CEN WORKSHOP AGREEMENT

CWA 16926-61

February 2020

ICS 35.200; 35.240.15; 35.240.40

English version

Extensions for Financial Services (XFS) interface specification Release 3.40 - Part 61: Application Programming Interface (API) - Service Provider Interface (SPI) - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

EUROPEAN COMMITTEE FOR STANDARDIZATION
COMITÉ EUROPÉEN DE NORMALISATION
EUROPÄISCHES KOMITEE FÜR NORMUNG

CEN-CENELEC Management Centre: Rue de la Science 23, B-1040 Brussels

Ref. No.:CWA 16926-61:2020 E

Table of Contents

European Foreword

This CEN Workshop Agreement has been developed in accordance with the CEN-CENELEC Guide 29 "CEN/CENELEC Workshop Agreements – The way to rapid consensus" and with the relevant provisions of CEN/CENELEC Internal Regulations - Part 2. It was approved by a Workshop of representatives of interested parties on 2019-10-08, the constitution of which was supported by CEN following several public calls for participation, the first of which was made on 1998-06-24. However, this CEN Workshop Agreement does not necessarily include all relevant stakeholders.
The final text of this CEN Workshop Agreement was provided to CEN for publication on 2019-12-12.
The following organizations and individuals developed and approved this CEN Workshop Agreement:

- ATM Japan LTD
- AURIGA SPA
- BANK OF AMERICA
- CASHWAY TECHNOLOGY
- CHINAL ECTRONIC FINANCIAL EQUIPMENT SYSTEM CO.
- CIMA SPA
- CLEAR2PAY SCOTLAND LIMITED
- DIEBOLD NIXDORF
- EASTERN COMMUNICATIONS CO. LTD – EASTCOM
- FINANZ INFORMATIK
- FUJITSU FRONTECH LIMITED
- FUJITSU TECHNOLOGY
- GLORY LTD
- GRG BANKING EQUIPMENT HK CO LTD
- HESS CASH SYSTEMS GMBH & CO. KG
- HITACHI OMRON TS CORP.
- HYOSUNG TNS INC
- JIANGSU GUOGUANG ELECTRONIC INFORMATION TECHNOLOGY
- KAL
- KEBA AG
- NCR FSG
- NEC CORPORATION
- OKI ELECTRIC INDUSTRY SHENZHEN
- OKI ELECTRONIC INDUSTRY CO
- PERTO S/A

- REINER GMBH & CO KG
- SALZBURGER BANKEN SOFTWARE
- SIGMA SPA
- TEB
- ZIJIN FULCRUM TECHNOLOGY CO

It is possible that some elements of this CEN/CWA may be subject to patent rights. The CEN-CENELEC policy on patent rights is set out in CEN-CENELEC Guide 8 “Guidelines for Implementation of the Common IPR Policy on Patents (and other statutory intellectual property rights based on inventions)”. CEN shall not be held responsible for identifying any or all such patent rights.
The Workshop participants have made every effort to ensure the reliability and accuracy of the technical and non-technical content of CWA 16926-61, but this does not guarantee, either explicitly or implicitly, its correctness. Users of CWA 16926-61 should be aware that neither the Workshop participants, nor CEN can be held liable for damages or losses of any kind whatsoever which may arise from its application. Users of CWA 16926-61 do so on their own responsibility and at their own risk.

The CWA is published as a multi-part document, consisting of:

Part 1: Application Programming Interface (API) - Service Provider Interface (SPI) - Programmer's Reference

Part 2: Service Classes Definition - Programmer's Reference

Part 3: Printer and Scanning Device Class Interface - Programmer's Reference

Part 4: Identification Card Device Class Interface - Programmer's Reference

Part 5: Cash Dispenser Device Class Interface - Programmer's Reference

Part 6: PIN Keypad Device Class Interface - Programmer's Reference

Part 7: Check Reader/Scanner Device Class Interface - Programmer's Reference

Part 8: Depository Device Class Interface - Programmer's Reference

Part 9: Text Terminal Unit Device Class Interface - Programmer's Reference

Part 10: Sensors and Indicators Unit Device Class Interface - Programmer's Reference

Part 11: Vendor Dependent Mode Device Class Interface - Programmer's Reference

Part 12: Camera Device Class Interface - Programmer's Reference

Part 13: Alarm Device Class Interface - Programmer's Reference

Part 14: Card Embossing Unit Device Class Interface - Programmer's Reference

Part 15: Cash-In Module Device Class Interface - Programmer's Reference

Part 16: Card Dispenser Device Class Interface - Programmer's Reference

Part 17: Barcode Reader Device Class Interface - Programmer's Reference

Part 18: Item Processing Module Device Class Interface - Programmer's Reference

Part 19: Biometrics Device Class Interface - Programmer's Reference

Parts 20 - 28: Reserved for future use.

Parts 29 through 47 constitute an optional addendum to this CWA. They define the integration between the SNMP standard and the set of status and statistical information exported by the Service Providers.

Part 29: XFS MIB Architecture and SNMP Extensions - Programmer’s Reference

Part 30: XFS MIB Device Specific Definitions - Printer Device Class

Part 31: XFS MIB Device Specific Definitions - Identification Card Device Class

Part 32: XFS MIB Device Specific Definitions - Cash Dispenser Device Class

Part 33: XFS MIB Device Specific Definitions - PIN Keypad Device Class

Part 34: XFS MIB Device Specific Definitions - Check Reader/Scanner Device Class

Part 35: XFS MIB Device Specific Definitions - Depository Device Class

Part 36: XFS MIB Device Specific Definitions - Text Terminal Unit Device Class

Part 37: XFS MIB Device Specific Definitions - Sensors and Indicators Unit Device Class

Part 38: XFS MIB Device Specific Definitions - Camera Device Class

Part 39: XFS MIB Device Specific Definitions - Alarm Device Class

Part 40: XFS MIB Device Specific Definitions - Card Embossing Unit Class

Part 41: XFS MIB Device Specific Definitions - Cash-In Module Device Class

Part 42: Reserved for future use.

Part 43: XFS MIB Device Specific Definitions - Vendor Dependent Mode Device Class

Part 44: XFS MIB Application Management

Part 45: XFS MIB Device Specific Definitions - Card Dispenser Device Class

Part 46: XFS MIB Device Specific Definitions - Barcode Reader Device Class

Part 47: XFS MIB Device Specific Definitions - Item Processing Module Device Class

Part 48: XFS MIB Device Specific Definitions - Biometrics Device Class

Parts 49 - 60 are reserved for future use.

Part 61: Application Programming Interface (API) - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Service Provider Interface (SPI) - Programmer's Reference

Part 62: Printer and Scanning Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 63: Identification Card Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 64: Cash Dispenser Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 65: PIN Keypad Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 66: Check Reader/Scanner Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 67: Depository Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 68: Text Terminal Unit Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 69: Sensors and Indicators Unit Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 70: Vendor Dependent Mode Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 71: Camera Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 72: Alarm Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 73: Card Embossing Unit Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 74: Cash-In Module Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 75: Card Dispenser Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 76: Barcode Reader Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

Part 77: Item Processing Module Device Class Interface - Migration from Version 3.30 (CWA 16926) to Version 3.40 (this CWA) - Programmer's Reference

In addition to these Programmer's Reference specifications, the reader of this CWA is also referred to a complementary document, called Release Notes. The Release Notes contain clarifications and explanations on the CWA specifications, which are not requiring functional changes. The current version of the Release Notes is available online from: https://www.cen.eu/work/Sectors/Digital_society/Pages/WSXFS.aspx.

The information in this document represents the Workshop's current views on the issues discussed as of the date of publication. It is provided for informational purposes only and is subject to change without notice. CEN makes no warranty, express or implied, with respect to this document.

1 Migration Information

XFS 3.40 has been designed to minimize backwards compatibility issues. This document highlights the changes made to the API/SPI between version 3.30 and 3.40, by highlighting the additions and deletions to the text.

2 References

1. XFS Service Classes Definition, Programmer's Reference Revision 3.~~30~~40
2. The Unicode Standard, Version 5.0, released on 9 November 2006. ISBN 0321480910

3 XFS (eXtensions for Financial Services) Overview

A key element of the Extensions for Financial Services is the definition of a set of APIs, a corresponding set of SPIs, and supporting services, providing access to financial services for Windows-based applications. The definition of the functionality of the services, of the architecture, and of the API and SPI sets, is outlined in this section, and described in detail in Sections 5 through 10.

The specification defines a standard set of interfaces such that, for example, an application that uses the API set to communicate with a particular Service Provider can work with a Service Provider of another conformant vendor, without any changes.

Although the Extensions for Financial Services define a general architecture for access to Service Providers from Windows-based applications, the initial focus of the CEN/~~ISSS~~ XFS Workshop has been on providing access to peripheral devices that are unique to financial institutions. Since these devices are often complex, difficult to manage and proprietary, the development of a standardized interface to them from Windows-based applications and Windows operating systems can offer financial institutions and their solution providers immediate enhancements to productivity and flexibility.

3.1 Architecture

The architecture of the Extensions for Financial Services (XFS) system is shown below.

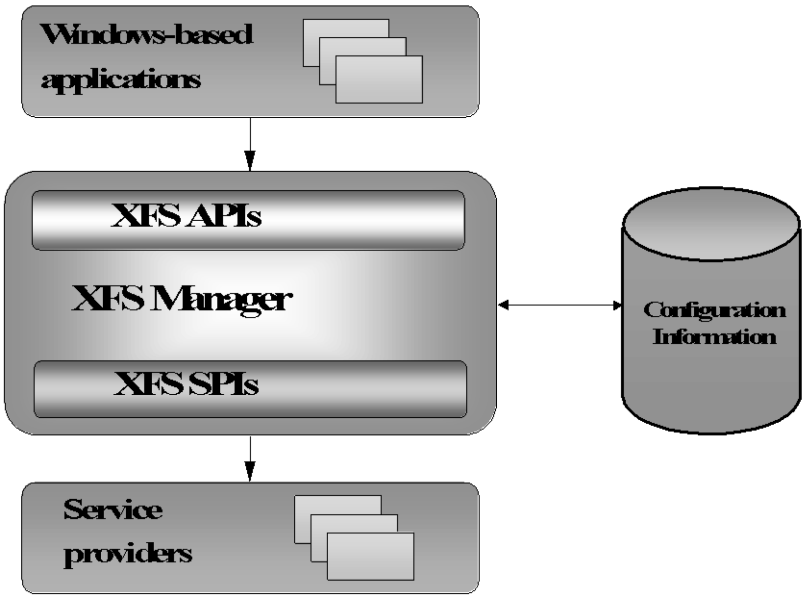


Figure 2.1 - Extensions for Financial Services Architecture

The applications communicate with Service Providers, via the Extensions for Financial Services Manager, using the API set. Most of these APIs can be invoked either "synchronously" (the Manager causes the application to wait until the API's function is completed) or "asynchronously" (the application regains control immediately, while the function is performed in parallel).

The common deliverable in all implementations of this Extensions for Financial Services specification is the Extensions for Financial Services Manager, which maps the specified API to the corresponding SPI, then routes this request to the appropriate Service Provider. Multiple implementations of the XFS Manager exist from different vendors. For the definition of the binary interface, see section 4.16.

The Manager uses the configuration information to route the API call (made to a "logical service" or a "logical device") to the proper Service Provider entry point (which is always local, even though the device or service that is the final target may be remote). Note that even though the API calls may be either synchronous or asynchronous, the SPI calls are always asynchronous.

The developers of financial services to be used via XFS and the manufacturers of financial peripherals will be responsible for the development and distribution of Service Providers for their services and devices. A setup routine for each device or service will also be necessary to define the appropriate configuration information. This information will allow an application to request capability and status information about the devices and services available at any point in time.

The primary functions of the Service Providers are to:

- Translate generic (e.g. forms-based) service requests to service-specific commands.
- Route the requests to either a local service or device, or to one on a remote system, effectively defining a peer-to-peer interface among Service Providers.
- Arbitrate access by multiple applications to a single service or device, providing exclusive access when requested.
- Manage the hardware interfaces to services or devices.
- Manage the asynchronous nature of the services and devices in an appropriate manner, always presenting this capability to the XFS Manager and the applications via Windows messages.

The system design supports solution of complex problems, often not addressed by current systems, by providing for maximum flexibility in all its capabilities:

- Multiple Service Providers, developed by multiple vendors, can coexist in a single system and in a network. This is ensured by a standard messaging/data interface and a standard binary interface for the XFS Manager.
- The service class definition is based on the logical functionalities of the service, with no assumption being made as to the physical configuration. A physical device that includes multiple distinct physical capabilities (referred to as a "compound device" in this specification) is treated as several logical services; the Service Provider resolves any conflicts. Note also that a logical service may include multiple physical devices (for example, a cash dispenser consisting of a note dispenser and coin dispenser).
- Similarly, a physical device may be shared between two or more users (e.g. tellers), and the physical device synchronization is managed at the Service Provider level.
- The API definition and associated services provide time-out functionality to allow applications to avoid deadlock of the type that can occur if two applications try to get exclusive access to multiple services at the same time.
- The architecture is designed to provide a framework for future development of network and system monitoring, measurement, and management.

Note that Figure 2.1 is a high level view of the architecture and, in particular, it makes no distinction between Service Providers and the services they manage. This specification focuses on Service Providers rather than on services, because the way a Service Provider communicates with a service is a vendor-specific internal design issue that applications and the XFS Manager are unaware of. In fact, there are many different ways that Service Providers can make services available to applications. Hence, this specification refers primarily to the Service Providers, since these are the modules with which the XFS Manager communicates. There are occasional references to 'service' where this is appropriate.

Example
Figure 2.2 below shows an XFS system supporting a set of financial peripherals. Note that in this framework the XFS Manager interfaces directly with a set of Service Providers that interface directly with the physical devices. Thus, the Service Providers are shown as implementing the Service Provider, service, and device driver functions, although these are more likely to be two or more separate layers. Many other configurations are possible.

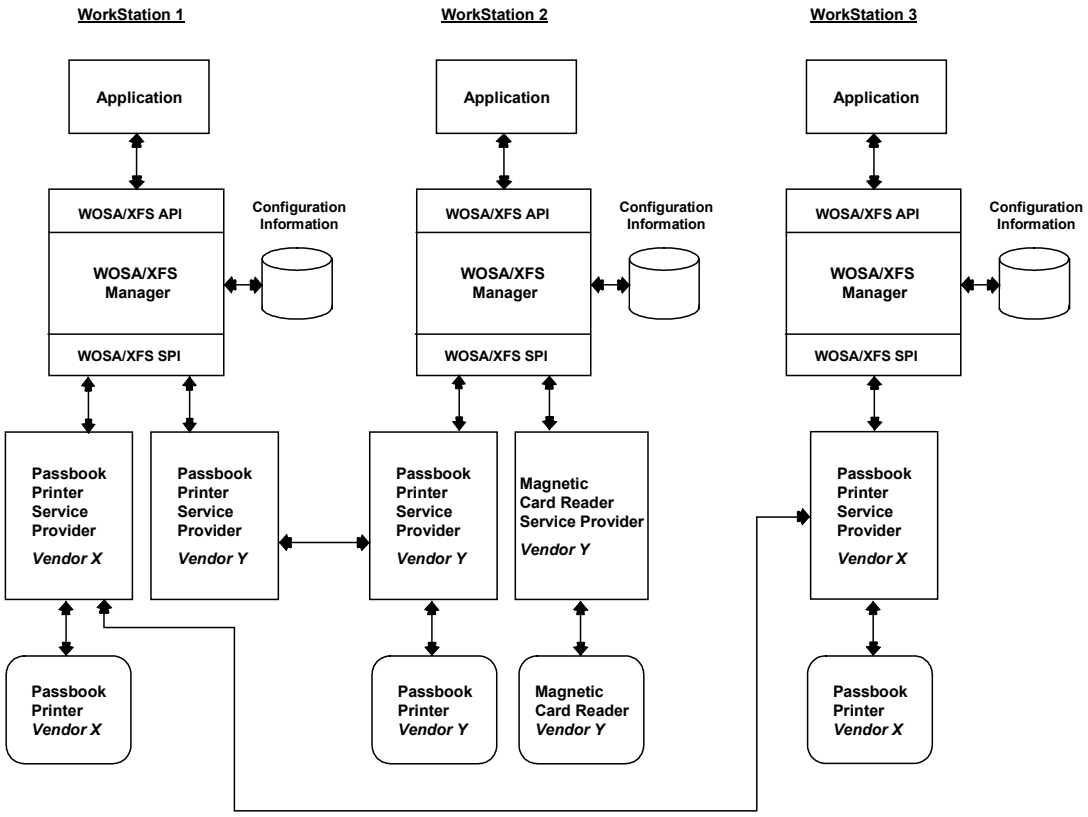


Figure 2.2 - An XFS architecture example for a branch office banking system.

It should also be noted that one vendor's Service Providers are not necessarily compatible with another vendor's, as

shown in Figure 2.2. If one application has to access the same service class as implemented by different vendors, a Service Provider is installed for each vendor.

3.2 API and SPI Summary

Sections 5 through 8 of this document present the interfaces that allow a financial application to communicate in a standard fashion with financial services and devices. The functions are at a sufficiently high level to allow for seamless redirection to other parts of the underlying operating system. A printer, for example, might rely on a set of services provided by the operating system, but in order to handle the unique characteristics of a financial printer and application, the Service Provider would pre-process the command, then redirect the derived commands to the operating system's printing services. In other implementations, the printer might be supported entirely by XFS service mechanisms, and not use the operating system printing services in any way.

The API is structured as sets of:

Basic functions, such as **StartUp**/**CleanUp**, **Open**/**Close**, **Lock**/**Unlock**, and **Execute**, that are common to all the Extensions for Financial Services device/service classes,

Administration functions, such as device initialization, reset, suspend or resume, used for managing devices and services, and

Specific commands, used to request information about a service/device, and to initiate device/service-specific functions; these are sent to devices and services as parameters of the **GetInfo** and **Execute** basic functions. These service-specific commands are specified in a set of separate specifications, one for each service class.

To the maximum extent possible, the syntax of specific commands that are used with multiple device/service classes is kept consistent across all devices. A primary objective is to standardize function codes and structures for the widest possible variety of devices.

The SPI is kept as similar as possible to the API. Some commands are processed exclusively by the XFS Manager, and so are not in the SPI, and there are minor differences in the specific parameters passed at the two interface levels.

A typical scenario showing the usage of the APIs is shown below. This example illustrates the functions used to print a form.

- **StartUp** (connects the application to the XFS Manager, including version negotiation)
  - **Open** (establishes a session between the application and the Service Provider)
    - **Register** (specifies the messages that the application should receive from the Service Provider)
      - **Lock** (obtains exclusive access to the service by the application)
        - multiple **Execute** functions, passing one or more specific commands:
        - **Print_Form**
        - etc.
      - **Unlock** (releases exclusive access to the service by the application)
    - **Deregister** (specifies that the application should no longer receive messages from the Service Provider)
  - **Close** (ends the session between the application and the Service Provider)
- **CleanUp** (disconnects the application from the XFS Manager)

Note that within a session (defined by **Open** and **Close**), an application may at any time change the classes of messages it wishes to receive from the Service Provider (using **Register**), and may either **Lock** the service only for specified periods (typically for each transaction), or for the entire session. Also, note that several of the commands are optional, depending on how the device is being managed and shared (i.e. **Lock**/**Unlock**, **Register**/**Deregister**).

3.3 Device Classes

The classes of devices that belong to the version 3.30 of the Extensions for Financial Services are described in the separate Service Class Definition Document.

3.4 Unicode Encoding Summary

If an XFS form or media file is UNICODE encoded then, consistent with the UNICODE standard [Ref. 2], the file must start with a Unicode Byte Order Mark (BOM) and the UTF-16 encoded data that follows must be in the byte order indicated by the BOM. The two-byte BOM prefix in a text file indicates a Little Endian (0xFFFE) or Big Endian (0xFEFF) notation. On a Windows operating system the byte order encoding is Little Endian.

If command parameter data is UNICODE encoded then this data will be UTF-16 encoded and the byte order must be Little Endian. UNICODE command parameter will **not** start with a BOM.

4 Architectural and Implementation Issues

The remainder of this document provides the technical specifications for the CEN~~/ISSS~~ eXtensions for Financial Services (referred to hereafter as "XFS" for brevity).

In this specification, the functions of the XFS Application Programming Interface (API) and Service Provider Interface (SPI) are always described in terms of providing a standardized, portable interface for applications to gain access to Service Providers. This architecture allows Service Providers to deliver an open-ended set of capabilities to financial applications based on the Microsoft Windows operating systems, including access to peripheral devices unique to financial institutions. Since the first priority of the CEN members for XFS implementations has been to provide this peripheral device access capability, the examples used relate primarily to device control and physical input/output.

The key elements of the Extensions for Financial Services are the API definition and the corresponding SPI definition, used by the XFS Manager to communicate with the Service Providers, together with the set of supporting services provided by the XFS Manager. These elements are combined in an XFS implementation, providing access to financial devices and services for Windows-based applications.

The specification defines a standard set of interfaces in order to provide multi-vendor interoperability: if an application uses the API to communicate successfully with a Service Provider, it should work with another conformant Service Provider of the same type, developed by another vendor, without any changes. To work with more than one hardware implementation of a device, an application must retrieve the device capability information - this will allow the application to successfully interact with different variants of the same hardware device. Applications that use the vendor specific fields of XFS commands may not be able to interact successfully with another vendor's conformant Service Provider. Applications should isolate vendor specific access to devices in order to maximize consistent device control across multiple device Service Provider implementations. Any Service Provider that conforms to the SPI definition can work with a range of conformant applications.

As new versions of the XFS device classes are developed and released, changes to the device class interface specifications are inevitable. Application exposure to these changes is controlled via the version negotiation process described later in this specification. Applications need to be updated to support new releases of XFS, but to minimize the migration effort it is recommended that they should be developed in such a way that they can handle additional error codes and new output literal values being added to existing commands within future versions of XFS in a graceful manner. In addition, applications must release the memory for all events received, this includes events that the application may be unaware at development time, i.e. the minimum processing for any XFS event must be the release of the memory associated with the event.

For clarity, three prefixes are used in naming the function interfaces in XFS:

Function type: Prefix	Functions called by	Functions provided by
• API functions: **WFS...**	• Applications	• XFS Manager (and typically passed through to **WFP** functions)
• SPI functions: **WFP...**	• XFS Manager	• Service Providers
• Support/Configuration functions: **WFM...**	• Service Providers • Applications	• XFS Manager

4.1 The XFS Manager

The XFS Manager provides overall management of the XFS subsystem. The XFS Manager is responsible for mapping the API (**WFS...**) functions to SPI (**WFP...**) functions, and calling the appropriate vendor-specific Service Providers. Note that the calls are ***always*** to a local Service Provider.

The XFS Manager determines which Service Provider to call using the logical name parameter of the **WFSOpen** or **WFSAsyncOpen** function. The logical name is the key providing access to the configuration information that defines the Service Class (e.g. printer, cash dispenser, etc.), the Service Type (e.g. receipt printer, journal printer, etc.) and the Service Provider (DLL file name), as well as additional information. The logical name must be unique at least within each workstation. See Sections 4.7 and 8 for discussions of configuration information access and management.

The XFS Manager also provides the Support Functions (**WFM...**) defined in Section 7 and the Configuration Functions (also **WFM...**) defined in Section 8.

Before an application is allowed to utilize any of the services managed by the XFS subsystem, it must first identify itself to the subsystem. This is accomplished using the **WFSStartUp** function. An application is only required to perform this function once, regardless of the number of XFS services it utilizes, so this function would typically be called during application initialization. Similarly, the complementary function, **WFSCleanUp**, is typically called during application shutdown. If an application exits or is shut down without issuing the **WFSCleanUp** function, the XFS Manager does the cleanup automatically, including the closing of any sessions with Service Providers the application has left open.

The XFS Manager's binary interface is described in section 4.16.

4.2 Service Providers

Each XFS service, for ***each*** vendor, is accessed via a service-specific module called a Service Provider. For example, vendor A's journal printer is accessed via vendor A's journal printer Service Provider, and vendor B's receipt printer is accessed via vendor B's receipt printer Service Provider.

The following sections describe the functionality and packaging of Service Providers.

4.2.1 Service Provider Functionality

The primary functions of XFS Service Providers, working in conjunction with their respective services and/or device drivers, are as follows. Note that ***how*** these functions are implemented is left to the Service Provider developer.

- Route the requests to the device or service, which may be on a remote workstation.
  Service Providers may communicate with remote services in a variety of ways, such as NetBIOS, named pipes, RPC (Remote Procedure Calls), Windows Sockets, proprietary network programming interfaces, etc.
- Translate the generic requests to resource specific commands.
  Note that this involves translation not just to service-specific commands, but to the commands native to the resource being used. For example, the commands would not be translated to "Receipt Printer Service" commands, but to "Brand X, Model Y Receipt Printer" commands. For example, a driver may implement device-specific translation tables or processes itself, or utilize standard operating system device interfaces (such as the Windows GDI), if they exist for the particular peripheral.
- Arbitrate access to the resource by multiple applications.
  Note that when a physical device includes multiple peripherals (for example, a receipt and journal printer in a single unit), this may also include arbitration of the sub-devices.
- Manage the interface to the resource.
  When physical devices are being controlled, this includes managing the hardware interface to the device. For example, the Service Providers may use standard operating system device drivers, vendor-written proprietary device drivers, etc.
- Manage the asynchronous nature of the services in a consistent manner with respect to the applications.
  The asynchronous nature of the SPI must always be presented back to the XFS Manager and the applications in the form of Windows messages.
- Error recovery.
  In some kinds of software failures, such as an application crash, the Service Provider loses connection with the application. In this situation, the Service Provider is responsible for an “orderly” shutdown of the session with that application. In particular, the Service Provider generates a system event (see Section 4.11) indicating that the connection was lost, and if any requests from the application were outstanding, it generates a system event for each completion that would normally have generated a completion message to the application.

4.2.2 Service Provider “Packaging”

XFS Service Providers can be “packaged” into DLLs in a variety of ways:

- One Service Provider per DLL; for example, a vendor might produce a journal printer DLL, a receipt printer DLL, a cash dispenser DLL, etc.
- Multiple Service Providers per DLL; for example a vendor might produce a DLL which contains the Service Providers for all XFS-compliant printers.
- All Service Providers for a specific vendor in a single DLL.

4.3 Asynchronous, Synchronous and Immediate Functions

Windows and XFS are built on an event-driven, asynchronous model. However, the XFS design allows an application using its interfaces to behave in either an asynchronous or synchronous manner. Thus the API supports two versions of each of the appropriate functions (e.g. an application can request to lock a service using either the asynchronous **WFSAsyncLock** function or the synchronous **WFSLock** function).

Each XFS API function operates in one of three synchronization modes: asynchronous, synchronous or immediate. These are described in the following sections.

Note that the SPI is purely an asynchronous interface, so all SPI functions are either asynchronous or immediate; there are no synchronous SPI functions.

See Sections 5 and 6 for a summary of the API and SPI functions and their synchronization modes.

