwitchOver, AutoSwitchOverDelay
- AutoSwitchOverForce keywords, for a list of conditions that must be met before automatic switch-over will occur
- the recommended network hardware installation and configuration to use with automatic switch-over.

When an automatic switch-over activation occurs, the result is the same as if a manual (user-initiated) activation were performed using the Node Control Utility. In particular, the FDSAP batch file will be executed. Configuring DDS for automatic switch-over does not prevent you from initiating a manual activation or deactivation. Automatic switch-over has no effect on the manual activation and deactivation functions. However, when the acting primary is manually deactivated, that same node will not automatically activate until it has successfully reconciled with an acting primary distributor.

The typical, automatic-switch-over scenario occurs when the machine that is performing the acting-primary role fails or is powered off. The following events occur to complete the automatic switch-over:

1. The acting primary distributor fails or is powered off.
2. The acting backup distributor:
    - detects the loss of communication with the acting primary distributor
    - continually tries to reestablish communication with the acting primary distributor for the amount of time specified by the AutoSwitchOverDelay configuration keyword
    - fails to reestablish communication and automatically activates as the acting primary distributor.
3. The original primary distributor is restarted and detects the new acting primary distributor. It establishes communication with the new primary distributor, reconciles its files with those on the new acting primary distributor, and becomes the acting backup distributor.

When DDS is configured for automatic switch-over, automatic deactivation is also enabled. Automatic deactivation entails the deactivation of the acting primary distributor whenever both of the following conditions are met:

1. The acting primary distributor can accurately determine when it is disconnected from the LAN, or is unable to communicate with the LAN.

Notification of disconnection from the LAN is not communicated by all network adapters to applications such as DDS. For LAN configuration requirements when using the DDS Automatic Switch-Over feature, refer to the Automatic Primary Distributor Switch-Over section in the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide**.

2. The acting primary distributor has communicated with an acting backup distributor after DDS became the acting primary distributor.

Under these conditions, DDS automatically deactivates the acting primary distributor and executes the FDSDP batch file to prevent two machines from both performing the acting primary distributor role.

Automatic switch-over is always initiated after the acting backup distributor detects a loss of communication with the acting primary distributor. The configured role of the machine is not considered. This independence from the configured roles allows automatic switch-over to occur repeatedly in either direction to insure that there is always an active primary distributor.

Consider a system with a configured primary distributor (node CPD) and a configured backup distributor (node CBD). The following scenario demonstrates the flexibility of the DDS automatic switch-over capability.

1. DDS is started on CPD and CBD. CPD assumes the acting primary-distributor role and CBD the acting backup-distributor role.
2. CPD fails, resulting in CBD's assuming the acting primary-distributor role. Some time later, CPD resumes normal operation and becomes the acting backup distributor. At this point, you could continue running with the roles reversed or node CBD could be manually deactivated and CPD could be manually activated as the primary distributor, so that the acting and configured distributor roles again match.
3. Assume the system was allowed to continue with CBD as the acting primary distributor and CPD as the acting backup distributor. If CBD fails or is manually deactivated at this point, the acting primary role would switch back to CPD. When CBD resumed normal operation, it would automatically assume the role of acting backup distributor.
4. The roles could continue to be switched between the two machines indefinitely, as the acting primary distributor fails or is taken offline.

Automatic switch-over should be used when it is critical that your DDS system always has a primary distributor available. However, the decision to use the automatic switch-over function should be made carefully; in some cases automatically activating the acting backup distributor as the acting primary distributor can lead to loss of data. For example, when the acting backup distributor is activated as the primary, all updates since the last flush made to open files with a distribution frequency of distribute-on-close are lost. When manually activating, you may be able to assure that all distributed files have been closed on the acting primary distributor before deactivating and then activating the backup distributor as the primary. Automatic activation does not give you that opportunity.

Performance

The number of distributed files and the block size of keyed files can affect system performance.

Number of Distributed Files

DDS performance is affected by the number of distributed files in the system. As the number of distributed files increases, the response time of file operations can become slower.

There is no specific upper limit on the number of distributed files that will guarantee good performance. Performance is greatly affected by many factors in the system. However, systems with 10,000 or fewer distributed files are less likely to experience performance degradation related to the total number of distributed files.

When totaling the number of distributed files, count files in distributed directories as well as those which are explicitly distributed. Distributed files with long file names should be counted as two files when considering performance implications.

Keyed Files

Update and distribution throughput for distribute-on-update keyed files is reduced as the record size or key size increases. A distribute-on-update keyed file can be updated while a node is performing a full reconciliation of the same file. However, the full reconciliation of the file reduces the throughput of the updates. The larger the block size of the keyed file, the greater the reduction in throughput.

A distribute-on-update keyed file can also be updated while a node is performing a partial reconciliation of the same file. However, the partial reconciliation of the file reduces the throughput of the updates.

Restrictions

Data Distribution has the following restrictions:

- The following restrictions apply to rename operations, such as a user performing a rename function at the command line or a program calling an API to rename the file:
  - A distributed directory cannot be renamed.
  - A directory that contains a distributed file, either directly or in a descendant subdirectory, cannot be renamed.

- If a directory has been set to be distributed with a scope qualifier of **FDS_SCOPE_TREE**, you cannot move another directory into the distributed directory. For example, if directory x has been set to be distributed with a scope of **FDS_SCOPE_TREE**, you cannot make directory y (using the MOVE command) a subdirectory of x.
- The following restrictions apply to using the distribution directory:
  - Files that reside on the boot partition cannot be distributed.
  - You cannot distribute the root directory of a drive.
  - No file or directory in the DDS installation or WORK directories can be distributed.
  - All possible combinations of subdirectory indicator, domain type, distribution frequency, and file type (byte stream or keyed) are supported, with the following exception:

    Byte stream files (and subdirectories with byte-stream files) that have

distribution frequencies of distribute-on-update should not be distributed to a broadcast domain if updates to these files are more than 4 KB each. Such directory entries are allowed, but updates to these files of more than 4 KB each are rejected by Data Distribution.

- The key length of distributed keyed files is limited to a maximum of 255 bytes.

FdsActivateAsPrimary()

Purpose

Activate the local node as the acting primary distributor.

Syntax

```
#include <fds/dist.h>

long FdsActivateAsPrimary( int ForceFlag );
```

Parameters

ForceFlag — input
A flag that indicates whether to force activation. Valid values are:

FDS_FORCE
Force activation

FDS_NO_FORCE
Do not force activation

Remarks

FdsActivateAsPrimary() activates the local node as the acting primary distributor. Two nodes are eligible to be the acting primary distributor: the configured primary distributor and the configured backup distributor. Whenever one node is activated as the acting primary distributor, the other node assumes the role of the acting backup distributor. If the local node is not fully reconciled and the **ForceFlag** does not specify force activation, the API fails with the -380 FDSERR_NOT_RECONCILED error.

This API does not start the FDSAP.BAT batch file (Windows). The FDSAP command or batch file is started by the Node Control Utility, which uses this API to activate the configured backup distributor or the configured primary distributor as the acting primary distributor. See the **IBM Distributed Data** *Services/Controller Services Feature for Windows User's Guide* for more information about the Node Control Utility.

This API can succeed only when called on either the configured backup distributor or configured primary distributor, and neither is the acting primary distributor.

Error Conditions

FdsActivateAsPrimary() returns the following values:

-60 FDSERR_CONFIG
-170 FDSERR_EXISTS
-210 FDSERR_FLAG

-360 FDSERR_NODE_TYPE
-380 FDSERR_NOT_RECONCILED
-560 FDSERR_SEQUENCE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>
long rc; // Return from API Call

// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------
   // The following call to FdsActivateAsPrimary API must be issued
   // from either the Configured Primary or the Configured Backup
   // Distributor
   //-------------------------------------------------------------------------------------------------------------
   rc = FdsActivateAsPrimary( FDS_FORCE );
   printf( "FdsActivateAsPrimary completed with return
            code = (%d).\n", rc );
} // end if
else
{
  // else process errors
}
```

FdsAddDomainNode()

Purpose

Add a node to a broadcast domain.

Syntax

```
#include <fds/dist.h>

long FdsAddDomainNode(const FDS_DOMAIN_NAME DomainName, const
                                FDS_NODE_NAME NodeID );
```

Parameters

DomainName — input
Indicates the broadcast domain name to which you want to add the node.

NodeID — input
Indicates the node ID of the node you want to add to the broadcast domain.

Remarks

FdsAddDomainNode() adds a node to a broadcast domain. All files distributed to the domain are loaded onto the node.

This API can be called from an application running on the acting primary distributor only.

Error Conditions

FdsAddDomainNode() returns the following values:
-60 FDSERR_CONFIG
-90 FDSERR_DISK
-120 FDSERR_DOMAIN_NAME
-130 FDSERR_DOMAIN_NOT_FOUND
-170 FDSERR_EXISTS
-340 FDSERR_NODE_NAME
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long          rc;                                  // Return from API Call

// Modify Domain Name
FDS_DOMAIN_NAME DomainName = "DOMAINxx";
FDS_NODE_NAME NodeID = "Node_A";     // Node to Add to Domain
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();

// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------------
   // Call FdsAddDomainNode to add Node ID "Node_A" to DOMAINxx
   //-------------------------------------------------------------------------------------------------------------------
   rc = FdsAddDomainNode( DomainName, NodeID );
   printf("FdsAddDomainNode completed with return code = (%d) \n",
               rc);
} // end if
else
{
// else process errors
}
```

FdsCreateBcastDomain()

Purpose

Create a broadcast domain.

Syntax

```
#include <fds/dist.h>

long FdsCreateBcastDomain(const FDS_DOMAIN_NAME DomainName, unsigned int BufferSize,
                          FDS_NODE_NAME *NodeList );
```

Parameters

DomainName — input
Indicates the name of the broadcast domain to be created. It must not be equal to the node ID of any node in the system.

BufferSize — input Indicates the size of the **NodeList** buffer in bytes.

NodeList — input
A pointer to an array of FDS_NODE_NAME elements. Each FDS_NODE_NAME contains a node ID.

Remarks

FdsCreateBcastDomain() can be called from an application running on the acting primary distributor only. It creates a broadcast domain with the nodes indicated in **NodeList.** If no **NodeList** is passed (**NodeList** is zero), an empty broadcast domain is created.

This API should not be used to create a broadcast domain that begins with the prefix FDS. The characters FDS are reserved. This function cannot be used to create a broadcast domain named MIRRORED.

Only one broadcast domain is supported.

Error Conditions

FdsCreateBcastDomain() returns the following values:
-20 FDSERR_ADDRESS
-60 FDSERR_CONFIG
-120 FDSERR_DOMAIN_NAME
-170 FDSERR_EXISTS
-340 FDSERR_NODE_NAME
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <string.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long          rc;                                    // Return from API Call
FDS_DOMAIN_NAME DomainName = "DOMAINxx"; // New Domain Name
FDS_NODE_NAME     NameList[5];                       // List of Nodes in Domain
unsigned int            BufferSize = sizeof(NameList);              // Size of NameList
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();

// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------
   // Set up list of Nodes in Domain
```

```
   //-------------------------------------------------------------
   strcpy( NameList[0], "Node_1" );
   strcpy( NameList[1], "Node_2" );
   strcpy( NameList[2], "Node_3" );
   strcpy( NameList[3], "Node_4" );
   strcpy( NameList[4], "Node_5" );
   //-------------------------------------------------------------------------------------------------
   // Call FdsCreateBcastDomain API to create Domain DOMAINxx
   // - create DOMAINxx with 5 Node IDs
   //-------------------------------------------------------------------------------------------------
   rc = FdsCreateBcastDomain( DomainName, BufferSize, NameList );
   printf("FdsCreateBcastDomain completed with
            return code= (%d) \n", rc);
} // end if
else
{
   // else process errors
}
```

FdsCreateSyncID()

Purpose

Create a synchronization ID associated with a particular, distributed-file update.

Syntax

```
#include <fds/file.h>

long FdsCreateSyncID( long FileHandle, FDS_SYNC_ID *SyncID );
```

Parameters

FileHandle — input

Indicates the file handle returned by DDS when the sequential, keyed, or binary file was opened.

SyncID — output

Pointer to the location where the synchronization ID is stored.

Remarks

FdsCreateSyncID() can be called from an application running on any node. It returns a synchronization identifier. The synchronization ID allows an application to determine when a certain set of file updates have been distributed to one or more nodes within a distribution domain. For files with a distribution frequency of distribute-on-update (DOU), this API should be called by the application immediately after the file update for which the synchronization ID is to be recorded.

For keyed files with a distribution frequency of distribute-on-close (DOC), this API should be called by the application immediately after the file has been flushed using FdsCloseKeyedFile().For binary files with a distribution frequency of distribute-on-close (DOC), this API should be called by the application immediately

after the file has been flushed using FdsFlushBinFile().

Note: This API should not be used with sequential DOC files.

The synchronization ID identifies the current state of the file in the distribution domain in terms of the last update applied at the acting primary distributor. The synchronization ID can be used by an application at any node within the distribution domain to wait until the file at that node is brought to the state identified by the synchronization ID obtained at the acting primary distributor.

Error Conditions

FdsCreateSyncID() returns the following values:

-60 FDSERR_CONFIG
-220 FDSERR_HANDLE
-350 FDSERR_NODE_NOT_FOUND
-375 FDSERR_NOT_DISTRIBUTED
-530 FDSERR_ROLE_CHANGE
-560 FDSERR_SEQUENCE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/dist.h>
#include <fds/errno.h>

long          rc;                          // Return from API Call
FDS_SYNC_ID   SyncID;                      // SyncID
long          FileHandle;                  // FileHandle
unsigned int  KeySize;   // Size of key
unsigned int  RecordSize;                  // Size of records to write

// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();

// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------
   // The following call to FdsCreateSyncID API can be invoked from an
   // application running on any node.
   // For this example, assume this program is running on the
   // acting primary and you want to notify a remote node regarding
   // updates to a keyed file.
   //-------------------------------------------------------------------------------------------------------------
   //-------------------------------------------------------------------------------------------------------------
   // Open the distribute on update file, "d:\dou\itemrec.dat", so
   // a SyncID can be created for this file.
   //-------------------------------------------------------------------------------------------------------------
   rc = FdsOpenKeyedFile( &FileHandle,
                          "d:\\dou\\itemrec.dat",
                          &KeySize,
                          &RecordSize,
```

```
                                        FDS_FILE_ACCESS_READ_WRITE );

   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------------------------------------------------------
      // Call FdsCreateSyncID API for file "d:\xxx\itemrec.dat"
      //-------------------------------------------------------------------------------------------------------------
      rc = FdsCreateSyncID( FileHandle, &SyncID );
      printf( "FdsCreateSyncID completed with return code = (%d).\n",
              rc );
      //-------------------------------------------------------------------------------------------------------------
      // Open a queue on the remote node and write the SyncID
      // returned from FdsCreateSyncID into it.
      // (See the FdsSetupSyncIDNotify API for more information as to
      // what occurs on the remote node.)
      //-------------------------------------------------------------------------------------------------------------
   } // end if
} // end if
else
{
// else process errors
}
```

FdsDeactivatePrimary()

Purpose

Deactivate the primary distributor role at the local node.

Syntax

```
#include <fds/dist.h>

long FdsDeactivatePrimary( ) ;
```

Remarks

FdsDeactivatePrimary() deactivates the acting primary distributor role at the local node. This API will succeed only when called on the acting primary distributor. It will fail if the acting backup distributor is not online and fully reconciled.

This API may take a long time to complete. Running this API when there is a lull in system activity improves the time required.

This API does not start the FDSDP.BAT batch file.. The FDSDP command and batch files are started by the Node Control Utility, which uses this API to deactivate the acting primary distributor. See the **IBM Distributed Data** *Services/Controller Services Feature for Windows User's Guide* for more information about the Node Control Utility.

Error Conditions

FdsDeactivatePrimary() returns the following values:

-60 FDSERR_CONFIG

-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE
-380 FDSERR_NOT_RECONCILED
-560 FDSERR_SEQUENCE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long rc; // Return from API Call
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------
   // The following call to FdsDeactivatePrimary API must be issued
   // from the Acting Primary Distributor
   //-------------------------------------------------------------------------------------------------------
   rc = FdsDeactivatePrimary ();
   printf( "FdsDeactivatePrimary completed with return code =
               (%d).       \n", rc );
} // end if
else
{
  // else process errors
}
```

FdsDeleteBcastDomain()

Purpose

Delete a broadcast domain.

Syntax

```
#include <fds/dist.h>

long FdsDeleteBcastDomain( const FDS_DOMAIN_NAME DomainName );
```

Parameters

DomainName — input
Indicates the name of the broadcast domain to be deleted.

Remarks

FdsDeleteBcastDomain() can be called from an application running only on the acting primary distributor. It deletes the broadcast domain indicated by **DomainName**. This deletion causes all files and subdirectories distributed to the broadcast domain to be made local (deleted from the distribution directory). All

files distributed to this domain must be closed when this function is called.

This API should not be used to delete a broadcast domain that begins with the prefix FDS. The characters FDS are reserved. This function cannot be used to delete a broadcast domain named MIRRORED.

Error Conditions

FdsDeleteBcastDomain() returns the following values:
-10 FDSERR_ACCESS
-60 FDSERR_CONFIG
-90 FDSERR_DISK
-120 FDSERR_DOMAIN_NAME
-130 FDSERR_DOMAIN_NOT_FOUND
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long        rc;                    // Return from API Call

// Choose Domain Name to delete
FDS_DOMAIN_NAME DomainName = "DOMAINxx";
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------
   // Call FdsDeleteBcastDomain API to delete DOMAINxx
   //-------------------------------------------------------------
   rc = FdsDeleteBcastDomain( DomainName );
   printf("FdsDeleteBcastDomain completed with return code =
     (%d) \n", rc);
} // end if
else
{
   // else process errors
}
```

FdsDeleteDomainNode()

Purpose

Remove a node from a broadcast domain.

Syntax

```
#include <fds/dist.h>

long FdsDeleteDomainNode(const FDS_DOMAIN_NAME DomainName, const FDS_NODE_NAME
                                        NodeID );
```

Parameters

DomainName — input

Indicates the broadcast domain name from which you want to remove the node.

NodeID — input

Indicates the node ID of the node to be removed.

Remarks

FdsDeleteDomainNode() can be called from an application running on the acting primary distributor only. It deletes a node from a broadcast domain. All files distributed to the domain are deleted from the node.

Error Conditions

FdsDeleteDomainNode() returns the following values:

-60 FDSERR_CONFIG
-90 FDSERR_DISK
-120 FDSERR_DOMAIN_NAME
-130 FDSERR_DOMAIN_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>


long rc; // Return from API Call FDS_DOMAIN_NAME DomainName = "DOMAINxx"; //
Domain Name to modify FDS_NODE_NAME NodeID = "Node_A"; // Node to Add to
Domain

// Initialize DDS. Could use FdsInit2() instead of FdsInit(). rc = FdsInit();

// If initialization was successful
if ( rc == FDS_SUCCESS )
{

    //-------------------------------------------------------------------------------------------------------------
    // Call FdsDeleteDomainNode API to delete Node ID "Node_A" from
    // DOMAINxx
    //-------------------------------------------------------------------------------------------------------------
    rc = FdsDeleteDomainNode( DomainName, NodeID );
    printf("FdsDeleteDomainNode completed with return
            code = (%d)          \n", rc);
  } // end if else
  {
    // else process errors
  }
```

FdsGetDomainList()

Purpose

Obtain the list of all the domains in the system.

Syntax

```
#include <fds/dist.h>

long FdsGetDomainList( unsigned int *BufferSize, FDS_DOMAIN_NAME *DomainList
                                    );
```

Parameters

BufferSize — input/output

Input A pointer to the location where the size of the **DomainList** buffer (in bytes) is stored.

Output

When this API completes successfully, the data in the location pointed to by **BufferSize** is replaced with the size of the returned data in bytes or the required buffer size in bytes if the input buffer is too small. In the latter case, the list of domains is not returned.

DomainList —Output

Pointer to the location where the array of FDS_DOMAIN_NAME elements is stored. Each FDS_DOMAIN_NAME structure contains a domain name.

Remarks

FdsGetDomainList() can be called from an application running on the acting primary distributor only. It returns a list of all the domains in the system and their update status.

Error Conditions

FdsGetDomainList() returns the following values:

-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-60 FDSERR_CONFIG
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long          rc;                                    // Return from API Call
FDS_DOMAIN_NAME          DomainList[2];       // List of Domains
unsigned int          BufferSize = sizeof(DomainList);   // Size of DomainList

// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
```

```
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------------
   // Call FdsGetDomainList API to get a list of all the Domains
   Chapter 6. Data Distribution 143
   //---------------------------------------------------------------
   rc = FdsGetDomainList( &BufferSize, DomainList );
   printf("FdsGetDomainList completed with return code = (%d)\n"
            " ----> Domain #1 = (%s) <---- \n"
            " ----> Domain #2 = (%s) <---- \n",
            rc,
            DomainList[0], // Domain #1
            DomainList[1] ); // Domain #2
} // end if
else
{
// else process errors
}
```

FdsGetDomainNodes()

Purpose

Obtain the list of nodes that are in a broadcast domain.

Syntax

```
#include <fds/dist.h>

long FdsGetDomainNodes(const FDS_DOMAIN_NAME DomainName, unsigned int *BufferSize,
                                    FDS_NODE_STATE *NodeList );
```

Parameters

DomainName — input

Indicates the broadcast domain name that contains the nodes.

BufferSize — input/output

Input The size of the **NodeList** buffer in bytes.

Output

The size of the returned data in bytes or the required buffer size in bytes if the input buffer is too small. In the latter case, the list of node IDs is not returned.

NodeList — output

Pointer to the location where the array of FDS_NODE_STATE elements is stored. Each FDS_NODE_STATE structure contains a node ID.

Remarks

FdsGetDomainNodes() can be called from an application running on the acting

primary distributor only. It returns the list of nodes that belong to a broadcast domain.

Error Conditions

FdsGetDomainNodes() returns the following values:
-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-60 FDSERR_CONFIG
-120 FDSERR_DOMAIN_NAME
-130 FDSERR_DOMAIN_NOT_FOUND
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long          rc;                       // Return from API Call
// Domain Name to query
FDS_DOMAIN_NAME             DomainName = "DOMAINxx";
FDS_NODE_STATE    NodeList[100];                  // Node List for Domain
unsigned int            BufferSize = sizeof(NodeList);              // Size of NodeList
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------------
   // Call FdsGetDomainNodes API to get a list of all the Node IDs in
   // DOMAINxx
   //---------------------------------------------------------------
   rc = FdsGetDomainNodes( DomainName, &BufferSize, NodeList );
   printf("FdsDomainName completed with return code = (%d) \n"
            " DomainName = (%s) \n"
            " Node List: \n"
            " NodeList[0] -- NodeID = (%s), state = (%d) \n"
            " NodeList[1] -- NodeID = (%s), state = (%d) \n"
            " NodeList[2] -- NodeID = (%s), state = (%d) \n"
            " NodeList[3] -- NodeID = (%s), state = (%d) \n"
            " NodeList[4] -- NodeID = (%s), state = (%d) \n",
            rc,
            DomainName,
            NodeList[0].Name, NodeList[0].State, // Node_1
            NodeList[1].Name, NodeList[1].State, // Node_2
            NodeList[2].Name, NodeList[2].State, // Node_3
            NodeList[3].Name, NodeList[3].State, // Node_4
            NodeList[4].Name, NodeList[4].State ); // Node_5
} // end if
else
{
  // else process errors
}
```

FdsQueryBackupState()

Purpose

Query the state of the backup distributor.

Syntax

```
#include <fds/dist.h>

long FdsQueryBackupState( int *State );
```

Parameters

State — output

Pointer to the location of the state of the backup distributor. Possible values are:

FDS_ACTIVE

Fully reconciled

FDS_JOINING

Reconciling

FDS_INACTIVE

Not online

Remarks

FdsQueryBackupState() can be called from an application that is running on the primary distributor only.

Error Conditions

FdsQueryBackupState() returns the following values:
-60 FDSERR_CONFIG
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long        rc;                 // Return from API Call
int         State = 0;          // Initialize State
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------------
   // The following call to FdsQueryBackupState API must be issued from
   // the Acting Primary Distributor
   //-------------------------------------------------------------------------------------------------------------------
   rc = FdsQueryBackupState( &State );
   printf( "FdsQueryBackupState completed with return
     code = (%d).\n", rc );
   if ( rc == FDS_SUCCESS )
   {
      switch (State)
```

```
      {
         case FDS_ACTIVE:
         printf("... Backup State = FDS_ACTIVE ... \n");
         break;
         case FDS_JOINING:
         printf("... Backup State = FDS_JOINING ... \n");
         break;
         case FDS_INACTIVE:
         printf("... Backup State = FDS_INACTIVE ... \n");
         break;
         default:
         printf("No state was returned from FdsQueryBackupState \n");
         break;
      } // end switch
   } // end if
} // end if
else
{
// else process errors
}
```

FdsQueryDistribution()

Purpose

Query a distribution directory entry for a file or subdirectory.

Syntax

```
#include <fds/dist.h>

long FdsQueryDistribution(const char *OsPath,                    int *DirIndicator,
                          int *DomainType,                  FDS_DOMAIN_NAME
                          DomainName,           int *DistFrequency,
                          int *Scope );
```

Parameters

OsPath — input Indicates the file or subdirectory path name. **OSPath** can be a logical name, but it must resolve to a file or subdirectory on the local node, or to a retail path specification that includes either the role name or the node ID of the acting primary distributor. See “File Names and Queue Names” for more information.

DirIndicator — output

Indicates whether the entry refers to a file or a subdirectory. Possible values are:

FDS_FILE

Indicates that the entry refers to a file

FDS_DIRECTORY

Indicates that the entry refers to a subdirectory

DomainType — output

Indicates the domain type. Possible values are:

FDS_MIRRORED

Indicates that the domain type is a mirrored domain

FDS_BROADCAST

Indicates that the domain type is a broadcast domain

DomainName — output

The broadcast domain name is returned if the entry is for a broadcast domain. Otherwise, the value is undefined.

DistFrequency — output

Indicates the distribution frequency. Possible values are:

FDS_DOU

Indicates distribute on update

FDS_DOC

Indicates distribute on close

Scope — output

Indicates the scope qualifier. Possible values are:

FDS_SCOPE_FILE

Only the files in the directory are distributed

FDS_SCOPE_UNDEFINED_FOR_FILE

FDS_FILE was specified for **DirIndicator**.

FDS_SCOPE_TREE

All files and subdirectories are distributed. See “Restrictions” for information about specifying a scope for FDS_SCOPE_TREE.

Remarks

FdsQueryDistribution() queries a distribution directory entry for a file or subdirectory specified by **OsPath**.

Error Conditions

FdsQueryDistribution() returns the following values:

-60 FDSERR_CONFIG
-70 FDSERR_CORRUPT
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-410 FDSERR_OVERFLOW
-500 FDSERR_REMOTE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/dist.h>
#include <fds/errno.h>

long        rc;                                 // Return from API Call
int         DirectoryIndicator = 0;         // Directory Indicator
FDS_DOMAIN_NAME          DomainName;       // Domain Name
int         DomainType = 0;                 // Domain Type
int         DistributionFrequency = 0;    // Distribution Frequency
int         Scope = 0;                          // Scope
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
```

```
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------------------
   // The following calls to FdsQueryDistribution API must be issued from
   // the Acting Primary Distributor
   //-------------------------------------------------------------------------------------------------------------------------
   //-----------------------------------------------------------------------------
   // Query the Distribution of file "d:\dou\itemrec.dat"
   //-----------------------------------------------------------------------------
   rc = FdsQueryDistribution( "d:\\dou\\itemrec.dat",
                     &DirectoryIndicator,
     &DomainType,
       DomainName,
                     &DistributionFrequency,
                &Scope );
               printf("FdsQueryDistribution completed with return code = (%d).\n"
                 " Directory Indicator = (%d)\n"
                 " Domain Type = (%d)\n"
                 " Domain Name = (%s)\n"
                 " Distribution Frequency = (%d)\n"
                 " Scope = (%d)\n",
                 rc,
                 DirectoryIndicator,
                 DomainType,
                 DomainName,
                 DistributionFrequency,
                 Scope );
   //-----------------------------------------------------------------------
   // Query the Distribution of directory "d:\doc"
   //-----------------------------------------------------------------------
   rc = FdsQueryDistribution( "d:\\doc\\",
                                          &DirectoryIndicator,
                                          &DomainType,
                                          DomainName,
                                          &DistributionFrequency,
                                          &Scope );
   printf("FdsQueryDistribution completed with
               return code = (%d).\n"
               " Directory Indicator = (%d)\n"
               " Domain Type = (%d)\n"
               " Domain Name = (%s)\n"
               " Distribution Frequency = (%d)\n"
               " Scope = (%d)\n",
               rc,
               DirectoryIndicator,
               DomainType,
               DomainName,
               DistributionFrequency,
               Scope );
} // end if
else
{
  // else process errors
}
```

FdsSetDistribution()

Purpose

Modify the distribution characteristics of files and subdirectories.

Syntax

```
#include <fds/dist.h>

long FdsSetDistribution(    const char *OsPath, int DirIndicator,    int DomainType,
                            const FDS_DOMAIN_NAME DomainName, int DistFrequency,
                            int Scope );
```

Parameters

OsPath — input Indicates the file or subdirectory path name. **OSPath** can be a logical name, but it must resolve to a file or subdirectory on the local node, or to a retail path specification that includes either the role name or the node ID of the acting primary distributor. The resolved name, not the logical name, is stored in the distribution directory (distribution characteristics are associated with resolved-to names, not logical names). See “File Names and Queue Names” for more information.

Note: DDS supports the distribution of files up to a maximum size of 4 GB.

DirIndicator — input

Indicates whether the entry refers to a file or subdirectory. Valid values are:

FDS_FILE

Indicates that the entry refers to a file

FDS_DIRECTORY

Indicates that the entry refers to a directory

This characteristic cannot be changed for an existing distribution directory entry.

DomainType — input

Indicates the domain type. Valid values are:

FDS_MIRRORED

Indicates a mirrored domain.

FDS_BROADCAST

Indicates a broadcast domain. The file or subdirectory is distributed to the acting backup distributor and all nodes in the specified broadcast domain.

FDS_LOCAL

Indicates a local domain. The file or subdirectory is removed from the distribution directory and all image copies are deleted. Paths (the parent directory of a file or subdirectory) are not deleted at image nodes. An attempt to change an individual file to **FDS_LOCAL** is rejected if the file exists as part of a distributed subdirectory.

For an existing entry in the distribution directory, this characteristic can

only be changed to **FDS_LOCAL**, which deletes the entry from the distribution directory.

DomainName — input

Indicates the broadcast domain name if the entry is for a broadcast domain. This is the case whether the caller wants to add a distributed object, modify the characteristics of an existing distributed object, or make a distributed object non-distributed by setting **DomainType** to **FDS_LOCAL**. This parameter is ignored if **DomainType** is **FDS_MIRRORED**.

This characteristic cannot be changed for an existing distribution directory entry.

DistFrequency — input

Indicates the distribution frequency. Valid values are:

FDS_DOU
Indicates distribute on update

FDS_DOC
Indicates distribute on close

This characteristic can be changed for an existing distribution directory entry.

Scope — input

Indicates the scope qualifier. Valid values are:

FDS_SCOPE_FILE
Only the files in the directory are distributed.

FDS_SCOPE_TREE
All files and subdirectories are distributed. See “Restrictions” for information about specifying a scope of **FDS_SCOPE_TREE**.

FDS_SCOPE_UNDEFINED_FOR_FILE
FDS_FILE was specified for **DirIndicator**.

This characteristic cannot be changed for an existing distribution directory entry.

Remarks

FdsSetDistribution() updates a distribution directory entry for a file or subdirectory specified by **OsPath** and distributes the file or subdirectory.

If **OsPath** specifies a file, the file must not be open. If **OsPath** specifies a directory, the directory must not contain any open files, nor can any descendant subdirectory contain any open files, even if the scope qualifier is **FDS_SCOPE_FILE**.

Note: No file or directory in the directories where DDS is installed (or the directory pointed to by the **WorkDirectory** configuration keyword) can be distributed.

The file or subdirectory identified by **OSPath** must exist when this API is called.

One exception is that a distribution directory entry can be removed (the domain type set to local), even if the corresponding file or subdirectory does not exist on the acting primary distributor. Such a situation should not normally occur.

The distribution directory entry is deleted by data distribution when the corresponding file or subdirectory is deleted.

If there is no previous entry in the distribution directory for **OsPath**, an entry is created. Otherwise, the only modifications that can be made are changing the domain type to **FDS_LOCAL** (remove the entry from the distribution directory) or changing the distribution frequency.

There can be no more than one entry for a particular file or subdirectory in the distribution directory. That is, a file or subdirectory can be distributed to no more than one distribution domain. An entry for a file or subdirectory cannot be added to the distribution directory if the file or subdirectory is contained in a directory that is already distributed. An entry for a directory cannot be added to the distribution directory if it contains a file that is already distributed.

In all valid cases where **FDS_LOCAL** is not specified for **DomainType**, the file is distributed by this API to all nodes in the domain.

To force distribution for a particular file or subdirectory, query its directory entry using FdsQueryDistribution() and pass the results to FdsSetDistribution(). If FdsSetDistribution() is called for an existing entry and no attributes are changed, the effect is the same as forcing the distribution of that file or subdirectory. The file or subdirectory will be distributed to all nodes, even if the file or subdirectory is already current on those nodes.

The root directory of a drive cannot be added to the distribution directory.

Error Conditions

FdsSetDistribution returns the following values:
-10 FDSERR_ACCESS
-60 FDSERR_CONFIG
-80 FDSERR_DIR_INDICATOR
-90 FDSERR_DISK
-110 FDSERR_DIST_FREQ
-120 FDSERR_DOMAIN_NAME
-130 FDSERR_DOMAIN_NOT_FOUND
-140 FDSERR_DOMAIN_TYPE
-170 FDSERR_EXISTS
-190 FDSERR_FILE_NAME
-200 FDSERR_FILE_NOT_FOUND
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-360 FDSERR_NODE_TYPE
-375 FDSERR_NOT_DISTRIBUTED
-410 FDSERR_OVERFLOW
-500 FDSERR_REMOTE
-555 FDSERR_SCOPE
-560 FDSERR_SEQUENCE

Examples

```
#include <stdio.h>
```

```
#include <fds/fds.h>
#include <fds/file.h>
#include <fds/dist.h>
#include <fds/errno.h>

long        rc;                                 // Return from API Call
long        FileHandle = 0;       // Keyed File Handle
// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------------
   // The following calls to FdsSetDistribution API must be issued from
   // the Acting Primary Distributor
   //-------------------------------------------------------------------------------------------------------------------
   //-----------------------------------------------------------------
   // Create a keyed file to be distributed
   //-----------------------------------------------------------------
   rc = FdsCreateKeyedFile( &FileHandle,
                                    "d:\\dou\\itemrec.dat",        // Keyed file to create
                                    10,                            // Key Size
                                    65,                            // Record Size
                                    512,                           // Block Size
                                    10,                            // Number of Blocks
                                    0,                             // Randomizing Divisor
                                    2,                             // Chaining Threshold
                                    FDS_FILE_EXIST_REPLACE );   // Create Flag
   if ( rc != FDS_SUCCESS )
   {
      printf("FdsCreateKeyedFile failed with return code = (%d).\n", rc);
      return (-1);
   } // end if
   //-------------------------------------------------------------------------------------------------------------------
   // Close the file before distributing
   //-------------------------------------------------------------------------------------------------------------------
   rc = FdsCloseKeyedFile( FileHandle, FDS_FILE_CLOSE_TYPE_FULL );
   if ( rc != FDS_SUCCESS )
   {
      printf("FdsCloseKeyedFile failed with return code = (%d).\n", rc);
      return (-2);
   } // end if
   //-------------------------------------------------------------------------------------------------------------------
   // Now distribute a file to be Mirrored with a distribution type of
   // distribute on update (DOU)
   //-------------------------------------------------------------------------------------------------------------------
   rc = FdsSetDistribution( "d:\\dou\\itemrec.dat", // OS Path
                                    FDS_FILE,                      // Distributed File
                                    FDS_MIRRORED,                  // Mirrored Distribution
                                    "DOMAINxx",         // Domain name ignored
                                    FDS_DOU,            // Distribute on Update
                                    FDS_SCOPE_FILE );  // Scope ignored
   printf("FdsSetDistribution completed with return code = (%d).\n",
            rc);
   //-------------------------------------------------------------------------------------------------------------------
   // Now distribute a directory in Broadcast Domain DOMAINyy with a
   // distribution type of distribute on close (DOC)
   //-------------------------------------------------------------------------------------------------------------------
```

```
   rc = FdsSetDistribution( "d:\\doc\\", // OS Path Specification
                                    FDS_DIRECTORY,             // Distributed Directory
                                    FDS_BROADCAST,        // Broadcast Distribution
                                    "DOMAINyy",               // Broadcast Domain name
                                    FDS_DOC,                     // Distribute on Close
                                    FDS_SCOPE_FILE );      // File Scope
   printf("FdsSetDistribution completed with return code = (%d).\n",
               rc);
} // end if
else
{
  // else process errors
}
```

FdsSetupDistMonitor()

Purpose

Prepare to receive notification of data-distribution events.

Syntax

```
#include <fds/dist.h>

long FdsSetupDistMonitor( const char *QName );
```

Parameters

QName — input
Indicates the IPC queue name where the results are to be placed.

Remarks

FdsSetupDistMonitor() can be called from an application that is running on the acting primary distributor or acting backup distributor only. It notifies the Data Distribution component of an IPC queue that can be used for notification of data-distribution, role-related state changes. This information is saved by the Data Distribution component and control is returned.

The following messages are placed in the specified queue to indicate data-distribution state changes:

- The local node is in transition to the acting primary distributor role.
- The local node is the acting primary distributor.
- The local node is in transition to the acting backup distributor role.
- The local node is the acting backup distributor.

One of these messages is immediately generated as the result of calling FdsSetupDistMonitor.

Data Distribution does not perform validation on **QName.** If **QName** does not identify a valid IPC queue, the notification is lost and no error is returned.

Error Conditions

FdsSetupDistMonitor() returns the following values:
-60 FDSERR_CONFIG
-360 FDSERR_NODE_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/ipc.h>
#include <fds/dist.h>
#include <fds/errno.h>

long          rc;                                   // Return from API Call
unsigned long      MaxQSize = 500;    // Maximum Queue Size
long          QueueHandle = 0;              // Queue Handle
char          ReadBuffer[100];              // Message from Read Queue
unsigned int BufferLength = sizeof(ReadBuffer); // Length of Message
long          Timeout = 120;                // Time out value
int           MsgType;                              // Type of Message Read
FDS_DIST_STATE*    DistStateptr;          // Role State structure

// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   rc = FdsCreateQ( "MyQueue", MaxQSize, &QueueHandle );
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------
      // The following call to FdsSetupDistMonitor API must be issued from
      // either the Acting Primary or the Acting Backup Distributor
      //-------------------------------------------------------------
      rc = FdsSetupDistMonitor( "MyQueue" );
      printf( "FdsSetupDistMonitor completed with return code = (%d).\n", rc );
      if ( rc == FDS_SUCCESS )
      {
         //-------------------------------------------------------------------------------------------------------------
         // Read the message returned from calling FdsSetupDistMonitor
         //
         // The messages will be written to "MyQueue". To read the
         // message in "MyQueue" use the FdsReadQ API. The message
         // type for these messages is FDS_DIST_STATE_NOTIFY_MSG. See
         // the FdsReadQ API for more information.
         //-------------------------------------------------------------------------------------------------------------
         rc = FdsReadQ( QueueHandle, &BufferLength, ReadBuffer,
                                    Timeout, &MsgType );
         DistStateptr = (FDS_DIST_STATE*) ReadBuffer;
         printf( "MyQueue completed with return code = (%d).\n"
                        " ---- MyQueue contains: ---- \n"
                        " - Role State = (%d) \n"
                        " - NodeID = (%s) \n",
                        rc,
                        DistStateptr->RoleState,
                        DistStateptr->NodeID );
      }
```

```
        rc = FdsCloseQ(QueueHandle);
    } // end if
} // end if
else
{
// else process errors
}
```

FdsSetupSyncIDNotify()

Purpose

Prepare to receive notification of file or directory synchronization.

Syntax

```
#include <fds/dist.h>

long FdsSetupSyncIDNotify(const FDS_SYNC_ID *SyncID,        const char *QName );
```

Parameters

SyncID — input
Indicates the synchronization ID.

QName — input
Indicates the IPC queue name where the results are to be placed.

Remarks

FdsSetupSyncIDNotify() can be called from an application running on any node. It notifies the Data Distribution component of a synchronization ID and IPC queue name. This information is saved by Data Distribution and control is returned to the caller.

When the file or directory on the local node, associated with synchronization ID **SyncID**, has been brought to the state identified by **SyncID**, the Data Distribution component writes a message with the synchronization ID to the queue specified by **QName.** If the local node is already at this state when this API is called, or has been at this state and has had additional updates applied, this message is written immediately to the queue. If additional updates have been applied, the SequenceNumber portion of the returned SyncID may be higher than the SequenceNumber provided on the call to FdsSetupSyncIDNotify. See “FdsReadQ()”
for more information about the messages.

Data Distribution does not perform validation on **QName.** If **QName** does not identify a valid IPC queue, the notification is lost and no error is returned.

Using FdsCreateSyncID() and FdsSetupSyncIDNotify()

FdsCreateSyncID() is used to uniquely identify a specific set of updates to a distributed sequential, keyed, or binary file. FdsSetupSyncIDNotify() is used

to determine when those updates have been distributed to a particular node.

Consider the following example:

The application APPLA, which runs on the backup distributor, opens and updates a file called FILE.DAT, which resides on the primary distributor. The distribution frequency for FILE.DAT is distribute on update (DOU).

Note: Only the prime copy of a distributed file can be modified. The prime copy of a distributed file resides on the acting primary distributor.

Another application, APPLB, runs on Subordinate 1, a subordinate node in the domain. APPLB uses data from the image copy of FILE.DAT and needs to be notified when the latest updates have been applied.

Figure 1 shows how APPLA issues a call to FdsCreateSyncID() to obtain the synchronization identifier, **sync_id**, and sends it to APPLB (via IPC). APPLB issues a call to FdsSetupSyncIDNotify() to receive notification when the updates have been applied to the copy of FILE.DAT residing on the Subordinate 1 node.

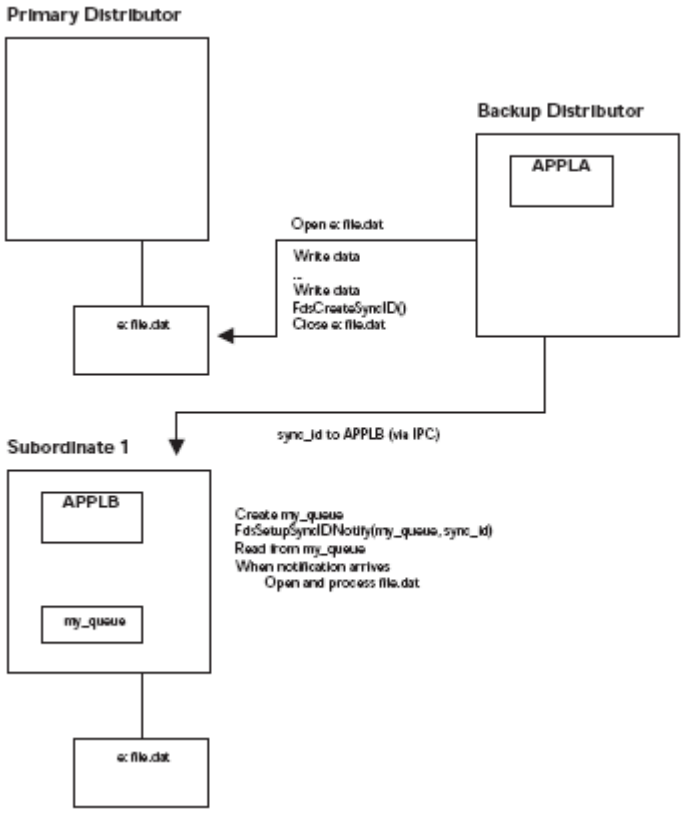


Figure 1. Using FdsCreateSyncID() and FdsSetupSyncIDNotify()

In Figure 1:

- APPLA on the backup distributor opens E:FILE.DAT on the primary distributor and writes data to the file. Because the distribution frequency is distribute on update, APPLA issues a call to FdsCreateSyncID() after the updates have

been made.