4.3.1 Asynchronous Functions

Asynchronous mode is used for operations which may take an indeterminate amount of time to complete. Performing an operation in an asynchronous, as opposed to a synchronous, mode allows the application to operate in Windows' native event-driven, message-based manner. The processing of an asynchronous request (e.g. **WFSAsyncExecute)** is as follows:

The application calls the XFS Manager.
The XFS Manager generates a sequence number, the *RequestID*, assigns it to the request, and calls the Service Provider.
The Service Provider schedules the request for deferred processing and immediately returns to the XFS Manager.
The XFS Manager returns the *RequestID* to the application, with a status indicating that the request has been initiated and is being processed.
At some point, the Service Provider processes the deferred request.
On completion, the Service Provider posts a completion message to the window handle specified by the application in its original call. For flexibility, an application using asynchronous functions can specify a different window for each request. The message contains a pointer to a WFSRESULT data structure defining the results of the request, including the *RequestID*, the status code and the other relevant data.

4.3.2 Synchronous Functions

Synchronous mode is also used when an operation can take an indeterminate amount of time to complete, but the application wishes to handle the function in a sequential manner. The XFS Manager does not return control to the application until the operation has completed, thus synchronous functions are referred to as blocking. Each synchronous call made by an application is translated by the XFS Manager into its asynchronous SPI counterpart before being passed to the Service Provider.

If a blocking operation is not completed immediately ~~in a Windows 3.x system~~, the XFS Manager executes a Windows message loop on behalf of the calling thread, thereby keeping the Windows system running. See Section 4.12 for a more detailed discussion of process, threads and message loops. ~~In Windows NT, the~~The calling application thread is blocked on request completion. A thread may have only ***one*** blocking XFS call outstanding at any one time. See Section 4.12 for additional discussion of the management of synchronous functions, including replacement of the default message loop.

The processing of a synchronous request (e.g. **WFSExecute)** is as follows:

- The application calls the XFS Manager.
- The XFS Manager translates the request into an asynchronous SPI, generates a *RequestID* to track the request, provides its own window handle to receive the completion message, and calls the Service Provider DLL.
- The Service Provider schedules the request for deferred processing and immediately returns to the XFS Manager.
- The XFS Manager simulates synchronous processing as described above and in Section 4.12.
- At some point, the Service Provider processes the deferred request.
- On completion, the Service Provider posts a completion message to the window handle specified by the XFS Manager. The message contains a pointer to a WFSRESULT data structure defining the results of the request, including the *RequestID*, the status code and the other relevant data.
- The XFS Manager unpacks the information from the completion message into the appropriate parameters, and

returns them to the application, unblocking the original application request.

4.3.3 Immediate Functions

These are API functions that are not either asynchronous or synchronous. Typically, immediate APIs are those which do not communicate with a service or a physical device (or use the network in any other way) and are thus guaranteed to complete immediately, whether successfully or not. They are handled in two ways:

Processed entirely by the XFS Manager, which returns immediately to the application. Examples include **WFSStartUp**, and **WFSSetBlockingHook**.

Passed by the XFS Manager to the Service Provider as an immediate SPI. The Service Provider processes the request and immediately returns to the XFS Manager, which returns immediately to the application. Examples include **WFSCancelAsyncRequest** and **WFMSetTraceLevel**.

4.4 Processing API Functions

When an application calls an XFS API function one of the following processing scenarios takes place. Note that this classification is distinct from the API synchronization modes discussed above. See Section 6 for the mapping of API functions to SPI functions.

- The function is converted by the XFS Manager directly into the corresponding SPI function (e.g. **WFSAsyncRegister**).
- The XFS Manager performs some preprocessing and then converts the function into the corresponding SPI function (e.g. **WFSAsyncExecute**).
- The XFS Manager performs some preprocessing and then translates the API function to a different SPI function, which it passes to the Service Provider. Most of the synchronous API functions (e.g. **WFSLock**) are of this type, since they are translated to their asynchronous SPI equivalents.
- The XFS Manager performs some preprocessing and then translates the API function to multiple SPI functions, which it passes to the Service Provider (e.g. **WFSOpen**).
- The function is completely processed inside the XFS Manager (e.g. **WFSIsBlocking, WFSSetBlockingHook**).

Service Providers (and sometimes applications) call the XFS Manager for the support functions defined in Section 7 and for the configuration functions defined in Section 8.

4.5 Opening a Session

Once a connection between an application and the XFS Manager has successfully been negotiated (via **WFSStartUp**), the application establishes a virtual session with a Service Provider by issuing a **WFSOpen** (or **WFSAsyncOpen)** request. Opens are directed towards "logical services" as defined in the XFS configuration. A service handle (*hService*) is assigned to the session, and is used in all the calls to the service in the lifetime of the session.

Note that applications may optionally choose to explicitly manage the concept of "application identity" when they need to use interdependent compound devices (see Section 4.8.2). This is achieved by using the **WFSCreateAppHandle** function to get an application handle (*hApp*), which is unique within the system. This function can be called multiple times to obtain multiple unique handles. An application handle parameter is then used in the **WFSOpen** function, directing the Service Provider to bind the specified application handle to the session being initiated. This allows a single application process (potentially multi-threaded) to act as multiple applications to the XFS subsystem, to allow effective use of interdependent compound devices. An example of a case in which this could be useful is an application using the Multiple Document Interface (MDI); the application could associate an application handle with each MDI child window. See Section 4.8.2 for additional discussion of the use of application handles with compound devices. Note that neither service nor application handles may be shared among two or more applications.

The actions performed by the XFS Manager on an open are as follows:

Retrieves the configuration information defining the specified logical service, in order to determine the DLL name of the Service Provider. The logical service name is the key to the configuration information.
Loads the DLL containing the requested Service Provider, if it is not already loaded.
Performs pre-processing and translation as necessary, depending on whether the synchronous or asynchronous open API has been issued.
Generates a unique service handle (*hService*) that identifies the session with the Service Provider that is being established, to be passed back to the application as a parameter.
Calls the Service Provider's **WFPOpen** function, passing the parameters needed.

The Service Provider does the following:

Performs version negotiation, using the parameters specifying the SPI version requested by the XFS Manager, and the service-specific interface version requested by the application.
Retrieves the configuration information.
Asynchronously establishes a session with the service specified in the configuration on the specified workstation, if necessary, relying on the transport facilities provided.
Upon completion of the request, posts a completion message (WFS_OPEN_COMPLETE), which goes to the application for a **WFSAsyncOpen** call, and to the XFS Manager for a **WFSOpen** call.

Note that even if the service is locked by another application, the open function succeeds, as defined in Section 4.8, "Exclusive Service and Device Access."

An application programmer has at least two obvious choices as to when to perform the **WFSOpen** (and the complementary **WFSClose**) of the services it utilizes:

- Open the services during application initialization, keep them open, and close them during application shutdown.
- Perform the open each time the service is required, utilize it, and immediately close it.

Each technique has its own advantages. For example, while the first example might provide better performance, the second might be easier to program. In any case, upon a successful completion of an open, the XFS subsystem returns a service handle which must be used for all subsequent communication with the service.

Note that an application must perform an open for ***each*** logical service that it wishes to utilize, even if the services are of the same type. For example, if an application wishes to utilize two separate receipt printers, it must open two separate logical services.

Furthermore, an application may need to open multiple logical services, even when a set of devices are housed in a single device. For example, consider a compound printer which includes both a receipt and a journal printer. If the application requires access to both the receipt and journal printer functions, it must open both a receipt logical service and a journal logical service.

4.6 Closing a Session

When an application no longer requires the use of a particular service, it issues a **WFSClose** or **WFSAsyncClose** request. The XFS subsystem then closes that session as follows:

The XFS Manager calls the Service Provider's **WFPClose** function.

The Service Provider schedules the request for deferred processing, and returns immediately to the XFS Manager. Note that at this point the service handle, *hService*, is no longer valid.

At some point, the Service Provider processes the deferred close request, communicating with the service as necessary to accomplish the request.

Requests that were issued by the application before the close are executed.

If the calling application has the service locked under the same *hService*, the Service Provider unlocks it automatically (following the standard lock policy as defined in Section 4.8).

The service cleans up its administrative information (removes **WFSRegister** entries etc.).

If the XFS subsystem loses connection to an application, it closes the session as described above, and:

An “application disconnect” event (SYSTEM_EVENT class) is generated.

Since messages can no longer be posted to the application, any command completion and event notification messages from this service for the application are converted to “undeliverable message” events (SYSTEM_EVENT class).

Note that it is required that some applications have registered for system events, or these events are effectively not reported.

When a Service Provider receives a Close request for a session, its behavior may vary as follows,

When the session has no outstanding requests the Service Provider will complete the Close request (even if it is executing a command from another session or has outstanding deferred requests from another session).

When the session that issues the close request has an outstanding request then the Service Provider will defer the Close until all outstanding requests are complete.

4.7 Configuration Information

The XFS Manager uses its configuration information to define the relationships among the applications and the Service Providers. In particular, this information defines the mapping between the logical service interface presented at the API (via logical service name) and the appropriate Service Provider entry points.

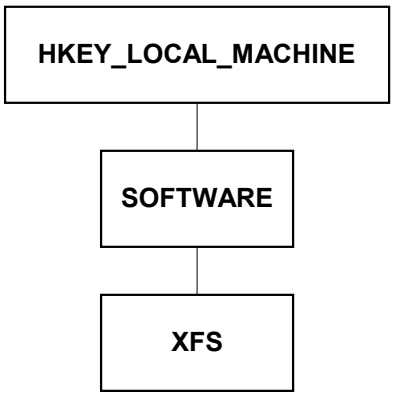
The configuration information also includes specific information about logical services and Service Providers, some of which is common to all solution providers; it may also include information about physical services, if any are present on the system, and vendor-specific information. The location of the information is transparent to both applications and Service Providers; they always store and retrieve it using the configuration functions provided by the XFS Manager, as described in Section 8, for portability across Windows platforms.

It is the responsibility of solution providers, and the developers of each Service Provider, to implement the appropriate setup and management utilities, to create and manage the configuration information about the XFS subsystem configuration and its Service Providers, using the configuration functions.

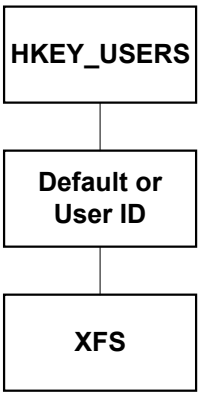
These functions are used by Service Providers and applications to write and retrieve the configuration information for an XFS subsystem, which is stored in a hierarchical structure called the Windows Registry. The structure and the functions are based on the Win32/Win64 Registry architecture and API functions, and are implemented ~~in Windows NT/98 and future versions of Windows~~ using the Registry and the associated functions.

Each node in the configuration registry is called a key, each having a name and (optionally) values. All values consist of a name and data pair, both null-terminated character strings. There are two logical groupings of XFS Registry information; local PC dependent configuration information and user dependent configuration information.

The local PC dependent configuration information is stored beneath the following Registry key. A pre-defined handle (WFS_CFG_HKEY_MACHINE_XFS_ROOT) can be used to access this key in the configuration functions defined in Section 8.



User dependent configuration information is stored in the HKEY_USERS section of the Registry. Pre-defined handles (WFS_CFG_HKEY_USER_DEFAULT_XFS_ROOT and WFS_CFG_CURRENT_USER_XFS_ROOT) can be used to access these keys in the configuration functions defined in Section 8.



Within the local PC dependent configuration information are stored the following XFS related keys;

XFS_MANAGER - Beneath this key are values and/or keys for information that the XFS Manager creates and uses.

SERVICE_PROVIDERS - Beneath this key is a key for each XFS compliant Service Provider.

PHYSICAL_SERVICES - Beneath this key are physical attachment configuration information, defined by the solution provider.

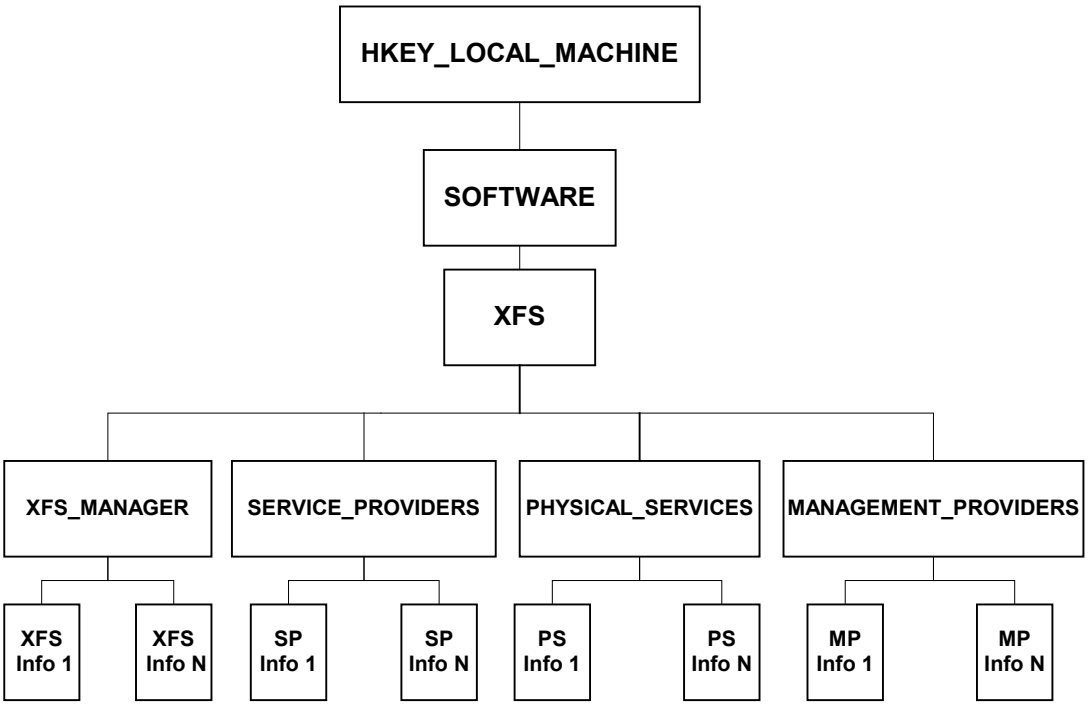
MANAGEMENT_PROVIDERS - Reserved for XFS SNMP Management. Beneath this key is a key for each XFS SNMP Managed Service.

Within the user dependent configuration information is stored the following LOGICAL_SERVICES key:

- LOGICAL_SERVICES - Beneath this key is defined a key for each XFS logical service (i.e.: the *lpszLogicalName* parameter of the **WFSOpen**, **WFSAsyncOpen** and **WFPOpen** functions).

The configuration functions provide the capabilities to create, enumerate, open and delete keys, and to set, query and delete values within each key. Vendor-provided configuration utility programs set up the registry structure and its contents, using these functions. Configured Registry values and keys define how the XFS subsystem, services and providers are configured. These are used by the XFS Manager, applications and Service Providers. Note that vendor-specific information may be added to any key in this structure, using optional values.

The figure below illustrates the full structure of the local PC dependent configuration information.



The XFS_MANAGER key has the following optional values:

- TraceFile — the name of the file containing trace data. If this value is not set in the configuration, trace data is written to the default file path\name C:\XFSTRACE.LOG.
- ShareFilename — the name of the memory mapped file used by the memory management functions of the XFS Manager.
- ShareFilesize — the size of the memory mapped file used by the memory management functions of the XFS Manager.
- ShareMapAddr — the address of the beginning of the XFS Manager Shared Memory. Care should be taken when using this value to control the load address of shared memory. When used, the address chosen should be consistently accessed across all XFS processes. A value of zero will result in the shared memory allocation being dynamic.

Some additional values may also be defined in the implementation of the XFS Manager. Please refer to the related document for more information.

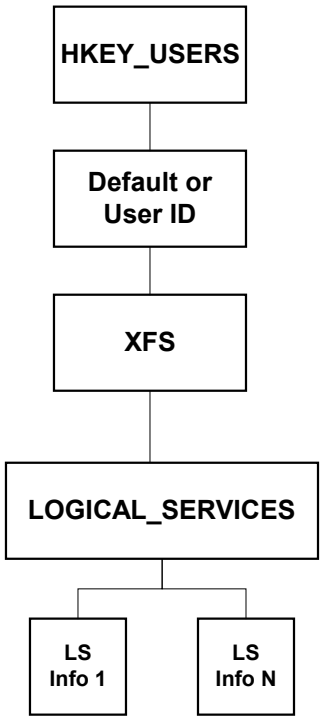
Beneath the SERVICE_PROVIDERS key there are keys for each individual Service Providers, the keys are the Service Provider names. Each of these keys has three mandatory values:

- dllname — The name of the file containing the Service Provider DLL.
- vendor_name — The name of the supplier of this Service Provider.
- version — The version number of this Service Provider. This version number is a vendor specific Service Provider implementation version; it has no relation to the version of the standard.

The PHYSICAL_SERVICES keys are fully vendor dependent.

Beneath the MANAGEMENT_PROVIDERS key there are keys for each XFS SNMP Managed Service, the keys are the managed service names. The structure of these keys is defined within the XFS MIB Architecture specification.

The figure below illustrates the full structure of the user dependent configuration information.



Beneath the LOGICAL_SERVICES keys there are keys for each individual Service Provider the keys are the logical service names: Each of these keys have two mandatory values:

- class — the service class of the logical service; (see the Service Class Definition Document for the standard values)
- provider — the name of the Service Provider that provides the logical service (the key name of the corresponding Service Provider key)

The ‘User Id’ key is only applicable to the Windows Terminal Server platform. The ‘User Id’ is the user name associated with the session in which the application is executing.

An example of the content of the configuration information for is shown below. See Section 8 for the definitions of the configuration functions.

```
[HKEY_USERS\.DEFAULT\XFS\LOGICAL_SERVICES\MyCurrencyDispenser]
"class"="CDM"
"provider"="CDM"

[HKEY_USERS\.DEFAULT\XFS\LOGICAL_SERVICES\MyCardReader]
"class"="IDC"
"provider"="IDC"

[HKEY_USERS\.DEFAULT\XFS\LOGICAL_SERVICES\MyJournalPrinter]
"class"="PTR"
"provider"="JPTR"

[HKEY_USERS\.DEFAULT\XFS\LOGICAL_SERVICES\MyPassbookPrinter]
"class"="PTR"
"provider"="PPTR"

[HKEY_USERS\.DEFAULT\XFS\LOGICAL_SERVICES\MyPinpad]
"class"="PIN"
"provider"="PIN"

[HKEY_USERS\.DEFAULT\XFS\LOGICAL_SERVICES\MyReceiptPrinter]
"class"="PTR"
"provider"="RPTR"

[HKEY_USERS\.DEFAULT\XFS\LOGICAL_SERVICES\MyStatementPrinter]
"class"="PTR"
"provider"="SPTR"

[HKEY_LOCAL_MACHINE\SOFTWARE\XFS\SERVICE_PROVIDERS\CDM]
"dllname"="C:\ Program Files \ABCTech\XFS PRODUCT\XFS CDM Service Provider\ABCTech_9827SP.dll"
"vendor_name"="ABCTech Corporation"
"version"="1.0.0"

[HKEY_LOCAL_MACHINE\SOFTWARE\XFS\SERVICE_PROVIDERS\IDC]
"dllname"="C:\Program Files\ABCTech\XFS PRODUCT\XFS IDC Service Provider\ABCTech_1212SP.dll"
"vendor_name"="ABCTech Corporation"
"version"="1.0.1"

[HKEY_LOCAL_MACHINE\SOFTWARE\XFS\SERVICE_PROVIDERS\JPTR]
"vendor_name"="ABCTech Corporation"
"dllname"="C:\Program Files\ABCTech\XFS PRODUCT\XFS PTR Service Provider\ABCTech_9001SP.dll"
"version"="1.2.4"

[HKEY_LOCAL_MACHINE\SOFTWARE\XFS\SERVICE_PROVIDERS\PIN]
"dllname"="C:\Program Files\ABCTech\XFS PRODUCT\XFS PIN Service Provider\ABCTech_1234SP.DLL"
"vendor_name"="ABCTech Corporation"
"version"="1.34.8"

[HKEY_LOCAL_MACHINE\SOFTWARE\XFS\SERVICE_PROVIDERS\PPTR]
"dllname"="C:\Program Files\ABCTech\XFS PRODUCT\XFS PTR Service Provider\ABCTech_2411SP.dll"
"vendor_name"="ABCTech Corporation"
"version"="1.2.3"

[HKEY_LOCAL_MACHINE\SOFTWARE\XFS\SERVICE_PROVIDERS\RPTR]
"dllname"="C:\Program Files\ABCTech\XFS PRODUCT\XFS PTR Service Provider\ABCTech_1028SP.dll"
"vendor_name"="ABCTech Corporation"
"version"="1.9.4"

[HKEY_LOCAL_MACHINE\SOFTWARE\XFS\SERVICE_PROVIDERS\SPTR]
"dllname"="C:\Program Files\ABCTech\XFS PRODUCT\XFS PTR Service Provider\ABCTech_1028SP.dll"
"vendor_name"="ABCTech Corporation"
"version"="1.9.4"
```

Notes:

In the above example the receipt and statement printer services are all implemented through a single physical printer and Service Provider DLL. The Service Provider determines which type of service the application has requested by the vendor specific configuration information.

4.8 Exclusive Service and Device Access

This section describes how application access to services and devices is handled by XFS subsystems, using the lock facility. It discusses the meaning of timers within the context of a lock request and issues that arise when multiple applications have issued lock requests. It also describes how requests that were submitted to the Service Provider prior to a lock request are managed. Furthermore, it describes how compound devices (physical devices that include two or more logical devices, such as a passbook printer that also includes a magnetic stripe reader) are handled.

Typically, an application requires ***exclusive*** access to a particular service when it is about to utilize it, particularly in combination with other services. For example, an application may need to use a PIN pad, magnetic stripe reader, receipt printer and journal printer to complete a transaction. The application must be guaranteed that it has access to ***all*** the devices before starting on the transaction, and that no other application will be able to use them until the transaction is complete and it has explicitly released them. This is accomplished by using the **WFSLock** (or **WFSAsyncLock**) function and the complementary **WFSUnlock** function.

An application should act in a cooperative manner when locking a service, by keeping it locked for the minimum time period that it requires exclusive access to the service. Typically, this means locking a set of services, performing a series of requests to the services to complete a transaction, and immediately unlocking the services.

However, an application which has obtained a lock on a device will be informed via the WFS_SYSE_LOCK_REQUESTED system event whenever another application requests a lock on the device (i.e. potentially multiple lock request events will occur - one for each request by another application). Therefore an alternative strategy is for the application to register for system events and unlock the device only when it receives the event notification that another application has requested a lock on the device.

Applications must use appropriate techniques to avoid deadlock when locking multiple services, typically by making use of the timeout parameter in the lock functions.

Also, note that there are cases in which exclusive access is not a requirement, so that it is not always required that an application lock a service before issuing execute operations to it.

The lock policy describes the rules that services use in managing lock requests. In the description of this policy, XFS requests are categorized into three types:

- **Non-deferred**: Requests that can be processed completely by a service as soon as they arrive (e.g. **WFPOpen**, **WFPRegister** and most **WFPGetInfo** calls.
- **Deferred**: Requests which may not be able to be processed completely as soon as they arrive, typically because they require hardware and/or operator interaction (e.g. **WFPExecute** and some **WFPGetInfo** calls).
- **Lock**: **WFPLock** calls.

The lock policy is described first for independent devices, i.e. logical services that correspond to devices whose operation is not interdependent with any other (even though they may be housed in the same physical enclosure). The following section describes the special requirements involved in managing compound interdependent devices.

4.8.1 Lock Policy for Independent Devices

The following describes how the categories of requests are handled, in each of the lock states of a service. Note that although the description refers to queues and other implied implementation characteristics, this is only for convenience; no particular implementation techniques are required.

Service state: UNLOCKED

- Non-deferred requests are processed on arrival.
- Deferred requests are placed in the deferred queue and processed FIFO.
- When a **WFPLock** request arrives:
  - The lock request is placed in the lock queue.
  - The service state changes to LOCK_PENDING.

Service state: LOCK_PENDING

- All requests in the deferred queue that arrived ***before*** the pending lock request are processed FIFO; after all are processed, the lock queue is processed. Note that depending on the nature of the service/device, lock requests may be granted FIFO or in some other order, e.g. when an operator takes an action such as pressing a station button.
- When a lock request has been granted:
  - The service state changes to LOCKED.
  - Any other pending lock requests from the same "owner" are also granted. (The owner is the same if it comes from the same workstation and has the same application and service handles.)

Service state: LOCKED

Arriving requests (except lock requests) are handled as follows:

Non-deferred requests are processed on arrival.

Deferred requests that are ***not*** **WFPExecute** requests are placed in the deferred queue.

WFPExecute requests from the owner of the lock are placed in the deferred queue.

WFPExecute requests that are not from the owner of the lock are rejected (with error code WFS_ERR_LOCKED).

WFPUnlock and **WFPClose** requests from the owner of the lock are placed in the deferred queue. (Note that a close request to a locked service is treated as an unlock followed by a close.)

WFPUnlock and **WFPClose** requests that are ***not*** from the owner of the lock are treated as non-deferred requests, i.e. processed on arrival.

The deferred queue is processed FIFO.

When a **WFPLock** request arrives:

If it is from the owner of the lock, it is granted.

If it is not from the owner of the lock, it is placed in the lock queue, a WFS_SYSE_LOCK_REQUESTED event is posted to the owner of the lock.

When a **WFPUnlock** or **WFPClose** request is processed from the deferred queue, or the connection between the service and the owner of the lock is lost:

If the lock queue is not empty, the service state changes to **LOCK_PENDING**.

If the lock queue is empty, the service state changes to **UNLOCKED**.

Note that most requests include a timeout parameter which must be managed appropriately, i.e. when the specified time expires, the request is rejected with the error code WFS_ERR_TIMEOUT. The timeout parameter is particularly important with the **WFSLock** request, since it allows applications to set a maximum time to wait for a lock to be granted, to allow prevention of deadlock situations when requesting locks of multiple devices.

4.8.2 Compound Devices

Compound devices are very common in the financial services industry. For the purposes of this discussion, there are three types of compound devices:

Two or more separate logical devices that share a physical housing (or perhaps some other attribute), but function completely independently of one another.

Two or more distinct logical devices that are functionally interdependent in some way, such as a journal printer and passbook printer that use the same print head mechanism.

Two or more logical devices that are simply different logical views of a single physical device, such as a single printer that is managed as two separate logical devices, a document printer and a passbook printer.

The first of these types has no special significance from the XFS point of view. Each of the devices is managed as a separate logical and physical device, and the system configuration issues (e.g. making sure that devices that are packaged together are assigned to the same workstation) are left to application utilities outside the scope of this specification.

The latter two types are treated identically in an XFS system. When any one of a set of interdependent logical devices that forms a compound device is locked, all the other logical devices in that compound device are also ***implicitly*** locked on behalf of the requesting application. (The specific policy is described below.) If the ***same*** application (see the discussion of "application identity" below and in Section 4.5) explicitly requests a lock of another of these logical devices, the lock is granted. In order to allow the application to "know" that the devices are part of a compound device, and therefore interdependent, the **WFSLock** function returns an array of service handles, defining the set of other devices within the compound device that are now explicitly locked by the application. This allows the application to manage its use of these devices accordingly. Normally, it must use them in a strictly sequential manner to avoid any possible conflicts, but if it has some special knowledge of how the devices are related, it may be able to multiplex requests in some ways.