Note: If the distribution frequency for file.dat were distribute-on-close (DOC), APPLA would first flush the file by calling FdsCloseKeyedFile() or FdsFlushBinFile() and then calling FdsCreateSyncID().

- FdsCreateSyncID() returns a synchronization identifier, *sync_id*, which uniquely identifies these changes to FILE.DAT. APPLA is responsible for sending *sync_id* (via IPC) to any other programs that may need it.

- APPLB on Subordinate 1 sets up a queue called *my_queue*. (This queue can be used by APPLB to receive a variety of messages.) When APPLB receives *sync_id* from APPLA, APPLB issues a call to FdsSetupSyncIDNotify(*my_queue*, *sync_id*). After DDS applies the updates to FILE.DAT on Subordinate 1, a notification message is written to *my_queue*.

  APPLB, based on the parameters passed to FdsReadQ(), determines how many times *my_queue* will be queried and how long to wait for each query.

Note: The calling program must ensure that the correct queue name is passed to FdsSetupSyncIDNotify(). If the queue name is incorrect, no error is returned.

Error Conditions

FdsSetupSyncIDNotify() returns the following values:
-60 FDSERR_CONFIG
-570 FDSERR_SYNCID

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/ipc.h>
#include <fds/dist.h>
#include <fds/errno.h>

long          rc;                          // Return from API Call
FDS_SYNC_ID   SyncID;                      // SyncID received from the primary
long          QueueHandle = 0;             // QueueHandle
unsigned long MaxQSize = 500;              // Maximum Queue size

// Initialize DDS. Could use FdsInit2() instead of FdsInit().
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------------------------------------------------------
   // The following call to FdsSetupSyncIDNotify API can be invoked from
   // an application running on any node; however, it is usually called
   // on image nodes for notification of file updates on the primary.
   //-------------------------------------------------------------------------------------------------------------
   //-------------------------------------------------------------------------------------------------------------
   // A queue was created on this node and the SyncID generated from
   // the FdsCreateSyncID call has been read from it. The distribute
```

```
   // on update file, "d:\dou\itemrec.dat", was opened on the primary.
   // (See the example in the FdsCreateSyncID API for more information
   // about what occurs on the primary node.)
   //-------------------------------------------------------------------------------------------------------------------
   //-----------------------------------------------------------------
   // Create "MyQueue" for SyncID Notify
   //-----------------------------------------------------------------
   rc = FdsCreateQ( "MyQueue", MaxQSize, &QueueHandle );
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------------------------------------------------------
      // Call FdsSetupSyncIDNotify API to be notified when changes
      // to "d:\dou\itemrec.dat" are distributed. The input SyncID
      // parameter was sent to this node by the application on
      // the primary via a queue.
      //
      // The messages will be written to "MyQueue". To read the
      // message in "MyQueue" use the FdsReadQ API. The message type
      // for these messages is FDS_DIST_SYNC_NOTIFY_MSG. See the
      // FdsReadQ API for more information.
      //-------------------------------------------------------------------------------------------------------------
      rc = FdsSetupSyncIDNotify ( &SyncID, "MyQueue" );
      printf( "FdsSetupSyncIDNotify completed with return code = (%d).\n",
      rc );
   } // end if
} // end if
else
{
  // else process errors
}
```

Chapter 7. Name Services

This chapter describes the Name Services component. It also describes the following APIs, which are available for the Name Services component:

- **FdsChangeLogicNm()** — Change a logical name
- **FdsCreateLogicNm()** — Create a logical name
- **FdsDeleteLogicNm()** — Delete a logical name
- **FdsResolveLogicNm()** — Resolve a logical name
- **FdsResolveRoleNm()** — Resolve a role name
- **FdsSetResetRole()** — Assume or relinquish a role name
- **FdsVerifyRole()** — Verify that the local node is acting a role

The Name Services component provides a name-resolution capability, allowing applications to use logical names instead of hard-coded file names, IPC queue names, and node IDs. These logical names are dynamically resolved when the application runs.

Some names are rarely changed, such as the name of a configuration file. While this name is not likely to change, it is still desirable to avoid using the name in an application program. The use of a logical name allows the file name to be changed without having to rebuild the application. A logical name has the following format:

<**name**>

Where **name** is 1 to FDS_MAX_LOGICAL_NAME_SIZE-2 characters and the less-than and greater-than characters (< and >) are required delimiters.

Others names are more dynamic, for example, the node ID of the primary distributor. This name changes whenever the backup distributor takes over for the primary distributor. In this case, a **role** (the primary distributor) is assumed by a particular node. A logical name can be used to identify this role, and is termed the role name. A role name has the following format:

<**name::**>

Where **name** is 1 to 8 characters, the less than and greater than characters (< and >) are required delimiters, and double colons (::) indicate that this is a role name.

An instance of the Name Services component exists on each node. Each instance maintains a cache of:

- Logical names read from the logical-names file at initialization
- Logical names and role names that have been set for the local node via an API
- All role names that have previously been resolved at the local node

This cache of logical names is unique to each node and global to all processes on a node. Therefore, any application process on the local node using a logical name will have the name resolved to the same string, whereas an application on a remote node could have the same logical name resolve to a different string. For example:

Logical Name	**Resolved Name on Node 1**	**Resolved Name on Node 2**	**Resolved Name on Node 3**
drive	C:	D:	C:

Creating Logical Names

Logical names are created through the DDS Configuration and Response File Utility or the FdsCreateLogicNm() API. The logical name represents a string of characters defined by the person or process creating the logical name. This string is called the **resolved name**. The resolved name can contain other logical names. If a resolved name contains a logical name, the logical name must be delimited by the less-than and greater-than characters (< and >). For example, the following logical names definitions are valid:

Logical Name	**Resolved Name**
<drive>	C:
<prices>	\dept72\prices.dat
<pricefile1>	<drive><prices>
<pricefile2>	<FDSFDXAP::><drive><prices>
<badcheck>	\secur\badcheck.dat
<fileserver>	fsnode::
<badcheckfile1>	<drive><badcheck>
<badcheckfile2>	<fileserver><drive><badcheck>

Note: FDSFDXAP:: is the reserved role name for the acting primary controller. Because it is a role name, it is also a logical name and must be delimited by the less than and greater than characters (< and >).

In the previous example, the double colons (::) are required delimiters for the resolved name of the logical name, **fileserver**. These delimiters separate the node ID from the path name.

The same results could have been achieved by the following logical-name definitions:

Logical Name	Resolved Name
<fileserver>	fsnode
<badcheckfile2>	<fileserver>::<drive><badcheck>

There are two types of logical names for DDS:

Persistent logical names

Persistent logical names are retained in memory across a restart of DDS. Persistent logical names must be defined using the DDS Configuration and Response File Utility. They are stored in the logical-names file and loaded into memory when DDS is initialized. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for information about creating logical names to be used with IPC and File Services.

Temporary logical names

Temporary logical names are not retained across a restart of DDS. Temporary logical names are defined by the FdsCreateLogicNm() API, and are created by the application for special purposes. They are stored in memory and are lost when an IPL is performed at a node or when DDS is restarted.

Logical-Names File

The logical-names file contains the persistent logical names that are to be initially loaded into the Name Services cache. This file can be changed using the DDS Configuration and Response File Utility. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about the management and distribution of the logical-names file. DDS must be restarted on a node for the logical name changes to take effect for that node.

Changing Logical Names

Persistent logical-name definitions contained in the logical names file are changed using the DDS Configuration and Response File Utility. DDS must be restarted on a node for the logical name changes to take effect for that node. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and**

Configuration Guide for more information about the Configuration and Response File Utility.

Persistent and temporary logical names stored in memory are changed using the FdsChangeLogicNm() API. This API has no effect on the persistent logical-name definitions contained in the logical names file. These changes are not retained across IPLs or restarts of DDS. The effect of this API is the same as deleting an existing logical name and creating a new one.

Deleting Logical Names

Persistent logical-name definitions contained in the logical names file are deleted using the DDS Configuration and Response File Utility. DDS must be restarted on a node for the logical-name changes to take effect for that node. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about the Configuration and Response File Utility.

Persistent and temporary logical names stored in memory are deleted using the FdsDeleteLogicNm() API. This API has no effect on the persistent logical-name definitions contained in the Logical-Names File. These deletions are not retained across IPLs or restarts of DDS.

Logical Name Resolution

When the application calls the FdsResolveLogicNm() API, the Name Services component resolves any logical names that appear in the input string. The only exception to this is when a remote role name or node ID is encountered. In that case, this API returns the error -500 FDSERR_REMOTE. Because this resource is remote, it must be resolved on the remote node indicated by the role name or node ID. Your applications must use the IPC component or the File Services component to access the remote resource represented by this logical name.

The logical names in the input string to be resolved must be delimited by the less-than and greater-than characters (< and >). The following table shows how the FdsResolveLogicNm() API resolves the logical names shown in “Creating Logical Names”.

In the previous example, the logical names <badcheckfile2> and <pricefile2> contain an imbedded role name or node ID. Therefore, they must be used with the IPC component or the File Services component, and cannot be resolved directly by the application.

A null node ID indicates that the logical name resolved locally.

Creating Role Names

Although role names are conceptually logical names for a particular node, they are not created by the FdsCreateLogicNm() API, nor can they be defined in the logical names file (fdsln.ln). They are created by DDS or by your application by calling the FdsSetResetRole() API on the node that is to assume the role.

Your application can create roles for its own use. For example, if you have one node perform all of the communications with outside networks, you could assign it the role name <GATEWAY::>. Role names that begin with FDS are reserved.

Role Name Resolution

When the application calls an IPC or File Services API using a logical name, the Name Services component is indirectly called to resolve the logical name. If a role name is found, role name resolution is performed for the application by DDS. The application does not need to know the details of how a role is resolved.

However, application programmers should be aware that if a logical name contains a role name, it is possible that some network communication might occur to resolve the role. Under normal circumstances, the resolution of a role name should take no longer than that of any other logical name. Under some circumstances, however, the role-name information in the Name Services cache might become outdated and have to be refreshed. If the application programmer needs to resolve a role to a node ID, the FdsResolveRoleNm() should be used.

Verifying Role Names

An application can determine whether it is running on a node that is acting a role by using the FdsVerifyRole() API. If this API returns 0 (zero), the role name is local. If it returns the error -550 FDSERR_ROLE_NOT_FOUND, the role name is not local.

FdsChangeLogicNm()

Purpose

Change a logical name.

Syntax

```
#include <fds/names.h>

long FdsChangeLogicNm( const char *LogicalName,
                       const char *ResolvedName );
```

Parameters

LogicalName — input

Specifies the logical name to be changed in the Name Services component. This parameter points to a null-terminated string. The string length, including the less than and greater than (< and >) delimiters, must not be more than the value of FDS_MAX_LOGICAL_NAME_SIZE. Logical names cannot be a null string and cannot start with the prefix FDS. Logical names cannot end with two colons (::). The logical name must be delimited by the less-than and greater-than characters (< and >).

ResolvedName — input

Specifies the string that the logical name represents. This parameter points to a null-terminated string. The string length must not be more than the value of FDS_MAX_RESOLVED_NAME_SIZE. Resolved names cannot be a null string. The name to be resolved can have other

logical names imbedded; if it does, the logical name must be delimited by the less-than and greater-than characters (< and >).

Remarks

FdsChangeLogicNm changes the resolved name for a logical name. This API only affects the Name Services component on the local node. It does not change a logical name in the logical names file. Therefore, persistent logical names that are listed in the logical names file are reset to their original values when DDS is restarted.

Applications can use this API to change the resolved name of temporary and unreserved persistent logical names in the Name Services component on the local node. The DDS Configuration and Response File Utility must be used to permanently change the persistent logical names. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about permanently changing persistent logical names.

Error Conditions

FdsChangeLogicNm returns the following values:

-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-510 FDSERR_RESOLVED_NAME

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/names.h>
#include <fds/errno.h>

long          rc = 0;                                  // Return from API call
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if (rc == FDS_SUCCESS)
{
   //-------------------------------------------------------------------------------------------------------------
   // Change the definition of <drive> in memory. This does not
   // affect any defintion in the logical names file.
   //-------------------------------------------------------------------------------------------------------------
   rc = FdsChangeLogicNm("<drive>", "e:");
   printf("FdsChangeLogicNm completed with return code = (%d).\n",
            rc);
} // end if
else
{
  // else process errors
}
```

FdsCreateLogicNm()

Purpose

Create a logical name.

Syntax

```
#include <fds/names.h>

long FdsCreateLogicNm(      const char *LogicalName, const char
                            *ResolvedName );
```

Parameters

LogicalName — input

Specifies the logical name to be created in the Name Services component. This parameter points to a null-terminated string. The string length, including the less than and greater than (< and >) delimiters, must not be more than the value of FDS_MAX_LOGICAL_NAME_SIZE. A logical name cannot be a null string and cannot start with the prefix FDS. A logical name cannot contain two consecutive colons (::). The logical name must be delimited by the less than and greater than characters (< and >). These delimiters (< and >) may not be imbedded within the logical name.

ResolvedName— input

Specifies the string that the logical name represents. This parameter points to a null-terminated string. The string length must not be more than the value of FDS_MAX_RESOLVED_NAME_SIZE. A resolved name cannot be a null string. The name to be resolved can have other logical names imbedded. If it does, the logical name must be delimited by the less than and greater than characters (< and >).

Remarks

FdsCreateLogicNm() adds a temporary logical name to the Name Services component. This API affects only the Name Services component on the local node. Applications can call this API to create a logical name to be used in resolving file names, IPC queue names, or other special names. Temporary logical names are created in memory and are not persistent across IPLs or restarts of DDS. Persistent logical names must be created using the DDS Configuration and Response File Utility.

Logical names are used by the Name Services component to resolve names. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about creating logical names to be used with IPC and File Services.

Error Conditions

FdsCreateLogicNm() returns the following values:
-170 FDSERR_EXISTS
-300 FDSERR_LOGICAL_NAME
-510 FDSERR_RESOLVED_NAME

Examples

```
#include <stdio.h>
```

```
#include <fds/fds.h>
#include <fds/names.h>
#include <fds/errno.h>

long          rc = 0;                                  // Return from API call
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if (rc == FDS_SUCCESS)
{
   //-------------------------------------------------------------------------------
   // Create a temporary logical name for the drive
   //-------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<drive>", "d:" );
   printf( "FdsCreateLogicNm completed with return code = (%d).\n",
               rc);
   //-------------------------------------------------------------------------------------------------------------
   // Create a temporary logical Nm for the Nm of the price file
   //-------------------------------------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<prices>", "\\registerdata\\prices.dat" );
   printf( "FdsCreateLogicNm completed with return code = (%d).\n",
               rc);
   //-------------------------------------------------------------------------------
   // Create a temporary logical Nm for the price file
   //-------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<pricefile>", "<drive><prices>" );
   printf( "FdsCreateLogicNm completed with return code = (%d).\n",
               rc);
   //-------------------------------------------------------------------------------------------------------
   // Create a temporary logical Nm for the master price file
   // on the primary controller using role Nm <FDSFDXAP::>
   //-------------------------------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<masterpricefile>",
                                          "<FDSFDXAP::><drive><prices>" );
   printf( "FdsCreateLogicNm completed with return code = (%d).\n",
               rc);
   //-------------------------------------------------------------------------------------
   // Create a temporary logical Nm for the price file on
   // node REG51
   //-------------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<term51pricefile>",
                                                "REG51::<drive><prices>" );
   printf( "FdsCreateLogicNm completed with return code = (%d).\n",
               rc);
} // end if
else
{
// else process errors
}
```

FdsDeleteLogicNm()

Purpose

Delete a logical name.

Syntax

```
#include <fds/names.h>

long FdsDeleteLogicNm( const char *LogicalName );
```

Parameters

LogicalName— input

Specifies the logical name to be deleted from the Name Services component. This parameter points to a null-terminated string. The string length, including the less than and greater than (< and >) delimiters, must not be more than the value of FDS_MAX_LOGICAL_NAME_SIZE. A logical name cannot be a null string and cannot start with the prefix FDS. A logical name cannot end with two colons (::). The logical name must be delimited by the less-than and greater-than characters (< and >).

Remarks

FdsDeleteLogicNm() deletes a logical name from the Name Services component. This API affects only the Name Services component on the local node. This API does not delete a logical name from the logical-names file. Therefore, persistent logical names listed in the logical-names file are added back to the Name Services component when DDS is restarted.

Applications can use this API to remove temporary and unreserved persistent logical names from the Name Services component on the local node. The DDS Configuration and Response File Utility must be used to permanently remove persistent logical names. Refer to the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about removing persistent logical names.

Error Conditions

FdsDeleteLogicNm() returns the following values:

-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/names.h>
#include <fds/errno.h>

long          rc = 0;                              // Return from API call
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if (rc == FDS_SUCCESS)
{
   //-----------------------------------------------------------------
   // Delete the definition of <drive> in memory. This does not
   // affect any defintion in the logical names file.
   //-----------------------------------------------------------------
   rc = FdsDeleteLogicNm( "<drive>" );
   printf( "FdsDeleteLogicNm completed with return code = (%d).\n",
               rc);
```

```
} // end if
else
{
  // else process errors
}
```

FdsResolveLogicNm()

Purpose

Resolve a logical name.

Syntax

```
#include <fds/names.h>

long FdsResolveLogicNm(const char *InputString, char *OutputString, unsigned int
                                        *OutputStringLen );
```

Parameters

InputString — input

Points to the null-terminated input string that is to be resolved. Imbedded logical names must be delimited by the less-than and greater-than characters (< and >).

OutputString — output

Points to the buffer where the resolved string will be stored.

OutputStringLen — input/output

Input A pointer to the location where the length of the buffer parameter is 0 (zero) when the API is called, the error -20_ADDRESS will be returned.

Output

When this API completes successfully, the data in the location pointed to by **OutputStringLen** is replaced with the length of the null-terminated string placed in **OutputString**. This length includes the null terminator.

If this API returns the error -40 FDSERR_BUFFER_SIZE, this parameter is set to the required buffer size.

Remarks

FdsResolveLogicNm() submits a string to the Name Services component for resolution. This API provides a simple, string-substitution function. It does not allow resolution of remote role names or node IDs. All imbedded logical names are resolved from left to right. If an undefined logical name is encountered, the -310 FDSERR_LOGICAL_NAME_NOT_FOUND error is returned.

If a remote role name or node ID is encountered, this API returns the -500 FDSERR_REMOTE error because the imbedded role name or node ID indicates that this resource is remote; the resource must be resolved on the

remote node indicated by the role name or node ID. Your applications must use IPC or File Services to access the remote resource represented by this logical name.

Because the resource is local, resolution will If a local role name or node ID is encountered, it is resolved to a null string because the imbedded role name or node ID indicates that this resource is local. continue until complete or until an error is encountered. In the case where **InputString** resolves to only a local node ID, the buffer pointed to by **OutputString** will contain a null string.

The logical names in the input string to be resolved must be delimited by the less-than and greater-than characters (< and >). If no logical name is found in **InputString**, the string is copied to the buffer pointed to by **OutputString**.

To determine the local node ID, use the FdsQueryConfig() API.

Error Conditions

FdsResolveLogicNm() returns the following values:
-40 FDSERR_BUFFER_SIZE
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-410 FDSERR_OVERFLOW
-500 FDSERR_REMOTE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/names.h>
#include <fds/errno.h>

long        rc = 0;                                 // Return from API call
char        OutBuffer[500];     // Resolve name buffer
unsigned int            OutBufferSize = sizeof(OutBuffer); // Size of OutBuffer
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if (rc == FDS_SUCCESS)
{
   //-------------------------------------------------------------------------
   // Create a temporary logical name for the drive
   //-------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<drive>", "d:" );
   //-------------------------------------------------------------------------------------------------------------
   // Create a temporary logical Nm for the Nm of the price file
   //-------------------------------------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<prices>", "\\registerdata\\prices.dat" );
   //-------------------------------------------------------------------------
   // Create a temporary logical Nm for the price file
   //-------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<pricefile>", "<drive><prices>" );
   //-------------------------------------------------------------------------------------------------------------
```

```
   // Create a temporary logical Nm for the master price file
   // on the primary controller using role Nm <FDSFDXAP::>
   //-------------------------------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<masterpricefile>",
   "<FDSFDXAP::><drive><prices>" );
   //-------------------------------------------------------------------------------
   // Create a temporary logical Nm for the price file on
   // node REG51
   //-------------------------------------------------------------------------------
   rc = FdsCreateLogicNm( "<term51pricefile>",
   "REG51::<drive><prices>" );
   //-------------------------------------------------------------------------------------------------------
   // Change the definition of <drive> in memory. This does not
   // affect any defintion in the logical names file.
   //-------------------------------------------------------------------------------------------------------
   rc = FdsChangeLogicNm( "<drive>", "e:" );
   //-------------------------------------------------------------------------------
   // Resolve the logical name for <prices>
   //-------------------------------------------------------------------------------
   OutBuffer[0] = 0;
   OutBufferSize = sizeof(OutBuffer);
   rc = FdsResolveLogicNm( "<prices>", OutBuffer, &OutBufferSize );
   printf( "<prices> resolves to = (%s)\n", OutBuffer );
   //-------------------------------------------------------------------------------
   // Resolve the logical name for <pricefile>
   //-------------------------------------------------------------------------------
   OutBuffer[0] = 0;
   OutBufferSize = sizeof(OutBuffer);
   rc = FdsResolveLogicNm( "<pricefile>", OutBuffer, &OutBufferSize);
   printf( "<pricefile> resolves to = (%s)\n", OutBuffer );
   //-------------------------------------------------------------------------------
   // Resolve the logical name for <masterpricefile>
   //-------------------------------------------------------------------------------
   OutBuffer[0] = 0;
   OutBufferSize = sizeof(OutBuffer);
   rc = FdsResolveLogicNm( "<masterpricefile>", OutBuffer,
   &OutBufferSize );
   if ( rc == FDSERR_REMOTE )
      printf( "<masterpricefile> refers to a remote file. "
      "Use file services to access it\n");
   else
      printf( "<masterpricefile> resolves to = (%s)\n", OutBuffer );
   //-------------------------------------------------------------------------------
   // Resolve the logical name for <term51pricefile>
   //-------------------------------------------------------------------------------
   OutBuffer[0] = 0;
   OutBufferSize = sizeof(OutBuffer);
   rc = FdsResolveLogicNm( "<term51pricefile>", OutBuffer,
   &OutBufferSize );
   if ( rc == FDSERR_REMOTE )
      printf( "<term51pricefile> refers to a remote file. "
      "Use file services to access it\n");
   else
      printf( "<term51pricefile> resolves to = (%s)\n", OutBuffer );
} // end if
else
{
   // else process errors
```

```
}
```
The output of this code on a machine that is acting as the role <FDSFDXAP::> and
has a node ID of REG51 is:
```
<prices> resolves to \registerdata\prices.dat
<pricefile> resolves to e:\registerdata\prices.dat
<masterpricefile> resolves to e:\registerdata\prices.dat
<term51pricefile> resolves to e:\registerdata\prices.dat
```
The output of this code on a machine that is not acting as the role <FDSFDXAP::> and does not have a node ID of REG51 is:
```
<prices> resolves to \registerdata\prices.dat
<pricefile> resolves to e:\registerdata\prices.dat
<masterpricefile> refers to a remote file. Use file services to access.
<term51pricefile> refers to a remote file. Use file services to access.
```

FdsResolveRoleNm()

Purpose

Resolve a role to a node ID.