Note that an application can also determine whether a device is compound by using the device capabilities query function of **WFSGetInfo**.

There are many different ways in which programmers can make use of multiple threads and/or processes in financial applications. Each XFS service can be controlled from its own thread; all services can be controlled from a single thread, with other threads/processes used for other application functions; several identical threads can handle all open services as needed; etc. In some of these models, the "user" of a service could be considered to be the process as a whole; in other models, the "user" is a single thread. The XFS design allows for both models by providing the programmer the capability to explicitly control the "identity" of an application. The programmer can make all the threads in a process appear to a Service Provider as one "application," identify each thread as a different "application," or create some hybrid of these approaches, allowing interdependent compound devices to be managed correctly no matter what application architecture is used.

In order to allow this flexibility in application architecture, the "identity" of an application can optionally be managed explicitly using the concept of application handles. An application handle (*hApp*) is created using the **WFSCreateAppHandle** function, and is guaranteed unique within the system. The **WFSOpen** function takes an optional application handle parameter which is bound to the service handle (*hService*) returned by the open function. This approach allows applications that use interdependent compound devices to be implemented with any combination of single or multiple processes and/or threads, by explicitly managing an appropriate set of application handles. If this facility is not used (indicated by the application using the value WFS_DEFAULT_HAPP for the *hApp* parameter in **WFSOpen**), the XFS subsystem automatically treats each process as having a single, unique application handle. See Section 4.5 for additional discussion of this topic.

The lock policy for interdependent compound devices uses the same rules as for independent devices, with some additional constraints. In order to synchronize access via multiple logical services to a single physical device, or to interdependent devices, the service manages a single lock queue and a single deferred queue for the set of related logical services. The additional constraints are:

Service state: LOCK_PENDING

When a lock request has been granted to one of a set of related logical services:

All the other related services in the set change to a "reserved" state in which they are treated as being in the LOCKED state for requests not from the owner.

Any lock request from the owner for one of the reserved services is granted on arrival.

Lock requests that are not from the owner of the reserved devices are placed in the lock queue.

Service state: LOCKED

Any lock request from the owner for one of the reserved services is granted on arrival.

Lock requests that are not from the owner of the reserved devices are placed in the lock queue.

Note that if a **WFPUnlock** or **WFPClose** request is processed for the service, and any other logical service that is related to this service is in the LOCKED state, then the service state is set to "reserved," ***not*** UNLOCKED.

Note also, that if a **WFPUnlock** or **WFPClose** request is processed for the service, and the other logical services that are related to this service are in the "reserved" state, then all these services change to the UNLOCKED state.

4.9 Timeout

There are two fundamentally different time domains in a system, each having a different implication on the concept of timeout:

"user time" = real time; timeout here says simply "this job is taking too long" as defined by the application and/or the user (indicated by a WFS_ERR_TIMEOUT error code).

"service time" = the time taken by the service request ***within*** the service; typically, the physical device operation (indicated by WFS_ERR_DEV_NOT_READY or WFS_ERR_HARDWARE_ERROR error code).

In XFS systems, the service manages the latter, ***without*** needing any input from the application, since it "knows" the characteristics of the device, and can generate a timeout event if the device takes too long, even if the application timeout value (if any) has not been exceeded. Therefore, the timeout value provided in the API is treated by the Service Provider as user/real time. If the time is exceeded, the Service Provider cancels the request and returns a timeout event to the application. An application can also specify that a request should wait until completion, no matter how long the request takes, by specifying the special value WFS_INDEFINITE_WAIT.

4.10 Function Status Return

When an XFS API or SPI function call completes, it returns a value that either defines the completion status, or in the case of asynchronous functions, the status of the initial processing of the request. When an asynchronous function completes, the completion message includes the final status of the request. The return value of most functions is a "result handle," *hResult*, of type HRESULT. *hResult* values are defined to be WFS_SUCCESS (zero) for success; other values indicate the specific error that occurred, as defined in each function specification.

The XFS Manager and the Service Providers return status from a function call, in the form of an *hResult* result handle, in two manners:

By returning an *hResult* value as the function return.

By posting a completion message to the window specified in the request. The message contains a pointer to a structure that includes the *hResult*.

The mechanism depends on the category of function being processed, as follows:

Immediate API

The XFS Manager processes the request, and immediately returns a result handle. In some cases, the XFS Manager calls the Service Provider to process the request, then returns the result handle from the Service Provider to the application.

Asynchronous API

Since the processing is performed in a number of steps, as described earlier, return status is generated at a number of levels:

The Service Provider performs any validations which can be processed immediately.

If an error is detected, the Service Provider returns the *hResult* to the XFS Manager, which immediately returns it to the application.

Otherwise, the request is scheduled and an *hResult* of WFS_SUCCESS is immediately returned to the XFS Manager, which immediately returns it to the application. This informs the application that the request has been accepted and is being processed.

Upon completion of the deferred request, a completion message is posted to the application's window. This message points to the structure that includes the *hResult* indicating the completion status of the request.

Synchronous API

Since a synchronous API call is translated by the XFS Manager to an asynchronous SPI, the Service Provider behaves the same as in asynchronous API processing. Specifically, the Service Provider performs any validations which can be processed immediately.

If an error is detected, the Service Provider returns the *hResult* to the XFS Manager, which immediately returns it to the application.

Otherwise, the request is scheduled and an *hResult* of WFS_SUCCESS is immediately returned to the XFS Manager, indicating that the request has been accepted and is being processed.

Upon completion of the deferred request, a completion message is posted to the XFS Manager window. The XFS Manager retrieves the *hResult* from the structure pointed to by the message and returns it to the application.

4.11 Notification Mechanisms - Registering for Events

The **WFSRegister** and **WFSDeregister** functions (and their asynchronous counterparts) are used to register and deregister the window procedures which are to receive Windows messages when particular unsolicited, asynchronous events occur, either during request processing or at other times. In other words, they are used to enable or disable the reception of event notifications. By providing notifications of this type to applications, the requirement to poll for status is removed, and a simple method for implementing "monitoring" applications is provided. Each **WFSRegister** call specifies a service handle (*hService*), one or more event classes, and an application window handle (*hWnd*) which is to receive all the messages of the specified class(es). The corresponding SPI functions, **WFPRegister** and **WFPDeregister**, implement the API functions.

There are four classes of events:

- SERVICE_EVENTS
- USER_EVENTS
- SYSTEM_EVENTS
- EXECUTE_EVENTS

For the first three of these event classes, if a class is being monitored and an event occurs in that class, a message is broadcast to every *hWnd* registered for that class, containing the service handle of the session that the event is sent to. The exception to this is the WFS_SYSE_LOCK_REQUESTED system event, this event is posted only to the application which owns the lock on the device. The events are generated when:

- The service status changes (SERVICE_EVENTS), e.g. a printer is suspended or is no longer available.
- The service needs an operation from the user to take place (USER_EVENTS), e.g. a device needs “abnormal” attention, such as adding paper or toner to a printer.
- A system event occurs (SYSTEM_EVENTS), e.g. a hardware error occurs, a version negotiation fails, the network is no longer available or there is no more disk space.

The EXECUTE_EVENTS class is different from the other three. These are events which occur as a normal part of processing a **WFSExecute** command and they are always sent before the completion of the command. Examples include the need to interact with the user or operator to request an action such as inserting a passbook into a printer, “swiping” a magnetic stripe card, etc. A message generated by one of these events is sent ***only*** to the application that issued the **WFSExecute** that caused the event, even though other applications are registered for EXECUTE_EVENTS. In this case an application is defined as all window handles associated with the *hService* through a **WFSRegister** call requesting EXECUTE_EVENTS. Note that an application must explicitly register for these events; if it has not, and such an event occurs, the event is not deliverable and the **WFSExecute** completes normally.

The logic of **WFSRegister** is cumulative: for a given service the number of notification messages sent may be increased by specifying additional event classes. Since the XFS Manager does not keep track of what events the application is registered for and the logic of the register/deregister mechanism is cumulative, the Service Providers are responsible for implementing the logic of this process.

An application requests registration for more than one event class in a single call by using a logical ‘OR’:

```
hr = WFSRegister( hService,USER_EVENTS|SERVICE_EVENTS,hWnd );
```

Note that services always monitor their resources, regardless of whether any application has registered for event monitoring or not. Issuing **WFSRegister** simply causes a service to send notifications to the Service Provider, which, in turn, sends notifications to one or more applications.

To communicate to the XFS Manager that it no longer wishes to receive messages in one or more event classes, an application can cancel any previous registration using the **WFSDeregister** function. The logic of **WFSRegister** and **WFSDeregister** is symmetric: the application can deregister one or more classes of events monitored for each window, by properly specifying them in the parameter list. To deregister completely (e.g. every event class for every window), an application uses NULL event class and window handle values in the parameter list.

Although the **WFSDeregister** takes effect immediately, it is possible that messages may be waiting in the application's message queue. A robust application must therefore be prepared to receive event messages even after deregistration.

Note that an event notification message always passes the information describing the event to an application by pointing to a **WFSRESULT** data structure. After the application has used the data in the structure, it ***must*** free the memory that the Service Provider allocated for the **WFSRESULT** data structure, using the **WFSFreeResult** function. The *hResult* field of the **WFSRESULT** structure is not used unless the event is a command completion event or explicitly defined in this specification.

4.12 Application Processes, Threads and Blocking Functions

~~An application process contains one or more threads of execution. The XFS interface is designed to work in both the single-threaded versions of the Windows operating systems (Windows 3.1 and Windows for Workgroups) and in the multi-threaded versions of Windows (Windows NT and future versions of Windows). All references to threads in this document refer to actual threads in multi-threaded Windows environments. In single-threaded environments, "thread" is synonymous with "process."~~

Within the XFS Manager, a blocking (synchronous) function is handled as follows:

1. The XFS Manager ~~initiates~~ creates a transitory HWND on the calling thread to receive the completion message for the operation~~, and then enters~~ e.g. WFS_EXECUTE_COMPLETE.
2. The XFS Manager calls the Service Provider WFP API, passing the transitory HWND.

The XFS Manager waits for the completion message to be received. It does this by entering a loop ~~in which it dispatches any Windows~~equivalent to the following pseudo code, calling the current blocking hook (a Windows message dispatch routine) waiting for the completion message to be received from the Service Provider.

```
for(;;) {
/* flush messages (thus yielding the processor to other applications as necessary) and checks for the completion of the operation. When the operation completes, orfor good user response */
for(;;) {
BlockingHook();
/* check for WFSCancelBlockingCall is invoked, the blocking () */
if ( operation_cancelled() )
    break;
/* check to see if operation completed */
if( operation completes with an appropriate result.
```

3. ~~When a Windows message is received for a thread for which~~_completed())
```
    break;    /* normal completion */
}

where the Default Blocking Hook is equivalent to:

BOOL DefaultBlockingHook(void) {
    MSG msg = {0};
    BOOL ret = GetMessage(&msg, NULL, 0, 0);
    if( (int) ret != -1 ) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    /* FALSE if we got a WM_QUIT message */
    return( ret );
}
```

4. On reception of the completion message, the XFS Manager exits the loop.
5. The XFS Manager destroys the transitory HWND.
6. The blocking operation ~~is in progress, the thread~~ completes. The blocking function return code is copied from the completion message *lpWFSResult hResult* field. If applicable, the *lpWFSResult* is also returned.

The thread, on which the blocking function has been called, is not permitted to issue any XFS calls ~~during the processing of the message,~~ other than the following two specific functions provided to assist the ~~programmer~~developer in this situation~~:~~.

- **WFSIsBlocking** determines whether or not a blocking ~~call~~function is in progress.
- **WFSCancelBlockingCall** cancels a blocking ~~call~~function in progress.

Any other XFS function, called ~~when~~from a thread with a blocking ~~call is~~function in progress ~~fails~~, will fail with the error WFS_ERR_OP_IN_PROGRESS. ~~This restriction applies to requests for both blocking and non-blocking operations~~

Developers must be aware that **WFSIsBlocking** cannot simply be called in a loop waiting for the blocking function to complete. The application must allow the message handler to return to allow control to return to the blocking

hook. Otherwise, the blocking function will not complete.

Although this mechanism is sufficient for simple applications, it cannot support those applications which require more complex message processing while ~~blocked for~~ a ~~synchronous call~~blocking function is executing, such as processing messages relating to MDI (~~multiple document interface~~Multiple Document Interface) events, accelerator key translations, and modeless dialogs. For such applications, the XFS API includes the function **WFSSetBlockingHook**, which allows the ~~programmer~~developer to define a ~~special routine~~custom blocking hook which will be called instead of the default ~~message dispatch routine~~blocking hook described above. ~~This function gives an application the ability to execute its own routine at blocking time in place of the default routine.~~ It is ***not*** intended as a mechanism for performing general application functions while blocked; it is still true that the ***only*** XFS functions that may be called from a blocking routine are **WFSIsBlocking** and **WFSCancelBlockingCall**. The asynchronous versions of the XFS functions must be used to allow an application to continue processing while an operation is in progress. Developers must be aware of their responsibility when replacing the default blocking hook. The developer must ensure:

~~This mechanism is provided to allow a Windows 3.x or Windows for Workgroups application to make blocking calls without blocking the rest of the system. Under Windows NT and future multi-threaded, preemptive versions of Windows, the default blocking action is to suspend the calling application's thread until the request completes. This is because the system is not blocked by a single application waiting for an operation to complete, and hence not calling **PeekMessage** or **GetMessage,** which are required in the non-preemptive systems in order to cause the application to yield control.~~

~~Therefore, if a single-threaded application is targeted at both single- and multi-threaded environments, and relies on this functionality, it should install a specific blocking hook by calling **WFSSetBlockingHook**, even if the default hook would suffice. This maximizes the portability of applications that depend on the blocking hook behavior. Programmers who are constrained to use blocking mode - for example, as part of an existing application which is being ported - should be aware of the semantics of blocking operations.~~

~~In the XFS implementation in a single-threaded environment, the blocking function operates as follows. When an application requests a blocking XFS API function, the XFS Manager initiates the requested function and then enters a loop which is equivalent to the following pseudo-code:~~

```
for(;;) {
    /* flush messages for good user response */
    DefaultBlockingHook();
    /* check for WFSCancelBlockingCall() */
    if( operation_cancelled() )
        break;
    /* check to see if operation completed */
    if( operation_complete() )
        break;          /* normal completion */
}
```

~~The **DefaultBlockingHook** routine is equivalent to:~~

```
BOOL DefaultBlockingHook(void) {
    MSG msg;
    BOOL ret;
    /* Wait for the next message */
    ret = GetMessage(&msg, NULL, 0, 0);
    if( (int) ret != -1 ) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    /* FALSE if we got a WM_QUIT message */
    return( ret );
}
```

- All messages are processed in the order received. If not, the potential exists for the Service Provider to be blamed for sending messages in the wrong order e.g. a WFS_EXECUTE_EVENT message after a WFS_EXECUTE_COMPLETE.
- All messages are processed. If not, the potential exists that the thread message queue will fill preventing other messages being added to the queue, including the Service Provider attempt to post the completion message being waited on.

The developer must be aware that replacing the default blocking hook impacts the process. The custom blocking hook will be called from every thread which makes use of XFS blocking functions.

In a multi-threaded environment, the developer of a multi-threaded application must be aware that it is the responsibility of the application, not the XFS Manager, to synchronize access to a service by multiple threads. Failure to synchronize calls to a service leads to unpredictable results; for example, if two threads "simultaneously" issue **WFSExecute** requests to send data to the same service, there is no guarantee as to the order in which the data is sent. This is true in general; the application is responsible for coordinating access by multiple threads to any object (e.g. other forms of I/O, such as file I/O), using appropriate synchronization mechanisms. The XFS Manager can not, and will not, address these issues. The possible consequences of failing to observe these rules are beyond the scope of this specification.

In order to allow maximum flexibility in the design and implementation of applications, especially in multi-threaded environments, the concept of "application identity" can optionally be managed explicitly by the application developer using the concept of application handles. See Sections 4.5 and 4.8.2 for additional discussion of this concept.

4.13 Vendor Dependent Mode

XFS compliant applications must comply with the following:

- Every XFS application should open a session with the VDM Service Provider passing a valid Application ID and then register for all VDM entry and exit notices.
- Before opening any session with any other XFS Service Provider, check the status of the VDM Service Provider. If Vendor Dependent Mode is not “Inactive”, do not open a session.
- When getting a VDM entry notice, close all open sessions with all other XFS Service Providers as soon as possible and issue an acknowledgement for the entry to VDM.
- When getting a VDM exit notice, acknowledge at once.
- When getting a VDM exited notice, re-open any required sessions with other XFS Service Providers.

This is mandatory for self-service but optional for branch.

4.14 Memory Management

XFS specifies a protocol for dynamic allocation and release of memory. The general strategy is that the Service Providers allocate memory as they need it, and the applications specify when it can be released. This is implemented using a standard structure (WFSRESULT, defined in Section 9.1) that is always used to pass information to the applications from the services.

Most Service Provider function calls are asynchronous, and return their results via a completion message, which contains a pointer to a WFSRESULT structure, containing the function return status (*hResult*) and optional data. The Service Provider allocates the memory for this structure, using the memory management framework described below. The deallocation of the structure is done as follows:

- Asynchronous API functions
  The application receives the structure from the Service Provider via a completion message, and is responsible for deallocation.
- Synchronous **WFSExecute**, **WFSGetInfo** and **WFSLock** API functions
  The XFS Manager passes through the WFSRESULT structure to the application as a returned parameter, and the application is then responsible for deallocation, just as for asynchronous calls.
- All other synchronous API functions
  The XFS Manager unpacks the required information from the WFSRESULT structure into returned parameters to the application, deallocates the structure, and returns to the application.

Four functions are provided by the XFS Manager to implement this protocol: **WFMAllocateBuffer**, **WFMAllocateMore**, **WFMFreeBuffer**, and **WFSFreeResult**. Using these functions, two widely applicable allocation policies are supported:

A linear allocation policy

A linked allocation policy

Linear allocation can be used for any flat or contiguously allocated data structure. Such structures are returned in a single block of allocated memory by the **WFMAllocateBuffer** function.

Linked allocation can be used as an efficient way of managing complex data structures, permitting the Service Provider some flexibility while allowing the application to release the entire structure with a single call. In cases in which the Service Provider does not know a priori the size of the result data set, it makes an initial estimate, and uses **WFMAllocateBuffer**. If the Service Provider later determines that more space is required by the data, new memory is requested using the function **WFMAllocateMore**, and is automatically linked to the originally allocated block. The new memory block returned by **WFMAllocateMore** is, in general, not contiguous with the root block, and the user of this function should behave in all circumstances as if it is not.

The Service Provider is free to choose whatever allocation granularity is most convenient. This is completely transparent to the application or XFS Manager, which frees the entire WFSRESULT structure with a single **WFSFreeResult** call (the XFS Manager can also use this call as an indication that it can clean up any other objects associated with the request). Applications must be sure ***always*** to free a returned WFSRESULT structure. Note that a WFSRESULT structure may be returned even if the Service Provider has returned an error; if no WFSRESULT is returned, the pointer to the structure is NULL. A Service Provider may use also this facility for its "private" memory management requirements; it then uses the **WFMFreeBuffer** support function to free the allocated memory.

> **NOTE**:
> Applications and Service Providers ***must*** use the facilities provided by the XFS Manager for XFS-related memory allocation and deallocation, in order to avoid memory management conflicts among the applications, the XFS Manager and the Service Providers.

The following example illustrates how a Service Provider dynamically allocates a WFSRESULT buffer structure and an additional data buffer. Note that **WFMAllocateMore** automatically links these, allowing the application to free both structures with a single call.

```
WFSRESULT *  lpResultBuffer;

// Service Provider allocates a WFSResult buffer structure

result = WFMAllocateBuffer(sizeof(WFSRESULT), ulMemFlags, &lpResultBuffer);
•
•
•
// Service Provider allocates additional memory

hr = WFMAllocateMore(evenMoreMemory, lpResultBuffer, &lpResultBuffer->lpBuffer);
•
•
•
```

Once the application has retrieved all the information it needs from the WFSRESULT buffer and any associated structures, it must free the memory, which requires only a single call:

```
•
•
•
// application deallocates the structure when it is finished with it

hr = WFSFreeResult(lpResultBuffer);  // frees both the result buffer and
                                     // any additional buffers
```

NOTE:
When an application invokes an asynchronous or immediate (i.e. non-blocking) function which takes a pointer to a memory object as an argument, it is the responsibility of the Service Provider to ensure that it no longer needs access to the object before returning control to the application. This allows the application to release (deallocate) the memory object immediately upon the return from the call.

4.15 Command Synchronization

When the Service Provider supports command synchronization, the application can synchronize a command with another action (e.g. another command, screen change, etc.). For example, if both a receipt printer Service Provider and a card reader Service Provider support command synchronization for media ejection, the application can call synchronization preparation commands to both Service Providers and then the application can synchronize the media ejections (a receipt and a card) by calling the actual eject commands at the same time. For sample flows of command synchronization, see chapter 14.

4.16 Binary Interface

All applications and Service Providers should be fully compliant with the exported WFS and WFP interfaces in order to be compliant with any vendor's implementation of the XFS Manager. The CEN XFS SDK provides a reference XFS Manager and matching LIB files which are compliant with the interface defined below.

The following table lists the XFS Manager's API functions and their DLL locations, together with their fixed ordinal values.

	DLL and Ordinal Number		
API Call	**MSXFS**	**XFS_SUPP**	**XFS_CONF**
WFMAllocateBuffer	1	4	
WFMAllocateMore	2	5	
WFMFreeBuffer	3	6	
WFMGetTraceLevel	4		
WFMKillTimer	5	7	
WFMOutputTraceData	7	9	
WFMReleaseDLL	8		
WFMSetTimer	9	10	
WFMSetTraceLevel	10	11	
WFSAsyncClose	11		
WFSAsyncDeregister	12		
WFSAsyncExecute	13		
WFSAsyncGetInfo	14		
WFSAsyncLock	15		
WFSAsyncOpen	16		
WFSAsyncRegister	17		
WFSAsyncUnlock	18		
WFSCancelAsyncRequest	19		
WFSCancelBlockingCall	20		
WFSCleanUp	21		
WFSClose	22		
WFSCreateAppHandle	23		
WFSDeregister	24		
WFSDestroyAppHandle	25		
WFSExecute	26		
WFSFreeResult	27		
WFSGetInfo	28		
WFSIsBlocking	30		
WFSLock	31		
WFSOpen	32		
WFSRegister	33		
WFSSetBlockingHook	34		
WFSStartUp	35		
WFSUnhookBlockingHook	36		
WFSUnlock	37		
WFMCloseKey			4
WFMCreateKey			5
WFMDeleteKey			6
WFMDeleteValue			7
WFMEnumKey			8
WFMEnumValue			9
WFMOpenKey			10
WFMQueryValue			11
WFMSetValue			12

5 Application Programming Interface (API) Functions

The functions defined by the XFS API are divided into:

- ***Basic functions*** that are common to all classes of financial services.
- ***Administration functions***, used for the special purpose of administering services.
- ***Service-specific commands*** that are peculiar to a single service class or a group of them and that are sent to services using basic functions (**WFSExecute**, **WFSAsyncExecute**, **WFSGetInfo**, **WFSAsyncGetInfo**).

The benefit of grouping functions that are common to all services is evident: programmers can immediately focus on those operations that are common through all services and thus can easily build a high level model of interaction with the Service Providers.

The basic functions are defined in this section, in alphabetical order, except that the asynchronous version of each command is described immediately following the synchronous version. For example, **WFSAsyncExecute** is placed immediately following **WFSExecute**. The table on the next page lists all the basic functions. This set of basic functions may be expanded in future releases of this specification, if new functions are determined to be useful for all Service Providers.

The administration functions have not yet been fully defined; they are outlined in Appendix A. - Planned Enhancements and Extensions.

The service-specific commands are defined in separate specifications-one for each service class. In addition, the XFS SNMP MIB architecture specification defines a number of category codes that are common across all service classes.

The table below summarizes the XFS API functions, and the sections in which they are defined.

Section	Function	Mode	Description
5.1	**WFSCancelAsyncRequest**	Immediate	Cancel an outstanding asynchronous request
5.2	**WFSCancelBlockingCall**	Immediate	Cancel an outstanding blocking operation
5.3	**WFSCleanUp**	Synchronous	Terminate a connection between an application and the XFS Manager
5.4	**WFSClose**	Synchronous	Close a session between an application and a Service Provider
5.5	**WFSAsyncClose**	Asynchronous	The asynchronous version of **WFSClose**
5.6	**WFSCreateAppHandle**	Immediate	Create a new application handle to be used in a subsequent **WFSOpen** call
5.7	**WFSDeregister**	Synchronous	Disable monitoring of a class of events by an application
5.8	**WFSAsyncDeregister**	Asynchronous	The asynchronous version of **WFSDeregister**
5.9	**WFSDestroyAppHandle**	Immediate	Destroy the specified application handle
5.10	**WFSExecute**	Synchronous	Send service-specific commands to a Service Provider
5.11	**WFSAsyncExecute**	Asynchronous	The asynchronous version of **WFSExecute**
5.12	**WFSFreeResult**	Immediate	Request the XFS Manager to free a result buffer
5.13	**WFSGetInfo**	Synchronous	Retrieve service-specific information from a Service Provider
5.14	**WFSAsyncGetInfo**	Asynchronous	The asynchronous version of **WFSGetInfo**
5.15	**WFSIsBlocking**	Immediate	Determine if a blocking call is in progress
5.16	**WFSLock**	Synchronous	Establish exclusive control by an application of a service
5.17	**WFSAsyncLock**	Asynchronous	The asynchronous version of **WFSLock**
5.18	**WFSOpen**	Synchronous	Open a session between an application and a Service Provider
5.19	**WFSAsyncOpen**	Asynchronous	The asynchronous version of **WFSOpen**
5.20	**WFSRegister**	Synchronous	Enable monitoring of a class of events by an application
5.21	**WFSAsyncRegister**	Asynchronous	The asynchronous version of **WFSRegister**
5.22	**WFSSetBlockingHook**	Immediate	Install an application-specific blocking routine
5.23	**WFSStartUp**	Immediate	Initiate a connection between an application and the XFS Manager
5.24	**WFSUnhookBlockingHook**	Immediate	Restore the default blocking routine
5.25	**WFSUnlock**	Synchronous	Release exclusive control by an application of a service
5.26	**WFSAsyncUnlock**	Asynchronous	The asynchronous version of **WFSUnlock**

5.1 WFSCancelAsyncRequest

HRESULT **WFSCancelAsyncRequest**(*hService, RequestID*)

Cancels the specified (or every) asynchronous request being performed on the specified service, before its (their) completion.

Parameters

HSERVICE *hService*
Handle to the service as returned by **WFSOpen** or **WFSAsyncOpen.**

REQUESTID *RequestID*
The request identifier for the request to be canceled, as returned by the original function call (NULL to cancel all).

Mode Immediate

Comments

If the *RequestID* parameter is set to NULL, the command will cancel ***all*** asynchronous requests that are in progress using the specified *hService*.

A previously initiated asynchronous request is canceled prior to completion by issuing the **WFSCancelAsyncRequest** function, specifying the request identifier returned by the asynchronous function. This function is immediate with respect to its calling application, but the cancellation process is inherently asynchronous. On completion, the specified request (or all requests) will have finished, with a completion message indicating a status of WFS_ERR_CANCELED, unless the cancel request was received by the service ***after*** the request had completed. Thus, **WFSCancelAsyncRequest** is not guaranteed to stop all asynchronous commands: normal completion messages may still be posted after the cancel. A robust application that uses asynchronous commands should be designed to accept these messages even after a cancel is issued.

The cancellation applies not only to the XFS Manager level, but also to the Service Provider level. The request is passed through the SPI, and the Service Provider normally then also cancels any physical I/O or other device operation in progress, in the appropriate manner for the device or service.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_REQ_ID
The *RequestID* parameter does not correspond to an outstanding request on the service.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFSAsyncExecute**

5.2 WFSCancelBlockingCall

HRESULT **WFSCancelBlockingCall**(dw*ThreadID*)

Cancels a blocking operation for the specified thread, if one is in progress.

Parameters **DWORD** *dwThreadID*
Identifies the thread for which the blocking operation is to be canceled; a NULL value indicates the calling thread.

Mode Immediate

Comments This function is used to cancel a blocking call (synchronous request) that is in progress. Since a thread may have only ***one*** blocking call in progress at any time, **WFSIsBlocking** and **WFSCancelBlockingCall** are the only XFS functions allowed with respect to a thread when it has a blocking call in progress.

The application that issued the blocking call receives a WFS_ERR_CANCELED return code if the operation is successfully canceled.

The cancellation applies not only to the XFS Manager level, but also to the Service Provider level. The request is passed through the SPI, and the Service Provider normally then also cancels any physical I/O or other device operation in progress, in the appropriate manner for the device or service.

Note: the cancel request is accepted and is honored as soon as all Windows messages have been removed from the message queue (i.e. GetMessage returns no more messages). Refer to **WFSSetBlockingHook** for more information.

Error Codes If the function return is not WFS_SUCCESS, it is the following error condition:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_NO_BLOCKING_CALL
There is no outstanding blocking call for the specified thread.

WFS_ERR_NO_SUCH_THREAD
The specified thread does not exist.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

See Also **WFSSetBlockingHook, WFSIsBlocking, WFSCancelAsyncRequest**

5.3 WFSCleanUp

HRESULT **WFSCleanUp**( )

Disconnects an application from the XFS Manager.

Parameters None

Mode Synchronous

Comments The **WFSCleanUp** call indicates disconnection of an XFS application from the XFS Manager. This function, for example, frees resources allocated to the specific application. **WFSCleanUp** applies to all threads of a multi-threaded application. If **WFSClose** has not been issued for one or more Service Providers, then the XFS Manager will automatically issue the close(s). Once the **WFSCleanUp** has been performed, subsequent attempts to issue any XFS function other than **WFSStartUp** will fail.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFSStartUp**

5.4 WFSClose

HRESULT **WFSClose(** *hService*)

Terminates a session (a series of service requests initiated with the **WFSOpen** or **WFSAsyncOpen** function) between the application and the specified service. The synchronous version of **WFSAsyncClose**.

Parameters **HSERVICE** *hService*
The service handle returned by **WFSOpen** or **WFSAsyncOpen**. Matches the close request to the open request, allowing an application to have multiple sessions open simultaneously with a single Service Provider.

Mode Synchronous

Comments **WFSClose** directs the service to free all resources associated with the series of requests made using the *hService* parameter since the **WFSOpen** that returned it. If there is a blocking call in progress the close fails. If the service is locked, the close automatically unlocks it. If no **WFSDeregister** has been issued, it is automatically performed.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFSAsyncClose, WFSOpen, WFSDeregister**

5.5 WFSAsyncClose

HRESULT **WFSAsyncClose**(*hService, hWnd, lpRequestID*)

Terminates a session (a series of service requests initiated with the **WFSOpen** or **WFSAsyncOpen** function) between the application and the specified service. The asynchronous version of **WFSClose**.

Parameters

HSERVICE *hService*
The service handle returned by **WFSOpen** or **WFSAsyncOpen.** Matches the close request to the open request, allowing an application to maintain several "open sessions" simultaneously.

HWND *hWnd*
The window handle which is to receive the completion message for this request.

LPREQUESTID *lpRequestID*
Pointer to the request identifier for this request (returned parameter).

Mode

Asynchronous

Comments

See **WFSClose**.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure which is pointed to by the completion message. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Messages

WFS_CLOSE_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

The following error condition can be returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

See Also

WFSOpen, WFSDeregister

5.6 WFSCreateAppHandle

HRESULT **WFSCreateAppHandle**(*lphApp*)

Requests a new, unique application handle value.

Parameters **LPHAPP** *lphApp*
A pointer to the application handle to be created (returned parameter).

Mode Immediate

Comments This function is used by an application to request a unique (within a single system) application handle from the XFS Manager (to be used in subsequent **WFSOpen**/**WFSAsyncOpen** calls). Note that an application may call this function multiple times in order to create multiple "application identities" for itself with respect to the XFS subsystem. See Sections 4.5 and 4.8.2 for additional discussion.

Error Codes If the function return is not WFS_SUCCESS, it is the following error condition.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

See Also **WFSDestroyAppHandle, WFSOpen, WFSAsyncOpen**

5.7 WFSDeregister

HRESULT **WFSDeregister**(*hService, dwEventClass, hWndReg*)

Discontinues monitoring of the specified message class(es) (or ***all*** classes) from the specified *hService*, by the specified *hWndReg* (or ***all*** the calling application's *hWnd*'s). The synchronous version of **WFSAsyncDeregister**.

Parameters

HSERVICE *hService*
Service handle returned by **WFSOpen** or **WFSAsyncOpen**. If this value is NULL, and *dwEventClass* is SYSTEM_EVENTS, the XFS manager deregisters the application for those system events generated by the Manager itself.

DWORD *dwEventClass*
The class(es) of messages from which the application is deregistering. Specified as a bit mask that can be a logical OR of the values for multiple classes. A NULL value requests that ***all*** message classes be deregistered from the specified window for this *hService*.

HWND *hWndReg*
The window which has been previously registered to receive notification messages, and is now to be deregistered. A NULL value requests that ***all*** the application's windows be deregistered from the specified message class(es) for this *hService*.

Mode

Synchronous

Comments

The functions of a **WFSDeregister** request are performed automatically if a **WFSClose** is issued without a previous **WFSDeregister**.

See section 4.11 for a description of the classes of events that may be monitored.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_EVENT_CLASS
The *dwEventClass* parameter specifies one or more event classes not supported by the service.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWNDREG
The *hWndReg* parameter is not a valid window handle.

WFS_ERR_NOT_REGISTERED
The specified *hWndReg* window was not registered to receive messages for any event classes.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also

WFSRegister, WFSClose

5.8 WFSAsyncDeregister

HRESULT **WFSAsyncDeregister**(*hService, dwEventClass, hWndReg, hWnd, lpRequestID*)

Discontinues monitoring of the specified message class(es) (or ***all*** classes) from the specified *hService*, by the specified *hWndReg* (or ***all*** the calling application's hWnd's). The asynchronous version of **WFSDeregister**.

Parameters

HSERVICE *hService*
Service handle returned by **WFSOpen** or **WFSAsyncOpen**. If this value is NULL, and *dwEventClass* is SYSTEM_EVENTS, the XFS manager deregisters the application for those system events generated by the Manager itself.

DWORD *dwEventClass*
The class(es) of events from which the application is deregistering. Specified as a bit mask that can be a logical OR of the values for multiple classes. A NULL value requests that ***all*** event classes be deregistered from the specified window for this *hService*.

HWND *hWndReg*
The window which has been previously registered to receive notification messages, and is now to be deregistered. A NULL value requests that ***all*** the application's windows be deregistered from the specified message class(es) for this *hService*.

HWND *hWnd*
The window handle which is to receive the completion message for this request.

LPREQUESTID *lpRequestID*
Pointer to the request identifier for this request (returned parameter).

Mode Asynchronous

Comments See **WFSDeregister**.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure which is pointed to by the completion message. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Messages WFS_DEREGISTER_COMPLETE

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_EVENT_CLASS
The *dwEventClass* parameter specifies one or more event classes not supported by the service.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_HWNDREG
The *hWndReg* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_REGISTERED
The specified *hWndReg* window was not registered to receive messages for any event classes.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

See Also **WFSRegister, WFSClose**

5.9 WFSDestroyAppHandle

HRESULT **WFSDestroyAppHandle**(*hApp*)

Makes the specified application handle invalid.

Parameters **HAPP** *hApp*
The application handle to be made invalid.

Mode Immediate

Comments This function is used by an application to indicate to the XFS Manager that it will no longer use the specified application handle (from a previous **WFSCreateAppHandle** call). See **WFSCreateAppHandle** and Sections 4.5 and 4.8.2 for additional discussion.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_INVALID_APP_HANDLE
The specified application handle is not valid, i.e. was not created by a preceding create call.

See Also **WFSCreateAppHandle**

5.10 WFSExecute

HRESULT **WFSExecute** (*hService, dwCommand, lpCmdData, dwTimeOut, lppResult*)

Sends a service-specific command to a Service Provider. The synchronous version of **WFSAsyncExecute**.

Parameters

HSERVICE *hService*
Handle to the service as returned by **WFSOpen** or **WFSAsyncOpen**.

DWORD *dwCommand*
Command to be executed by the Service Provider.

LPVOID *lpCmdData*
Pointer to a command data structure to be passed to the Service Provider.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

LPWFSRESULT * *lppResult*
Pointer to the pointer to the result data structure used to return the results of the execution. The Service Provider allocates the memory for this structure.

Mode

Synchronous

Comments

This function is used to execute service-specific commands. Each class of service includes a unique set of commands for the given class of device or service; they are defined in the service-specific command specifications. Each Service Provider developer is responsible for recognizing the complete set of commands for a given class, even if the Service Provider doesn't support them all. Each command, for each service class, defines a command data structure and/or a result data structure. See the separate specifications for each service class for more discussion of these issues, and the definitions of the service-specific commands and associated data structures.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure returned by this function. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_COMMAND
The *dwCommand* issued is not supported by this service class.

WFS_ERR_INVALID_DATA
The data structure passed as input parameter contains invalid data.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_LOCKED
The service is locked under a different *hService*.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_UNSUPP_COMMAND
The *dwCommand* issued, although valid for this service class, is not supported by this Service Provider or device.

WFS_ERR_USER_ERROR
A user is preventing proper operation of the device.

WFS_ERR_UNSUPP_DATA
The data structure passed as an input parameter although valid for this service class, is not supported by this Service Provider or device.

WFS_ERR_FRAUD_ATTEMPT
Some devices are capable of identifying a malicious physical attack which attempts to defraud valuable information or media. In these cases, this error code is returned to indicate the user is attempting a fraudulent act on the device.

WFS_ERR_SEQUENCE_ERROR
The requested operation is not valid at this time or in the devices current state.

WFS_ERR_AUTH_REQUIRED
The requested operation cannot be performed because it requires authentication.

See Also **WFSAsyncExecute**

5.11 WFSAsyncExecute

HRESULT **WFSAsyncExecute**(*hService, dwCommand, lpCmdData, dwTimeOut, hWnd, lpRequestID*)

Sends a service-specific command to a Service Provider. The asynchronous version of **WFSExecute**.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen.**

DWORD *dwCommand*
Command to be executed by the Service Provider.

LPVOID *lpCmdData*
Pointer to the data structure to be passed to the Service Provider.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

LPREQUESTID *lpRequestID*
Pointer to the request identifier for this request (returned parameter).

Mode

Asynchronous

Comments

See **WFSExecute**.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure which is pointed to by the completion message. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Messages

WFS_EXECUTE_COMPLETE
WFS_EXECUTE_EVENT

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_COMMAND
The *dwCommand* issued is not supported by this service class.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_UNSUPP_COMMAND
The *dwCommand* issued, although valid for this service class, is not supported by this Service Provider or device.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_DATA
The data structure passed as input parameter contains invalid data.

WFS_ERR_LOCKED
The service is locked under a different *hService*.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_UNSUPP_COMMAND
The *dwCommand* issued, although valid for this service class, is not supported by this Service Provider or device.

WFS_ERR_USER_ERROR
A user is preventing proper operation of the device.

WFS_ERR_UNSUPP_DATA
The data structure passed as an input parameter although valid for this service class, is not supported by this Service Provider or device.

WFS_ERR_FRAUD_ATTEMPT
Some devices are capable of identifying a malicious physical attack which attempts to defraud valuable information or media. In these cases, this error code is returned to indicate the user is attempting a fraudulent act on the device.

WFS_ERR_SEQUENCE_ERROR
The requested operation is not valid at this time or in the devices current state.

WFS_ERR_AUTH_REQUIRED
The requested operation cannot be performed because it requires authentication.

See Also **WFSCancelAsyncRequest, WFSExecute**

5.12 WFSFreeResult

HRESULT **WFSFreeResult** (*lpResult*)

Notifies the XFS Manager that a memory buffer (or linked list of buffers) that was dynamically allocated by a Service Provider is to be freed.

Parameters **LPWFSRESULT** *lpResult*
Pointer to a WFSRESULT data structure.

Mode Immediate

Comments The XFS Service Providers may allocate memory to send data to an application. This function is used by the application to deallocate the memory, and the application must call it when it no longer needs access to the memory. When the application calls **WFSFreeResult**, all memory allocated by the Service Provider for this result is deallocated. See Section 4.14.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_RESULT
The *lpResult* parameter is not a pointer to an allocated WFSRESULT structure.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

See Also **WFSExecute, WFSAsyncExecute, WFSGetInfo, WFSAsyncGetInfo**

5.13 WFSGetInfo

HRESULT **WFSGetInfo**(*hService, dwCategory, lpQueryDetails, dwTimeOut, lppResult*)

Retrieves information from the specified Service Provider. The synchronous version of **WFSAsyncGetInfo**.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen.**

DWORD *dwCategory*
Specifies the category of the query (e.g. for a printer, WFS_INF_PTR_STATUS to request status or WFS_INF_PTR_CAPABILITIES to request capabilities). The available categories depend on the service class, the Service Provider and the service. The information requested can be either static or dynamic, e.g. basic service capabilities (static) or current service status (dynamic).

LPVOID *lpQueryDetails*
Pointer to the data structure to be passed to the Service Provider, containing further details to make the query more precise, e.g. a form name. Many queries have no input parameters, in which case this pointer is NULL.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

LPWFSRESULT * *lppResult*
Pointer to the pointer to the data structure to be filled with the result of the execution. The Service Provider allocates the memory for the structure.

Mode

Synchronous

Comments

The XFS Manager passes the request to the Service Provider, and since the information may be stored remotely, the function cannot be immediate. Note that many requests ***can*** be satisfied by the Service Provider and will therefore complete immediately.

The definitions of the *dwCategory* and *lpQueryDetails* parameters are provided in the service-specific command sections of this specification. Note that these information retrieval functions are separate from the other service-specific commands, since those commands can be executed only via **WFSExecute** or **WFSAsyncExecute**, which require that the service be either locked by the application issuing the command, or unlocked. The **GetInfo** functions, however, can be used even when a service is locked by another application.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure which is returned by this function. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_CATEGORY
The *dwCategory* issued is not supported by this service class.

WFS_ERR_INVALID_DATA
The data structure passed as input parameter contains invalid data.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_UNSUPP_CATEGORY
The *dwCategory* issued, although valid for this service class, is not supported by this Service Provider.

WFS_ERR_USER_ERROR
A user is preventing proper operation of the device.

WFS_ERR_UNSUPP_DATA
The data structure passed as an input parameter although valid for this service class, is not supported by this Service Provider or device.

See Also **WFSAsyncGetInfo**

5.14 WFSAsyncGetInfo

HRESULT **WFSAsyncGetInfo**(*hService, dwCategory, lpQueryDetails, dwTimeOut, hWnd, lpRequestID*)

Retrieves information from the specified Service Provider. The asynchronous version of **WFSGetInfo**.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen.**

DWORD *dwCategory*
See **WFSGetInfo**.

LPVOID *lpQueryDetails*
See **WFSGetInfo**.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

LPREQUESTID *lpRequestID*
The request identifier for this request (returned parameter).

Mode

Asynchronous

Comments

See **WFSGetInfo**.

The only difference in the asynchronous version of the function is that the results (query details) returned to the application (in the WFSRESULT data structure) are pointed to by the WFS_GETINFO_COMPLETE message sent to the specified *hWnd*.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure which is pointed to by the completion message. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Messages

WFS_GETINFO_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_CATEGORY
The *dwCategory* issued is not supported by this service class.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_UNSUPP_CATEGORY
The *dwCategory* issued, although valid for this service class, is not supported by this Service Provider.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_DATA
The data structure passed as input parameter contains invalid data.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_USER_ERROR
A user is preventing proper operation of the device.

WFS_ERR_UNSUPP_DATA
The data structure passed as an input parameter although valid for this service class, is not supported by this Service Provider or device.

See Also **WFSGetInfo, WFSCancelAsyncRequest**

5.15 WFSIsBlocking

BOOL **WFSIsBlocking**( )

Determines whether a thread has a blocking operation in progress.

Parameters None

Return Value The return value is TRUE if a blocking operation is in progress and FALSE otherwise.

Mode Immediate

Comments Although a call issued on a synchronous (blocking) function appears to an application as though it blocks, the XFS Manager in fact relinquishes control of the processor to allow other Windows processes to run. Thus it is possible for an application that issues a blocking call to be re-entered, depending on the messages it receives. Since the XFS Manager prohibits more than one outstanding blocking call per thread, an application's message processing routines need a way to determine whether they have been re-entered while the application is waiting for an outstanding blocking call to complete. The **WFSIsBlocking** function provides this function, allowing an application to detect whether a blocking operation is already in progress, before it issues another XFS request.

Note that if another XFS call ***is*** issued in this situation, the XFS Manager returns with a WFS_ERR_OP_IN_PROGRESS error code. See Section 4.12 for additional discussion.

See Also **WFSCancelBlockingCall**

5.16 WFSLock

HRESULT **WFSLock**(*hService, dwTimeOut, lppResult*)

Establishes exclusive control by the calling application over the specified service. The synchronous version of **WFSAsyncLock**.

Parameters

HSERVICE *hService*
Service Provider handle as returned by **WFSOpen** or **WFSAsyncOpen.**

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

LPWFSRESULT **lppResult*
Pointer to the pointer to a WFSRESULT data structure (see Comments). The Service Provider allocates the memory for this structure.

Mode

Synchronous

Comments

A Service Provider can support a "shared" session, in which multiple applications' data are mixed in the service's I/O stream. More typically, a session is exclusive at any point in time; all I/O is for a single application. To define an exclusive use of the Service Provider, a lock function (synchronous or asynchronous) must be used. See Section 4.8 for more discussion of the lock concepts and policy.

The time to complete will depend on whether there is another application that has acquired exclusive access to the service. Note that trying to lock several services at the same time can lead to a deadlock. The timeout capability is provided in the API to allow applications to prevent this.

lppResult is a pointer to a pointer to a WFSRESULT data structure containing a null-terminated array of service handles (*hService* values), specifying any ***other*** services that are already locked by the application (i.e. under the same *hApp*), ***only if*** those services are part of a compound device that includes the service being locked, ***and*** are interdependent with it. The returned pointer is NULL if there are no such "associated" services locked. See Section 4.8.2 for more discussion of this subject.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure, if there is one. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_TIMEOUT
The timeout interval expired.

See Also **WFSAsyncLock, WFSUnlock, WFSCancelBlockingCall**

5.17 WFSAsyncLock

HRESULT **WFSAsyncLock**(*hService, dwTimeOut, hWnd, lpRequestID*)

Establishes exclusive control by the calling application over the specified service. The asynchronous version of **WFSLock.**

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen**.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

LPREQUESTID *lpRequestID*
Pointer to the request identifier for this request (returned parameter).

Mode

Asynchronous

Comments

See **WFSLock** and Section 4.8.2. In particular, note that if other services are locked as a result of this call (i.e. because the service specified is part of a compound device), the handles of these services are returned in the WFSRESULT data structure pointed to by the completion message.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Messages

WFS_LOCK_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_TIMEOUT
The timeout interval expired.

See Also **WFSLock, WFSUnlock, WFSCancelAsyncRequest**

5.18 WFSOpen

HRESULT **WFSOpen**(*lpszLogicalName, hApp, lpszAppID, dwTraceLevel, dwTimeOut, dwSrvcVersionsRequired, lpSrvcVersion, lpSPIVersion, lphService*)

Initiates a session (a series of service requests terminated with the **WFSClose** function) between the application and the specified service. This does not necessarily mean that the hardware is opened. This command will return with WFS_SUCCESS even if the hardware is inoperable, offline or powered off. The status of the device can be requested through a **WFSGetInfo** command.

The synchronous version of **WFSAsyncOpen.**

Parameters

~~LPSTR~~LPCSTR *lpszLogicalName*
Points to a null-terminated string containing the pre-defined logical name of a service. It is a high level name such as "SYSJOURNAL1", "PASSBOOKPTR3" or "CASHDISP02," that is used by the XFS Manager and the Service Provider solely as a key to obtain the specific configuration information they need.

HAPP *hApp*
The application handle to be associated with the session being opened. If this parameter is equal to WFS_DEFAULT_HAPP, the session is associated with the calling process as a whole (i.e. the calling process, not some subset of its threads, is the owner of the session and its *hService*). See **WFSCreateAppHandle** and Sections 4.5 and 4.8.2 for details.

~~LPSTR~~LPCSTR *lpszAppID*
Points to a null-terminated string containing the application ID; the pointer may be NULL if the ID is not used. This ID may be used by services in a variety of ways; e.g. it is included in the SYSTEM_EVENT message for undeliverable events, to aid in finding system problems

DWORD *dwTraceLevel*
See **WFMSetTraceLevel**. NULL turns off all tracing.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

DWORD *dwSrvcVersionsRequired*
Specifies the range of versions of the service-specific interface that the application can support. (See Comments.) The low-order word indicates the highest version of the interface the application can support; the high-order word indicates the lowest version of the interface the application can support. In each word, the low-order byte specifies the major version number and the high-order byte specifies the minor version number (i.e. the numbers before and after the decimal).
Note: in order to allow intermediate minor revisions (e.g. between 1.10 and 1.20), the minor version number should always be expressed as two decimal digits, i.e. 1.10, 1.11, 1.20, etc.

LPWFSVERSION *lpSrvcVersion*
Pointer to the data structure that is to receive version support information and other details about the service-specific interface implementation (returned parameter).

LPWFSVERSION *lpSPIVersion*
Pointer to the data structure that is to receive version support information and (optionally) other details about the SPI implementation of the Service Provider being opened (returned parameter). This pointer may be NULL if the application is not interested in receiving this information. See **WFPOpen**.

LPHSERVICE *lphService*
Pointer to the service handle that the XFS Manager assigns to the service on a successful open; the application uses this handle for communication with the Service Provider for the remainder of the session (returned parameter). If a process opens the same service twice, the XFS Manager generates and returns different *hService* values.

Mode Synchronous

Comments

This function is used by an application to initiate a session with a service; the session is terminated by **WFSClose.** After **WFSStartUp**, an application must use this function (or the asynchronous version) to access a service. The request is made in terms of a logical service name (*lpszLogicalName*) which is mapped by the XFS Manager to a Service Provider. The XFS Manager loads the Service Provider, if necessary, and returns a logical service handle to the application which is used during the session to refer to the service.

In order to support future XFS implementations with maximum flexibility, two version negotiations take place in **WFSOpen** processing. An application specifies in the *dwSrvcVersionsRequired* parameter the range of versions of the service-specific interface (as defined by the events and error codes within this specification and in the separate XFS specifications for specific classes of devices, such as banking printers and cash dispensers) that it can support. If the range of versions specified by the application overlaps the range of versions that the Service Provider's implementation can support, the call succeeds. Otherwise the call fails. (The other negotiation that takes place during the open process is between the XFS Manager and the Service Provider regarding the SPI level. See **WFPOpen** for details.)

Information describing the actual Service Provider implementation is returned in the WFSVERSION data structure (defined in Section 9.2). In particular, it returns the version the Service Provider expects the application to use (the highest common version), as well as the lowest and highest versions it is capable of. If the call fails, WFSVERSION is still returned, to help with analysis of the failure.

The version numbers refer to the complete interface specification: the service-specific **WFSExecute** and **WFSGetInfo** commands, parameters, data structures, error codes, and messages. If there are any changes to these, the version number should be changed.

This version negotiation allows an XFS application and a Service Provider to operate successfully if there is any overlap in their versions. The following chart gives examples of how **WFSOpen** works in conjunction with different application and Service Provider versions:

dwSrvcVersions-Required (Version required by Application):	lpSrvcVersion.wLowVerion lpSrvcVersion.wHighVersion (Service Provider versions):	Return status from **WFSOpen**:	lpSrvcVersion .wVersion (Result):
0x00010001 (1.00)	0x0001 0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x00010A02 (1.00 - 2.10)	0x0001 0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x0B010B01 (1.11)	0x0001 0x0002 (1.00 - 2.00)	WFS_SUCCESS	0x0B01 (use 1.11)
0x0B020003 (2.11 - 3.00)	0x0001 0x1402 (1.00 - 2.20)	WFS_SUCCESS	0x1402 (use 2.20)
0x00010001 (1.00)	0x1402 0x0003 (2.20 - 3.00)	WFS_ERR_SRVC_VERS_TOO_LOW	0x0000 (fails)
0x0B010003 (1.11 - 3.00)	0x0001 0x0001 (1.00)	WFS_ERR_SRVC_VERS_TOO_HIGH	0x0000 (fails)

Note that a version negotiation error also generates a system event (see Section 10.8).

If a valid Service Provider is available, the Open command will not complete until the Service Provider and all its dependencies are running. That is, if an out of process executable is required by this Service Provider, this executable should be running and fully initialized before completion of the Open command. The starting and stopping of external dependent processes is not defined as the responsibility of the Service Provider, but the latter has to be aware of and respond correctly to the Open command according to external dependent process state. In addition, if the specified timeout period expires before dependent external processes have correctly initialized, the Service Provider must complete and return WFS_ERR_TIMEOUT as expected.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_APP_HANDLE
The specified application handle is not valid, i.e. was not created by a preceding create call.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_INVALID_SERVPROV
The file containing the Service Provider is invalid or corrupted.

WFS_ERR_INVALID_TRACELEVEL
The *dwTraceLevel* parameter does not correspond to a valid trace level or set of levels.