Syntax

```
#include <fds/names.h>

long FdsResolveRoleNm( FDS_ROLE_NAME RoleName, int SeekMethod,
                       FDS_NODE_NAME *NodeID );
```

Parameters

RoleName — input

Points to a null-terminated string of the role name to be resolved. The role name must contain double colons (::) and cannot begin with the characters FDS. The role name must be delimited by the less-than and greater-than characters (< and >).

SeekMethod — input

Indicates where to look for the role name. This parameter is required. Valid values are:

FDS_CACHE_ONLY

Search cache on the local machine only.

FDS_NETWORK_ONLY

Query all nodes in the DDS system.

FDS_CACHE_FIRST

Search cache on the local machine first. If no match is found, query all nodes in the DDS system.

NodeID — output

Points to the location where the node ID string is stored.

Remarks

FdsResolveRoleNm() resolves a role to a node ID using cache, querying all nodes in the DDS system, or both. The **SeekMethod** parameter controls how the

role resolution is done.

Local roles will always resolve to the local node ID regardless of the value of the **SeekMethod** parameter.

Error Conditions

FdsResolveRoleNm() returns the following values:

-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND
-558 FDSERR_SEEK_TYPE

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/names.h>
#include <fds/errno.h>

long          rc = 0;                                    // Return from API call
FDS_NODE_NAME    OutBuffer;             // Resolve role name buffer
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if (rc == FDS_SUCCESS)
{
   //-----------------------------------------------------------------
   // Resolve role name for <MyRole::> using only the cache
   //-----------------------------------------------------------------
   rc = FdsResolveRoleNm( "<MyRole::>", FDS_CACHE_ONLY, &OutBuffer; );
   switch (rc)
   {
      case FDS_SUCCESS:
      printf("<MyRole::> resolves to (%s).\n", OutBuffer );
      break;
      case FDSERR_ROLE_NOT_FOUND:
      printf("The role name was not found.\n");
      break;
      case FDSERR_ROLE_NAME:
      printf("The role name syntax is invalid.\n");
      break;
      default:
      printf( "FdsResolveRoleNm completed with return code = (%d).\n", rc);
      break;
   }
   //-----------------------------------------------------------------
   // Resolve role name for <MyRole::> by querying all nodes in the
   // system to determine whether the role still exists
   //-----------------------------------------------------------------
   rc = FdsResolveRoleNm( "<MyRole::>", FDS_NETWORK_ONLY, &OutBuffer; );
   switch (rc)
   {
      case FDS_SUCCESS:
      printf("<MyRole::> resolves to (%s).\n", OutBuffer );
      break;
      case FDSERR_ROLE_NOT_FOUND:
      printf("The role name was not found.\n");
      break;
      case FDSERR_ROLE_NAME:
```

```
        printf("The role name syntax is invalid.\n");
        break;
        default:
        printf( "FdsResolveRoleNm completed with return code = (%d).\n", rc);
        break;
    }
    //-----------------------------------------------------------------
    // Resolve role name for <MyRole::> using the cache first. If the
    // name is not in cache, query all nodes in the system to
    // determine whether the role still exists
    //-----------------------------------------------------------------
    rc = FdsResolveRoleNm( "<MyRole::>", FDS_CACHE_FIRST, &OutBuffer; );
    switch (rc)
    {
        case FDS_SUCCESS:
        printf("<MyRole::> resolves to (%s).\n", OutBuffer );
        break;
        case FDSERR_ROLE_NOT_FOUND:
        printf("The role name was not found.\n");
        break;
        case FDSERR_ROLE_NAME:
        printf("The role name syntax is invalid.\n");
        break;
        default:
        printf( "FdsResolveRoleNm completed with return code = (%d).\n", rc);
        break;
    }
} // end if
else
{
// else process errors
}
```

The output of this code when role <MyRole::> exists on node MYSERVER and the local machine's cache does not have an entry for <MyRole::> is:

```
The role name was not found.
<MyRole::> resolves to MYSERVER.
<MyRole::> resolves to MYSERVER.
```

If this code is run a second time, the output will be different because the local machine's cache is updated whenever a role is discovered. The output for the second run is:

```
<MyRole::> resolves to MYSERVER.
<MyRole::> resolves to MYSERVER.
<MyRole::> resolves to MYSERVER.
```

If this code is run when MYSERVER is down and the local machine's cache has an entry for <MyRole::> set to MYSERVER, the output is:

```
<MyRole::> resolves to MYSERVER.
The role name was not found.
<MyRole::> resolves to MYSERVER.
```

FdsSetResetRole()

Purpose

Assume or relinquish a role name.

Syntax

```
#include <fds/names.h>

long FdsSetResetRole( FDS_ROLE_NAME RoleName, int Flag );
```

Parameters

RoleName — input

Points to a null-terminated string of the role name to be set. The role name must contain double colons (::) and cannot begin with the characters FDS. The role name must be delimited by the less-than and greater-than characters (< and >). This parameter cannot be a logical name.

Flag — input

Indicates whether the node is acting a role. One of the following attributes must be chosen:

FDS_RESET_ROLE

The node no longer has a role.

FDS_SET_ROLE

The node has assumed a role.

Remarks

If the **Flag** parameter is set to **FDS_SET_ROLE**, this API indicates to the Name Services component that the local node has assumed a role. As a result, strings containing the role name specified by the **RoleName** parameter and passed to IPC or File Services anywhere in the system will resolve to this node.

If the **Flag** parameter is set to **FDS_RESET_ROLE**, this API indicates to the Name Services component that the local node is no longer acting the role specified by **RoleName**.

The Name Services component does not attempt to prevent role conflicts. The application must ensure that two nodes in the same system are not acting the same role concurrently.

When a role is assumed on a node, DDS creates a thread to periodically announce, via a broadcast message, that the new role has been assumed. This thread exists for the announcement period or until the role is relinquished. If your application sets many roles in a short period of time, you may find it necessary to increase the number of threads available to the operating system.

Error Conditions

FdsSetResetRole() returns the following values:

-170 FDSERR_EXISTS
-210 FDSERR_FLAG
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/names.h>
#include <fds/errno.h>

long          rc = 0;                                    // Return from API call
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if (rc == FDS_SUCCESS)
{
   //---------------------------------------------------------------
   // Assume the role <MyRole::> on this machine
   //---------------------------------------------------------------
   rc = FdsSetResetRole( "<MyRole::>", FDS_SET_ROLE );
   //---------------------------------------------------------------
   // Check that <MyRole::> is set to this machine
   //---------------------------------------------------------------
   rc = FdsVerifyRole( "<MyRole::>" );
   if ( FDS_SUCCESS == rc )
   {
      printf("Role <MyRole::> is local\n");
   }
   else
   {
      printf( "Role <MyRole::> is remote \n");
   }
   //---------------------------------------------------------------
   // Relinquish the role <MyRole::> on this machine
   //---------------------------------------------------------------
   rc = FdsSetResetRole( "<MyRole::>", FDS_RESET_ROLE );
} // end if
else
{
// else process errors
}
```

The output of this code is:

```
Role <MYROLE::> is local
```

FdsVerifyRole()

Purpose

Verify that the local node is acting a role.

Syntax

```
#include <fds/names.h>

long FdsVerifyRole(FDS_ROLE_NAME RoleName);
```

Parameters

RoleName — input

Points to a null-terminated string of the role name to be verified. The role

name must contain double colons (::) and must be delimited by the less-than and greater-than characters (< and >).

Remarks

FdsVerifyRole() allows an application to determine whether it is running on a node that is acting a specified role. If this API returns 0 (zero), the role name is local. If it returns the error -550 FDSERR_ROLE_NOT_FOUND, the role name is not local.

Error Conditions

FdsVerifyRole() returns the following values:

-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND

Examples

```
#include <stdio.h>
#include <fds/fds.h>
#include <fds/names.h>
#include <fds/errno.h>

long          rc = 0;                                    // Return from API call
// Initialize DDS. Could use FdsInit2() instead of FdsInit()
rc = FdsInit();
// If initialization was successful
if (rc == FDS_SUCCESS)
{
   //---------------------------------------------------------------
   // Relinquish the role <MyRole::> on this machine
   //---------------------------------------------------------------
   rc = FdsSetResetRole( "<MyRole::>", FDS_RESET_ROLE );
   //---------------------------------------------------------------
   // Check that <MyRole::> is set to this machine
   //---------------------------------------------------------------
   rc = FdsVerifyRole( "<MyRole::>" );
   if ( FDS_SUCCESS == rc )
      printf("Role <MyRole::> is local\n");
else
      printf("Role <MyRole::> is remote\n");
} // end if
else
{
   // else process errors
}
```

The output of this code is:

```
Role <MYROLE::> is remote
```

Chapter 8. Interprocess Communication

The Interprocess Communication (IPC) component provides a set of APIs to facilitate the exchange of information between applications at local nodes and remote nodes. The information is exchanged in the form of messages. The IPC component maintains a message queue in which messages accumulate and from which messages can be removed. Message queues can be accessed

through the IPC APIs.

The IPC APIs for receiving messages are:

- **FdsCloseQ()** — Close a queue handle
- **FdsCreateQ()** — Create and open a queue on the local node
- **FdsLockQ()** — Lock a queue
- **FdsOpenQ()** — Open a queue on the local or remote node
- **FdsPurgeMsg()** — Purge the next message in the queue
- **FdsQueryQ()** — Query information about a local queue
- **FdsReadQ()** — Read the next message from a queue
- **FdsUnlockQ()** — Unlock a locked queue

The IPC APIs for sending messages using point-to-point messaging are:

- **FdsCloseQ()** — Close a queue handle
- **FdsOpenQ()** — Open a queue on the local or remote node
- **FdsWriteQ()** — Write a message to a queue

See “Writing Messages to Queues” for a description of point-to-point messaging.

The IPC API for sending messages using broadcast messaging is FdsBroadcastQ(). See “Writing Messages to Queues” for a description of broadcast messaging.

The messages contained in message queues are application messages. There is no fixed format for messages being written to and read from queues, and the IPC component does not interpret messages in any way.

An application must use the IPC component to create at least one queue for receiving input messages. The name of the queue can be a DDS logical name, and is required to be unique within the local node only. Once a queue is created, other local and remote applications can write messages to that queue. Applications do not need to detect whether another application is local or remote before writing a message to a queue.

When a queue is created, the returned queue handle has read/write permission, which can be used in all other IPC APIs that require a queue handle.

Writing Messages to Queues

You can write messages either to a single node or to a set of nodes.

Writing Messages to a Single Node

Writing messages to a single node is called **point-to-point messaging.**

Point-to-point messaging is a reliable method for sending messages. This method allows the application to write a message to a single queue on a single node within each FdsWriteQ() request.

The application must open the queue before writing a message to the queue, and must identify the node ID where the queue is located. When a queue is opened on a remote node, the IPC component locates the node, establishes a session with it, and saves information about it and the NetBIOS session or TCP/IP connection that was used to communicate with it.

A unique queue handle is returned. All messages that are written to the queue using this queue handle are sent to the remote node over the connection that was established during the FdsOpenQ() request.

In addition, because the node ID uniquely identifies a single node, the IPC component provides:

- Confirmation that a message has been successfully delivered to the destination node. (Retries are performed if necessary by the IPC component before returning to the calling application.)
- Confirmation that a message has been successfully delivered to the destination node and has been written to the destination queue. (Retries are performed if necessary by the IPC component before returning to the calling application.)
- Notification if communication with the node fails or if the queue is deleted (closed by the owner).
- Confirmation that the destination node is still acting the role that was specified during the FdsOpenQ() request.

If a single message needs to be sent to a queue that exists on multiple nodes, point-to-point messaging requires that the application open the queue on each of the nodes and then call the IPC component to send the message to each unique node. Alternatively, you can use the broadcast messaging method described below.

Writing Messages to a Set of Nodes

Writing a single message to a set of nodes is called **broadcast messaging.** The set of nodes to which the message is sent must be defined as a broadcast domain.

This method does not require the application to open a queue before writing a message; nor does it require the application to specify the unique node ID of the node or nodes where the queue exists. The application must only specify the broadcast domain name and the queue name. See Chapter 6. Data Distribution for more information about how to create the broadcast domain name.

When the message is sent, it is transmitted to all IPC nodes in the system. At each IPC node that receives the message, the message is discarded if the node ID is not defined as part of the specified broadcast domain, or if the specified queue does not exist, is full, or is locked on that node.

Note: Broadcast messaging does not provide confirmation of a successful write; nor does it automatically retry to send the

message in the event of a failure.

FdsBroadcastQ()

Purpose

Broadcast a message to a specified queue name on a node that is a member of a specified broadcast-domain name.

Syntax

```
#include <fds/ipc.h>

long FdsBroadcastQ(const char *BroadcastQPtr, unsigned int BuffSize, const void *BuffPtr );
```

Parameters

BroadcastQPtr — input

A pointer to a null-terminated string that contains a retail path specification or a logical name that resolves to a retail path specification. The retail path specification or resolved logical name must contain a destination broadcast domain as well as a destination queue name. See “File Names and Queue Names” for more information.

The retail path specification identifies the following names:

Destination broadcast-domain name

Identifies the list of node IDs that should handle the message if a message is received.

The predefined broadcast-domain name FDSS**xxxx**, where **xxxx** is the system ID, can be used to send a message to every node in the system. This broadcast domain name is available for use with this API only , and cannot be used with any Data Distribution APIs.

Destination queue name

Specifies the name or logical name of the queue at each node in the broadcast domain specified by the input broadcast-domain name to which the message should be written.

If a logical name is specified, each receiving node resolves the logical name of the queue to determine which queue should receive the message.

BuffSize — input

The length, in bytes, of the message to be broadcast. The range for this parameter is 1 to FDS_MAX_BCAST_SIZE. A **BuffSize** value of 0 (zero) is not valid and results in the error -20 FDSERR_ADDRESS.

BuffPtr — input

A pointer to a buffer that contains the message to be broadcast and written to the specified queue name on each node in the specified broadcast-domain name.

The data pointed to by **BuffPtr** does not need to be null-terminated. However, if it is, the value for **BuffSize** must include 1 byte if the

null terminator is to be copied with the message data.

Remarks

This API broadcasts a message on the network. Each node that receives the message processes the message only if the node is defined as part of the specified destination broadcast-domain name. If the node belongs to the specified broadcast domain name definition, the node writes the message to the destination queue if the queue exists on that node and is not locked or full. If the node does not belong to the specified broadcast domain name, or if the queue has not been created, is locked, or full, the node does not write the message to the queue.

If the sending node is also part of the destination broadcast domain and has a queue with the name of the destination queue, a copy of the message is placed in the local queue.

Refet to the **IBM Distributed Data Services/Controller Services Feature for** *Windows User's Guide* for more information about defining broadcast domain names.

Notes:

1. The IPC component does not validate that the specified broadcast-domain name is defined or that any or all of the nodes are active that are defined as part of the specified broadcast-domain name.

   The number of active nodes in the system defined as part of the specified broadcast-domain name does not affect how the broadcast function works. The message, along with the destination broadcast-domain name and destination queue name, is sent to all IPC nodes, but is discarded by nodes that do not belong to the specified broadcast domain. This should be taken into consideration when selecting this method for writing messages to one or more queues.
2. There is no guaranteed delivery with this method: data can be lost or duplicated, or messages can arrive out of order. No confirmation is provided to report that the message arrived successfully at any or all of the destination nodes defined in the specified broadcast domain name. In API will be completed successfully.
3. A unique node ID can be substituted for a broadcast-domain name and will not be detected. The message will be broadcast to all IPC nodes, and is then discarded at all nodes except theaddition, a broadcast to a non-existent queue name cannot be detected by the sending node; the FdsBroadcastQ single node with the specified node ID. If you need to send a message to only a single node, point-to-point messaging is the most direct method. The exception to this rule is when the unique node ID is the same as the node ID on which the caller is running; in this case, the error -120 FDSERR_DOMAIN_NAME is returned.

Error Conditions

This API returns the following values:

-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-120 FDSERR_DOMAIN_NAME
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-330 FDSERR_MESSAGE_SIZE
-410 FDSERR_OVERFLOW

-450 FDSERR_QUEUE_NAME
-500 FDSERR_REMOTE

Examples

```
#include <stdio.h>
#include <string.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long          rc;                                   // Return from API Call
unsigned long       MaxQSize = 500;     // Maximum Queue Size
long                CreateQHandle = 0;            // For Storing Queue Handle
typedef struct
{
   char MsgText[100];
   int MsgLen;
   int MsgId;
} MSG_DATA;
MSG_DATA            MsgData;                      // Message to be broadcast
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------
   // Create MyQueue
   //---------------------------------------------------------
   rc = FdsCreateQ( "MyQueue", MaxQSize, &CreateQHandle );
   if (rc == FDS_SUCCESS)
   {
      //---------------------------------------------------------
      // Message text to be broadcast
      //---------------------------------------------------------
      strcpy( MsgData.MsgText, "This is a broadcast message test");
      MsgData.MsgId = 2;
      //-------------------------------------------------------------------------------------------------
      // Determine the message length that is to be broadcasted
      //-------------------------------------------------------------------------------------------------
      MsgData.MsgLen = strlen(MsgData.MsgText);
      //-------------------------------------------------------------------------------------------------------------
      // Broadcast the message to the following:
      // BroadcastDomain name = DomainX
      // Queue name = DomainXQueue
      //
      // Assuming a default system ID of 0000, you could also specify
      // BroadcastDomain name = FDSS0000, which would broadcast
      // the message to all nodes in the Distributed Data Services
      // system.
      //-------------------------------------------------------------------------------------------------------------
      rc = FdsBroadcastQ("DomainX::DomainXQueue", MsgData.MsgLen,
                  &MsgData);
   printf( "FdsBroadcastQ completed with return code = (%d).\n", rc );
   }
   rc = FdsCloseQ( CreateQHandle );
   } // end if
else
{
```

```
  // else process errors
}
```

FdsCloseQ()

Purpose

Close a queue handle.

Syntax

```
#include <fds/ipc.h>

long FdsCloseQ( long QHandle );
```

Parameters

QHandle — input

Specifies the queue handle to be closed. The queue handle can have been returned in either the FdsCreateQ() or FdsOpenQ().

Remarks

This API closes the queue. Any pending commands with the specified queue handle are cancelled and returned with the error -420 FDSERR_QUEUE_CLOSED.

If the specified queue handle was the queue handle returned in the FdsCreateQ() API, all outstanding messages are purged from the queue and the queue is deleted. Any new requests with queue handles that were returned in the FdsOpenQ() API for this queue receive an error indicating that the queue is closed.

If the specified queue handle was one returned in the FdsOpenQ() API, the queue contents are not affected and the queue remains available to the application that created the queue as well as to other applications that have opened the queue.

The specified queue handle is no longer available for use after this API has completed successfully.

Error Conditions

FdsCloseQ() returns the following value:
-220 FDSERR_HANDLE

Examples

```
#include <stdio.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long    rc;                             // Return from API Call
long    NotificationQHandle = 0;    // No Notification Requested
long    OpenQHandle;                    // Queue Handle from OpenQ
```

```
long          timeout = 60;                          // timeout FdsOpenQ Value
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------
   // Open queue on node id NODE_A
   //-------------------------------------------------------------
   rc = FdsOpenQ( "NODE_A::MyQueue",
                  NotificationQHandle,
                  timeout,
                  &OpenQHandle );
   // Work with remote queue
   //-------------------------------------------------------------------------------------
   // Call FdsCloseQ to close queue on node id NODE_A
   //-------------------------------------------------------------------------------------
   rc = FdsCloseQ( OpenQHandle );
   printf( "FdsCloseQ completed with return code = (%d).\n", rc );
} // end if
else
{
   // else process errors
}
```

FdsCreateQ()

Purpose

Create and open a queue on the local node and assign a queue handle.

Syntax

```
#include <fds/ipc.h>

long FdsCreateQ(const char *QNamePtr, unsigned long MaxQSize, long
                *QHandlePtr );
```

Parameters

QNamePtr — input

A pointer to the name of the queue to be created on the local node. This name is chosen by your application and can be a logical name. The queue name or resolved logical name must be a null-terminated string of not more than 20 characters, and must be unique on the local node. If another queue with this name exists on the local node, an error is returned.

Queue names that begin with the prefix FDS or fds are reserved.

MaxQSize — input

The maximum number of bytes that can be written to the queue before the queue is full and additional application messages to be written to the queue are either discarded or blocked. See “FdsWriteQ()” for more information.

QHandlePtr — output

A pointer to the location where the assigned queue handle is stored. The queue handle can be used by your application to access the queue. It can also be used in any IPC API that requires a queue handle as input.

The creator of the queue is the queue owner.

If you call the FdsCloseQ() API with the returned queue handle, the queue is destroyed.

Remarks

If a queue with the specified name does not already exist on the local node, FdsCreateQ() creates and opens a queue on the local node, returning a queue handle that your application can use to access the queue. If a queue with the specified name already exists on the node, the error -170 FDSERR_EXISTS is returned.