WFS_ERR_NO_SERVPROV
The file containing the Service Provider does not exist.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_SERVICE_NOT_FOUND
The logical name is not a valid Service Provider name.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_SPI_VER_TOO_HIGH
The range of versions of XFS SPI support requested by the XFS Manager is higher than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SPI_VER_TOO_LOW
The range of versions of XFS SPI support requested by the XFS Manager is lower than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SRVC_VER_TOO_HIGH
The range of versions of the service-specific interface support requested by the application (in the *dwSrvcVersionsRequired* parameter of this call) is higher than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SRVC_VER_TOO_LOW
The range of versions of the service-specific interface support requested by the application (in the *dwSrvcVersionsRequired* parameter of this call) is lower than any supported by the Service Provider for the logical service being opened.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_VERSION_ERROR_IN_SRVC
Within the service, a version mismatch of two modules occurred.

See Also **WFSAsyncOpen, WFSClose, WFSCreateAppHandle**

5.19 WFSAsyncOpen

HRESULT **WFSAsyncOpen**(*lpszLogicalName, hApp, lpszAppID, dwTraceLevel, dwTimeOut, lphService, hWnd, dwSrvcVersionsRequired, lpSrvcVersion, lpSPIVersion, lpRequestID*)

Initiates a session (a series of service requests terminated with the **WFSClose** or **WFSAsyncClose** function) between the application and the specified service. This does not necessarily mean that the hardware is opened. This command will return with WFS_SUCCESS even if the hardware is inoperable, offline or powered off. The status of the device can be requested through a **WFSGetInfo** command.

The asynchronous version of **WFSOpen.**

Parameters

~~**LPSTR**~~**LPCSTR** *lpszLogicalName*
See **WFSOpen**.

HAPP *hApp*
The application handle to be associated with the session being opened.
See **WFSOpen**, **WFSCreateAppHandle** and Sections 4.5 and 4.8.2 for details.

~~**LPSTR**~~**LPCSTR** *lpszAppID*
Points to a null-terminated string containing the application ID. See **WFSOpen.**

DWORD *dwTraceLevel*
See **WFMSetTraceLevel**. NULL turns off all tracing.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

LPHSERVICE *lphService*
Pointer to the service handle (returned parameter).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

DWORD *dwSrvcVersionsRequired*
See **WFSOpen.**

LPWFSVERSION *lpSrvcVersion*
See **WFSOpen** (returned parameter).

LPWFSVERSION *lpSPIVersion*
See **WFSOpen** (returned parameter).

LPREQUESTID *lpRequestID*
Pointer to the request identifier for this request (returned parameter).

Mode Asynchronous

Comments See **WFSOpen**.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure which is pointed to by the completion message. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Messages WFS_OPEN_COMPLETE

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_APP_HANDLE
The specified application handle is not valid, i.e. was not created by a preceding create call.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_INVALID_SERVPROV
The file containing the Service Provider is invalid or corrupted.

WFS_ERR_INVALID_TRACELEVEL
The *dwTraceLevel* parameter does not correspond to a valid trace level or set of levels.

WFS_ERR_NO_SERVPROV
The file containing the Service Provider does not exist.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_SERVICE_NOT_FOUND
The logical name is not a valid Service Provider name.

WFS_ERR_SPI_VER_TOO_HIGH
The range of versions of XFS SPI support requested by the XFS Manager is higher than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SPI_VER_TOO_LOW
The range of versions of XFS SPI support requested by the XFS Manager is lower than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SRVC_VER_TOO_HIGH
The range of versions of the service-specific interface support requested by the application (in the *dwSrvcVersionsRequired* parameter of this call) is higher than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SRVC_VER_TOO_LOW
The range of versions of the service-specific interface support requested by the application (in the *dwSrvcVersionsRequired* parameter of this call) is lower than any supported by the Service Provider for the logical service being opened.

WFS_ERR_VERSION_ERROR_IN_SRVC
Within the service, a version mismatch of two modules occurred.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready timed out.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_VERSION_ERROR_IN_SRVC
Within the service, a version mismatch of two modules occurred.

See Also **WFSOpen, WFSClose, WFSCreateAppHandle, WFSCancelAsyncRequest, WFMSetTraceLevel**

5.20 WFSRegister

HRESULT **WFSRegister**(*hService, dwEventClass, hWndReg*)

Enables event monitoring for the specified service by the specified window; all messages of the specified class(es) are sent to the window specified in the *hWndReg* parameter. The synchronous version of **WFSAsyncRegister**.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen**. If this value is NULL, and *dwEventClass* is SYSTEM_EVENTS, the XFS manager registers the application for those system events generated by the Manager itself.

DWORD *dwEventClass*
The class(es) of events for which the application is registering. Specified as a set of bit masks that are logically ORed together into this parameter.

HWND *hWndReg*
The window handle which is to be registered to receive the specified messages.

Mode Synchronous

Comments Issuing a **WFSRegister** for a service enables event monitoring on that service. **WFSRegister** calls can be cumulative for the same window. For example, to receive notification for both system and user events, the application can call **WFSRegister** with both SYSTEM_EVENTS and USER_EVENTS, as follows:

```
hr = WFSRegister(hPassbook1, SYSTEM_EVENTS | USER_EVENTS, hWndReg1);
```

or call them in two phases:

```
hr = WFSRegister( hPassbook1, SYSTEM_EVENTS, hWndReg1);
. . . . . .
hr = WFSRegister( hPassbook1, USER_EVENTS, hWndReg1);
```

To cancel notifications use **WFSDeregister**.

Note that the Service Provider always monitors the service, regardless of whether an application has registered for event monitoring. Issuing **WFSRegister** simply causes the Service Provider to post messages to the application in addition to handling the messages itself. See the discussion in Section 4.11.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_EVENT_CLASS
The *dwEventClass* parameter specifies one or more event classes not supported by the service.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWNDREG
The *hWndReg* parameter is not a valid window handle.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFSAsyncRegister, WFSDeregister, WFSAsyncDeregister**

5.21 WFSAsyncRegister

HRESULT **WFSAsyncRegister**(*hService, dwEventClass, hWndReg, hWnd, lpRequestID*)

Enables event monitoring for the specified service by the specified window; all messages of the specified class(es) are sent to the window specified in the *hWndReg* parameter. The asynchronous version of **WFSRegister**.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen.** If this value is NULL, and *dwEventClass* is SYSTEM_EVENTS, the XFS manager registers the application for those system events generated by the Manager itself.

DWORD *dwEventClass*
See **WFSRegister**.

HWND *hWndReg*
The window handle which is to be registered to receive the specified messages.

HWND *hWnd*
The window handle which is to receive the completion message for this request.

LPREQUESTID *lpRequestID*
Pointer to the request identifier for this request (returned parameter).

Mode

Asynchronous

Comments

See **WFSRegister**.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure pointed to by the completion message. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.11.

Messages

WFS_REGISTER_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_EVENT_CLASS
The *dwEventClass* parameter specifies one or more event classes not supported by the service.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_HWNDREG
The *hWndReg* parameter is not a valid window handle.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

The following error conditions can be returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

See Also **WFSRegister, WFSDeregister, WFSAsyncDeregister**

5.22 WFSSetBlockingHook

HRESULT **WFSSetBlockingHook**(*lpBlockFunc, lppPrevFunc*)

Establishes an application-specific blocking routine.

Parameters

XFSBLOCKINGHOOK *lpBlockFunc*
Pointer to the procedure instance address of the blocking routine to be installed.

LPXFSBLOCKINGHOOK *lppPrevFunc*
Returned pointer to a pointer to the procedure instance of the ***previously*** installed blocking routine.

Mode Immediate

Comments When this function is successfully issued by an application, it returns a pointer to the previously installed blocking routine. The application may save this pointer so that it can be restored if desired. If such "nesting" is not required, the application can discard this value and simply use the **WFSUnhookBlockingHook** function to restore the default routine at any time.

See Section 4.12 for a complete discussion.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

See Also **WFSUnhookBlockingHook, WFSCancelBlockingCall, WFSIsBlocking**

5.23 WFSStartUp

HRESULT **WFSStartUp**(*dwVersionsRequired, lpWFSVersion*)

Establishes a connection between an application and the XFS Manager.

Parameters

DWORD *dwVersionsRequired*
Specifies the range of versions of the XFS Manager that the application can support. The low-order word indicates the highest version of the XFS Manager the application can support; the high-order word indicates the lowest version of the XFS Manager the application can support. In each word, the low-order byte specifies the major version number and the high-order byte specifies the minor version number (i.e. the numbers before and after the decimal).
Note: in order to allow intermediate minor revisions (e.g. between 1.10 and 1.20), the minor version number should always be expressed as two decimal digits, i.e. 1.10, 1.11, 1.20, etc.

LPWFSVERSION *lpWFSVersion*
Pointer to the data structure that is to receive version support information and other details about the current XFS implementation (returned parameter).

Mode Immediate

Comments

This function is used by an application to register itself with the XFS Manager and specify the version(s) of the XFS API specification it can use, and returns information on the specific XFS implementation. It ***must*** be the first XFS API function called by an application. An application may only issue further XFS functions after a successful **WFSStartUp** has completed.

In order to support future XFS implementations with maximum flexibility, a version negotiation process takes place in **WFSStartUp**. An application specifies in the *dwVersionsRequired* parameter the range of versions of the XFS API specification which it can support. If the range of versions specified by the application overlaps the range of versions that the current implementation of XFS Manager can support, the call succeeds. Otherwise the call fails.

Information describing the actual XFS implementation is returned by the XFS Manager in the WFSVERSION data structure (defined in Section 9.2). In particular, it returns the version it expects the application to use (the highest common version), as well as the lowest and highest versions it is capable of. If the call fails, WFSVERSION is still returned, to help with analysis of the failure.

The version numbers refer to the API specification, specifically functions, parameters, data structures, error codes, and messages. If there are any changes to these, the version number should be changed.

This version negotiation allows an XFS application and the XFS Manager to operate successfully if there is any overlap in their versions. The following chart gives examples of how **WFSStartUp** works in conjunction with different application and XFS Manager versions:

dwVersionsRequired (Versions required by Application):	lpWFSVersion.wLowVersion lpWFSVersion.wHighVersion (XFS Manager versions):	Return status from **WFSStartUp**:	lpWFSVersion .wVersion (Result):
0x00010001 (1.00)	0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x00010A02 (1.00 - 2.10)	0x0001 0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x0B010B01 (1.11)	0x0001 0x0002 (1.00 - 2.00)	WFS_SUCCESS	0x0B01 (use 1.11)
0x0B020003 (2.11 - 3.00)	0x0001 0x1402 (1.00 - 2.20)	WFS_SUCCESS	0x1402 (use 2.20)
0x00010001 (1.00)	0x1402 0x0003 (2.20 - 3.00)	WFS_ERR_API_VER_TOO_LOW	0x0000 (fails)
0x0B010003 (1.11 - 3.00)	0x0001 0x0001 (1.00)	WFS_ERR_API_VER_TOO_HIGH	0x0000 (fails)

Note that a version negotiation error also generates a system event (see Section 10.8).

After making its last XFS call, an application ***must*** call **WFSCleanUp** to allow the XFS Manager to release any resources allocated for the application.

Error Codes

The return value indicates whether the application was registered successfully (i.e. the XFS Manager can support requests from the application). If the function was successful, the returned value is WFS_SUCCESS; if not, it is one of the following error conditions:

WFS_ERR_ALREADY_STARTED
A **WFSStartUp** has already been issued by the application, without an intervening **WFSCleanUp**.

WFS_ERR_API_VER_TOO_HIGH
The range of versions of XFS API support requested by the application is higher than any supported by this particular XFS implementation.

WFS_ERR_API_VER_TOO_LOW
The range of versions of XFS API support requested by the application is lower than any supported by this particular XFS implementation.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

See Also

WFSCleanUp

5.24 WFSUnhookBlockingHook

HRESULT **WFSUnhookBlockingHook**( )

Removes any previous blocking hook that had been installed and reinstalls the default blocking mechanism.

Parameters None.

Mode Immediate

Comments The function will always install the ***default*** routine, not the ***previous*** routine. If an application wishes to nest blocking hook routines - i.e. to establish a temporary blocking call and then revert to the previous mechanism - it must save and restore the value returned by the **WFSSetBlockingHook** function. See Section 4.12.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFSSetBlockingHook**

5.25 WFSUnlock

HRESULT **WFSUnlock**(*hService*)

Releases a service that has been locked by a previous **WFSLock** or **WFSAsyncLock** function. The synchronous version of **WFSAsyncUnlock**.

Parameters **HSERVICE** *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen.**

Mode Synchronous

Comments See Section 4.8.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_CANCELED
The request was canceled by **WFSCancelBlockingCall**.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_NOT_LOCKED
The application requesting a service be unlocked had not previously performed a successful **WFSLock** or **WFSAsyncLock**.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFSAsyncUnlock, WFSLock, WFSAsyncLock**

5.26 WFSAsyncUnlock

HRESULT **WFSAsyncUnlock**(*hService, hWnd, lpRequestID*)

Releases a service that has been locked by a previous **WFSLock** or **WFSAsyncLock** function. The asynchronous version of **WFSUnlock**.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen.**

HWND *hWnd*
The window handle which is to receive the completion message for this request.

LPREQUESTID *lpRequestID*
Pointer to the request identifier for this request (returned parameter).

Mode Asynchronous

Comments See **WFSUnlock** and Section 4.8.

The application ***must*** call **WFSFreeResult** to deallocate the WFSRESULT data structure which is pointed to by the completion message. Note that a WFSRESULT structure may be returned even if the function completes with an error; see Section 4.14.

Messages WFS_UNLOCK_COMPLETE

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure:

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_NOT_LOCKED
The application requesting a service be unlocked had not previously performed a successful **WFSLock** or **WFSAsyncLock**.

See Also **WFSUnlock, WFSLock, WFSAsyncLock**

6 Service Provider Interface (SPI) Functions

The Service Provider functions are described in the following sections, in alphabetical order. The table below shows the SPI functions, the sections in which they are defined, their modes, and the API functions they implement. The asynchronous SPI functions behavior is influenced by whether the function is Deferred or Non-deferred [see section 4.8 Exclusive Service and Device Access]. An asynchronous non-deferred function (for example WFPRegister) can be processed completely by the service as soon as it is received. An asynchronous deferred function (for example WFPExecute) cannot be processed completely as soon as it arrives, because it may require hardware and/or operator interaction.

Section	XFS SPI	Mode	XFS API	Mode
6.1	**WFPCancelAsyncRequest**	Immediate	**WFSCancelAsyncRequest**	Immediate
6.1	**WFPCancelAsyncRequest**	Immediate	**WFSCancelBlockingCall**	Immediate
	(none)	-	**WFSCleanUp**	Synchronous
6.2	**WFPClose**	Asynchronous	**WFSClose**	Synchronous
6.2	**WFPClose**	Asynchronous	**WFSAsyncClose**	Asynchronous
	(none)	-	**WFSCreateAppHandle**	Immediate
6.3	**WFPDeregister**	Asynchronous	**WFSDeregister**	Synchronous
6.3	**WFPDeregister**	Asynchronous	**WFSAsyncDeregister**	Asynchronous
	(none)	-	**WFSDestroyAppHandle**	Immediate
6.4	**WFPExecute**	Asynchronous	**WFSExecute**	Synchronous
6.4	**WFPExecute**	Asynchronous	**WFSAsyncExecute**	Asynchronous
	(none)	-	**WFSFreeResult**	Immediate
6.5	**WFPGetInfo**	Asynchronous	**WFSGetInfo**	Synchronous
6.5	**WFPGetInfo**	Asynchronous	**WFSAsyncGetInfo**	Asynchronous
	(none)	-	**WFSIsBlocking**	Immediate
6.6	**WFPLock**	Asynchronous	**WFSLock**	Synchronous
6.6	**WFPLock**	Asynchronous	**WFSAsyncLock**	Asynchronous
6.7	**WFPOpen**	Asynchronous	**WFSOpen**	Synchronous
6.7	**WFPOpen**	Asynchronous	**WFSAsyncOpen**	Asynchronous
6.8	**WFPRegister**	Asynchronous	**WFSRegister**	Synchronous
6.8	**WFPRegister**	Asynchronous	**WFSAsyncRegister**	Asynchronous
	(none)	-	**WFSSetBlockingHook**	Immediate
6.9	**WFPSetTraceLevel**	Immediate	(none)	-
	(none)	-	**WFSStartUp**	Immediate
	(none)	-	**WFSUnhookBlockingHook**	Immediate
6.10	**WFPUnloadService**			
6.11	**WFPUnlock**	Asynchronous	**WFSUnlock**	Synchronous.
6.11	**WFPUnlock**	Asynchronous	**WFSAsyncUnlock**	Asynchronous

Note that in this section device drivers and devices are mentioned frequently, instead of Service Providers and services. This is due primarily to the fact that access to financial peripheral devices is the first category of financial services being addressed by the BSVC. However, note that in the future other financial services will be part of the Extensions to Financial Services, and will also use these interfaces, with additions as necessary. See Appendix A for more on this subject.

6.1 WFPCancelAsyncRequest

HRESULT **WFPCancelAsyncRequest**(*hService, RequestID*)

Cancels the specified (or every) asynchronous request being performed on the specified Service Provider, before its (their) completion.

Parameters

HSERVICE *hService*
Handle to the Service Provider.

REQUESTID *RequestID*
The request identifier (NULL to cancel all requests for the specified *hService*).

Mode

Immediate. Although the cancellation process itself is asynchronous, the completion message(s) are associated with the original request, not the cancel request (even if they indicate a WFS_ERR_CANCELED status).

Comments

If the *RequestID* parameter is set to NULL, the command will cancel ***all*** asynchronous requests on the specified service that are in progress on behalf of the calling application.

A previously initiated asynchronous request is canceled prior to completion by issuing the **WFSCancelAsyncRequest** function, specifying the request identifier returned by the asynchronous function. This function is immediate with respect to its calling application, but the cancellation process is inherently asynchronous. On completion, the specified request (or all the requests) will have finished, with a completion message indicating a status of WFS_ERR_CANCELED, unless the cancel request was made after the request had completed.

The cancellation applies to the Service Provider level. The request is passed through the SPI, and the Service Provider normally then also cancels any physical I/O or other device operation in progress, in the appropriate manner for the device or service.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_REQ_ID
The *RequestID* parameter does not correspond to an outstanding request on the service.

6.2 WFPClose

HRESULT **WFPClose**(*hService, hWnd, ReqID*)

Terminates a session (a series of service requests initiated with the **WFPOpen** SPI function) between the XFS Manager and the specified Service Provider.

Parameters

HSERVICE *hService*
Handle to the Service Provider.

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

Mode Asynchronous

Comments **WFPClose** directs the service to free all resources associated with the series of requests made using the *hService* parameter. If the service is locked by the application, the close automatically unlocks it. If no **WFPDeregister** has been issued, it is automatically performed.

See **WFPOpen** and Section 4.6 for further discussion.

Messages WFS_CLOSE_COMPLETE

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. The service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

6.3 WFPDeregister

HRESULT **WFPDeregister**(*hService, dwEventClass, hWndReg, hWnd, ReqID*)

Discontinues monitoring of the specified message class(es) from the specified Service Provider, by the specified *hWndReg* (or all *hWnd*'s).

Parameters

HSERVICE *hService*
Handle to the Service Provider

DWORD *dwEventClass*
The class(es) of messages from which the application is deregistering. Specified as a set of bit masks that can be logically ORed together. A NULL value requests that ***all*** message classes be deregistered from the specified window for this Service Provider.

HWND *hWndReg*
The window to which notification messages are posted. A NULL value requests that ***all*** the application's windows be deregistered from the specified message class(es) for this *hService*.

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

Mode

Asynchronous

Comments

WFPDeregister does not stop asynchronous command completion messages from being posted; a robust application should be designed to accept these messages even after a deregister is issued.

A **WFPDeregister** is performed automatically if a **WFPClose** is issued without a previous **WFPDeregister**.

To deregister ***all*** messages for ***all*** *hWnd*s, the call supplies NULL values for both the *dwEventClass* and *hWnd* parameters.

See the **WFPRegister** function for a description of the types of events that may be monitored.

Messages

WFS_DEREGISTER_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_EVENT_CLASS
The *dwEventClass* parameter specifies one or more event classes not supported by the service.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_HWNDREG
The *hWndReg* parameter is not a valid window handle.

WFS_ERR_NOT_REGISTERED
The specified *hWndReg* window was not registered to receive messages for any event classes.

The following error condition is returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

6.4 WFPExecute

HRESULT **WFPExecute**(*hService, dwCommand, lpCmdData, dwTimeOut, hWnd, ReqID*)

Sends asynchronous service class specific commands to a Service Provider.

Parameters

HSERVICE *hService*
Handle to the Service Provider.

DWORD *dwCommand*
Command to be executed.

LPVOID *lpCmdData*
Pointer to the data structure to be passed.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

Mode

Asynchronous

Comments

See **WFSExecute**.

Messages

WFS_EXECUTE_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_COMMAND
The *dwCommand* issued is not supported by this service class.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_UNSUPP_COMMAND
The *dwCommand* issued, although valid for this service class, is not supported by this Service Provider.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INVALID_DATA
The data structure passed as input parameter contains invalid data.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_LOCKED
The service is locked under a different *hService*.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_USER_ERROR
A user is preventing proper operation of the device.

WFS_ERR_UNSUPP_DATA
The data structure passed as an input parameter although valid for this service class, is not supported by this Service Provider or device.

WFS_ERR_FRAUD_ATTEMPT
Some devices are capable of identifying a malicious physical attack which attempts to defraud valuable information or media. In these cases, this error code is returned to indicate the user is attempting a fraudulent act on the device.

6.5 WFPGetInfo

HRESULT **WFPGetInfo**(*hService, dwCategory, lpQueryDetails, dwTimeOut, hWnd, ReqID*)

Retrieves various kinds of information from the specified Service Provider.

Parameters

HSERVICE *hService*
Handle to the Service Provider.

DWORD *dwCategory*
Specifies the category of the query (e.g. for a printer, WFS_INF_PTR_STATUS to request status or WFS_INF_PTR_CAPABILITIES to request capabilities). The available categories depend on the service class, the Service Provider and the service. The information requested can be either static or dynamic, e.g. basic service capabilities (static) or current service status (dynamic).

LPVOID *lpQueryDetails*
Pointer to the data structure to be passed to the Service Provider, containing further details to make the query more precise, e.g. a form name. (Many queries have no input parameters, in which case this pointer is NULL.)

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

Mode

Asynchronous

Comments

The XFS Manager retrieves the information requested from the Service Provider itself, and, since the information can be stored remotely, the function cannot be guaranteed to complete immediately. Note that, typically, requests for generic and class specific categories ***can*** complete immediately. See **WFSGetInfo** for additional discussion.

The specifications for the information structures for each service class can be found in the specifications for the service-specific commands.

Messages

WFS_GETINFO_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_CATEGORY
The *dwCategory* issued is not supported by this service class.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_UNSUPP_CATEGORY
The *dwCategory* issued, although valid for this service class, is not supported by this Service Provider.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_DATA
The data structure passed as input parameter contains invalid data.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_USER_ERROR
A user is preventing proper operation of the device.

WFS_ERR_UNSUPP_DATA
The data structure passed as an input parameter although valid for this service class, is not supported by this Service Provider or device.

6.6 WFPLock

HRESULT **WFPLock**(*hService, dwTimeOut, hWnd, ReqID*)

Establishes exclusive control by the calling application over the specified service.

Parameters

HSERVICE *hService*
Handle to the Service Provider.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

Mode Asynchronous

Comments See **WFSLock**.

Messages WFS_LOCK_COMPLETE

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_TIMEOUT
The timeout interval expired.

6.7 WFPOpen

HRESULT **WFPOpen**(*hService, lpszLogicalName, hApp, lpszAppID, dwTraceLevel, dwTimeOut, hWnd, ReqID, hProvider, dwSPIVersionsRequired, lpSPIVersion, dwSrvcVersionsRequired, lpSrvcVersion*)

Establishes a connection between the XFS Manager and the Service Provider that supports the specified service, and initiates a session (a series of service requests terminated with the **WFPClose** function).

Parameters

HSERVICE *hService*
The service handle to be associated with the session being opened.

~~**LPSTR**~~**LPCSTR** *lpszLogicalName*
Points to a null-terminated string containing the pre-defined logical name of a service. It is a high level name such as "SYSJOURNAL1," "PASSBOOKPTR3" or "ATM02," that is used by the XFS Manager and the Service Provider as a key to obtain the specific configuration information they need.

HAPP *hApp*
The application handle to be associated with the session being opened.
See **WFSCreateAppHandle** and Sections 4.5 and 4.8.2 for details.

~~**LPSTR**~~**LPCSTR** *lpszAppID*
Pointer to a null terminated string containing the application ID; the pointer may be NULL if the ID is not used.

DWORD *dwTraceLevel*
See **WFPSetTraceLevel**.

DWORD *dwTimeOut*
Number of milliseconds to wait for completion (WFS_INDEFINITE_WAIT to specify a request that will wait until completion).

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

HPROVIDER *hProvider*
Service Provider handle supplied by the XFS Manager - used by the Service Provider to identify itself when calling the **WFMReleaseDLL** function.

DWORD *dwSPIVersionsRequired*
Specifies the range of XFS SPI versions that the XFS Manager can support. (See Comments.) The low-order word indicates the highest version the XFS Manager can support; the high-order word indicates the lowest version the XFS Manager can support. In each word, the low-order byte specifies the major version number and the high-order byte specifies the minor version number (i.e. the numbers before and after the decimal).
Note: in order to allow intermediate minor revisions (e.g. between 1.10 and 1.20), the minor version number should always be expressed as two decimal digits, i.e. 1.10, 1.11, 1.20, etc.

LPWFSVERSION *lpSPIVersion*
Pointer to the data structure that is to receive SPI version support information and (optionally) other details about the SPI implementation (returned parameter).

DWORD *dwSrvcVersionsRequired*
Service-specific interface versions required; see *dwSPIVersionsRequired* above, and **WFSOpen**.

LPWFSVERSION *lpSrvcVersion*
Pointer to the service-specific interface implementation information; see *lpSPIVersion* above, and **WFSOpen** (returned parameter).

Mode Asynchronous

Comments

This function establishes the connection between the XFS Manager and the Service Provider, including version negotiation and passing of implementation information, and initiates a session between the application and the service. This call is made by the XFS Manager each time any application issues a **WFSOpen** or **WFSAsyncOpen** call to the specified service (immediately after loading the Service Provider DLL, if it is not already loaded).

In order to support future XFS implementations with maximum flexibility, two version negotiations take place in **WFPOpen**. In the first, the XFS Manager specifies in the *dwSPIVersionsRequired* parameter the range of versions of the XFS SPI specification which it can support. If the range of versions specified by the XFS Manager overlaps the range of versions that the Service Provider can support, the call succeeds. Otherwise the call fails.

The WFSVERSION data structure (described in Section 9.2) is used by the Service Provider to return the version of SPI support it expects the XFS Manager to use (the highest common version), as well as the lowest and highest versions it is capable of. In addition, this structure is used optionally by the XFS Manager to specify other information about the Service Provider implementation. If the call fails, WFSVERSION is still returned, to help with analysis of the failure.

The version numbers refer to the SPI specification, specifically functions, parameters, data structures, error codes, and messages. If there are any changes to these, the version number should be changed.