The queue can also be opened by other applications using the FdsOpenQ() API.

Upon successful completion, the returned queue handle can be used by all threads in your application.

When you close a queue whose specified queue handle was returned in the FdsCreateQ() API, the queue is destroyed. Any pending read requests will be completed with the error -420 FDSERR_QUEUE_CLOSED. In addition, any subsequent requests with a queue handle associated with this queue will result in the error -420 FDSERR_QUEUE_CLOSED. See “FdsCloseQ()” for more details.

Error Conditions

FdsCreateQ() returns the following values:
-170 FDSERR_EXISTS
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-410 FDSERR_OVERFLOW
-450 FDSERR_QUEUE_NAME
-470 FDSERR_QUEUE_SIZE
-500 FDSERR_REMOTE

Examples

```
#include <stdio.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>
long rc; // Return from API Call
unsigned long MaxQSize = 500; // Maximum Queue Size
long CreateQHandle = 0; // For Storing Queue Handle
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-----------------------------------------------------------------------------------------------------------------
   // The following call to FdsCreateQ API creates a queue named MyQueue
```

```
    // and returns a handle in CreateQHandle parameter
    //-------------------------------------------------------------------------------------------------------------------
    rc = FdsCreateQ( "MyQueue", MaxQSize, &CreateQHandle );
    printf( "FdsCreateQ completed with return code = (%d).\n", rc );
    // Work with created queue
    // Close the Queue
    rc = FdsCloseQ( CreateQHandle );
} // end if
else
{
  // else process errors
}
```

FdsLockQ()

Purpose

Lock the queue associated with the input **QHandle.** While the queue is locked, no additional messages can be added.

Syntax

```
#include <fds/ipc.h>

long FdsLockQ ( long QHandle );
```

Parameters

QHandle — input

Specifies the queue handle associated with the queue to be locked. Only the handle returned in the FdsCreateQ() API can be used to lock the queue. Queue handles returned in FdsOpenQ() requests for a queue are not valid for this request.

Remarks

FdsLockQ() locks the queue. No additional messages can be added to the queue while it is locked. All messages that are received while the queue is locked are discarded. At the time the queue is locked, all blocked writes (from FdsWriteQ()) are completed with the error -10 FDSERR_ACCESS and the contents of the queue remain unchanged. The messages already in the queue can be read, but no additional messages are added to the queue.

If the queue is already locked at the time of this request, the queue remains locked and this API is completed without an error.

Error Conditions

FdsLockQ() returns the following values:

-10 FDSERR_ACCESS
-220 FDSERR_HANDLE

Examples

```
#include <stdio.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long          rc;                                   // Return from API Call
unsigned long          MaxQSize = 500;     // Maximum Queue Size
long          CreateQHandle = 0;                    // For Storing Queue Handle
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   // Create MyQueue
   rc = FdsCreateQ( "MyQueue", MaxQSize, &CreateQHandle );
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------
      // Call FdsLockQ to lock MyQueue
      //-------------------------------------------------------
      rc = FdsLockQ( CreateQHandle );
      printf( "FdsLockQ completed with return code = (%d).\n", rc );
   }
   // Close the Queue
   rc = FdsCloseQ( CreateQHandle );
} // end if
else
{
   // else process errors
}
```

FdsOpenQ()

Purpose

Open a queue on the local or remote node and return a queue handle that can be used to write to the queue.

Syntax

```
#include <fds/ipc.h>

long FdsOpenQ(const char *RetPathSpecPtr,   long NotificationQHandle, long timeout, long
                        *QHandlePtr );
```

Parameters

RetPathSpecPtr — input

A pointer to a string that contains a retail path specification. The retail path specification can be a logical name, but must be a null-terminated string. The input retail path specification or resolved logical name must contain a queue name. It can optionally contain a node ID or RoleName. See “File Names and Queue Names” for more information about retail path specification formats. Each piece of the retail path specification or

resolved logical name determines what action the IPC component will take:

QName
The name of the queue to be opened. See FdsCreateQ() for more information about queue names. A queue with the specified queue name must have been created on the specified node using the FdsCreateQ() API before it can be opened using the FdsOpenQ() API.

RoleName or node ID A **RoleName** can be specified to tell the IPC component to open the queue on whatever node is acting the role defined by the **RoleName**. The IPC component resolves the **RoleName** to the actual **node ID**.

A **node ID** can be specified if the queue is to be opened on a particular node, no matter what role that node currently has. The **node ID** specifies the unique name of the node on which the queue should be opened.

Note: Neither **RoleName** nor **node ID** are required if you are opening a queue on the local node. However, if you are opening a queue on a remote node, you must specify the **node ID** or a **RoleName** of the remote node. If no **node ID** or **RoleName** is specified, the IPC component attempts to open the queue on the local node only.

NotificationQHandle — input
The handle of a local queue that has been opened or created by your application. The IPC component uses this handle to write a notification to the queue if one of the following conditions occurs:

- The queue being opened in this call is deleted (closed by the owner)
- Communication fails with the remote node where the queue being opened in this call is located.

If the queue associated with the specified **NotificationQHandle** is closed or locked when a notification is generated, the notification is discarded. The failure is returned as an error the next time your application attempts to write to the queue.

Valid values for **NotificationQHandle** are:

0 (zero)
No notifications are written to the queue if one of the above conditions occurs.

QHandle
the valid queue handle of a queue, reated or opened on the local node by your application, to which notifications will be written.

timeout — input
The time, in seconds, that an application is blocked if the IPC component is unable to open the queue on the first attempt, unless a non-recoverable error is detected (for example, the adapter is closed). If the queue is on a remote node that the IPC component has not communicated with before this call, the open process might be a lengthy operation because it

requires establishing communication with the node. Upon completion of the attempt to establish communication and open the queue. The IPC component waits and retries again if the operation was not successful and the specified time out has not already elapsed,. A time out will not occur while the IPC component is attempting to establish communication and opening the queue; a timeout occurs only when the IPC component is determining whether it should wait and try again. Therefore, the specified time out and the actual elapsed time before the IPC component returns with the error -580 FDSERR_TIMEOUT could vary by as much as 2 to 3 minutes. Valid values are:

0 (zero)
The IPC component attempts to open the queue once and return the results.

Greater than zero
The IPC component blocks the caller up to the specified time or until it successfully opens the queue, whichever is less.

Less than zero
The IPC component blocks the caller until it successfully opens the queue.

QHandlePtr — input/output
A pointer to the location where the assigned queue handle is stored. The queue handle can be used by your application to access the queue.

Remarks

FdsOpenQ() opens a queue on the local or a remote node. Upon successful completion, this API returns a queue handle for your application to use to write to the specified queue. The returned queue handle can be used by any thread within your application.

The queue must be created (using FdsCreateQ()) before it can be opened.

If a RoleName is specified for the node, the IPC component locates the node ID of the node that is acting that role. The IPC component saves the information about the role used to locate the node to open the queue. Using the FdsWriteQ() API, you can request that the IPC component confirm that the node is still acting the role specified in the FdsOpenQ() request. If requested, the IPC component verifies that the node is still acting that role. If not, the IPC component does not write the message to the queue and returns an error or posts a notification to the notification queue that was specified in the FdsOpenQ() request. See “FdsWriteQ()” for more information.
If the specified queue name is a logical name, the IPC component resolves the queue name at the node where the queue is to be opened.

Error Conditions

FdsOpenQ() returns the following values:
-300 FDSERR_LOGICAL_NAME
-310 FDSERR_LOGICAL_NAME_NOT_FOUND
-340 FDSERR_NODE_NAME
-350 FDSERR_NODE_NOT_FOUND
-370 FDSERR_NOTIFY_QUEUE

-410 FDSERR_OVERFLOW
-450 FDSERR_QUEUE_NAME
-460 FDSERR_QUEUE_NOT_FOUND
-500 FDSERR_REMOTE
-540 FDSERR_ROLE_NAME
-550 FDSERR_ROLE_NOT_FOUND
-580 FDSERR_TIMEOUT

Examples

```
#include <stdio.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long        rc;                                 // Return from API Call
long        NotificationQHandle = 0;      // No Notification Requested
long        OpenQHandle;                        // Queue Handle from OpenQ
long        timeout = 60;                       // timeout FdsOpenQ Value
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //---------------------------------------------------------------
   // Call FdsOpenQ to open queue on node id NODE_A
   //---------------------------------------------------------------
   rc = FdsOpenQ( "NODE_A::MyQueue",
      NotificationQHandle,
      timeout,
      &OpenQHandle );
   printf( "FdsOpenQ completed with return code = (%d).\n", rc );
   // Work with remote queue
   // Close the Queue
   rc = FdsCloseQ( OpenQHandle );
} // end if
else
{
// else process errors
}
```

FdsPurgeMsg()

Purpose

Purge the next message in the queue.

Syntax

```
#include <fds/ipc.h>

long FdsPurgeMsg( long QHandle );
```

Parameters

QHandle — input

Specifies the queue handle associated with the queue from which a message should be removed and discarded. Only the queue handle

returned in FdsCreateQ() can be used to purge a message from the queue. Queue handles returned in FdsOpenQ() requests for a queue are not valid for this request. See FdsCreateQ() and FdsOpenQ() for more information about queue handles.

Remarks

This API removes and discards the next message in the queue associated with the specified queue handle.

Error Conditions

FdsPurgeMsg() returns the following values:

- -10 FDSERR_ACCESS
- -220 FDSERR_HANDLE
- -430 FDSERR_QUEUE_EMPTY

Examples

```
#include <stdio.h>
#include <string.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long          rc;                                    // Return from API Call
unsigned long        MaxQSize = 500;     // Maximum Queue Size
long          QueueHandle = 0;
char          ReadBuffer[100];           // Message from Read Queue
unsigned int         BufferLength = sizeof(ReadBuffer); // Length of Message
int           MsgType;
long          timeout = 60;                          // timeout Value
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   rc = FdsCreateQ( "MyQueue", MaxQSize, &QueueHandle );
   if ( rc == FDS_SUCCESS )
   {
      // Write to MyQueue
      rc = FdsWriteQ( QueueHandle,
                      BufferLength,
                      "Message to be purged",
                      FDS_WRITTEN,
                      timeout );
      // Set BufferLength to value smaller than the actual message size
      BufferLength = 10;
      // Attempt to read message on MyQueue
      rc = FdsReadQ( QueueHandle,
                     &BufferLength,
                     ReadBuffer,
                     timeout,
                     &MsgType );
      // If the Read was unsuccessful
      if ( rc != FDS_SUCCESS )
      {
         printf( "Read failed. Return code = (%d).\n", rc );
```

```
            //-------------------------------------------------------------------------------------------------
            // If error is that read buffer is too small, purge the message
            //-------------------------------------------------------------------------------------------------
            if (FDSERR_BUFFER_SIZE == rc)
            //-------------------------------------------------------------------------------------------------
            // Call FdsPurgeMsg API to delete the message from MyQueue
            //-------------------------------------------------------------------------------------------------
            rc = FdsPurgeMsg ( QueueHandle );
            printf( "FdsPurgeMsg completed with return code = (%d).\n", rc );
        } // end if
        // Else process message
        // Close MyQueue
        rc = FdsCloseQ( QueueHandle );
    } // end if
} // end if
else
{
// else process errors
}
```

FdsQueryQ()

Purpose

Query information about a local queue.

Syntax

```
#include <fds/ipc.h>

long FdsQueryQ( long QHandle, unsigned int *QMsgCountPtr,
                unsigned long *QMaxSizePtr,
                unsigned long *QBytesLeftPtr,
                unsigned int *QueryFlagPtr );
```

Parameters

QHandle — input

A queue handle associated with the queue to be queried. Only the handle returned in the FdsCreateQ() API can be used to query the queue. Queue handles returned in FdsOpenQ() requests for a queue are not valid for this request.

QMsgCountPtr — output

An input pointer to the location where the number of messages currently in the queue is written.

QMaxSizePtr — output

An input pointer to the location where the maximum queue size is written. The value returned is the number of bytes.

QBytesLeftPtr — output

An input pointer to the location where the number of bytes that can be added to the queue before the queue is full. This number is the maximum queue size minus the number of bytes currently written in the queue.

QueryFlagPtr — output

An input pointer to the location where the query status is written. The query flag contains the following attributes:

Lock status
Indicates whether or not the queue is locked. See FdsLockQ() and FdsUnlockQ() for more information.

Block status
Indicates whether or not the queue is blocked. If it is blocked, at least one message has been blocked due to insufficient space available in the queue. See FdsWriteQ() for more information.

To determine whether a particular attribute is set, perform a bitwise AND operation of the value pointed to by **QueryFlagPtr** with the attribute you are testing. If the result is non-zero, the attribute is set.

Valid values are:
FDS_QUEUE_LOCKED
The queue is locked.
FDS_QUEUE_BLOCKED
The queue is blocked.

Remarks

This API queries and returns information about the local queue associated with the specified **QHandle** parameter.

Error Conditions

FdsQueryQ() returns the following values:
-10 FDSERR_ACCESS
-220 FDSERR_HANDLE

Examples

```
#include <stdio.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>
\
long          rc;                                   // Return from API Call
long          MaxQSize = 500;                 // Maximum queue size
long          CreateQHandle = 0;                    // Queue Handle from CreateQ
unsigned int          NumMessages;                  // Number of messages in the queue
unsigned long         BytesLeft;                    // Number of bytes left in the queue
unsigned int          QueryFlag;                    // Query flag value
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   // Create MyQueue
   rc = FdsCreateQ( "MyQueue", MaxQSize, &CreateQHandle );
} // end if
if (FDS_SUCCESS == rc)
{
   //-------------------------------------------------------------------------------------------
   // Call FdsQueryQ API to get information about MyQueue
   //-------------------------------------------------------------------------------------------
```

```
   rc = FdsQueryQ( QueueHandle,
                              &NumMessages,
                              &MaxQSize,
                              &BytesLeft,
   &QueryFlag );
} // end if
if (FDS_SUCCESS == rc)
{
   printf("Number of messages in queue = (%i). \n", NumMessages);
   printf("Maximum queue size = (%l) bytes. \n", MaxQSize);
   printf("Number of bytes left in queue = (%l). \n", BytesLeft);
   if (QueryFlag & FDS_QUEUE_LOCKED)
     printf(" ----> Queue is locked.\n");
   if (QueryFlag & FDS_QUEUE_BLOCKED)
     printf(" ----> Queue is blocked.\n");
} // end if
else
  printf(" Query Queue failed. Return Code = (%l).\n", rc);
```

FdsReadQ()

Purpose

Read the next message from a queue.

Syntax

```
#include <fds/ipc.h>

long FdsReadQ( long QHandle, unsigned int *BuffSizePtr, void *BuffPtr, long
                        timeout, int *MsgTypePtr)
```

Parameters

QHandle — input

Specifies a queue handle associated with a queue from which a message is to be read. Only the queue handle returned in the FdsCreateQ() API can be used to read from the queue. Queue handles returned in FdsOpenQ() requests for a queue are not valid for this request.

BuffSizePtr — input/output

Input When this API is called, this value must specify the length of memory pointed to by **BuffPtr**.

Output

When this API has completed successfully, the value pointed to by **BuffSizePtr** is replaced by the size of the message that was copied to the input buffer pointed to by the **BuffPtr** parameter.

If the read fails with the error -40 FDSERR_BUFFER_SIZE, the value pointed to by **BuffSizePtr** is replaced with the size of the next message in the queue.

If the read fails with any other error, the value pointed to

by **BuffSizePtr** is not modified.

BuffPtr — input
Specifies a pointer to a buffer where the message read from the queue is placed.

timeout — input
Indicates whether and for how long your application is suspended to wait for a message if there are no messages in the queue. Valid values are:

0 (zero)
If no messages are in the queue, FdsReadQ() returns immediately with the error -430 FDSERR_QUEUE_EMPTY.

Less than zero
If no messages are in the queue, FdsReadQ() suspends your thread until a message is in the queue or until the queue is closed (from another thread in your application).

Greater than zero
The maximum time in seconds that FdsReadQ() suspends your thread before returning the error -580 FDSERR_TIMEOUT if no messages are written to the queue.

MsgTypePtr — **output** Specifies an input pointer to an integer where the *MsgType* value of the message that was read is stored. The value of *MsgType* identifies whether the message was written to the queue by an application or by a DDS component. Possible values are:

FDS_IPC_MSG
If the queue being read from was designated as a notification queue when another queue was opened, the IPC component writes notifications to this queue if communication with the other queue fails. The format of the data copied to the memory pointed to by **BuffPtr** is defined by the FDS_IPC_MSG_STRUCT data structure.

FDS_DIST_SYNC_NOTIFY_MSG
This value identifies a message used by the Data Distribution component for file and directory synchronization notification. This message is generated as the result of a previous call to the FdsSetupSyncIDNotify() API. The format of the data copied to the memory pointed to by **BuffPtr** is defined by the FDS_SYNC_ID data structure.

FDS_DIST_STATE_NOTIFY_MSG
This value identifies a message used by the Data Distribution component for state change notifications. This message is generated as the result of a previous call to the FdsSetupDistMonitor() API. The format of the data copied to the memory pointed to by **BuffPtr** is defined by the FDS_DIST_STATE data structure.

FDS_APPL_MSG
If the source of the message is an application other than the IPC component, the **MsgType** parameter is set to FDS_APPL_MSG.

The format of the data copied to the memory pointed to by **BuffPtr** is defined by the application.

Remarks

FdsReadQ() reads the next message from the queue associated with the input queue handle, and copies the message into the buffer pointed to by the **BuffPtr** parameter.

Upon successful completion, the parameter pointed to by **BuffSizePtr** is replaced with the actual number of bytes placed in the buffer.

If the size of the message in the queue exceeds the length of the input buffer (specified by the value pointed to by the **BuffSizePtr** parameter), the error -40 FDSERR_BUFFER_SIZE is returned and the parameter pointed to by **BuffSizePtr** is replaced with the actual size of the next message. You can call another FdsReadQ() API with a larger buffer to read the message or you can purge the message from the queue using the FdsPurgeMsg() API.

Error Conditions

FdsReadQ() returns the following values:

-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-220 FDSERR_HANDLE
-420 FDSERR_QUEUE_CLOSED
-430 FDSERR_QUEUE_EMPTY
-580 FDSERR_TIMEOUT

Examples

```
#include <stdio.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long          rc;                                   // Return from API Call
unsigned long       MaxQSize = 500;     // Maximum Queue Size
long          CreateQHandle = 0;                    // Queue Handle from CreateQ
long          timeout = 60;                         // timeout Value
char          ReadBuffer[100];              // Message from Read Queue
unsigned int        BufferLen = sizeof(ReadBuffer);  // Length of Message Read
int           MsgType;                              // Type of Message Read
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   // Create MyQueue
   rc = FdsCreateQ( "MyQueue", MaxQSize, &CreateQHandle );
} // end if
// If create MyQueue was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------
   // Write to MyQueue
```

```
   //-------------------------------------------------------------
   rc = FdsWriteQ( CreateQHandle,
                   BufferLen,
                   "Read Message Data",
                   FDS_WRITTEN,
                   timeout );
   //-------------------------------------------------------------
   // Call FdsReadQ API to read message in "MyQueue"
   //-------------------------------------------------------------
   rc = FdsReadQ( CreateQHandle,
                  &BufferLen,
                  ReadBuffer,
                  timeout,
                  &MsgType );
   printf( "FdsReadQ completed with return code = (%d).\n", rc );
   //-------------------------------------------------------------
   // Process Message in Read Buffer
   // CloseQ
   //-------------------------------------------------------------
   rc = FdsCloseQ( CreateQHandle );
} // end if
else
{
   // else process errors
}
```

FdsUnlockQ()

Purpose

Unlock a locked queue. When a queue is unlocked, applications can resume writing messages to the queue.

Syntax

```
#include <fds/ipc.h>

long FdsUnlockQ( long QHandle );
```

Parameters

QHandle — input

Specifies the queue handle associated with the queue to be unlocked. Only the handle returned in the FdsCreateQ() API can be used to unlock the queue. Queue handles returned in FdsOpenQ() requests for a queue are not valid for this request.

Remarks

This API unlocks a locked queue to allow applications to resume writing messages to the queue.

If the queue is not locked at the time of this request, the queue remains unlocked and the API completes without an error.

Error Conditions

FdsUnlockQ() returns the following values:
-10 FDSERR_ACCESS
-220 FDSERR_HANDLE

Examples

```
#include <stdio.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long          rc;                                    // Return from API Call
unsigned long          MaxQSize = 500;     // Maximum Queue Size
long          CreateQHandle = 0;                     // For Storing Queue Handle
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   //-------------------------------------------------------------
   // Create MyQueue
   //-------------------------------------------------------------
   rc = FdsCreateQ( "MyQueue", MaxQSize, &CreateQHandle );
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------
      // Lock MyQueue
      //-------------------------------------------------------------
      rc = FdsLockQ( CreateQHandle );
   }
   //-------------------------------------------------------------
   // Work with locked queue
   //-------------------------------------------------------------
   if ( rc == FDS_SUCCESS )
   {
      //-------------------------------------------------------------
      // Call UnlockQ API to unlock the locked queue
      //-------------------------------------------------------------
      rc = FdsUnlockQ( CreateQHandle );
      printf( "FdsUnLockQ completed with return code = (%d).\n", rc );
   }
   //-------------------------------------------------------------
   // Close the Queue
   //-------------------------------------------------------------
   rc = FdsCloseQ( CreateQHandle );
} // end if
else
{
  // else process errors
}
```

FdsWriteQ()

Purpose

Write a message to the queue associated with the input **QHandle.**

Syntax

```
#include <fds/ipc.h>

long FdsWriteQ( long QHandle, unsigned int BuffSize,
                const void *BuffPtr, int WriteFlag,
                long timeout );
```

Parameters

QHandle — input

The handle of the queue associated with the queue to which the message is to be added. The **QHandle** must be one that was returned to your application in either FdsCreateQ() or FdsOpenQ().

BuffSize — input

The length, in bytes, of the message to be written to the queue.

The data pointed to by **BuffPtr** does not need to be null-terminated. If it is, the value for **BuffSize** must include 1 byte if the null terminator is to be copied with the message data. The maximum supported message size is the smaller of the following sizes:

- 60,000 bytes (the maximum message size supported by the IPC component)
- The maximum queue size of the queue to be written to (see FdsCreateQ() for more information)

A **BuffSize** value of 0 (zero) is not valid and results in the error -20 FDSERR_ADDRESS.

BuffPtr — input

A pointer to the buffer that contains the message to be written to the queue.

WriteFlag — input

A flag that contains one or more of the following write attributes:

RoleConfirm

Whether the role should be confirmed. The default is **FDS_NO_CONFIRM_ROLE**. Valid values are:

FDS_CONFIRM_ROLE

Set this flag if you specified a **RoleName** instead of a node ID in the FdsOpenQ() request, and you want the IPC component to confirm that the node on which the queue is opened is still acting that role. If you request this confirmation and the IPC component detects that the node is not acting the specified role, the message is discarded.

This flag can be used in combination with the **FDS_WRITTEN** flag only.

FDS_NO_CONFIRM_ROLE

The IPC component will not confirm the role on the destination node.

Note: If you specified a queue name only or if you specified a node ID instead of a **RoleName** in the

FdsOpenQ() request, this attribute is ignored.

WaitConfirm

The level of confirmation to be completed by the IPC component. The calling thread is suspended until the specified level of confirmation has been completed. The default is **FDS_REQUEST_COPIED**. Valid values are:

FDS_WRITTEN

The IPC component returns without an error after receiving
confirmation from the destination node that the message has been successfully written to the queue.

When this level of confirmation is requested and there is not enough room in the queue for the message, the queue becomes blocked. While a queue is blocked, all other messages written to the queue with this level of confirmation requested are blocked behind this request. All messages broadcast to this queue or written with any other level of confirmation are discarded.

As space become available in the queue, blocked messages are retrieved to be written in the same order that the messages were received and blocked.

If the caller specified a timeout other than 0 and the queue is blocked, the caller's thread remains suspended until space becomes available and the message has been written successfully to the queue or until an error is detected.

The IPC component waits for the confirmation until one of the following conditions occurs, at which time an error is returned:

- The API has timed out (see the timeout parameter)
- The destination queue is locked
- The IPC component detected that it can no longer communicate with the destination node
- The RoleConfirm attribute in the WriteFlag parameter was set to FDS_CONFIRM_ROLE and the destination node is no longer acting the role specified in the FdsOpenQ() API

If this flag is specified when writing to a remote destination node, the minimum timeout value used is the value specified for the configuration parameter **IPCtimeout**. Any timeout value that is less than the value specified for **IPCtimeout** (including 0) will be ignored and the **IPCtimeout** value will be used.

A minimum timeout value is required to allow the

FDS_WRITTEN confirmation to be returned to the sender. Your testing could determine that a longer timeout value is needed for high-volume systems.

If your application cannot wait the timeout value specified by **IPCtimeout** for a confirmation, it should not use the **FDS_WRITTEN** value.

FDS_REQUEST_COPIED

The IPC component returns without an error after the input request data has been validated and the request has been copied.

Note: The IPC component does not wait to receive a confirmation from the destination node confirming that the message was delivered and written to the destination queue. Therefore, this API might be completed without an error, but the message will be discarded if the IPC component detects any of the following conditions:

- The IPC component has lost communication with the destination node
- The destination queue is locked or blocked
- There is not enough room in the queue for the message
- The RoleConfirm attribute in the WriteFlag parameter was set to FDS_CONFIRM_ROLE and the destination node is no longer acting the role specified in the FdsOpenQ() API

timeout — input

Whether and for how long your application is suspended to wait for the specified level of confirmation to be completed. If the requested confirmation level was **FDS_REQUEST_COPIED**, this parameter does not apply and is ignored. Valid values are:

0 (zero)

FdsWriteQ() attempts to complete the write request once. If the **WaitConfirm** attribute in the **WriteFlag** parameter is set to **FDS_DELIVERED** and the IPC component fails to deliver the message, an error is returned and no retries are performed. If the **WaitConfirm** attribute in the **WriteFlag** parameter is set to **FDS_WRITTEN** and the IPC component fails to deliver the message or receive confirmation that the message was successfully written to the queue, an error is returned and no retries are performed. The IPC component does not suspend the caller's thread if there is not enough room in the queue for the message or the queue is already blocked.

Less than zero

FdsWriteQ() suspends the caller's thread indefinitely until the confirmation requested by the **WaitConfirm** attribute in the **WriteFlag** parameter is received or until an unrecoverable error is detected (for example, communication with the destination

node has failed). If there is not enough room in the queue or if the queue is already blocked, the caller's thread is suspended until there is room available in the queue only if the **WaitConfirm** attribute in the **WriteFlag** parameter value is **FDS_WRITTEN**.

Greater than zero

The time, in seconds, that the calling thread remains suspended while waiting for the correct level of confirmation to be completed. If the request could not be completed in the specified period of time, an error is returned.

Remarks

FdsWriteQ() writes a message to the queue associated with the input **QHandle** parameter. The specified **QHandle** parameter must be one that was returned in either FdsOpenQ() or FdsCreateQ().

Error Conditions

FdsWriteQ() returns the following values:
-10 FDSERR_ACCESS
-20 FDSERR_ADDRESS
-40 FDSERR_BUFFER_SIZE
-210 FDSERR_FLAG
-220 FDSERR_HANDLE
-325 FDSERR_MEMORY_CONSTRAINED
-330 FDSERR_MESSAGE_SIZE
-350 FDSERR_NODE_NOT_FOUND
-420 FDSERR_QUEUE_CLOSED
-440 FDSERR_QUEUE_FULL
-500 FDSERR_REMOTE
-530 FDSERR_ROLE_CHANGE
-580 FDSERR_TIMEOUT

Examples

```
#include <stdio.h>
#include <string.h>
#include <fds/ipc.h>
#include <fds/fds.h>
#include <fds/errno.h>

long          rc;                                   // Return from API Call
long          OpenQHandle = 0;              // Queue Handle from OpenQ
long          NotifyQHandle = 0;            // No notification requested
long          timeout = 60;                         // timeout Value
char          WriteBuffer[100];             // Message from Write Queue
unsigned int          BufferLen = sizeof(WriteBuffer);   // Length of Message Write
int           MsgType;                              // Type of Message Write
// Initialize DDS - could use FdsInit2()
rc = FdsInit();
// If initialization was successful
if ( rc == FDS_SUCCESS )
{
   rc = FdsOpenQ( "NODE_A::MyQueue",
                  NotifyQHandle,
                  timeout,
                  &OpenQHandle );
} // end if
if ( rc == FDS_SUCCESS )
```

```
{
   strcpy(WriteBuffer, "Write to Queue");
   //-------------------------------------------------------------------------------------------
   // Call FdsWriteQ API to write a message to the queue
   //-------------------------------------------------------------------------------------------
   rc = FdsWriteQ( OpenQHandle,
   BufferLen,
   "WriteBuffer",
   FDS_WRITTEN,
   timeout );
   printf( "FdsWriteQ completed with return code = (%d).\n", rc );
   //-----------------------------------------------------------
   // process message
   //-----------------------------------------------------------
   //-----------------------------------------------------------
   // CloseQ
   //-----------------------------------------------------------
   rc = FdsCloseQ(OpenQHandle);
   //-----------------------------------------------------------
   // CloseQ
   //-----------------------------------------------------------
   rc = FdsCloseQ(CreateQHandle);
} // end if
else
{
   // else process errors
}
```

Appendix A. Data Types

This section contains descriptions of the data types defined by the DDS APIs.

Note: DDS makes use of several program constants, which are used by the data types described in this section. These constants are defined in the C language header files provided with DDS. See “C Language Header Files” for more information about the header files.

FDS_NODE_NAME

Node ID.

```
typedef char FDS_NODE_NAME [FDS_MAX_NODE_NAME_LEN]
```

FDS_NODE_INFO

Node ID and communication status data structure.

```
typedef struct
    {
        FDS_NODE_NAME   NodeID;
        short                       NodeStatus;
    } FDS_NODE_INFO;
```

NodeID

Node ID

NodeStatus

Communications status with the acting primary distributor:

FDS_ACTIVE

The node is communicating with the acting primary distributor.

FDS_INACTIVE

The node is not communicating with the acting primary distributor.

FDS_NODE_STATE

Node ID and distribution state data structure.

```
typedef          struct
    {
        FDS_NODE_NAME      Name;
        int            State;
    } FDS_NODE_STATE ;
```

Name Node ID

State Reserved.

FDS_ROLE_NAME

Role name.

```
typedef    char      FDS_ROLE_NAME [FDS_MAX_ROLE_NAME_LEN]
```

FDS_DOMAIN_NAME

Domain name.

```
typedef    char      FDS_DOMAIN_NAME [FDS_MAX_DOMAIN_NAME_LEN]
```

FDS_SYNC_ID

Data Distribution synchronization ID.

```
typedef struct

  {
      unsigned long        ObjectHandle ;
      long                 ObjectCreationTime ;
      unsigned long        SequenceNumber ;
      long                 SequenceTimeStamp ;

  }  FDS+SYNC_ID;
```

FDS_DDS_BLOCKED_INTERFACE;

Interfaces that DDS will be blocked from using.

```
typedef struct
{
char Address[FDS_MAX_TCPIP_ADDR_LEN+1];
} FDS_DDS_BLOCKED_INTERFACE;
```

FDS_CFG

Installation and configuration data structure.

```
typedef struct
    {
        short              Adapter0Sessions;
        short              Adapter1Sessions;
        short              Adapter2Sessions;
        short              Adapter3Sessions;
        unsigned short     AdptrNumNames;
        unsigned short     AdptrResetValue;
        unsigned short     DDActive;
        unsigned short     DistributionRole;
        unsigned short     IPCTimeout;
        unsigned short     LocatePrimary;
        unsigned long      MaximumMemory;
        unsigned short     MaxRequestors;
```

```
    FDS_NODE_NAME       NodeID;
    short               NVRAMApplLine;
    char                ProductLevel[9];
    unsigned short      RemoteIPC;
    char                SystemID[5];
    char                WorkDirectory[FDS_MAX_WORK_DIR_LENGTH];
    char          ControlledDrives[FDS_MAX_CONTROLLED_DRIVES_SIZE];
    char          FDSInstallDirectory[FDS_MAX_PATH_LENGTH];
    unsigned short      IPCTransport;
    unsigned short      IPCPortStart;
    unsigned short      IPCPortCount;
    unsigned short      IPCHeartbeatInterval;
    unsigned short      NetworkRequestInterval;
    unsigned short      NetworkRequestRetries;
    FDS_DDS_BLOCKED_INTERFACE
                  DDSBlockedInterface[FDS_MAX_BLOCKED_INTERFACES];
    unsigned short      DistRenamedFile;
} FDS_CFG;
```

Adapter0Sessions
Number of NetBIOS sessions for LAN adapter 0.
Adapter1Sessions
Number of NetBIOS sessions for LAN adapter 1.

Adapter2Sessions
Number of NetBIOS sessions for LAN adapter 2.
Adapter3Sessions
Number of NetBIOS sessions for LAN adapter 3.
AdptrNumNames
Number of NetBIOS names used on each adapter.
AdptrResetValue
Number of seconds between adapter resets.
DDActive
FDS_CONFIG_YES
Data Distribution is configured on the node.
FDS_CONFIG_NO
Data Distribution is not configured on the node.
DistributionRole
FDS_CONFIG_NONE
Data Distribution is not configured on the node.
FDS_CONFIG_PRIMARY_DIST
The node is the configured primary distributor.
FDS_CONFIG_BACKUP_DIST
The node is the configured backup distributor.
FDS_CONFIG_SUBORDINATE
The node is a subordinate.
IPCTimeout
IPC time out, in seconds.

LocatePrimary

FDS_CONFIG_YES
Data Distribution is configured on the node.

FDS_CONFIG_NO
Data Distribution is not configured on the node.

DistributionRole

FDS_CONFIG_NONE
Data Distribution is not configured on the node.

FDS_CONFIG_PRIMARY_DIST
The node is the configured primary distributor.

FDS_CONFIG_BACKUP_DIST
The node is the configured backup distributor.

FDS_CONFIG_SUBORDINATE
The node is a subordinate.

IPCTimeout
IPC time out, in seconds.

LocatePrimary

FDS_CONFIG_YES
A primary distributor is present in the system.

FDS_CONFIG_NO
A primary distributor is not present in the system.

MaximumMemory
Amount of shared memory used, in kilobytes.

MaxRequestors
Number of file clients supported by this node.

NodeID
Node ID.

NVRAMApplLine
Amount of NVRAM, in kilobytes, reserved for application use.

ProductLevel
Encoding of product version, release, and modification level.

RemoteIPC

FDS_CONFIG_YES
Remote IPC is configured.

FDS_CONFIG_NO
Remote IPC is not configured.

SystemID
System ID.

WorkDirectory
Null-terminated (\0) path specification of work directory. The path must include the final directory separator (\).

ControlledDrives
Null-terminated (\0) string of controlled drive letters.

FDSInstallDirectory
Null-terminated (\0) path specification of install directory. The path must include the final directory separator (\).

IPCTransport

FDS_CONFIG_NETBIOS
IPC uses NetBIOS as the transport layer protocol.

FDS_CONFIG_TCPIP
IPC uses TCP/IP as the transport layer protocol.

IPCPortStart
First TCP/UDP port number used by IPC.

IPCPortCount
Number of TCP/UDP ports used by IPC.

IPCHeartbeatInterval
Time interval between IPC checks for a communication connection, in seconds.

NetworkRequestInterval
Time interval between IPC queries of the LAN for the TCP/IP port associated with a broadcast domain, in milliseconds.

NetworkRequestRetries
Maximum number of times IPC queries the LAN for the TCP/IP port associated with a broadcast domain.

DDSBlockedInterface
Maximum number of interfaces that DDS can be blocked from using.

DistRenamedFile
Whether distributed files (that are not part of a distributed subdirectory) remain distributed if they are renamed.

AutoSwitchOver

FDS_CONFIG_YES
Automatic Switch-Over is configured on this node.

FDS_CONFIG_NO
Automatic Switch-Over is not configured on this node.

AutoSwitchOverDelay
Time in minutes to wait before automatically activating the acting backup as the primary distributor.

AutoSwitchOverForce

FDS_CONFIG_YES
Force automatically activating the acting backup as the primary distributor.

FDS_CONFIG_NO
Do not force automatically activating the acting backup as the primary distributor if it is not fully reconciled.

PrimaryIPAdapter
The network adapter number specified using the PrimaryIPAddress keyword that will be added to during activation of the primary distributor.

PrimaryIPAddress
The IP address to be added to the node that is being activated
PrimarySubnetMask

PrimaryIPAddress
The subnet mask used with the PrimaryIPAddress.

PrimaryComputerName

The computer name (or NetBIOS name) to add to the node being activated as the primary distributor.

FDS_IPC_MSG_STRUCT

IPC notification data structure.

```
typedef struct
    {
        long ClosedQHndl ;
        long ReasonCode ;


    } FDS_IPC_MSG_STRUCT;
```

ClosedQHndl

IPC queue to which this notification refers.

ReasonCode

The reason the IPC queue was closed. Valid values are:

FDS_IPC_MSG_REASONCODE_COMM_FAILED

IPC remote communication has failed.

FDS_IPC_MSG_REASONCODE_Q_CLOSED

The remote IPC queue has been closed by the application.

FDS_DIST_STATE

Node ID and role-state data structure.

```
typedef struct
    {
    long            Reserved[3]
    long            RoleState;
    FDS_NODE_NAME       NodeID;


} FDS_DIST_STATE
```

Reserved

Reserved.

RoleState

Distributor state.

FDS_ACTING_PRIMARY

Current role is acting primary.

FDS_TRANS_TO_ACTING_PRIMARY

In transition to the acting primary role.

FDS_ACTING_BACKUP

Current role is acting backup.

FDS_TRANS_TO_ACTING_BACKUP

In transition to the acting backup role.

NodeID

Node ID.

FDS_DATE_TIME

Date and time data structure.

```
typedef struct
        unsigned short      Year ;
        unsigned char       Month ;
        unsigned char       Day ;
        unsigned char       Hour ;
        unsigned char       Minute ;
        unsigned char       Second ;
 } FDS_DATE_TIME
```

Appendix B. Error Codes

DDS return codes are 4-byte signed integers. A 0 (zero) return code indicates a successful function call. A negative return code indicates an unsuccessful function call.

This section contains a list of error codes in alphanumeric order.

-10 FDSERR_ACCESS

Explanation: A locking or sharing conflict occurred. Specific conditions for this error include:

Note: Some of the descriptions indicate that this error can be caused by a file being open. In such cases, the file may be opened by an application or by the Data Distribution component. See the note under “Reconciliation” for information about Data Distribution’s use of distributed files.

- The file is open or the file is read-only. These APIs are associated with this error:
  - “FdsDeleteFile()”
  - “FdsRenameFile()”
  - “FdsSetFileAttributes()”
- Another process has locked the physical drive, the file, or some portion of the file and the requested access would conflict with this lock, or another process has access to the file and the requested lock would conflict with this access; or, the file is considered read-only at the operating system level and **FDS_FILE_ACCESS_READ_WRITE** was specified.
  These APIs are associated with this error:
  – “FdsCreateKeyedFile()”
  – “FdsOpenBinFile()”
  – “FdsOpenKeyedFile()”
  – “FdsOpenSeqFile()”
- The file is locked. These APIs are associated with this error:
  – “FdsFindNextSeqRecord()”
  – “FdsFlushBinFile()”
  – “FdsReadBinFile()”
  – “FdsReadKeyedRecord()”
  – “FdsReadSeqRecord()”
  – “FdsReleaseKeyedRecord()”
  – “FdsSeekBinFilePos()”
  – “FdsSetBinFileLocks()”
  – “FdsSetBinFileSize()”
  – “FdsWriteBinFile()”

- The file is locked or **FDS_FILE_ACCESS_READ_WRITE** was not specified when the file was opened. These APIs are associated with this error:
  – “FdsDeleteKeyedRecord()”
  – “FdsWriteSeqRecord()”
- “FdsWriteKeyedRecord()” was called and the file is locked, **FDS_FILE_ACCESS_READ_WRITE** was not specified when the file was opened, or **FDS_FILE_RECORD_UNLOCK_NO** was specified and the record is locked.
- “FdsSetDistribution()” was called and the file or subdirectory is open.
- “FdsDeleteBcastDomain()” was called and a file distributed to the broadcast domain is open.
- “FdsWriteQ()” was called; **FDS_WRITTEN** was specified and the queue is locked.
- A write-only queue handle (returned by “FdsOpenQ()”) was specified for an operation that requires a read-write queue handle (returned by “FdsCreateQ()”).
- “FdsCreateDir()” was called and the directory already exists.
- “FdsRemoveDir()” was called and the directory is currently being used by another process or the directory is not empty.
- “FdsRestrictFile()” was called and an attempt was made to restrict a file that has already been restricted.
- “FdsUnrestrictFile()” was called and an attempt was made to remove restrictions from a file that is not restricted.
- “FdsQueryFileSystemInfo()” was called and **FileSystemID** refers to a locked drive specification.

-20 FDSERR_ADDRESS

Explanation: Either a pointer that is not valid was specified, indicating that the process does not have access to the full length of the buffer, or an input buffer size of 0 (zero) was specified.

These APIs are associated with this error:

- “FdsQueryConfig()”
- “FdsReadKeyedRecord()”
- “FdsReleaseKeyedRecord()”
- “FdsWriteKeyedRecord()”
- “FdsWriteSeqRecord()”
- “FdsReadSeqRecord()”
- “FdsReadBinFile()”
- “FdsWriteBinFile()”
- “FdsGetFileNames()”
- “FdsGetNodes()”
- “FdsCreateBcastDomain()”
- “FdsGetDomainList()”
- “FdsGetDomainNodes()”
- “FdsBroadcastQ()”
- “FdsReadQ()”
- “FdsWriteQ()”

-25 FDSERR_APPL_DOWN

Explanation: The DDS exit processing for this application has been run because the operating system indicated that the application has ended. The application can no longer make API calls to DDS. It must restart and reinitialize DDS before using the DDS API. This error occurs if the application's exit process or signal handling occurs after DDS exit processing.

-30 FDSERR_BLOCK_SIZE

Explanation: A block size that is not valid was specified to "FdsCreateKeyedFile()".

-40 FDSERR_BUFFER_SIZE

Explanation: The input buffer was too large to be contained within an internal buffer, or the input buffer was not large enough to contain all of the output data. No output data is returned. The size of the output data is returned.

Specific conditions for this error include:

-50 FDSERR_CHAIN_THRESH

Explanation: A chain threshold that is not valid was specified to "FdsCreateKeyedFile()".

-60 FDSERR_CONFIG

Explanation: The component required to support the API that returned this error is not installed or is not configured. These APIs are associated with this error:

- "FdsActivateAsPrimary()"
- "FdsAddDomainNode()"
- "FdsCreateBcastDomain()"
- "FdsCreateSyncID()"
- "FdsDeactivatePrimary()"
- "FdsDeleteBcastDomain()"
- "FdsDeleteDomainNode()"
- "FdsGetDomainList()"
- "FdsGetDomainNodes()"
- "FdsQueryBackupState()"
- "FdsQueryDistribution()"
- "FdsSetDistribution()"
- "FdsSetupDistMonitor()"
- "FdsSetupSyncIDNotify()"

-70 FDSERR_CORRUPT

Explanation: Damaged data was detected or created. Specific conditions for this error include:

- "FdsOpenSeqFile()" was called and the file is not the correct format for a sequential file.
- "FdsReadSeqRecord()" was called and the next record is not the correct format for a sequential record.
- "FdsWriteSeqRecord()" was called and a partial record was written to disk.
- "FdsOpenKeyedFile()" was called and the file is not the correct format for a keyed file.
- Damaged data was detected in the file.
  - "FdsDeleteKeyedRecord()"
  - "FdsReadKeyedRecord()"
  - "FdsReleaseKeyedRecord()"
  - "FdsWriteKeyedRecord()"
- "FdsQueryDistribution()" was called and damaged data was detected in the distribution directory. Recovery from this error requires that a new distribution directory be created. Follow these steps if the damaged distribution directory is on the acting backup distributor or a subordinate node.
  DDS should be started on the acting primary distributor:

  1. Stop DDS if it is running on the node on which the error exists.
  2. Erase all of the distribution directory files from the node where the error exists.
  3. Start DDS on the node where the error exists. The DDS reconciler will copy all of the distributed files from the acting primary distributor and rebuild the distribution directory.