This version negotiation allows the XFS Manager and a Service Provider to operate successfully if there is any overlap in their versions. The following chart gives examples of how **WFPOpen** works in conjunction with different XFS Manager and Service Provider versions:

dwSPIVersions-Required (Versions required by XFS Manager):	lpSPIVersion.wLowVersion lpSPIVersion.wHighVersion (Service Provider versions):	Return status from **WFPOpen**:	lpSPIVersion .wVersion (Result):
0x00010001 (1.00)	0x0001 0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x00010A02 (1.00 - 2.10)	0x0001 0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x0B010B01 (1.11)	0x0001 0x0002 (1.00 - 2.00)	WFS_SUCCESS	0x0B01 (use 1.11)
0x0B020003 (2.11 - 3.00)	0x0001 0x1402 (1.00 - 2.20)	WFS_SUCCESS	0x1402 (use 2.20)
0x00010001 (1.00)	0x1402 0x0003 (2.20 - 3.00)	WFS_ERR_SPI_VER_TOO_LOW	0x0000 (fails)
0x0B010003 (1.11 - 3.00)	0x0001 0x0001 (1.00)	WFS_ERR_SPI_VER_TOO_HIGH	0x0000 (fails)

The second negotiation is in relation to the service-specific interface, between the application program and the Service Provider. The following chart gives examples of how **WFPOpen** works in conjunction with different application and Service Provider versions. See **WFSOpen**, Section 5.19, for details.

dwSrvcVersions-Required (Versions required by the application):	lpSrvcVersion.wLowVersion lpSrvcVersion.wHighVersion (Service Provider versions):	Return status from **WFPOpen**:	lpSrvcVersion .wVersion (Result):
0x00010001 (1.00)	0x0001 0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x00010A02 (1.00 - 2.10)	0x0001 0x0001 (1.00)	WFS_SUCCESS	0x0001 (use 1.00)
0x0B010B01 (1.11)	0x0001 0x0002 (1.00 - 2.00)	WFS_SUCCESS	0x0B01 (use 1.11)
0x0B020003 (2.11 - 3.00)	0x0001 0x1402 (1.00 - 2.20)	WFS_SUCCESS	0x1402 (use 2.20)
0x00010001 (1.00)	0x1402 0x0003 (2.20 - 3.00)	WFS_ERR_SRVC_VER_TOO_LOW	0x0000 (fails)
0x0B010003 (1.11 - 3.00)	0x0001 0x0001 (1.00)	WFS_ERR_SRVC_VER_TOO_HIGH	0x0000 (fails)

Note that a version negotiation error also generates a system event (see Section 10.8).

Also, see **WFSStartUp**, Section 5.23.

Messages WFS_OPEN_COMPLETE

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_INVALID_TRACELEVEL
The *dwTraceLevel* parameter does not correspond to a valid trace level or set of levels.

WFS_ERR_SPI_VER_TOO_HIGH
The range of versions of XFS SPI support requested by the XFS Manager is higher than any supported by this particular Service Provider.

WFS_ERR_SPI_VER_TOO_LOW
The range of versions of XFS SPI support requested by the XFS Manager is lower than any supported by this particular Service Provider.

WFS_ERR_SRVC_VER_TOO_HIGH
The range of versions of the service-specific interface support requested by the application is higher than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SRVC_VER_TOO_LOW
The range of versions of the service-specific interface support requested by the application is lower than any supported by the Service Provider for the logical service being opened.

WFS_ERR_VERSION_ERROR_IN_SRVC
Within the service, a version mismatch of two modules occurred.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. The service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_VERSION_ERROR_IN_SRVC
Within the service, a version mismatch of two modules occurred.

6.8 WFPRegister

HRESULT **WFPRegister**(*hService, dwEventClass, hWndReg, hWnd, ReqID*)

Enables event monitoring for the specified service by the specified *hWndReg*; all events of the specified class(es) generate messages to the *hWndReg*.

Parameters

HSERVICE *hService*
Handle to the Service Provider.

DWORD *dwEventClass*
The class(es) of events for which the application is registering. Specified as a set of bit masks that can be logically ORed together.

HWND *hWndReg*
The window handle which is to be registered to receive the specified messages.

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

Mode

Asynchronous

Comments

WFPDeregister is used to cancel notifications. See **WFSRegister**.

Messages

WFS_REGISTER_COMPLETE

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_EVENT_CLASS
The *dwEventClass* parameter specifies one or more event classes not supported by the service.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_HWNDREG
The *hWndReg* parameter is not a valid window handle.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

6.9 WFPSetTraceLevel

HRESULT **WFPSetTraceLevel**(*hService, dwTraceLevel*)

Sets the specified trace level(s) at run time, in and/or below the Service Provider. See **WFMSetTraceLevel**.

Parameters

HSERVICE *hService*
Handle to the Service Provider.

DWORD *dwTraceLevel*
The level(s) of tracing being requested. See below.

Mode Immediate

Comments

Issuing **WFPSetTraceLevel** for a service enables tracing on that service at various levels. The predefined trace levels that can be used in this function, with their meanings to the Service Provider, are as follows (see **WFMSetTraceLevel** for the API and support function trace levels):

`WFS_TRACE_SPI 0x00000004`

Trace all the SPI calls to the Service Provider, and notification and event messages generated by the Service Provider, that are associated with the specified *hService*.

`WFS_TRACE_ALL_SPI 0x00000008`

Trace ***all*** SPI, notification and event activity of the Service Provider (the *hService* parameter is not relevant to this trace level).

Other standard trace levels may be defined in the future, and a range of trace level values (the high order 16 bits of this parameter) is reserved for use by individual Service Providers. Example of other functions that may be traced include network messages, interactions between the Service Provider and service, and device interface interaction.

Trace level values can be ORed together in a single *dwTraceLevel* parameter to request more than one kind of tracing be started. A NULL value stops all tracing in the Service Provider.

If more than one process may be using the trace facility, this function should always be preceded with the **WFMGetTraceLevel** function. This value returned by this function is ORed together with the new trace level(s), and the resulting value is used with **WFMSetTraceLevel**, thus adding the new trace level(s) to whatever the existing trace level(s) had been,

This function has the highest priority to the Service Provider; it activates the trace as soon as possible.

WFPOpen also includes an option to set these trace levels, to allow the open process itself to be traced.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_TRACELEVEL
The *dwTraceLevel* parameter does not correspond to a valid trace level or set of levels.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFMGetTraceLevel, WFSOpen, WFSAsyncOpen**

6.10 WFPUnloadService

HRESULT WFPUnloadService( )

Asks the called Service Provider whether it is OK for the XFS Manager to unload the Service Provider's DLL.

Parameters None

Mode Immediate

Comments This function is issued after the XFS Manager has received a **WFMReleaseDLL** request from the Service Provider or during the processing of the **WFSCleanUp** command. The Service Provider returns WFS_SUCCESS only if it has fully "cleaned up," i.e. has freed any resources it has allocated, has no separate threads running, etc. If this is not true, it returns the error below, and initiates or continues the clean up process.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_NOT_OK_TO_UNLOAD
The XFS Manager may not unload the Service Provider DLL at this time. It will repeat this request to the Service Provider until the return is WFS_SUCCESS, or until a new session is started by an application with this Service Provider.

6.11 WFPUnlock

HRESULT **WFPUnlock**(*hService, hWnd, ReqID*)

Releases a service that has been locked by a previous **WFPLock** function.

Parameters

HSERVICE *hService*
Handle to the Service Provider

HWND *hWnd*
The window handle which is to receive the completion message for this request.

REQUESTID *ReqID*
Request identification number.

Mode Asynchronous

Comments See **WFPLock**, **WFSLock**, **WFSUnlock** and Section 4.9.

Messages WFS_UNLOCK_COMPLETE

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions, indicating that the asynchronous operation was not initiated. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

The following error conditions are returned via the asynchronous command completion message, as the *hResult* from the WFSRESULT structure. Any service-specific errors that can be returned are defined in the specifications for each service class.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest**.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_NOT_LOCKED
The service to be unlocked is not locked under the calling *hService*.

7 Support Functions

Support functions are services of the XFS Manager used by Service Providers and applications. All the functions are ***immediate***, since they are completely processed inside the XFS Manager, or use only immediate functions of the Service Providers.

7.1 WFMAllocateBuffer

HRESULT **WFMAllocateBuffer**(*ulSize, ulFlags, lppvData*)

Allocates a memory buffer for the Service Provider in which to return results.

Parameters

ULONG *ulSize*
Size (in bytes) of the memory to be allocated.

ULONG *ulFlags*
Flags, see comments below.

LPVOID **lppvData*
Address of the variable in which the XFS Manager will place the pointer to the allocated memory.

Comments

A Service Provider ***must*** use this call when creating data structures for the XFS Manager or an application to use, and may use it when allocating memory for its own private use. The flags can be ORed together, and specify:

WFS_MEM_SHARE	Allocates shareable memory.
WFS_MEM_ZEROINIT	Initializes memory contents to zero (not required in Win32 or Win64).

The application, XFS Manager or Service Provider then ***must***, in turn, use the **WFSFreeResult** or **WFMFreeBuffer** functions to deallocate the memory.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_OUT_OF_MEMORY
There is not enough memory available to satisfy the request.

See Also

WFMAllocateMore, WFMFreeBuffer, WFSFreeResult and Section 4.14.

7.2 WFMAllocateMore

HRESULT **WFMAllocateMore**(*ulSize, lpvOriginal, lppvData*)

Allocates a memory buffer, linking it to a previously allocated one.

Parameters

ULONG *ulSize*
Size (in bytes) of the memory to be allocated

LPVOID *lpvOriginal*
Address of the original buffer to which the newly allocated buffer should be linked

LPVOID **lppvData*
Address of the variable in which the XFS Manager will place the pointer to the newly allocated memory.

Comments

This function allocates an additional memory buffer and link it to one previously allocated by **WFMAllocateBuffer**. The returned buffer has the same properties as the previous buffer (i.e. the WFS_MEM_SHARE and WFS_MEM_ZEROINIT flags) and it can be freed ***only*** by freeing the original buffer (using **WFMFreeBuffer** or **WFSFreeResult**).

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INVALID_ADDRESS
The *lpvOriginal* parameter does not point to a previously allocated buffer.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_OUT_OF_MEMORY
There is not enough memory available to satisfy the request.

See Also

WFMAllocateBuffer, WFMFreeBuffer, WFSFreeResult and Section 4.14.

7.3 WFMFreeBuffer

HRESULT **WFMFreeBuffer**(*lpvData*)

Releases the memory buffer(s) allocated by **WFMAllocateBuffer** and **WFMAllocateMore**.

Parameters **LPVOID** *lpvData*
Address of the memory buffer to free.

Comments See **WFMAllocateBuffer** and **WFSFreeResult**. This function frees a set of one or more linked buffers, as does the **WFSFreeResult** API function, except that it is used by Service Providers to free memory that they have allocated for "private" use, via the **WFMAllocateBuffer** and **WFMAllocateMore** functions.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INVALID_BUFFER
The *lpvData* parameter is not a pointer to an allocated buffer structure.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

See Also **WFMAllocateBuffer, WFMAllocateMore, WFSFreeResult and Section 4.14.**

7.4 WFMGetTraceLevel

HRESULT **WFMGetTraceLevel**(*hService, lpdwTraceLevel*)

Returns the trace level associated with the specified *hService* (at run time). See **WFMSetTraceLevel**.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen**.

LPDWORD *lpdwTraceLevel*
Pointer to the value defining the current trace level (returned parameter).

Mode Immediate

Comments This function returns the current tracing levels in the XFS Manager and the Service Provider specified by *hService*. See **WFMSetTraceLevel**.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFMSetTraceLevel, WFSOpen, WFSAsyncOpen**

7.5 WFMKillTimer

HRESULT **WFMKillTimer**(*wTimerID*)

Cancels the timer identified by the *wTimerID* parameter. Any pending WFS_TIMER_EVENT message associated with the timer is removed from the message queue.

Parameters **WORD** *wTimerID*
ID of the timer to be canceled.

Comments See **WFMSetTimer**.

Error Codes If the function return is not WFS_SUCCESS, it is the following error condition:

WFS_ERR_INVALID_TIMER
The *wTimerID* parameter does not correspond to a currently active timer.

7.6 WFMOutputTraceData

HRESULT **WFMOutputTraceData**(*lpszData*)

Requests the XFS Manager to output the specified data to the current trace destination.

Parameters ~~LPSTR~~LPCSTR *lpszData*
Pointer to a null-terminated string containing the trace data.

Comments Normally used by a Service Provider that has been requested via **WFMSetTraceLevel** to trace its operation. The XFS Manager adds standard header information (timestamp, etc.) to the data before writing it to the trace stream. Note that the XFS Manager also writes data to the trace stream if the appropriate trace level(s) have been requested.

Error Codes If the function return is not WFS_SUCCESS, it is the following error condition:

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

7.7 WFMReleaseDLL

HRESULT **WFMReleaseDLL**(*hProvider*)

Notifies the XFS Manager that the Service Provider is available to be unloaded from memory.

Parameters **HPROVIDER** *hProvider*
Handle to the Service Provider, obtained from the XFS Manager in the **WFPOpen** call.

Comments This function initiates the process in which the Service Provider is unloaded from memory by the XFS Manager. However, note that the Manager must issue the **WFPUnloadService** function to the Service Provider before it actually unloads the Service Provider DLL. The recommended procedure is as follows:

- The Service Provider finishes processing the **WFPClose** for its last open session
- The Service Provider does appropriate cleanup (deallocating memory, killing separate threads, etc.)
- The Service Provider posts the WFS_CLOSE_COMPLETE message for the final close
- The Service Provider calls **WFMReleaseDLL**, and after the return from this call, does nothing other than return from the procedure that called **WFMReleaseDLL**
- The XFS Manager calls **WFPUnloadService**, verifying that the Service Provider is in fact still ready to be unloaded
- If the Service Provider says OK, the XFS Manager unloads the Service Provider DLL

Error Codes If the function return is not WFS_SUCCESS, it is the following error condition:

WFS_ERR_INVALID_HPROVIDER
The *hProvider* parameter is not a valid provider handle.

7.8 WFMSetTimer

HRESULT **WFMSetTimer**(*hWnd, lpContext, dwTimeVal, lpwTimerID*)

Starts a system timer.

Parameters

HWND *hWnd*
The window to which the requested timer message is to be posted.

LPVOID *lpContext*
Context pointer supplied by the Service Provider requesting the timer; may be NULL.

DWORD *dwTimeVal*
Timer value (in milliseconds).

LPWORD *lpwTimerID*
Pointer to the timer identifier (returned parameter).

Comments

The **WFMSetTimer** function requests the XFS Manager to start a system timer with the specified time value; when that time interval expires, the XFS Manager posts a WFS_TIMER_EVENT message to the specified *hWnd*, containing the *wTimerID* value and the *lpContext* pointer.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

7.9 WFMSetTraceLevel

HRESULT **WFMSetTraceLevel**(*hService, dwTraceLevel*)

Sets the specified trace level(s) at run time; to be used for debugging and testing purposes.

Parameters

HSERVICE *hService*
Handle to the Service Provider as returned by **WFSOpen** or **WFSAsyncOpen**.

DWORD *dwTraceLevel*
The level(s) of tracing being requested. See below.

Mode Immediate

Comments Issuing **WFMSetTraceLevel** for a service enables tracing on that service at various levels. Five standard trace levels are predefined:

`WFS_TRACE_API` `0x00000001`

Trace all input and output parameters of all API function calls using the specified *hService*.

`WFS_TRACE_ALL_API` `0x00000002`

Trace all input and output parameters of ***all*** API function calls associated with the Service Provider identified by the specified *hService*, ***not*** just the ones associated with the specified *hService*.

`WFS_TRACE_SPI` `0x00000004`

Trace all input and output parameters of all SPI function calls associated with the specified *hService*, as well as all notification and event messages generated by the Service Provider for the *hService*.

`WFS_TRACE_ALL_SPI` `0x00000008`

As for WFS_TRACE_ALL_API, but trace ***all*** SPI, notification and event activity on the Service Provider, ***not*** just that associated with the specified *hService*.

`WFS_TRACE_MGR` `0x00000010`

Trace the support functions (**WFMxxxxx**) of the XFS Manager.

Other standard trace levels may be defined in the future, and a range of trace level values (the high order 16 bits of this parameter) is reserved for use by individual Service Providers. Examples of other functions that may be traced include network messages, interactions between the Service Provider and service, and device interface interaction.

Trace level values can be ORed together in a single *dwTraceLevel* parameter to request more than one kind of tracing be started. A NULL value stops all tracing.

If more than one process may be using the trace facility, this function should always be preceded with a call to the **WFMGetTraceLevel** function. This value returned by this function is ORed together with the new trace level(s), and the resulting value is used with **WFMSetTraceLevel**, thus adding the new trace level(s) to whatever the existing trace level(s) had been,

This function has the highest priority to the XFS Manager and the Service Provider; they activate the trace as soon as possible. Note that the XFS Manager performs all the traces defined above, other than the completion and event messages posted by the Service Providers.

WFSOpen and **WFSAsyncOpen** also include an option to set these trace levels, to allow the open process itself to be traced.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_TRACELEVEL
The *dwTraceLevel* parameter does not correspond to a valid trace level or set of levels.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

See Also **WFMGetTraceLevel, WFPSetTraceLevel, WFSOpen, WFSAsyncOpen**

8 Configuration Functions

See Section 4.7 for the overall discussion of configuration information and how it is stored within the Windows Registry.

8.1 WFMCloseKey

HRESULT **WFMCloseKey** (*hKey*)

Closes the specified key.

Parameters **HKEY** *hKey*
Handle to the currently open key that is to be closed.

Comments The *hKey* handle can not be used after it has been closed, because it will no longer be valid. Note that it is not valid to close the XFS root key (passing one of the pre-defined handles as the value for the *hKey* parameter).

Error Codes If the function return is not WFS_SUCCESS, it is the following error condition:

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key, or it is the XFS root.

8.2 WFMCreateKey

HRESULT **WFMCreateKey** (*hKey, lpszSubKey, phkResult, lpdwDisposition*)

Creates a new key, or if the specified key exists, opens it.

The first use of *hKey* by a process sets the migration mode for that process. The use of this function is an application decision: the XFS Manager must not automatically migrate the registry values at load time.

Be aware that when the WFMCreateKey is used for the first time and the *hKey* parameter is set to WFS_CFG_HKEY_XFS_ROOT then the existing registry structure will be migrated from HKEY_CLASSES_ROOT to HKEY_LOCAL_MACHINE. If ~~either~~any of the ~~new~~other values WFS_CFG_HKEY_MACHINE_XFS_ROOT, WFS_CFG_HKEY_USER_DEFAULT_XFS_ROOT or WFS_CFG_CURRENT_USER~~_DEFAULT~~_XFS_ROOT are used then no migration will take place for this process. The assumption is that any process using the new key values will be doing its own migration. The reason migration does not always take place is that some applications will require access to both the old and new key roots so that they can migrate their non-CEN keys and values.

WFS_CFG_HKEY_XFS_ROOT is defined in XFS 2.x as HKEY_CLASSES_ROOT\WOSA/XFS_ROOT.

Parameters

HKEY *hKey*
Handle to a currently open key, or one of the predefined handles.

The key opened or created by this function is a subkey of the key identified by this parameter.

~~**LPSTR**~~**LPCSTR** *lpszSubKey*
Pointer to a null-terminated string containing the name of the key to be created or opened.

PHKEY *phkResult*
Pointer to a variable that receives the handle of the created or opened key.

LPDWORD *lpdwDisposition*
Pointer to a variable that receives one of the disposition values:

WFS_CFG_CREATED_NEW_KEY
WFS_CFG_OPENED_EXISTING_KEY

Comments

If this function creates a new key, it has no values. The **WFMSetValue** function is used to create values.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions:

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

8.3 WFMDeleteKey

HRESULT **WFMDeleteKey** (*hKey, lpszSubKey*)

Deletes the specified key. This function cannot delete a key that has subkeys.

Parameters

HKEY *hKey*
Handle to a currently open key, or one of the predefined handles.

The key specified by the *lpszSubKey* parameter must be a subkey of the key identified by this parameter.

~~LPSTR~~LPCSTR *lpszSubKey*
Pointer to a null-terminated string specifying the name of the key to be deleted.

Comments

If this function succeeds, the specified key is removed from the configuration information. The entire key, including all its values, is removed.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_CFG_INVALID_SUBKEY
The key specified by *lpszSubKey* does not exist.

WFS_ERR_CFG_KEY_NOT_EMPTY
The specified key has subkeys and cannot be deleted. The subkeys must be deleted first.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

8.4 WFMDeleteValue

HRESULT **WFMDeleteValue** (*hKey, lpszValue*)

Deletes the specified value (both name and data).

Parameters **HKEY** *hKey*
Handle to a currently open key, or one of the predefined handles.

~~**LPSTR**~~**LPCSTR** *lpszValue*
Pointer to a null-terminated string specifying the name of the value to be deleted.

Comments The specified value is removed from the specified open key. The **WFMSetValue** function is used to create or modify values.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_CFG_INVALID_VALUE
The specified value does not exist within the specified open key.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

8.5 WFMEnumKey

HRESULT **WFMEnumKey** (*hKey, iSubKey, lpszName, lpcchName, lpftLastWrite*)

Enumerates the subkeys of the specified open key. Retrieves information about one subkey each time it is called.

Parameters

HKEY *hKey*
Handle to a currently open key, or one of the predefined handles.

The keys enumerated by this function are subkeys of the key identified by this parameter.

DWORD *iSubKey*
The index of the subkey to retrieve. This parameter should be zero for the first call to this function, then incremented for each subsequent call, in order to enumerate all the subkeys of the specified open key.

Because subkeys are not ordered, any new subkey will have an arbitrary index. This means that the function may return subkeys in any order.

LPSTR *lpszName*
Pointer to a buffer that receives the name of the subkey, including the terminating null character.

LPDWORD *lpcchName*
Pointer to a variable that specifies the size, in characters, of the buffer specified by the *lpszName* parameter, including the terminating null character. When the function returns, this variable contains the number of characters actually stored in the buffer, ***not*** including the terminating null character.

PFILETIME *lpftLastWrite*
Pointer to a variable that receives the time the enumerated subkey was last written to, in the form of a FILETIME structure (see *Microsoft Win32 Programmer's Reference, Vol. 5*):

```
typedef struct _FILETIME {
    DWORD  dwLowDateTime;
    DWORD  dwHighDateTime;
} FILETIME;
```

Comments

While a program is using this function iteratively, it should not call any other configuration functions that would change the key being enumerated.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_CFG_NO_MORE_ITEMS
There are no more subkeys to be returned (the *iSubKey* parameter is greater than the index of the last subkey).

WFS_ERR_CFG_NAME_TOO_LONG
The length of the name to be returned exceeds the length of the buffer.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

8.6 WFMEnumValue

HRESULT **WFMEnumValue** (*hKey, iValue, lpszValue, lpcchValue, lpszData, lpcchData*)

Enumerates the values of the specified open key. Retrieves the name and data for one value each time it is called.

Parameters

HKEY *hKey*
Handle to a currently open key, or one of the predefined handles.

The value enumerated by this function is a value of the key identified by this parameter.

DWORD *iValue*
The index of the value to retrieve. This parameter should be zero for the first call to this function, then incremented for each subsequent call, in order to enumerate all the values of the specified open key.

Because values are not ordered, any new value will have an arbitrary index. This means that the function may return values in any order.

LPSTR *lpszValue*
Pointer to a buffer that receives the name of the value, including the terminating null character.

LPDWORD *lpcchValue*
Pointer to a variable that specifies the size, in characters, of the buffer pointed to by the *lpszValue* parameter. This size should include the terminating null character. When the function returns, this variable contains the number of characters actually stored in the buffer, ***not*** including the terminating null character.

LPSTR *lpszData*
Pointer to a buffer that receives the data for the value entry, including the terminating null character. This parameter can be NULL, if the data is not required.

LPDWORD *lpcchData*
Pointer to a variable that specifies the size, in characters, of the buffer pointed to by the *lpszData* parameter, including the terminating null character. When the function returns, this variable contains the number of characters actually stored in the buffer, ***not*** including the terminating null character. Ignored if *lpszData* is NULL.

Comments

While a program is using this function iteratively, it should not call any other configuration functions that would change the key being queried.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_CFG_NO_MORE_ITEMS
There are no more values to be returned (the *iValue* parameter is greater than the index of the last value).

WFS_ERR_CFG_NAME_TOO_LONG
The length of the name to be returned exceeds the length of the buffer.

WFS_ERR_CFG_VALUE_TOO_LONG
The length of the value to be returned exceeds the length of the buffer.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

8.7 WFMOpenKey

HRESULT **WFMOpenKey** (*hKey, lpszSubKey, phkResult*)

Opens the specified key.

Parameters

HKEY *hKey*
Handle to a currently open key, or one of the predefined handles.

The key opened by this function is a subkey of the key identified by this parameter.

~~**LPSTR**~~**LPCSTR** *lpszSubKey*
Pointer to a null-terminated string containing the name of the key to be opened. If this parameter is NULL, or points to an empty string, the function opens another handle to the key identified by the *hKey* parameter (and does ***not*** close any previously opened handles).

PHKEY *phkResult*
Pointer to a variable that receives the handle of the opened key.

Comments

In contrast with the **WFMCreateKey** function, this function does not create the specified key if it does not exist.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_CFG_INVALID_SUBKEY
The key specified by *lpszSubKey* does not exist.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

8.8 WFMQueryValue

HRESULT **WFMQueryValue** (*hKey, lpszValueName, lpszData, lpcchData*)

Retrieves the data for the value with the specified name, within the specified open key.

Parameters

HKEY *hKey*
Handle to a currently open key, or one of the predefined handles.

The value data returned is within the key identified by this parameter.

~~**LPSTR**~~**LPCSTR** *lpszValueName*
Pointer to a null-terminated string containing the name of the value being queried.

LPSTR *lpszData*
Pointer to a buffer that receives the data for the value entry, including the terminating null character.

LPDWORD *lpcchData*
Pointer to a variable that specifies the size, in characters, of the buffer pointed to by the *lpszData* parameter, including the terminating null character. When the function returns, this variable contains the number of characters actually stored in the buffer, ***not*** including the terminating null character.

Comments None.

Error Codes If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_CFG_INVALID_NAME
The value specified by the *lpszValueName* parameter does not exist in the specified key.

WFS_ERR_CFG_VALUE_TOO_LONG
The length of the value to be returned exceeds the length of the buffer.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

8.9 WFMSetValue

HRESULT **WFMSetValue** (*hKey, lpszValueName, lpszData, cchData*)

Stores data in the specified value of the specified key. If the value does not exist, it is created.

Parameters

HKEY *hKey*
Handle to a currently open key, or one of the predefined handles.

The value set or created is within the key identified by this parameter.

~~**LPSTR**~~**LPCSTR** *lpszValueName*
Pointer to a null-terminated string containing the name of the value being set. If a value with this name does not already exist in the specified key, it is added to the key.

~~**LPSTR**~~**LPCSTR** *lpszData*
Pointer to a buffer containing the data (a null-terminated character string) to be stored with the specified value name.

DWORD *cchData*
The size, in characters, of the string pointed to by the *lpszData* parameter, including the terminating null character.