  If this error occurs on the acting primary distributor, the acting primary distributor must be deactivated and stopped. Next, the acting backup distributor must be activated as the acting primary distributor. The steps listed above can then be followed to recover the damaged distribution directory.

-75 FDSERR_DATE_TIME

Explanation: A date or time that is not valid was specified to "FdsSetFileAttributes()".

-80 FDSERR_DIR_INDICATOR

Explanation: A directory indicator that is not valid was specified to "FdsSetDistribution()".

-90 FDSERR_DISK

Explanation: An error occurred while writing the distribution information manager to disk. These APIs are associated with this error:

- "FdsDeleteBcastDomain()"
- "FdsDeleteDomainNode()"
- "FdsQueryFileSystemInfo()"
- "FdsSetDistribution()"

-100 FDSERR_DISK_FULL

Explanation: The disk is full. These APIs are associated with this error:

- "FdsWriteSeqRecord()"
- "FdsCreateKeyedFile()"
- "FdsWriteBinFile()"
- "FdsSetBinFileSize()"

-110 FDSERR_DIST_FREQ

Explanation: A distribution frequency that is not valid was specified to "FdsSetDistribution()".

-120 FDSERR_DOMAIN_NAME

Explanation: A domain name that is not valid was specified. These APIs are associated with this error:

- "FdsAddDomainNode()"
- "FdsBroadcastQ()"
- "FdsCreateBcastDomain()"
- "FdsDeleteBcastDomain()"
- FdsDeleteDomainNode()"
- "FdsGetDomainNodes()"
- "FdsSetDistribution()"

-130 FDSERR_DOMAIN_NOT_FOUND

Explanation: The domain does not exist. These APIs are associated with this error:

- "FdsAddDomainNode()"
- "FdsDeleteBcastDomain()"
- The handle is not a valid queue handle. These APIs are associated with this error:
  –"FdsCloseQ()"

–“FdsLockQ()”
-“FdsPurgeMsg()”
–“FdsQueryQ()”
–“FdsReadQ()”
–“FdsUnlockQ()”
–“FdsWriteQ()”

- The handle is not a valid, binary-file handle. These APIs are associated with this error:
  –“FdsCloseBinFile()”
  –“FdsFlushBinFile()”
  –“FdsSetBinFileSize()”
  –“FdsQueryBinFileSize()”
  –“FdsReadBinFile()”
  –“FdsSeekBinFilePos()”
  –“FdsSetBinFileLocks()”
  –“FdsWriteBinFile()”

-140 FDSERR_DOMAIN_TYPE

Explanation: A domain type that is not valid was specified to “FdsSetDistribution()”.

-150 FDSERR_DOWN

Explanation: DDS was not started, is shutting down, or has shut down. If your application had already successfully completed either an FdsInit() or FdsInit2() call, the application must be shut down and restarted in order to successfully reinitialize.

-160 FDSERR_EOF

Explanation: The end of the file has been reached. Specific conditions for this error include:

- There are no more valid records in the file. These APIs are associated with this error:
  – “FdsFindNextSeqRecord()”
  – “FdsReadSeqRecord()”
- “FdsReadBinFile()” was called and the value specified by *NBytesPtr* is greater than the number of bytes read.

-170 FDSERR_EXISTS

Explanation: An object exists. Specific conditions for this error include:

- “FdsActivateAsPrimary()” was called and another node is currently the acting primary distributor.
- “FdsRenameFile()” was called and the target file exists.
- “FdsCreateKeyedFile()” was called; **FDS_FILE_EXIST_FAIL** as specified and the file exists.
- “FdsSetDistribution()” was called and changing the

broadcast domain name for a distributed file or subdirectory is not allowed.

- "FdsCreateBcastDomain()" was called and a broadcast domain already exists, a node ID was specified more than once, or this node ID was assigned to more domains than are currently supported by DDS.
- "FdsAddDomainNode()" was called and the node is already a member of the domain.
- "FdsCreateLogicNm()" was called and the logical name already exists.
- "FdsSetResetRole()" was called and the role is already active on the node.
- "FdsCreateQ()" was called and the queue already exists.
- "FdsInit()" or "FdsInit2()" was called and a resource that was needed could not be obtained. See the Event Viewer for detail

-180 FDSERR_FILE_FULL

Explanation: The keyed file is full. No additional records can be added to the file. Space will become available as existing records are deleted. To increase the capacity of the file, it must be rebuilt specifying a larger file size (that is, a larger number of blocks, a larger block size, or both). This error is associated with "FdsWriteKeyedRecord()".

-190 FDSERR_FILE_NAME

Explanation: The file or path name is not valid. These APIs are associated with this error:

- "FdsCreateDir()"
- "FdsCreateKeyedFile()"
- "FdsExistFile()"
- "FdsGetFileAttributes()"
- "FdsGetFileNames()"
- "FdsOpenBinFile()"
- "FdsOpenSeqFile()"
- "FdsOpenKeyedFile()"
- "FdsQueryFileSystemInfo()"
- "FdsQueryDistribution()"
- "FdsRemoveDir()"
- "FdsRenameFile()"
- "FdsRestrictFile()"
- "FdsSetFileAttributes()"
- "FdsUnrestrictFile()"
- "FdsSetDistribution()"
- "FdsSetDistribution()" was called and:
  – The drive specified by the path is not valid.
  – The file or path name refers to a file or subdirectory that is not on a controlled drive and therefore cannot be distributed.

-200 FDSERR_FILE_NOT_FOUND

Explanation: The file, path, or directory does not exist. If you are using the Name Services component, verify that the resolved name is correct. Specific conditions for this error include:

- The file or path does not exist on disk. These APIs are associated with this error:
  – "FdsDeleteFile()"
  – "FdsExistFile()"
  – "FdsGetFileAttributes()"
  – "FdsGetFileNames()"
  – "FdsOpenBinFile()"
  – "FdsOpenKeyedFile()"
  – "FdsSetFileAttributes()"
- "FdsRenameFile()" was called and the source file, source path, or the target path does not exist on disk.
- The path does not exist on disk. These APIs are associated with this error:

– "FdsCreateKeyedFile()"
– "FdsOpenSeqFile()"

- "FdsSetDistribution()" was called, and either the file or directory does not exist on the disk, or the root directory has been specified.
- "FdsQueryDistribution()" was called and the file or directory was not found in the distribution directory. It is possible that the file or directory is not distributed or that it is distributed but only as a result of being in a distributed directory.
- "FdsInit()" or "FdsInit2()" was called and a resource that was needed could not be obtained. See the Event Viewer for details.
- Files are contained in the directory; the directory is not empty. These APIs are associated with this error:
  – "FdsCreateDir()"
  – "FdsRemoveDir()"

-210 FDSERR_FLAG

Explanation: A flag that is not valid was specified. These APIs are associated with this error:

- "FdsActivateAsPrimary()"
- "FdsCloseKeyedFile()"
- "FdsCreateKeyedFile()"
- "FdsInit2()"
- "FdsOpenBinFile()"
- "FdsOpenKeyedFile()"
- "FdsOpenSeqFile()"
- "FdsReadKeyedRecord()"
- "FdsSetBinFileLocks()"
- "FdsSetFileAttributes()"
- "FdsSetResetRole()"
- "FdsWriteKeyedRecord()"
- "FdsWriteQ()"

-220 FDSERR_HANDLE

Explanation: A handle that is not valid was specified. Specific conditions for this error include:

- The handle is not a valid, sequential-file handle. These APIs are associated with this error:
  – "FdsCloseSeqFile()"
  – "FdsFindNextSeqRecord()"
  – "FdsReadSeqRecord()"
  – "FdsReturnSeqFilePos()"
  – "FdsSeekSeqFilePos()"
  – "FdsWriteSeqRecord()"
- The handle is not a valid, keyed-file handle. These APIs are associated with this error:
  – "FdsCloseKeyedFile()"
  – "FdsDeleteKeyedRecord()"
  – "FdsReadKeyedRecord()"
  – "FdsReleaseKeyedRecord()"
  – "FdsWriteKeyedRecord()"
- "FdsCreateSyncID()" was called and the handle is not a valid sequential- or keyed-file handle.
- The handle is not a valid queue handle. These APIs are associated with this error:

– "FdsCloseQ()"
– "FdsLockQ()"
– "FdsPurgeMsg()"
– "FdsQueryQ()"
– "FdsReadQ()"
– "FdsUnlockQ()"
– "FdsWriteQ()"

- The handle is not a valid, binary-file handle. These APIs are associated with this error:
  – "FdsCloseBinFile()"
  – "FdsFlushBinFile()"
  – "FdsSetBinFileSize()"
  – "FdsQueryBinFileSize()"
  – "FdsReadBinFile()"
  – "FdsSeekBinFilePos()"
  – "FdsSetBinFileLocks()"
  – "FdsWriteBinFile()"

-222 FDSERR_HANDLE_FORCED_CLOSED

Explanation: A handle to a file that has been restricted has been specified. When file access to a file has been restricted using FdsRestrictFile(), file handles to that file cannot be specified. To resolve this error:

1. Close the file handle.
2. Remove access restrictions to the file using FdsUnrestrictFile().
3. Open a new file handle to the file.

These APIs are associated with this error:

- "FdsCloseBinFile()"
- "FdsCloseKeyedFile()"

- “FdsCloseSeqFile()”
- “FdsDeleteKeyedRecord()”
- “FdsFlushBinFile()”
- “FdsReadBinFile()”
- “FdsReadKeyedRecord()”
- “FdsReadSeqRecord()”
- “FdsReleaseKeyedRecord()”
- “FdsRestrictFile()”
- “FdsReturnSeqFilePos()”
- “FdsSeekBinFilePos()”
- “FdsSeekSeqFilePos()”
- “FdsSetBinFileLocks()”
- “FdsWriteBinFile()”
- “FdsWriteKeyedRecord()”
- “FdsWriteSeqRecord()”

-230 FDSERR_INIT

Explanation: Initialization has not occurred or has failed. Be sure that you have initiated the FdsInit() or FdsInit2() API before using any other APIs.

-240 FDSERR_INTERNAL

Explanation: An internal error occurred. Contact your IBM representative.

-250 FDSERR_INTERRUPT

Explanation: An API call was interrupted and was not completed.

-260 FDSERR_IO

Explanation: An error occurred while accessing a physical I/O device. These APIs are associated with this error:

- “FdsCloseKeyedFile()” (returned only if **FDS_FILE_RESET_YES** is specified)
- “FdsCreateDir()”
- “FdsCreateKeyedFile()”
- “FdsDeleteFile()”
- “FdsDeleteKeyedRecord()”
- “FdsExistFile()”
- “FdsFindNextSeqRecord()”
- “FdsFlushBinFile()”
- “FdsGetFileAttributes()”
- “FdsGetFileNames()”
- “FdsOpenBinFile()”

- “FdsOpenKeyedFile()”
- “FdsOpenSeqFile()”
- “FdsQueryBinFileSize()”
- “FdsQueryFileSystemInfo()”
- “FdsReadBinFile()”
- “FdsReadKeyedRecord()”
- “FdsReadSeqRecord()”
- “FdsRemoveDir()”
- “FdsRenameFile()”
- “FdsRestrictFile()”
- “FdsSetBinFileSize()”
- “FdsSetFileAttributes()”
- “FdsUnrestrictFile()”
- “FdsWriteBinFile()”
- “FdsWriteKeyedRecord()”
- “FdsWriteSeqRecord()”

-270 FDSERR_KEY

Explanation: The key is not valid. Null keys are not allowed. These APIs are associated with this error:

- “FdsDeleteKeyedRecord()”
- “FdsReadKeyedRecord()”
- “FdsReleaseKeyedRecord()”
- “FdsWriteKeyedRecord()”

-280 FDSERR_KEY_NOT_FOUND

Explanation: The key does not exist in the file. These APIs are associated with this error:

- “FdsDeleteKeyedRecord()”
- “FdsReadKeyedRecord()”
- “FdsReleaseKeyedRecord()”

-290 FDSERR_KEY_SIZE

Explanation: The key size is not valid. Specific conditions for this error include:

- “FdsCreateKeyedFile()” was called and the key size is out of range.
- The key size does not match the size specified when the file was created. These APIs are associated with this error:
  – “FdsDeleteKeyedRecord()”
  – “FdsReadKeyedRecord()”
  – “FdsReleaseKeyedRecord()”
  – “FdsWriteKeyedRecord()”

-300 FDSERR_LOGICAL_NAME

Explanation: The logical name or input string is not valid. One of the following conditions could have caused the error:

- The string is too long.
- The string is not null terminated.
- The string contains a delimiter mismatch.

Specific conditions for this error include:

- The input string is not valid. The string contains more than two colons at the end of a role name or node ID. These APIs are associated with this error:
  – "FdsBroadcastQ()"
  – "FdsCreateDir()"
  – "FdsCreateKeyedFile()"
  – "FdsCreateQ()"
  – "FdsDeleteFile()"
  – "FdsExistFile()"
  – "FdsGetFileAttributes()"
  – "FdsGetFileNames()"
  – "FdsOpenBinFile()"
  – "FdsOpenKeyedFile()"
  – "FdsOpenQ()"
  – "FdsOpenSeqFile()"
  – "FdsQueryDistribution()"
  – "FdsQueryFileSystemInfo()"
  – "FdsRemoveDir()"
  – "FdsRenameFile()"
  – "FdsResolveLogicNm()"
  – "FdsRestrictFile()"
  – "FdsSetDistribution()"
  – "FdsSetFileAttributes()"
  – "FdsUnrestrictFile()"

- The logical name is not valid. One of the following conditions could have caused the error:
  –The string contains double colons.
  –The string begins with the characters FDS.
  –The string does not match the form **<name>** where the less-than and greater-than characters (< and >) are required delimiters.
  These APIs are associated with this error:
  - "FdsCreateLogicNm()"
  - "FdsDeleteLogicNm()"
  - "FdsChangeLogicNm()"

-310 FDSERR_LOGICAL_NAME_NOT_FOUND

Explanation: The logical name is not defined. These APIs are associated with this error:

- "FdsCreateDir()"
- "FdsCreateKeyedFile()"

- “FdsCreateQ()”
- “FdsDeleteFile()”
- “FdsExistFile()”
- “FdsGetFileAttributes()”
- “FdsGetFileNames()”
- “FdsOpenBinFile()”
- “FdsOpenKeyedFile()”
- “FdsOpenQ()”
- “FdsOpenSeqFile()”
- “FdsQueryDistribution()”
- “FdsQueryFileSystemInfo()”
- “FdsRemoveDir()”
- “FdsRenameFile()”
- “FdsResolveLogicNm()”
- “FdsRestrictFile()”
- “FdsSetDistribution()”
- “FdsSetFileAttributes()”
- ”FdsUnrestrictFile()”
- “FdsRenameFile()”
- “FdsOpenSeqFile()”
- “FdsOpenKeyedFile()”
- “FdsCreateKeyedFile()”
- “FdsOpenQ()”
- “FdsBroadcastQ()”
- “FdsCreateQ()”
- “FdsSetDistribution()”
- “FdsQueryDistribution()”
- “FdsDeleteLogicNm()”
- “FdsChangeLogicNm()”
- “FdsResolveLogicNm()”
- “FdsOpenBinFile()”

-320 FDSERR_MEMORY

Explanation: DDS or the operating system is out of memory. To increase the available memory for DDS, use the **MaximumMemory** configuration keyword and restart DDS. To increase the available memory for the operating system, refer to the operating system documentation.

-325 FDSERR_MEMORY_CONSTRAINED

Explanation: DDS is in a low-memory condition on the target node of this operation. Use the **MaximumMemory** configuration keyword to allocate more memory and restart DDS on the target node. For example, if your application issued “FdsWriteQ()” to a queue on a remote node and received this error code, the low-memory problem exists on the remote node.

-330 FDSERR_MESSAGE_SIZE

Explanation: A message size that was not valid was specified to FdsWriteQ() or FdsBroadcastQ().

-340 FDSERR_NODE_NAME

Explanation: A node ID was specified that is not valid. These APIs are associated with this error:

- "FdsAddDomainNode()"
- "FdsCreateBcastDomain()"
- "FdsCreateDir()"
- "FdsCreateKeyedFile()"
- "FdsDeleteDomainNode()"
- "FdsDeleteFile()"
- "FdsExistFile()"
- "FdsGetFileAttributes()"
- "FdsOpenBinFile()"
- "FdsOpenKeyedFile()"
- "FdsOpenQ()"
- "FdsOpenSeqFile()"
- "FdsQueryFileSystemInfo()"
- "FdsRemoveDir()"
- "FdsRenameFile()"
- "FdsRestrictFile()"
- "FdsSetFileAttributes()"
- "FdsUnrestrictFile()"

-350 FDSERR_NODE_NOT_FOUND

Explanation: Communication with a node could not be established or was lost, or the node does not exist. The list below describes the specific condition for this error for each API.

- Communication with the node could not be established. The node might not exist, might be malfunctioning, or might not be configured as a file server. These APIs are associated with this error:
  –"FdsCreateDir()"
  –"FdsCreateKeyedFile()"
  –"FdsDeleteFile()"
  –"FdsExistFile()"
  –"FdsGetFileAttributes()"
  –"FdsGetFileNames()"
  –"FdsOpenBinFile()"
  –"FdsOpenKeyedFile()"
  –"FdsOpenQ()"
  –"FdsOpenSeqFile()"
  –"FdsQueryBinFileSize()"
  –"FdsQueryFileSystemInfo()"
  –"FdsRemoveDir()"
  –"FdsRenameFile()"
  –"FdsRestrictFile()"
  –"FdsSetBinFileSize()"
  –"FdsSetFileAttributes()"
  –"FdsUnrestrictFile()"

- Communication with the node has been lost. The file must be closed. These APIs are associated with this error:
  –"FdsCloseBinFile()"

– "FdsCloseKeyedFile()" (returned only if **FDS_CLOSE_TYPE_FLUSH** was specified.)
– "FdsCreateSyncID()"
– "FdsCreateSyncID()"
– "FdsDeleteKeyedRecord()"
– "FdsFindNextSeqRecord()"
– "FdsFlushBinFile()"
– "FdsQueryBinFileSize()"
– "FdsReadBinFile()"
– "FdsReadKeyedRecord()"
– "FdsReadSeqRecord()"
– "FdsReleaseKeyedRecord()"
– "FdsReturnSeqFilePos()"
– "FdsSeekBinFilePos()"
– "FdsSeekSeqFilePos()"
– "FdsSetBinFileLocks()"
– "FdsSetBinFileSize()"
– "FdsWriteBinFile()"
– "FdsWriteKeyedRecord()"
– "FdsWriteSeqRecord()"

-360 FDSERR_NODE_TYPE

Explanation: The specified operation is not allowed on this node. Specific conditions for this error include:

- This operation is valid only on the acting primary distributor. These APIs are associated with this error:
  – "FdsAddDomainNode()"
  – "FdsCreateBcastDomain()"
  – "FdsDeactivatePrimary()"
  – "FdsDeleteBcastDomain()"
  – "FdsDeleteDomainNode()"
  – "FdsGetDomainList()"
  – "FdsGetDomainNodes()"
  – "FdsQueryBackupState()"
  – "FdsQueryDistribution()"
  – "FdsSetDistribution()"
- "FdsActivateAsPrimary()" was called and this operation is valid only on the configured backup distributor or configured primary distributor when neither is the acting primary distributor.
- "FdsSetupDistMonitor()" was called and this operation is valid only on the acting primary distributor or acting backup distributor.
- An attempt was made to obtain write access to the image copy of a distributed file (returned only if **FDS_FILE_ACCESS_READ_WRITE** was specified). These APIs are associated with this error:
  – "FdsCreateKeyedFile()"
  – "FdsOpenBinFile()"
  – "FdsOpenKeyedFile()"
  – "FdsOpenSeqFile()"
- An attempt was made to update the image copy of a distributed file. These APIs are associated with this error:
  – "FdsDeleteFile()"
  – "FdsDeleteKeyedRecord()"
  – "FdsRemoveDir()"
  – "FdsRenameFile()"
  – "FdsSetBinFileSize()"
  – "FdsWriteBinFile()"

– “FdsWriteKeyedRecord()”
– “FdsWriteSeqRecord()”

-370 FDSERR_NOTIFY_QUEUE

Explanation: A notification queue handle that is not valid was specified to “FdsOpenQ()”. Possible problems are:

- The queue with which the queue handle is associated has been closed.
- The specified queue handle has not been initialized (using “FdsCreateQ()”).
-

-375 FDSERR_NOT_DISTRIBUTED

Explanation: The file or directory is not distributed. This error is associated with “FdsCreateSyncID()” and “FdsSetDistribution()”.

-380 FDSERR_NOT_RECONCILED

Explanation: The acting backup distributor is not fully reconciled. This error is associated with “FdsActivateAsPrimary()” and “FdsDeactivatePrimary()”.

-390 FDSERR_NUM_BLOCKS

Explanation: A number of blocks that is not valid was specified to “FdsCreateKeyedFile()”.

-400 FDSERR_OS

Explanation: An unexpected, operating-system condition occurred. See the event logs for details.

-410 FDSERR_OVERFLOW

Explanation: An internal buffer has reached its capacity. Specific conditions for this error include:

- The logical-name resolution was too complex or the system is out of file handles. One of the following conditions could have caused the error:
  – The input string resolves to a recursive, logical-name definition.
  – The input string takes more than 500 logical-name resolutions to completely resolve.
  – The output string was longer than 2 times the maximum path length allowed by the operating system.

  These APIs are associated with this error:
  – “FdsBroadcastQ()”
  – “FdsCreateDir()”
  – “FdsCreateKeyedFile()”
  – “FdsDeleteFile()”
  – “FdsExistFile()”
  – “FdsGetFileAttributes()”
  – “FdsGetFileNames()”
  – “FdsOpenBinFile()”

– “FdsOpenKeyedFile()”
– “FdsOpenSeqFile()”
– “FdsQueryDistribution()”
– “FdsQueryFileSystemInfo()”
– “FdsRemoveDir()”
– “FdsRenameFile()”
– “FdsRestrictFile()”
– “FdsSetDistribution()”
– “FdsSetFileAttributes()”
– “FdsUnrestrictFile()”

- “FdsResolveLogicNm()” was called and the logical-name resolution was too complex. One of the following conditions could have caused the error:
  – The input string resolves to a recursive, logical-name definition.
  – The input string takes more than 500 logical name resolutions to completely resolve.
  – The output string was longer than 2 times the maximum path length allowed by the operating system.
  The partially resolved name is returned.
- The logical name resolution was too complex or the system is out of queue handles. These APIs are associated with this error:
  – “FdsCreateQ()”
  – “FdsOpenQ()”

-420 FDSERR_QUEUE_CLOSED

Explanation: The queue no longer exists. The queue handle must be closed. These APIs are associated with this error:

- “FdsReadQ()”
- “FdsWriteQ()”

-430 FDSERR_QUEUE_EMPTY

Explanation: There are no more messages in the queue. These APIs are associated with this error:

- “FdsPurgeMsg()”
- “FdsReadQ()”

-440 FDSERR_QUEUE_FULL

Explanation: “FdsWriteQ()” was called and the queue is full.
Your request has timed out and the message was not written to the queue as a result of one of the following conditions:

- There was not enough space available in the queue for your message.
- The queue was blocked by another write request.

A queue becomes blocked when a write request is received with the WaitConfirm parameter set to **FDS_WRITTEN**, but there is not enough space in the destination queue for the message. While a queue is blocked, all subsequent write requests become blocked, in the order that the write requests were received. As space becomes available in the queue, IPC completes the blocked write requests in the order that they were blocked. When IPC has completed all of the blocked write requests,

the queue is no longer blocked.

-450 FDSERR_QUEUE_NAME

Explanation: A queue name that is not valid was specified. Specific conditions for this error include:

- Either the specified queue name begins with the letters FDS, or the specified queue name or resolved logical name exceeds the maximum queue name length. These APIs are associated with this error:
  – “FdsCreateQ()”
  – “FdsOpenQ()”
- FdsBroadcastQ()” was called and the queue name was not a string, or it was a null string.

-460 FDSERR_QUEUE_NOT_FOUND

Explanation: The queue does not exist. The list below describes the specific condition for this error for each API.

- “FdsOpenQ()” was called and the queue does not exist on the specified node.
- At the file server, the **MaxRequesters** keyword must be set to the number of workstations requesting file services. These APIs are associated with this error:
  – “FdsCreateDir()”
  – “FdsCreateKeyedFile()”
  – “FdsDeleteFile()”
  – “FdsExistFile()”
  – “FdsGetFileAttributes()”
  – “FdsOpenKeyedFile()”
  – “FdsOpenSeqFile()”
  – “FdsQueryFileSystemInfo()”
  – “FdsRemoveDir()”
  – “FdsRenameFile()”
  – “FdsRestrictFile()”
  – “FdsSetFileAttributes()”
  – “FdsUnrestrictFile()”

-470 FDSERR_QUEUE_SIZE

Explanation: A queue size that is not valid was specified to “FdsCreateQ()” .

-480 FDSERR_RAND_DIV

Explanation: A randomizing divisor that is not valid was specified to “FdsCreateKeyedFile()”.

-490 FDSERR_REC_SIZE

Explanation: A record size that is not valid was specified. Specific conditions for this error include:

- The record size is out of range. These APIs are associated with this error:
  – “FdsCreateKeyedFile()”

– “FdsReadBinFile()”
– “FdsReadKeyedRecord()”
– “FdsWriteBinFile()”
– “FdsWriteSeqRecord()”

- “FdsWriteKeyedRecord()” was called and the record size does not match the size that was specified when the file was created.

-500 FDSERR_REMOTE

Explanation: A remote object was specified or remote communication was requested when remote IPC has not been configured. Specific conditions for this error include:

- “FdsResolveLogicNm()” was called and a non-local node ID or role name was encountered. The string that caused the error is returned.
- “FdsCreateQ()” was called and a non-local queue name was specified. Queues can be created locally only.
- A non-local queue name was specified and remote communication is not configured. These APIs are associated with this error:
  – “FdsOpenQ()”
  – “FdsWriteQ()”
- “FdsBroadcastQ()” was called and remote communication is not configured.
- A non-local file or directory was specified. These APIs are associated with this error:
  – “FdsQueryDistribution()”
  – “FdsSetDistribution()”

-510 FDSERR_RESOLVED_NAME

Explanation: A definition that is not valid was provided for a logical name. One of the following conditions could have caused the error:

- The string is too long.
- The string is not null terminated.
- The string contains a delimiter mismatch.

These APIs are associated with this error:

- “FdsChangeLogicNm()”
- “FdsCreateLogicNm()”

-520 FDSERR_RESOURCE

Explanation: The application has too many concurrent requests running.

-530 FDSERR_ROLE_CHANGE

Explanation: The handle was associated with a role that has moved. Specific conditions for this error include:

- Either the file was opened with a role that has moved or the prime copy of a distributed file was opened and the acting primary distributor has been deactivated. The file must be closed.
  These APIs are associated with this error:
  – “FdsCreateSyncID()”

– “FdsDeleteKeyedRecord()”
– “FdsFindNextSeqRecord()”
– “FdsFlushBinFile()”
– “FdsQueryBinFileSize()”
– “FdsReadBinFile()”
– “FdsReadKeyedRecord()”
– “FdsReadSeqRecord()”
– “FdsReleaseKeyedRecord()”
– “FdsReturnSeqFilePos()”
– “FdsSeekBinFilePos()”
– “FdsSeekSeqFilePos()”
– “FdsSetBinFileLocks()”
– “FdsSetBinFileSize()”
– “FdsWriteBinFile()”
– “FdsWriteKeyedRecord()”
– “FdsWriteSeqRecord()”

- “FdsWriteQ()” was called and
- This operation is valid only on the acting primary distributor. These APIs are associated with this error:
  – “FdsAddDomainNode()”
  – “FdsCreateBcastDomain()”
  – “FdsDeactivatePrimary()”
  – “FdsDeleteBcastDomain()”
  – “FdsDeleteDomainNode()”
  – “FdsGetDomainList()”
  – “FdsGetDomainNodes()”
  – “FdsQueryBackupState()”
  – “FdsQueryDistribution()”
  – “FdsSetDistribution()”
- “FdsActivateAsPrimary()” was called and this operation is valid only on the configured backup distributor or configured primary distributor when neither is the acting primary distributor.
- “FdsSetupDistMonitor()” was called and this operation is valid only on the acting primary distributor or acting backup distributor.
- An attempt was made to obtain write access to the image copy of a distributed file (returned only if **FDS_FILE_ACCESS_READ_WRITE** was specified). These APIs are associated with this error:
  – “FdsCreateKeyedFile()”
  – “FdsOpenBinFile()”
  – “FdsOpenKeyedFile()”
  – “FdsOpenSeqFile()”
- An attempt was made to update the image copy of a distributed file. These APIs are associated with this error:
  – “FdsDeleteFile()”
  – “FdsDeleteKeyedRecord()”
  – “FdsRemoveDir()”
  – “FdsRenameFile()”
  – “FdsSetBinFileSize()”
  – “FdsWriteBinFile()”
  – “FdsWriteKeyedRecord()”
  – “FdsWriteSeqRecord()”

-370 FDSERR_NOTIFY_QUEUE

Explanation: A notification queue handle that is not valid was specified to

"FdsOpenQ()". Possible problems are:

- The queue with which the queue handle is associated has been closed.
- The specified queue handle has not been initialized (using "FdsCreateQ()").

-375 FDSERR_NOT_DISTRIBUTED

Explanation: The file or directory is not distributed. This error is associated with "FdsCreateSyncID()" and "FdsSetDistribution()".

-380 FDSERR_NOT_RECONCILED

Explanation: The acting backup distributor is not fully reconciled. This error is associated with "FdsActivateAsPrimary()" and "FdsDeactivatePrimary()".

-390 FDSERR_NUM_BLOCKS

Explanation: A number of blocks that is not valid was specified to "FdsCreateKeyedFile()".

-400 FDSERR_OS

Explanation: An unexpected, operating-system condition occurred. See the event logs for details.

-410 FDSERR_OVERFLOW

Explanation: An internal buffer has reached its capacity. Specific conditions for this error include:

- The logical-name resolution was too complex or the system is out of file handles. One of the following conditions could have caused the error:
  – The input string resolves to a recursive, logical-name definition.
  – The input string takes more than 500 logical-name resolutions to completely resolve.
  – The output string was longer than 2 times the maximum path length allowed by the operating system.

  These APIs are associated with this error:
  – "FdsBroadcastQ()"
  – "FdsCreateDir()"
  – "FdsCreateKeyedFile()"
  – "FdsDeleteFile()"
  – "FdsExistFile()"
  – "FdsGetFileAttributes()"
  – "FdsGetFileNames()"
  – "FdsOpenBinFile()"
  – "FdsOpenKeyedFile()"
  – "FdsOpenSeqFile()"
  – "FdsQueryDistribution()"
  – "FdsQueryFileSystemInfo()"
  – "FdsRemoveDir()"
  – "FdsRenameFile()"

– “FdsRestrictFile()”
– “FdsSetDistribution()”
– “FdsSetFileAttributes()”
– “FdsUnrestrictFile()”

- “FdsResolveLogicNm()” was called and the logical-name resolution was too complex. One of the following conditions could have caused the error:
  – The input string resolves to a recursive, logical-name definition.
  – The input string takes more than 500 logical name resolutions to completely resolve.
  – The output string was longer than 2 times the maximum path length allowed by the operating system.

  The partially resolved name is returned.
- The logical name resolution was too complex or the system is out of queue handles. These APIs are associated with this error:
  – “FdsCreateQ()”
  – “FdsOpenQ()”

-420 FDSERR_QUEUE_CLOSED

Explanation: The queue no longer exists. The queue handle must be closed. These APIs are associated with this error:

- “FdsReadQ()”
- “FdsWriteQ()”

-430 FDSERR_QUEUE_EMPTY

Explanation: There are no more messages in the queue. These APIs are associated with this error:

- “FdsPurgeMsg()”
- “FdsReadQ()”

-440 FDSERR_QUEUE_FULL

Explanation: “FdsWriteQ()” was called and the queue is full.
Your request has timed out and the message was not written to the queue as a result of one of the following conditions:

- There was not enough space available in the queue for your message.
- The queue was blocked by another write request.

A queue becomes blocked when a write request is received with the WaitConfirm parameter set to **FDS_WRITTEN**, but there is not enough space in the destination queue for the message. While a queue is blocked, all subsequent write requests become blocked, in the order that the write requests were received. As space becomes available in the queue, IPC completes the blocked write requests in the order that they were blocked. When IPC has completed all of the blocked write requests, the queue is no longer blocked.

-450 FDSERR_QUEUE_NAME

Explanation: A queue name that is not valid was specified. Specific conditions for this error include:

- Either the specified queue name begins with the letters FDS, or the specified queue name or resolved logical name exceeds the maximum queue name length. These APIs are associated with this error:
  – “FdsCreateQ()”
  – “FdsOpenQ()”
- “FdsBroadcastQ()” was called and the queue name was not a string, or it was a null string.

-460 FDSERR_QUEUE_NOT_FOUND

Explanation: The queue does not exist. The list below describes the specific condition for this error for each API.

- “FdsOpenQ()” was called and the queue does not exist on the specified node.
- At the file server, the **MaxRequesters** keyword must be set to the number of workstations requesting file services. These APIs are associated with this error:
  – “FdsCreateDir()”
  – “FdsCreateKeyedFile()”
  – “FdsDeleteFile()”
  – “FdsExistFile()”
  – “FdsGetFileAttributes()”
  – “FdsOpenKeyedFile()”
  – “FdsOpenSeqFile()”
  – “FdsQueryFileSystemInfo()”
  – “FdsRemoveDir()”
  – “FdsRenameFile()”
  – “FdsRestrictFile()”
  – “FdsSetFileAttributes()”
  – “FdsUnrestrictFile()”

-470 FDSERR_QUEUE_SIZE

Explanation: A queue size that is not valid was specified to “FdsCreateQ()” .

-480 FDSERR_RAND_DIV

Explanation: A randomizing divisor that is not valid was specified to “FdsCreateKeyedFile()”.

-490 FDSERR_REC_SIZE

Explanation: A record size that is not valid was specified. Specific conditions for this error include:

- The record size is out of range. These APIs are associated with this error:
  – “FdsCreateKeyedFile()”
  – “FdsReadBinFile()”
  – “FdsReadKeyedRecord()”
  – “FdsWriteBinFile()”
  – “FdsWriteSeqRecord()”
- “FdsWriteKeyedRecord()” was called and the record size

does not match the size that was specified when the file was created.

-500 FDSERR_REMOTE

Explanation: A remote object was specified or remote communication was requested when remote IPC has not been configured. Specific conditions for this error include:

- "FdsResolveLogicNm()" was called and a non-local node ID or role name was encountered. The string that caused the error is returned.
- "FdsCreateQ()" was called and a non-local queue name was specified. Queues can be created locally only.
- A non-local queue name was specified and remote communication is not configured. These APIs are associated with this error:
  – "FdsOpenQ()"
  – "FdsWriteQ()"
- "FdsBroadcastQ()" was called and remote communication is not configured.
- A non-local file or directory was specified. These APIs are associated with this error:
  – "FdsQueryDistribution()"
  – "FdsSetDistribution()"

-510 FDSERR_RESOLVED_NAME

Explanation: A definition that is not valid was provided for a logical name. One of the following conditions could have caused the error:

- The string is too long.
- The string is not null terminated.
- The string contains a delimiter mismatch.

These APIs are associated with this error:

- "FdsChangeLogicNm()"
- "FdsCreateLogicNm()"
-

-520 FDSERR_RESOURCE

Explanation: The application has too many concurrent requests running.

-530 FDSERR_ROLE_CHANGE

Explanation: The handle was associated with a role that has moved. Specific conditions for this error include:

- Either the file was opened with a role that has moved or the prime copy of a distributed file was opened and the acting primary distributor has been deactivated. The file must be closed.
  These APIs are associated with this error:
  – "FdsCreateSyncID()"
  – "FdsDeleteKeyedRecord()"
  – "FdsFindNextSeqRecord()"
  – "FdsFlushBinFile()"
  – "FdsQueryBinFileSize()"
  – "FdsReadBinFile()"

– “FdsReadKeyedRecord()”
– “FdsReadSeqRecord()”
– “FdsReleaseKeyedRecord()”
– “FdsReturnSeqFilePos()”
– “FdsSeekBinFilePos()”
– “FdsSeekSeqFilePos()”
– “FdsSetBinFileLocks()”
– “FdsSetBinFileSize()”
– “FdsWriteBinFile()”
– “FdsWriteKeyedRecord()”
– “FdsWriteSeqRecord()”

- “FdsWriteQ()” was called and the queue was opened with a role that has moved and **FDS_CONFIRM_ROLE** was specified.

-540 FDSERR_ROLE_NAME

Explanation: A role name that is not valid was specified. One of the following conditions could have caused the error:

- The string is too long.
- The string is not null terminated.
- The string does not match the form <name::> where name is1to8 characters, and the less-than and greater-than characters (< and >) and a double colon (::) are required characters.
- The string begins with the prefix FDS.
- The string contains more than two colons at the end of the role name. These APIs are associated with this error:
  - “FdsCreateDir()”
  - “FdsCreateKeyedFile()”
  - “FdsDeleteFile()”
  - “FdsExistFile()”
  - “FdsGetFileAttributes()”
  - “FdsOpenBinFile()”
  - “FdsOpenKeyedFile()”
  - “FdsOpenQ()”
  - “FdsOpenSeqFile()”
  - “FdsQueryFileSystemInfo()”
  - “FdsRemoveDir()”
  - “FdsRenameFile()”
  - “FdsResolveRoleNm()”
  - “FdsRestrictFile()”
  - “FdsSetFileAttributes()”
  - “FdsSetResetRole()”
  - “FdsUnrestrictFile()”
  - “FdsVerifyRole()”

-550 FDSERR_ROLE_NOT_FOUND

Explanation: The role is not active or could not be found. Specific

conditions for this error include:

- The role is not active on any node in the system. These APIs are associated with this error:
  – "FdsCreateDir()"
  – "FdsCreateKeyedFile()"
  – "FdsDeleteFile()"
  – "FdsExistFile()"
  – "FdsGetFileAttributes()"
  – "FdsGetNodes()"
  – "FdsOpenBinFile()"
  – "FdsOpenKeyedFile()"
  – "FdsOpenQ()"
  – "FdsOpenSeqFile()"
  – "FdsQueryFileSystemInfo()"
  – "FdsRemoveDir()"
  – "FdsRenameFile()"
  – "FdsRestrictFile()"
  – "FdsSetFileAttributes()"
  – "FdsUnrestrictFile()"
- "FdsSetResetRole()" was called and the role is not active on the local node (returned only if **FDS_RESET_ROLE** is specified).
- "FdsVerifyRole()" was called and the role is not active on the local node.
- "FdsResolveRoleNm()" was called; the role is not active on the local node and **FDS_CACHE_ONLY** is specified, or the role is not active on any node within the system.

-555 FDSERR_SCOPE

Explanation: A scope that is not valid was specified to "FdsSetDistribution()".

-558 FDSERR_SEEK_TYPE

Explanation: The specified parameter is not valid. Specific conditions for this error include:

- The value provided for the *Origin* parameter is not valid. You must provide one of the following values:

– **FDS_FILE_START_OF_FILE**
– **FDS_FILE_CURRENT_POS**
– **FDS_FILE_END_OF_FILE**

The *Origin* parameter is used with these APIs:
– "FdsReadBinFile()"
– "FdsSeekBinFilePos()"
– "FdsWriteBinFile()"

- The value provided for the *SeekMethod* parameter is not valid. You must provide one of the following values:
  – **FDS_CACHE_ONLY**
  – **FDS_NETWORK_ONLY**
  – **FDS_CACHE_FIRST**
  The *SeekMethod* parameter is used with "FdsResolveRoleNm()" .

-560 FDSERR_SEQUENCE

Explanation: An operation occurred out of sequence. Specific conditions for this error include:

- "FdsInit()" or "FdsInit2()" has already been called successfully by this process or is currently being called by another thread in this process.
- "FdsReadKeyedRecord()" was called and the record is already locked (returned only if **FDS_FILE_RECORD_LOCK_YES** is specified).
- "FdsReleaseKeyedRecord()" was called and the record is not locked.
- "FdsWriteKeyedRecord()" was called and the record is not locked (returned only if **FDS_FILE_RECORD_UNLOCK_YES** is specified).
- "FdsSetDistribution()" was called and the path name is contained in a directory that is already distributed, or it is a directory that contains a file which is already distributed.
- "FdsActivateAsPrimary()" was called and an activation or deactivation of the primary distributor is already in progress.
- "FdsDeactivatePrimary()" was called and an activation or deactivation of the primary distributor is already in progress.
- "FdsCreateSyncID()" was called and an attempt is being made to create a synchronization ID without having previously performed a distributed file operation against the file. Most file operations performed on DOU files are distributed, but only some operations performed on DOC files are distributed.

-570 FDSERR_SYNCID

Explanation: A synchronization ID that is not valid was specified to "FdsSetupSyncIDNotify()".

-575 FDSERR_THREAD_LIMIT

Explanation: This request could not be completed because too many application threads currently have incomplete API calls to DDS. The calling application might pause and then try the API call again. If this problem persists, the number of application threads that can concurrently call DDS APIs should be reduced. This error could be associated with any DDS API.

-580 FDSERR_TIMEOUT

Explanation: The request could not be completed within the specified time limit. These APIs are associated with this error:

- "FdsGetNodes()"
- "FdsOpenQ()"
- "FdsReadQ()"
- "FdsWriteQ()"

Appendix C. Operating-System Error Codes

The File System Interface component must return standard, operating-system error codes for errors resulting from file-system operations. These error codes are returned from file system APIs such as:

- CreateFile() on Windows NT or Windows 2000
- WriteFile() on Windows NT or Windows 2000
- ReadFile() on Windows NT or Windows 2000

For normal file-system errors, the File System Interface component returns the error code that it receives from the underlying, operating-system file system. The error-code descriptions provided by the operating system include the information necessary for diagnosing the cause of these errors.

However, in some cases, the File System Interface component returns operating system error codes for problems unique to DDS. This chapter lists the operating-system error codes that can be returned in these situations, and explains the problems that will cause one of these errors to be returned. Because these situations are unique to the File System Interface component of DDS, the errors described in this section can only occur durng I/O operations against files stored on controlled drives. See the **IBM Distributed Data Services/Controller Services Feature for Windows Installation and Configuration Guide** for more information about controlled drives.

Error Codes from Windows NT or Windows 2000

This section contains a list of error codes in decimal numeric order.

5 ERROR_ACCESS_DENIED

Explanation: Only the prime copy of a distributed file, or files in distributed subdirectories, can be modified. Image copies of distributed files can only be read. This error is returned under the following conditions:

- On any attempt to modify an image copy of a file that has been opened on the acting primary distributor when it is no longer the acting primary distributor (after it has been deactivated)
- On any attempt to modify an image copy of a file or subdirectory using a name-based API (for example, deleting a file, removing a directory, or changing extended attributes)
- On any attempt to remove a directory or create a file or directory in an image copy of a distributed directory
- On an open of an image copy with an access mode of read/write or write-only

Services APIs map this error to The DDS File FDSERR_ACCESS. See “-10 FDSERR_ACCESS” for more information.

6 ERROR_INVALID_HANDLE

Explanation: All distributed files opened for write access should be closed before DDS is stopped or the acting primary distributor is deactivated. If a file is not closed, this error might be returned the next time the file handle is used.

This error is also returned if a write is attempted to the prime copy of a distributed file after the acting primary distributor has been deactivated. When this error is received, the file should be closed and then reopened after DDS has been restarted or the primary distributor has been activated.

21 ERROR_NOT_READY

Explanation: This error occurs during all attempts to modify any file on a controlled drive when DDS is not running; the File System Interface component cannot access the information required to determine if a file is distributed unless DDS is running. Attempts to open a file with an access mode of read/write or write-only also result in this error, even though the open operation itself does not cause the file to be modified.

Once DDS has been started, this error will be returned if an attempt is made to modify a distributed file or a file in a distributed subdirectory, when data distribution has not yet completed initialization.

The DDS File Services APIs map this error to FDSERR_DOWN. See "-150 FDSERR_DOWN" for more information.