Comments

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration information.

Error Codes

If the function return is not WFS_SUCCESS, it is one of the following error conditions.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

9 Data Structures

9.1 WFSRESULT

This structure has three functions:

- It is the parameter which returns the results of the synchronous **WFSLock**, **WFSExecute** and **WFSGetInfo** commands.
- It is pointed to by ***all*** command completion messages, and delivers completion status (as a result handle) and results data (if any) for ***all*** asynchronous API and SPI calls.
- It is pointed to by ***all*** event notification messages to deliver their contents.

Note that even though in many cases one or more members of this structure are not used, the adoption of a single, standard structure for request results simplifies the implementation and maintenance of applications, Service Providers and the XFS Manager itself.

```
typedef struct _wfs_result {
    REQUESTID RequestID;
    HSERVICE  hService;
    TIMESTAMP tsTimestamp;
    HRESULT   hResult;
    union {
        DWORD     dwCommandCode;
        DWORD     dwEventID;
    } u;
    LPVOID    lpBuffer;
} WFSRESULT, *LPWFSRESULT;
```

The members of this structure are:

Field	Description
RequestID	Request ID of the completed command; not used for event notifications other than Execute events.
hService	Service handle identifying the session that created the result, i.e. the service handle of the session that the event is sent to.
tsTimestamp	Time the event occurred (local time, in a Win32/Win64 SYSTEMTIME structure).
hResult	Result handle (note that for synchronous **WFSExecute** and **WFSGetInfo** commands, this value is identical to the synchronous function return value).
u.dwCommandCode	**WFSExecute** "command" code or **WFSGetInfo** "category" code; not used for other command completions.
u.dwEventID	ID of the event (for unsolicited events).
lpBuffer	Pointer to the results of the command (if any) or the contents of the event notification.

9.2 WFSVERSION

This structure is used to return version information from **WFSStartUp**, **WFSOpen** and **WFPOpen**.

```
typedef  struct _wfsversion {
   WORD      wVersion;
   WORD      wLowVersion;
   WORD      wHighVersion;
   char      szDescription[WFSDDESCRIPTION_LEN+1];
   char      szSystemStatus[WFSDSYSSTATUS_LEN+1];
} WFSVERSION, *LPWFSVERSION;
```

The members of this structure are (note that this structure is used to report version information for three distinct XFS interfaces: API, SPI, and the service-specific interface):

Element	Usage
wVersion	The version number to be used.
wLowVersion	The lowest version number that the called DLL can support.
wHighVersion	The highest version number that the called DLL can support.
szDescription	A null-terminated ASCII string into which the called DLL copies a description of the implementation. The text (up to 256 characters in length) may contain any characters: the most likely use that an application will make of this is to display it (possibly truncated) in a status message.
szSystemStatus	A null-terminated ASCII string into which the called DLL copies relevant status or configuration information. Not to be considered as an extension of the *szDescription* field. Used only if the information might be useful to the user or support staff.

10 Messages

This section defines the Windows messages used in the XFS subsystem.

10.1 Command Completions and Events

The following messages are sent to indicate:

- the completion of an asynchronous command, or
- the occurrence of an unsolicited event (execute, service, user, or system events).

All these messages have the same definition:

wParam:	not used
lParam:	points to a WFSRESULT data structure

```
WFS_<message_name>
wParam;   /*  not used  */
lParam = LPWFSRESULT lpWFSResult;
```

10.1.1 Command Completion Messages

WFS_OPEN_COMPLETE

WFS_CLOSE_COMPLETE

WFS_LOCK_COMPLETE

WFS_UNLOCK_COMPLETE

WFS_REGISTER_COMPLETE

WFS_DEREGISTER_COMPLETE

WFS_GETINFO_COMPLETE

WFS_EXECUTE_COMPLETE

10.1.2 Event Messages

WFS_EXECUTE_EVENT

WFS_SERVICE_EVENT

WFS_USER_EVENT

WFS_SYSTEM_EVENT

The *hService* parameter of the WFSRESULT structure, in the above event messages, contains the service handle of the session that the event is sent to.

10.2 WFS_TIMER_EVENT

The timer event message has the following format (see **WFMSetTimer**, **WFMKillTimer)**:

```
WFS_TIMER_EVENT
wParam  =  wTimerID;    /* timer ID returned by the WFMSetTimer function */
lParam  =  lpContext;   /* context pointer supplied by the Service Provider */
                        /* that requested the timer; may be NULL */
```

10.3 WFS_SYSE_DEVICE_STATUS

Status changes of logical services (which typically reflect changes in physical devices) are reported as system events. This is in addition to being reported by the WFS_INF_xxx_STATUS query of the **WFSGetInfo** or **WFSAsyncGetInfo** functions. The WFSRESULT data structure (defined in Section 9.1) is utilized as follows:

Field	Description
RequestID	(not used)
hService	Service handle identifying the session that created the result, i.e. the service handle of the session that the event is sent to.
tsTimestamp	Time the status change occurred (local time, in a Win32/Win64 SYSTEMTIME structure).
hResult	(not used)
u.dwEventID	= WFS_SYSE_DEVICE_STATUS
lpBuffer	Pointer to a WFSDEVSTATUS structure:

```
typedef struct _wfs_devstatus {
   LPSTR          lpszPhysicalName;
   LPSTR          lpszWorkstationName;
   DWORD          dwState;
} WFSDEVSTATUS, *LPWFSDEVSTATUS;
```

The members of this structure are:

Field	Description
lpszPhysicalName	Pointer to the physical service name of the service that changed its state.
lpszWorkstationName	Pointer to the name of the workstation in which the logical service name is defined.
dwState	Specifies the new state of the physical device managed by the service as one of the following:

Value	Meaning
WFS_STAT_DEVONLINE	The device is online (i.e. powered on and operable).
WFS_STAT_DEVOFFLINE	The device is offline (e.g. the operator has taken the device offline by turning a switch).
WFS_STAT_DEVPOWEROFF	The device is powered off or physically not connected.
WFS_STAT_DEVNODEVICE	There is no device intended to be there; e.g. this type of self service machine does not contain such a device or it is internally not configured.
WFS_STAT_DEVHWERROR	The device is inoperable due to a hardware error.
WFS_STAT_DEVUSERERROR	The device is inoperable because a person is preventing proper device operation.
WFS_STAT_DEVFRAUDATTEMPT	Some devices are capable of identifying a malicious physical attack which attempts to defraud valuable information or media. In this circumstance, this status code is returned to indicate the device is inoperable because a person attempted a fraudulent act on the device.
WFS_STAT_DEVPOTENTIALFRAUD	The device has detected a potential fraud attempt and is capable of remaining in service. In this case the application should make the decision as to whether to take the device offline.

10.4 WFS_SYSE_UNDELIVERABLE_MSG

If a command completion or event message cannot be delivered, it is reported as a system event. The WFSRESULT data structure (defined in Section 9.1) is utilized as follows:

Field	Description
RequestID	(not used)
hService	Service handle identifying the session that the event is sent to.
tsTimestamp	Time the event occurred (local time, in a Win32/Win64 SYSTEMTIME structure).
hResult	(not used)
u.dwEventID	= WFS_SYSE_UNDELIVERABLE_MSG
lpBuffer	Pointer to a WFSUNDEVMSG structure:

```
typedef struct _wfs_undevmsg {
    LPSTR        lpszLogicalName;
    LPSTR        lpszWorkstationName;
    LPSTR        lpszAppID;
    DWORD        dwSize;
    LPBYTE       lpbDescription;
    DWORD        dwMsg;
    LPWFSRESULT  lpWFSResult;
} WFSUNDEVMSG, *LPWFSUNDEVMSG;
```

The members of this structure are:

Field	Description
lpszLogicalName	Pointer to the logical service name of the service that generated the original undeliverable message.
lpszWorkstationName	Pointer to the name of the workstation in which the logical service name is defined.
lpszAppID	Pointer to the application ID associated with the session that generated the original message.
dwSize	The size in bytes of the following description.
lpbDescription	Pointer to a vendor-specific description of the reason why the message could not be delivered.
dwMsg	The message identifier of the original message.
lpWFSResult	Pointer to the WFSRESULT structure of the original message (which has the *lpBuffer* parameter set to NULL). This structure includes the *hService* of the session where the message could not be delivered.

10.5 WFS_SYSE_APP_DISCONNECT

If the XFS subsystem loses connection to an application, it closes the session (see Section 4.6) and generates this system event. The WFSRESULT data structure (defined in Section 9.1) is utilized as follows:

Field	Description
RequestID	(not used)
hService	Service handle identifying the session that the event is sent to.
tsTimestamp	Time the event occurred (local time, in a Win32/Win64 SYSTEMTIME structure).
hResult	(not used)
u.dwEventID	= WFS_SYSE_APP_DISCONNECT
lpBuffer	Pointer to a WFSAPPDISC structure:

```
typedef struct _wfs_appdisc {
   LPSTR        lpszLogicalName;
   LPSTR        lpszWorkstationName;
   LPSTR        lpszAppID;
} WFSAPPDISC, *LPWFSAPPDISC;
```

The members of this structure are:

Field	Description
lpszLogicalName	Pointer to the logical service name of the service that the application was connected to.
lpszWorkstationName	Pointer to the name of the workstation in which the logical service name is defined.
lpszAppID	Pointer to the application ID associated with the session that generated the event.

10.6 WFS_SYSE_HARDWARE_ERROR, WFS_SYSE_SOFTWARE_ERROR, WFS_SYSE_USER_ERROR and WFS_SYSE_FRAUD_ATTEMPT

Hardware and software errors are reported as system events. In most cases, this is in addition to being reported via the WFS_ERR_HARDWARE_ERROR (or device class specific error code), the WFS_ERR_SOFTWARE_ERROR or WFS_ERR_USER_ERROR error code that is returned as the command completion.

In order to supply the maximum information, these events should be sent as soon as an error is detected. In particular, if an error is detected during the processing of an execute command, then the event should be sent before the command completion event.

The WFSRESULT data structure (defined in Section 9.1), is utilized as follows:

Field	Description
RequestID	Request ID of the request being processed when the error occurred, zero if no request was being processed when the error occurred.
hService	Service handle identifying the session that the event is sent to.
tsTimestamp	Time the error occurred (local time, in a Win32/Win64 SYSTEMTIME structure).
hResult	Result handle of the request being processed when the error occurred, zero if no request was being processed.
u.dwEventID	The ID of the error.

Value	Meaning
WFS_SYSE_HARDWARE_ERROR	The error is a hardware error
WFS_SYSE_SOFTWARE_ERROR	The error is a software error
WFS_SYSE_USER_ERROR	The error is a user error
WFS_SYSE_FRAUD_ATTEMPT	Some devices are capable of identifying a malicious physical attack which attempts to defraud valuable information or media. In this circumstance, this error event is returned to indicate a fraud attempt has occurred.

lpBuffer Pointer to a WFSHWERROR structure:

```
typedef struct _wfs_hwerror {
    LPSTR        lpszLogicalName;
    LPSTR        lpszPhysicalName;
    LPSTR        lpszWorkstationName;
    LPSTR        lpszAppID;
    DWORD        dwAction;
    DWORD        dwSize;
    LPBYTE       lpbDescription;
} WFSHWERROR, *LPWFSHWERROR;
```

The members of this structure are:

Field	Description
lpszLogicalName	Pointer to the logical service name of the service that generated the error
lpszPhysicalName	Pointer to the physical service name of the service that generated the error
lpszWorkstationName	Pointer to the name of the workstation in which the logical service name is defined (if any)
lpszAppID	Pointer to the application ID associated with the session that generated the error (if any)
dwAction	The action required to manage the error. Possible values are:

Value	Meaning
WFS_ERR_ACT_NOACTION	No action required or error was autorecovered.
WFS_ERR_ACT_RESET	Reset device to attempt recovery using WFS_CMD_XXX_RESET, but should not be used excessively. Intervention is not required although if repeated attempts are unsuccessful then WFS_ERR_ACT_HWMAINT may be reported.
WFS_ERR_ACT_SWERROR	A software error occurred. Contact software vendor.

	WFS_ERR_ACT_CONFIG	A configuration error occurred. Check configuration.
	WFS_ERR_ACT_HWCLEAR	Recovery is not possible. A manual intervention for clearing the device is required. This value is only used for hardware errors. This value is typically returned when a hardware error has occurred which requires banking personnel specific maintenance, e.g. 'replace paper', or 'remove cards from retain bin'.
	WFS_ERR_ACT_HWMAINT	Recovery is not possible. A technical maintenance intervention is required. This value is only used for hardware errors and fraud attempts. This value is typically returned when a hardware error or fraud attempt has occurred which requires field engineer specific maintenance activity. WFS_CMD_XXX_RESET may be used to attempt recovery after intervention, but should not be used excessively – Vendor Dependent Mode may be required to recover the device.
	WFS_ERR_ACT_SUSPEND	Device will attempt auto recovery and will advise any further action required via a Device Status Event.
dwSize	The size in bytes of the following description	
lpbDescription	Pointer to a vendor-specific description of the error.	

Note:
The following table describes what *dwAction* may be returned for the various Hardware, Software, User Error and Fraud Attempt Events. The *dwAction* definitions above give guidance on what an application should do next when one of these events is received. Care should be taken to avoid calling WFS_CMD_XXX_RESET excessively without intervention, as this may lead to damage to the device or media contained in the device if for example media is jammed in the device:

	Generated on Hardware Event?	Generated on Software Event?	Generated on User Event?	Generated on Fraud Event?
_NOACTION	Yes	Yes	Yes	Yes
_RESET	Yes	Yes	Yes	No
_SWERROR	No	Yes	No	No
_CONFIG	Yes	Yes	No	No
_HWCLEAR	Yes	No	No	No
_HWMAINT	Yes	No	No	Yes
_SUSPEND	No	No	Yes	No

10.7 WFS_SYSE_LOCK_REQUESTED

The Lock requested system event is sent to any application which currently has a device locked whenever a request for a lock on the same device is received from another application or service handle. Note that this event is generated each time another application requests a lock on the same device. This system event differs from other system events in that it is only posted to the owner of the lock; it is NOT posted to any other application.

Field	Description
RequestID	(not used)
hService	Service handle identifying the device and session which has obtained the lock.
tsTimestamp	Time the status change occurred (local time, in a Win32/Win64 SYSTEMTIME structure).
hResult	(not used)
u.dwEventID	= WFS_SYSE_LOCK_REQUESTED
lpBuffer	(not used)

10.8 WFS_SYSE_VERSION_ERROR

Failures in version negotiation are reported as system events. This is in addition to being reported by the version error code returned by the **WFSStartUp** or **WFSOpen** functions. The WFSRESULT data structure (defined in Section 9.1) is utilized as follows:

Field	Description
RequestID	(not used)
hService	(not used)
tsTimestamp	Time the error occurred (local time, in a Win32/Win64 SYSTEMTIME structure).
hResult	The version error code (e.g. WFS_ERR_SPI_VER_TOO_HIGH).
u.dwEventID	= WFS_SYSE_VERSION_ERROR
lpBuffer	Pointer to a WFSVRSNERROR structure:

```
typedef struct _wfs_vrsnerror {
    LPSTR         lpszLogicalName;
    LPSTR         lpszWorkstationName;
    LPSTR         lpszAppID;
    DWORD         dwSize;
    LPBYTE        lpbDescription;
    LPWFSVERSION  lpWFSVersion;
} WFSVRSNERROR, *LPWFSVRSNERROR
```

The members of this structure are:

Field	Description
lpszLogicalName	Pointer to the logical service name of the service being opened (NULL if **WFSStartUp**).
lpszWorkstationName	Pointer to the name of the workstation in which the application made the **WFSStartUp** or **WFSOpen** request.
lpszAppID	Pointer to the application ID from the open request that failed (NULL if **WFSStartUp**).
dwSize	The size in bytes of the following description.
lpbDescription	Pointer to a vendor-specific description of the version negotiation failure.
lpWFSVersion	Pointer to the WFSVERSION structure reporting the results of the version negotiation.

11 Error Codes

The following are the error codes that can be returned from a call to an XFS API or SPI function, either as a function return or in a result structure pointed to by a completion message. Errors from service-specific commands are defined in the specifications for each service class.

WFS_ERR_ALREADY_STARTED
A **WFSStartUp** has already been issued by the application, without an intervening **WFSCleanUp**.

WFS_ERR_API_VER_TOO_HIGH
The range of versions of XFS API support requested by the application is higher than any supported by this particular XFS Manager implementation.

WFS_ERR_API_VER_TOO_LOW
The range of versions of XFS API support requested by the application is lower than any supported by this particular XFS Manager implementation.

WFS_ERR_CANCELED
The request was canceled by **WFSCancelAsyncRequest** or **WFSCancelBlockingCall**.

WFS_ERR_CFG_INVALID_HKEY
The specified *hKey* parameter does not correspond to a currently open key.

WFS_ERR_CFG_INVALID_NAME
The value specified by the *lpszValueName* parameter does not exist in the specified key.

WFS_ERR_CFG_INVALID_SUBKEY
The key specified by *lpszSubKey* does not exist.

WFS_ERR_CFG_INVALID_VALUE
The specified value does not exist within the specified open key.

WFS_ERR_CFG_KEY_NOT_EMPTY
The specified key has subkeys and cannot be deleted. The subkeys must be deleted first.

WFS_ERR_CFG_NAME_TOO_LONG
The length of the name to be returned exceeds the length of the buffer.

WFS_ERR_CFG_NO_MORE_ITEMS
There are no more subkeys to be returned (the *iSubKey* parameter is greater than the index of the last subkey).

WFS_ERR_CFG_VALUE_TOO_LONG
The length of the value to be returned exceeds the length of the buffer.

WFS_ERR_CONNECTION_LOST
The connection to the service is lost.

WFS_ERR_DEV_NOT_READY
The function required device access, and the device was not ready or timed out.

WFS_ERR_HARDWARE_ERROR
The function required device access, and an error occurred on the device.

WFS_ERR_INTERNAL_ERROR
An internal inconsistency or other unexpected error occurred in the XFS subsystem.

WFS_ERR_INVALID_ADDRESS
The *lpvOriginal* parameter does not point to a previously allocated buffer.

WFS_ERR_INVALID_APP_HANDLE
The specified application handle is not valid, i.e. was not created by a preceding create call.

WFS_ERR_INVALID_BUFFER
The *lpvData* parameter is not a pointer to an allocated buffer structure.

WFS_ERR_INVALID_CATEGORY
The *dwCategory* issued is not supported by this service class.

WFS_ERR_INVALID_COMMAND
The *dwCommand* issued is not supported by this service class.

WFS_ERR_INVALID_EVENT_CLASS
The *dwEventClass* parameter specifies one or more event classes not supported by the service.

WFS_ERR_INVALID_HSERVICE
The *hService* parameter is not a valid service handle.

WFS_ERR_INVALID_HPROVIDER
The *hProvider* parameter is not a valid provider handle.

WFS_ERR_INVALID_HWND
The *hWnd* parameter is not a valid window handle.

WFS_ERR_INVALID_HWNDREG
The *hWndReg* parameter is not a valid window handle.

WFS_ERR_INVALID_POINTER
A pointer parameter does not point to accessible memory.

WFS_ERR_INVALID_DATA
The data structure passed as input parameter contains invalid data.

WFS_ERR_INVALID_REQ_ID
The *RequestID* parameter does not correspond to an outstanding request on the service.

WFS_ERR_INVALID_RESULT
The *lpResult* parameter is not a pointer to an allocated WFSRESULT structure.

WFS_ERR_INVALID_SERVPROV
The file containing the Service Provider is invalid or corrupted.

WFS_ERR_INVALID_TIMER
The *hWnd* and *wTimerID* parameters do not correspond to a currently active timer.

WFS_ERR_INVALID_TRACELEVEL
The *dwTraceLevel* parameter does not correspond to a valid trace level or set of levels.

WFS_ERR_LOCKED
The service is locked under a different *hService*.

WFS_ERR_NO_BLOCKING_CALL
There is no outstanding blocking call for the specified thread.

WFS_ERR_NO_SERVPROV
The file containing the Service Provider does not exist.

WFS_ERR_NO_SUCH_THREAD
The specified thread does not exist.

WFS_ERR_NO_TIMER
The timer could not be created.

WFS_ERR_NOT_LOCKED
The application requesting a service be unlocked had not previously performed a successful **WFSLock** or **WFSAsyncLock**.

WFS_ERR_NOT_OK_TO_UNLOAD
The XFS Manager may not unload the Service Provider DLL.

WFS_ERR_NOT_STARTED
The application has not previously performed a successful **WFSStartUp**.

WFS_ERR_NOT_REGISTERED
The specified *hWndReg* window was not registered to receive messages for any event classes.

WFS_ERR_OP_IN_PROGRESS
A blocking operation is in progress on the thread; only **WFSCancelBlockingCall** and **WFSIsBlocking** are permitted at this time.

WFS_ERR_OUT_OF_MEMORY
There is not enough memory available to satisfy the request.

WFS_ERR_SERVICE_NOT_FOUND
The logical name is not a valid Service Provider name.

WFS_ERR_SOFTWARE_ERROR
The function required access to configuration information, and an error occurred on the software.

WFS_ERR_SPI_VER_TOO_HIGH
The range of versions of XFS SPI support requested by the XFS Manager is higher than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SPI_VER_TOO_LOW
The range of versions of XFS SPI support requested by the XFS Manager is lower than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SRVC_VER_TOO_HIGH
The range of versions of the service-specific interface support requested by the application is higher than any supported by the Service Provider for the logical service being opened.

WFS_ERR_SRVC_VER_TOO_LOW
The range of versions of the service-specific interface support requested by the application is lower than any supported by the Service Provider for the logical service being opened.

WFS_ERR_TIMEOUT
The timeout interval expired.

WFS_ERR_UNSUPP_CATEGORY
The *dwCategory* issued, although valid for this service class, is not supported by this Service Provider.

WFS_ERR_UNSUPP_COMMAND
The *dwCommand* issued, although valid for this service class, is not supported by this Service Provider or device.

WFS_ERR_UNSUPP_DATA
The data structure passed as an input parameter although valid for this service class is not supported by this Service Provider or device.

WFS_ERR_USER_ERROR
A user is preventing proper operation of the device.

WFS_ERR_VERSION_ERROR_IN_SRVC
Within the service, a version mismatch of two modules occurred.

WFS_ERR_FRAUD_ATTEMPT
Some devices are capable of identifying a malicious physical attack which attempts to defraud valuable information or media. In these cases, this error code is returned to indicate the user is attempting a fraudulent act on the device.

WFS_ERR_SEQUENCE_ERROR
The requested operation is not valid at this time or in the devices current state.

WFS_ERR_AUTH_REQUIRED
The requested operation cannot be performed because it requires authentication.

12 Common GetInfo, Execute Commands and Messages

12.1 Common GetInfo Commands

12.1.1 WFS_INF_API_TRANSACTION_STATE

Description This command can be used to get the transaction state.

Input Param None.

Output Param LPWFSAPITRANSACTIONSTATE lpTransactionState;

```
typedef struct _wfs_api_transaction_state
    {
    WORD                        fwState;
    LPWFSAPITRANSACTIONINFO     lpTransactionInfo;
    } WFSAPITRANSACTIONSTATE, * LPWFSAPITRANSACTIONSTATE;
```

fwState
Specifies the transaction state. The value is set to one of the following values:

Value	Meaning
WFS_API_TRANS_ACTIVE	A customer transaction is in progress.
WFS_API_TRANS_INACTIVE	No customer transaction is in progress.

lpTransactionInfo
Specifies the transaction information. If *fwState* is WFS_API_TRANS_INACTIVE, this value will be NULL.

```
typedef struct _wfs_api_transaction_info
    {
    LPWSTR                  lpszTransactionID;
    LPSTR                   lpszExtra;
    } WFSAPITRANSACTIONINFO, * LPWFSAPITRANSACTIONINFO;
```

lpszTransactionID
Specifies a UNICODE string which identifies the transaction ID. The value returned in this parameter is an application defined customer transaction identifier, which was previously set in the WFS_CMD_API_SET_TRANSACTION_STATE command.

lpszExtra
Pointer to a list of vendor-specific, or any other extended, transaction information. The information is set as a series of *"key=value"* strings. Each string is null-terminated, with the final string terminating with two null characters. An empty list may be indicated by either a NULL pointer or a pointer to two consecutive null characters.

Error Codes Only the generic error codes defined in Section 11 can be generated by this command.

Events None.

Comments None.

12.1.2 WFS_INF_API_SERVICE_INFO

Description This command is used to retrieve service information which is common to service providers of all XFS device classes, e.g. firmware versions, which commands and events are supported and which other devices the device is compounded with (if part of a compound device). A reference to this command is included in the WFSGetInfo section of every device class interface document.

Input Param None.

Output Param LPWFSSERVICEINFO lpServiceInfo;

```
typedef struct _wfs_service_info
      {
      LPWFSDEVICEINFO             *lppDeviceInformation;
      LPWFSVENDORMODEINFO         lpVendorModeInformation;
      LPWFSSERVICEINTERFACE       lpServiceInterface;
      LPWFSCOMPOUNDDEVICE         *lppCompoundDevices;
      LPWSTR                      lpszServiceProviderVersion;
      LPSTR                       lpszExtra;
      } WFSSERVICEINFO, *LPWFSSERVICEINFO;
```

lppDeviceInformation
Specifies a NULL-terminated array of pointers to WFSDEVICEINFO structures. If the Service Provider is comprised of more than one device then there will be a WFSDEVICEINFO structure for each device. If no information is available then this will be NULL.

```
typedef struct _wfs_device_info
      {
      LPWSTR          lpszModelName;
      LPWSTR          lpszSerialNumber;
      LPWSTR          lpszRevisionNumber;
      LPWSTR          lpszModelDescription;
      LPWFSFIRMWARE   *lppFirmware;
      LPWFSSOFTWARE   *lppSoftware;
      } WFSDEVICEINFO, *LPWFSDEVICEINFO;
```

lpszModelName
Specifies a UNICODE string which identifies the device model name. This string value is terminated with a null character. *lpszModelName* is NULL when the device model name is unknown.

lpszSerialNumber
Specifies a UNICODE string which identifies the unique serial number of the device. This string value is terminated with a null character. *lpszSerialNumber* is NULL when the serial number is unknown.

lpszRevisionNumber
Specifies a UNICODE string which identifies the unique revision number of the device. This string value is terminated with a null character. *lpszRevisionNumber* is NULL when the revision number is unknown.

lpszModelDescription
Specifies a UNICODE string which contains a description of the device. This string value is terminated with a null character. *lpszModelDescription* is NULL when the model description is unknown.

lppFirmware
A NULL-terminated array of pointers to WFSFIRMWARE structures specifying the names and version numbers of the firmware that is present. Single or multiple firmware versions can be reported. If the firmware versions are not reported, then *lppFirmware* is NULL

```
typedef struct _wfs_firmware
   {
   LPWSTR          lpszFirmwareName;
   LPWSTR          lpszFirmwareVersion;
   LPWSTR          lpszHardwareRevision;
   } WFSFIRMWARE, *LPWFSFIRMWARE;
```

lpszFirmwareName
Specifies a UNICODE string which identifies the firmware name. *lpszFirmwareName* is NULL when the firmware name is unknown.

lpszFirmwareVersion
Specifies a UNICODE string which identifies the firmware version. *lpszFirmwareVersion* is NULL when the firmware version is unknown.

lpszHardwareRevision
Specifies a UNICODE string which identifies the hardware revision. *lpszHardwareRevision* is NULL when the hardware revision is unknown.

lppSoftware
A NULL-terminated array of pointers to WFSSOFTWARE structures specifying the names and version numbers of the software components that are present. Single or multiple software versions can be reported. If the software versions are not reported, then *lppSoftware* is NULL

```
typedef struct _wfs_software
    {
    LPWSTR          lpszSoftwareName;
    LPWSTR          lpszSoftwareVersion;
    } WFSSOFTWARE, *LPWFSSOFTWARE;
```

lpszSoftwareName
Specifies a UNICODE string which identifies the software component. *lpszSoftwareName* is NULL when the software component name is unknown.

lpszSoftwareVersion
Specifies a UNICODE string which identifies the software component version. *lpszSoftwareVersion* is NULL when the software component version is unknown.

lpVendorModeInformation
Specifies additional information about the Service Provider while in Vendor Dependent Mode. If NULL, all sessions must be closed before entry to VDM.

```
typedef struct _wfs_vendor_mode_info
      {
      BOOL            bAllowOpenSessions;
      LPDWORD         lpdwAllowedExecuteCommands;
      } WFSVENDORMODEINFO, *LPWFSVENDORMODEINFO;
```

bAllowOpenSessions
If TRUE, sessions with this Service Provider may remain open during Vendor Dependent Mode for the purposes of monitoring events, sending Info commands, or sending Execute commands listed in *lpdwAllowedExecuteCommands*. If FALSE, all sessions must be closed before entering Vendor Dependent Mode.

lpdwAllowedExecuteCommands
A zero terminated list of Execute command IDs representing commands which can be accepted while in Vendor Dependent Mode. Any Execute command which is not included in this list will be rejected with WFS_ERR_SEQUENCE_ERROR as control of the device has been handed to the Vendor Dependent Application. If NULL, no Execute commands can be accepted.

lpServiceInterface
Specifies the WFSExecute commands, WFSGetInfo commands and service specific events which are supported by this service.

```
typedef struct _wfs_service_interface
      {
      LPDWORD         lpdwExecuteCommands;
      LPDWORD         lpdwGetInfoCategories;
      LPDWORD         lpdwEvents;
      DWORD           dwMaximumRequests;
      LPDWORD         lpdwAuthenticationRequired;
      } WFSSERVICEINTERFACE, *LPWFSSERVICEINTERFACE;
```

lpdwExecuteCommands
A zero terminated list of command IDs representing the WFSExecute commands which are supported by this service, e.g. WFS_CMD_CDM_DISPENSE.

lpdwGetInfoCategories
A zero terminated list of command IDs representing the WFSGetInfo categories which are supported by this service, e.g. WFS_INF_CDM_STATUS.

lpdwEvents
A zero terminated list of event IDs representing the service specific events which are supported by this service, e.g. WFS_SRVE_CDM_ITEMSTAKEN.

dwMaximumRequests
Specifies the maximum number of requests which can be queued by the Service Provider. This will be zero if not reported. This will also be zero if the maximum number of requests is unlimited.

lpdwAuthenticationRequired
A zero terminated list of command IDs representing the commands and categories which need to be authenticated using the compounding mechanism that is described in Appendix E – XFS Authentication.

lppCompoundDevices
If this is a compound device then this is a NULL-terminated array of pointers to WFSCOMPOUNDDEVICE structures that report the details of all other devices that this device is compounded with. If this is not a compound device then *lppCompoundDevices* is NULL.

```
typedef struct _wfs_compound_device
      {
      LPWSTR                lpszLogicalServiceName;
      LPWSTR                lpszProviderName;
      LPWSTR                lpszPhysicalServiceName;
      LPWSTR                lpszDeviceClass;
      } WFSCOMPOUNDDEVICE, *LPWFSCOMPOUNDDEVICE;
```

lpszLogicalServiceName
Specifies a UNICODE string which identifies the logical service name of the device in the Windows registry (see section 4.7). This string value is terminated with a null character.

lpszProviderName
Specifies a UNICODE string which identifies the provider name of the device in the Windows registry (see section 4.7). This string value is terminated with a null character.

lpszPhysicalServiceName
Specifies a UNICODE string which identifies the physical service name of the device in the Windows registry (see section 4.7). This string value is terminated with a null character.

lpszDeviceClass
Specifies a UNICODE string which identifies the service class of this device, e.g. “CDM” for the Cash Dispenser, This string value is terminated with a null character.

lpszServiceProviderVersion
Specifies a UNICODE string which identifies the Service Provider version. This string value is terminated with a null character. *lpszServiceProviderVersion* is NULL when the device model name is unknown.

lpszExtra
Pointer to a list of vendor-specific, or any other extended, information. The information is returned as a series of *“key=value”* strings so that it is easily extensible by Service Providers. Each string is null-terminated, with the final string terminating with two null characters. An empty list may be indicated by either a NULL pointer or a pointer to two consecutive null characters.

Error Codes Only the generic error codes defined in Section 11 can be generated by this command.

Comments None.

12.2 Common Execute Commands

12.2.1 WFS_CMD_API_SET_TRANSACTION_STATE

Description	This command allows the application to specify the transaction state, which the Service Provider can then utilize in order to optimize performance. After receiving this command, this Service Provider can perform the necessary processing to start or end the customer transaction. This command should be called for every Service Provider that could be used in a customer transaction. The transaction state applies to every session.
Input Param	LPWFSAPITRANSACTIONSTATE lpTransactionState; The LPWFSAPITRANSACTIONSTATE structure is specified in the documentation for the WFS_INF_API_TRANSACTION_STATE command.
Output Param	None.
Error Codes	In addition to the generic error codes defined in Section 11, any service-specific errors that can be returned are defined in the specifications for each service class.
Events	In addition to the generic events defined in Section 11, any service-specific events that can be generated are defined in the specifications for each service class.
Comments	None.

12.3 Common Messages

12.3.1 WFS_SRVE_API_STATUS_CHANGED

Description This service event specifies that a status has changed. It is non-mandatory. The *lpStatus* parameter points to the changed status structure. This event can be used to report non-critical status changes in the Service Provider which are not reported by the WFS_SYSE_DEVICE_STATUS event.

Event Param LPWFSSTATUSCHANGED lpStatus;

```
typedef struct _wfs_api_status_changed
      {
      LPVOID lpvOldStatus;
      LPVOID lpvNewStatus;
      } WFSSTATUSCHANGED, *LPWFSSTATUSCHANGED;
```

lpvOldStatus
Pointer to a structure containing the previous status structure. For a description of the status structure see the definition in the relevant device class specification.

lpvNewStatus
Pointer to a structure containing the status structure that was updated. For a description of the status structure see the definition in the relevant device class specification.

Comments The *lpvOldStatus* and *lpvNewStatus* parameters are specific to the Service Provider that sends this event. For example, if the device class is a PIN Service Provider then both of these parameters will point to a WFSPINSTATUS structure.

12.3.2 WFS_EXEE_API_ERROR_INFO

Description This optional execute event may be sent prior to completion of a command that completes with an error. It provides additional information detailing what specifically is causing the error. For example, the additional information can identify what specifically is invalid with any input parameters to the Execute command. The information is returned in a string with one or more "key=value pairs", where key = an input parameter and value = the reason why the parameter is invalid.

Event Param LPSTR lpszExtra;

lpszExtra
Pointer to a list of vendor-specific, or any other extended, information. The information is returned as a series of "key=value" strings so that it is easily extensible by Service Providers. Each string is null-terminated, with the final string terminating with two null characters.

It is recommended that the format of the key is the programmatic definition of the input value, for example "lpCUInfo->lppList[0].usNumber" and the value should indicate the reason the input value is invalid, for example "Must be less than 5".

For example, if the CDM Service Provider doesn't support an input parameter passed in by the command WFS_CMD_CDM_END_EXCHANGE, the key and value pair provides the error information:

lpszExtra = "lpCUInfo->lppList[0].usNumber=7 is invalid, expecting a value less than 5\0lpCUInfo->lppList[0].ulValues=20 is invalid, check currency\0\0"

Comments None.

~~12~~13 Appendix A - Planned Enhancements and Extensions

This section describes functions and facilities that are not fully defined in this version of the Extensions for Financial Services specification; modifications and complete definitions will be supplied in later versions. Vendor and user input is encouraged on these functions and facilities, as well as suggestions as to additional functionality.

XFS currently includes specifications for access to the key classes of financial peripherals for attended and self-service environments. These existing specifications will be extended and enhanced based on vendor and user experience with them. The Service Class Definition Document lists the classes of devices or services that, together with others that customers and vendors request, will be evaluated for inclusion in future versions of this specification.

Also to be considered for future versions of XFS are other types of services, such as financial transaction messaging and management, as well as related services for financial networks such as network and systems management and security. As with the current specification, all these capabilities will be specified for access from the familiar, consistent Microsoft Windows user interface and programming environments. Another portion of the XFS API set will deal with administration issues.

~~12.1~~13.1 Event and System Management

The XFS subsystem will need additional facilities for managing exception conditions (i.e. those that are not anticipated in the error codes, events, etc., that are defined in this specification). One general facility for this is the system event capability, as described in Sections 4.11 and 10. This will utilize a combination of one or more functions provided by the XFS Manager and other methods for applications, the XFS Manager, Service Providers, and services to report exception conditions in special circumstances (e.g. when the XFS Manager is not available). Such conditions would presumably be monitored by a system management agent responsible for logging and reporting them via a network management facility.

~~13~~14 Appendix B - XFS Workshop Contacts

Please submit comments and questions to
xfs-helpdesk@cenorm.be

Or contact
Luc Van den Berghe
CEN~~/ISSS~~ Workshop Manager
Rue de Stassart 36
B-1050 Brussels
Luc.vandenberghe@cenorm.be
Tel: + 32 2 55 00 957

~~14~~15 Appendix C - ATM Devices Synchronization Flow

The following section describes the flow of a synchronization use case using WFS_CMD_XXX_SYNCHRONIZE_COMMAND. This application flow is provided as a guideline only.

~~14.1~~15.1 Synchronized Media Ejection

The following table describes the flow of a transaction where the receipt ejection is synchronized with the card ejection during the transaction. Both Service Providers and the devices support the synchronization in this example.

Step	Application/XFS Commands	Service Providers / Devices
The next step is to eject the receipt and the card at the end of the transaction. The application would like to synchronize the receipt ejection with the card ejection.		
1.	Informs the PTR class Service Provider that the subsequent command is the "eject receipt" and that this command needs to be executed without delay for synchronization purposes. WFS_CMD_PTR_SYNCHRONIZE_COMMAND (*dwCommand*: WFS_CMD_PTR_CONTROL_MEDIA) (specifying WFS_PTR_CTRLEJECT as its parameter)	PTR class Service Provider sends a synchronization command to the receipt printer device for the receipt ejection.
2.	Informs the IDC class Service Provider that the subsequent command is the "eject card" and that this command needs to be executed without delay for synchronization purposes. WFS_CMD_IDC_SYNCHRONIZE_COMMAND (*dwCommand*: WFS_CMD_IDC_EJECT_CARD) (specifying WFS_IDC_EXITPOSITION as its parameter)	IDC class Service Provider sends a synchronization command to the card reader device for the card ejection.
3.	WFS_CMD_PTR_SYNCHRONIZE_COMMAND completion event. WFS_CMD_IDC_SYNCHRONIZE_COMMAND completion event.	
4.	The application executes the following commands at the same time. - Initiates via a WFSAsyncExecute WFS_CMD_PTR_CONTROL_MEDIA. - Initiates via a WFSAsyncExecute WFS_CMD_IDC_EJECT_CARD. (The parameters are the same as specified in the WFS_CMD_XXX_SYNCHRONIZATION_COMMANDs)	The following actions are performed at the same time. - The receipt is ejected. - The card is ejected.
5.	WFS_CMD_PTR_CONTROL_MEDIA completion event. WFS_CMD_IDC_EJECT_CARD completion event.	

~~15~~16 Appendix D – Win64 Migration Considerations

Users must ensure that when porting their XFS applications to the Win64 environment that care is taken to update their existing code correctly in order to avoid issues. Microsoft state that porting 32-bit applications to 64-bit will be easier than it was porting 16-bit applications to 32-bit Windows, but care must still be taken in certain areas.

On 64 bit operating systems it is possible to run either a complete 32 bit XFS software stack, or a complete 64 bit software stack. Where a native 64 bit application is being run a 64 bit XFS Manager must be used. A sample XFS Manager is supplied with the XFS SDK, however this is a 32 bit XFS Manager only.

By far the biggest change when migrating C code is the change in pointer size from 32 to 64 bits. As the XFS architecture makes extensive use of pointers this change may have a significant impact on native XFS applications that currently run on Win32 environments. The following are some considerations for developers with regard to the XFS architecture:

1. As it is impossible for a 64-bit process to load a 32-bit DLL directly it is recommended that the entire software stack from the application to the Service Providers should be native 64-bit where possible. While this is the most ideal solution it is not mandatory but a recommendation, as feasibly some dependencies could run out of process to the application and/or the Service Providers. The XFS Manager that is used will always need to be 64-bit for a 64-bit application.

2. All declarations, use and storage of pointers should be checked. In C code memory addresses are often stored as a ULONG value, because on 32-bit Windows an address, a pointer, and a ULONG are all 32 bits. On 64-bit Windows a ULONG is also 32 bits long, but all pointers are 64-bit values. Functions such as the C sizeof() method will also need to be checked as the value that they return for the size of a pointer will be 8 bytes rather than 4 bytes. C style casts will also need to be scrutinized as potentially they may cast a pointer to a 32 bit value.

3. In order to prepare for the possibility of future porting to 64-bit code, developers should consider using the latest Windows header files that contain the portable pointer precision type ULONG_PTR. This data type can be used in current 32-bit code to store pointer values instead of a ULONG. The ULONG_PTR data type is a portable value that is 32 bits when compiled with a 32-bit compiler and 64 bits when compiled with a 64-bit compiler, thus ensuring good compatibility when compiled in either environment.

4. If running a 32 bit application on a 64 bit operating system then the operating system may manage the precise location of the XFS registry locations in 32 bit compatible areas of the registry.

~~16~~17 Appendix D - C-Header files

~~16.1~~17.1 XFSAPI.H

```
/******************************************************************************
*                                                                             *
* xfsapi.h      XFS - API functions, types, and definitions                   *
*                                                                             *
*               Version 3.~~30  (March 19 2015)~~ 40  (December 6 2019)
*
*                                                                             *
******************************************************************************/

#ifndef __inc_xfsapi__h
#define __inc_xfsapi__h

#ifdef __cplusplus
extern "C" {
#endif

#include <windows.h>

/*   be aware of alignment   */
#pragma pack(push,1)

/****** Common ***************************************************************/

typedef unsigned short USHORT;
typedef char CHAR;
typedef short SHORT;
typedef unsigned long ULONG;
typedef unsigned char UCHAR;
typedef SHORT * LPSHORT;
typedef LPVOID * LPLPVOID;
typedef ULONG * LPULONG;
typedef USHORT * LPUSHORT;

typedef HANDLE HPROVIDER;

typedef ULONG REQUESTID;
typedef REQUESTID * LPREQUESTID;

typedef HANDLE HAPP;
typedef HAPP * LPHAPP;

typedef USHORT HSERVICE;
typedef HSERVICE * LPHSERVICE;

typedef LONG HRESULT;
typedef HRESULT * LPHRESULT;

typedef BOOL (WINAPI * XFSBLOCKINGHOOK)(VOID);
typedef XFSBLOCKINGHOOK * LPXFSBLOCKINGHOOK;

/****** Common Commands ******************************************************/

#define API_SERVICE_OFFSET                      (0)

/* API Info Commands */
#define WFS_INF_API_TRANSACTION_STATE           (API_SERVICE_OFFSET + 1)
#define WFS_INF_API_SERVICE_INFO                (API_SERVICE_OFFSET + 2)

/* API Execute Commands */
#define WFS_CMD_API_SET_TRANSACTION_STATE       (API_SERVICE_OFFSET + 1)

/* API Messages */
#define WFS_SRVE_API_STATUS_CHANGED             (API_SERVICE_OFFSET + 1)
#define WFS_EXEE_API_ERROR_INFO                 (API_SERVICE_OFFSET + 2)
```

```
/****** String lengths ***********************************************/

#define WFSDDESCRIPTION_LEN                     256
#define WFSDSYSSTATUS_LEN                       256

/****** Values of WFSDEVSTATUS.fwState ********************************/

#define WFS_STAT_DEVONLINE                      (0)
#define WFS_STAT_DEVOFFLINE                     (1)
#define WFS_STAT_DEVPOWEROFF                    (2)
#define WFS_STAT_DEVNODEVICE                    (3)
#define WFS_STAT_DEVHWERROR                     (4)
#define WFS_STAT_DEVUSERERROR                   (5)
#define WFS_STAT_DEVBUSY                        (6)
#define WFS_STAT_DEVFRAUDATTEMPT                (7)
#define WFS_STAT_DEVPOTENTIALFRAUD              (8)

/****** Value of WFS_DEFAULT_HAPP *************************************/

#define WFS_DEFAULT_HAPP                        (0)

/****** Values of WFSAPITRANSACTIONSTATE.fwState ***********************/

#define WFS_API_TRANS_ACTIVE                    (0)
#define WFS_API_TRANS_INACTIVE                  (1)

/****** Data Structures ***********************************************/

typedef struct _wfs_result
{
    REQUESTID       RequestID;
    HSERVICE        hService;
    SYSTEMTIME      tsTimestamp;
    HRESULT         hResult;
    union {
        DWORD       dwCommandCode;
        DWORD       dwEventID;
    } u;
    LPVOID          lpBuffer;
} WFSRESULT, *LPWFSRESULT;

typedef struct _wfsversion
{
    WORD            wVersion;
    WORD            wLowVersion;
    WORD            wHighVersion;
    CHAR            szDescription[WFSDDESCRIPTION_LEN+1];
    CHAR            szSystemStatus[WFSDSYSSTATUS_LEN+1];
} WFSVERSION, *LPWFSVERSION;

/*=====================================================*/
/* Common Info, Execute Command and Message Structures */
/*=====================================================*/

typedef struct _wfs_firmware
{
    LPWSTR          lpszFirmwareName;
    LPWSTR          lpszFirmwareVersion;
    LPWSTR          lpszHardwareRevision;
} WFSFIRMWARE, *LPWFSFIRMWARE;

typedef struct _wfs_software
{
    LPWSTR          lpszSoftwareName;
    LPWSTR          lpszSoftwareVersion;
} WFSSOFTWARE, *LPWFSSOFTWARE;

typedef struct _wfs_device_info
```

```
{
    LPWSTR          lpszModelName;
    LPWSTR          lpszSerialNumber;
    LPWSTR          lpszRevisionNumber;
    LPWSTR          lpszModelDescription;
    LPWFSFIRMWARE   *lppFirmware;
    LPWFSSOFTWARE   *lppSoftware;
} WFSDEVICEINFO, *LPWFSDEVICEINFO;

typedef struct _wfs_vendor_mode_info
{
    BOOL            bAllowOpenSessions;
    LPDWORD         lpdwAllowedExecuteCommands;
} WFSVENDORMODEINFO, *LPWFSVENDORMODEINFO;

typedef struct _wfs_service_interface
{
    LPDWORD         lpdwExecuteCommands;
    LPDWORD         lpdwGetInfoCategories;
    LPDWORD         lpdwEvents;
    DWORD           dwMaximumRequests;
    LPDWORD         lpdwAuthenticationRequired;
} WFSSERVICEINTERFACE, *LPWFSSERVICEINTERFACE;

typedef struct _wfs_compound_device
{
    LPWSTR          lpszLogicalServiceName;
    LPWSTR          lpszProviderName;
    LPWSTR          lpszPhysicalServiceName;
    LPWSTR          lpszDeviceClass;
} WFSCOMPOUNDDEVICE, *LPWFSCOMPOUNDDEVICE;

typedef struct _wfs_service_info
{
    LPWFSDEVICEINFO       *lppDeviceInformation;
    LPWFSVENDORMODEINFO   lpVendorModeInformation;
    LPWFSSERVICEINTERFACE lpServiceInterface;
    LPWFSCOMPOUNDDEVICE   *lppCompoundDevices;
    LPWSTR                lpszServiceProviderVersion;
    LPSTR                 lpszExtra;
} WFSSERVICEINFO, *LPWFSSERVICEINFO;

typedef struct _wfs_api_transaction_info
{
    LPWSTR          lpszTransactionID;
    LPSTR           lpszExtra;
} WFSAPITRANSACTIONINFO, *LPWFSAPITRANSACTIONINFO;

typedef struct _wfs_api_transaction_state
{
    WORD            fwState;
    LPWFSAPITRANSACTIONINFO lpTransactionInfo;
} WFSAPITRANSACTIONSTATE, *LPWFSAPITRANSACTIONSTATE;

/****** Message Structures ********************************************/

typedef struct _wfs_devstatus
{
    LPSTR           lpszPhysicalName;
    LPSTR           lpszWorkstationName;
    DWORD           dwState;
} WFSDEVSTATUS, *LPWFSDEVSTATUS;

typedef struct _wfs_undevmsg
{
    LPSTR           lpszLogicalName;
    LPSTR           lpszWorkstationName;
    LPSTR           lpszAppID;
    DWORD           dwSize;
    LPBYTE          lpbDescription;
```

```
    DWORD           dwMsg;
    LPWFSRESULT     lpWFSResult;
} WFSUNDEVMSG, *LPWFSUNDEVMSG;

typedef struct _wfs_appdisc
{
    LPSTR           lpszLogicalName;
    LPSTR           lpszWorkstationName;
    LPSTR           lpszAppID;
} WFSAPPDISC, *LPWFSAPPDISC;

typedef struct _wfs_hwerror
{
    LPSTR           lpszLogicalName;
    LPSTR           lpszPhysicalName;
    LPSTR           lpszWorkstationName;
    LPSTR           lpszAppID;
    DWORD           dwAction;
    DWORD           dwSize;
    LPBYTE          lpbDescription;
} WFSHWERROR, *LPWFSHWERROR;

typedef struct _wfs_vrsnerror
{
    LPSTR           lpszLogicalName;
    LPSTR           lpszWorkstationName;
    LPSTR           lpszAppID;
    DWORD           dwSize;
    LPBYTE          lpbDescription;
    LPWFSVERSION    lpWFSVersion;
} WFSVRSNERROR, *LPWFSVRSNERROR;

typedef struct _wfs_api_status_changed
{
    LPVOID          lpvOldStatus;
    LPVOID          lpvNewStatus;
} WFSSTATUSCHANGED, *LPWFSSTATUSCHANGED;

/****** Error codes ******************************************************/

#define WFS_SUCCESS                         (0)
#define WFS_ERR_ALREADY_STARTED             (-1)
#define WFS_ERR_API_VER_TOO_HIGH            (-2)
#define WFS_ERR_API_VER_TOO_LOW             (-3)
#define WFS_ERR_CANCELED                    (-4)
#define WFS_ERR_CFG_INVALID_HKEY            (-5)
#define WFS_ERR_CFG_INVALID_NAME            (-6)
#define WFS_ERR_CFG_INVALID_SUBKEY          (-7)
#define WFS_ERR_CFG_INVALID_VALUE           (-8)
#define WFS_ERR_CFG_KEY_NOT_EMPTY           (-9)
#define WFS_ERR_CFG_NAME_TOO_LONG           (-10)
#define WFS_ERR_CFG_NO_MORE_ITEMS           (-11)
#define WFS_ERR_CFG_VALUE_TOO_LONG          (-12)
#define WFS_ERR_DEV_NOT_READY               (-13)
#define WFS_ERR_HARDWARE_ERROR              (-14)
#define WFS_ERR_INTERNAL_ERROR              (-15)
#define WFS_ERR_INVALID_ADDRESS             (-16)
#define WFS_ERR_INVALID_APP_HANDLE          (-17)
#define WFS_ERR_INVALID_BUFFER              (-18)
#define WFS_ERR_INVALID_CATEGORY            (-19)
#define WFS_ERR_INVALID_COMMAND             (-20)
#define WFS_ERR_INVALID_EVENT_CLASS         (-21)
#define WFS_ERR_INVALID_HSERVICE            (-22)
#define WFS_ERR_INVALID_HPROVIDER           (-23)
#define WFS_ERR_INVALID_HWND                (-24)
#define WFS_ERR_INVALID_HWNDREG             (-25)
#define WFS_ERR_INVALID_POINTER             (-26)
#define WFS_ERR_INVALID_REQ_ID              (-27)
#define WFS_ERR_INVALID_RESULT              (-28)
#define WFS_ERR_INVALID_SERVPROV            (-29)
```

```
#define WFS_ERR_INVALID_TIMER                   (-30)
#define WFS_ERR_INVALID_TRACELEVEL              (-31)
#define WFS_ERR_LOCKED                          (-32)
#define WFS_ERR_NO_BLOCKING_CALL                (-33)
#define WFS_ERR_NO_SERVPROV                     (-34)
#define WFS_ERR_NO_SUCH_THREAD                  (-35)
#define WFS_ERR_NO_TIMER                        (-36)
#define WFS_ERR_NOT_LOCKED                      (-37)
#define WFS_ERR_NOT_OK_TO_UNLOAD                (-38)
#define WFS_ERR_NOT_STARTED                     (-39)
#define WFS_ERR_NOT_REGISTERED                  (-40)
#define WFS_ERR_OP_IN_PROGRESS                  (-41)
#define WFS_ERR_OUT_OF_MEMORY                   (-42)
#define WFS_ERR_SERVICE_NOT_FOUND               (-43)
#define WFS_ERR_SPI_VER_TOO_HIGH                (-44)
#define WFS_ERR_SPI_VER_TOO_LOW                 (-45)
#define WFS_ERR_SRVC_VER_TOO_HIGH               (-46)
#define WFS_ERR_SRVC_VER_TOO_LOW                (-47)
#define WFS_ERR_TIMEOUT                         (-48)
#define WFS_ERR_UNSUPP_CATEGORY                 (-49)
#define WFS_ERR_UNSUPP_COMMAND                  (-50)
#define WFS_ERR_VERSION_ERROR_IN_SRVC           (-51)
#define WFS_ERR_INVALID_DATA                    (-52)
#define WFS_ERR_SOFTWARE_ERROR                  (-53)
#define WFS_ERR_CONNECTION_LOST                 (-54)
#define WFS_ERR_USER_ERROR                      (-55)
#define WFS_ERR_UNSUPP_DATA                     (-56)
#define WFS_ERR_FRAUD_ATTEMPT                   (-57)
#define WFS_ERR_SEQUENCE_ERROR                  (-58)
#define WFS_ERR_AUTH_REQUIRED                   (-59)


#define WFS_INDEFINITE_WAIT                     0

/****** Messages **********************************************************/

/* Message-No = (WM_USER + No) */

#define WFS_OPEN_COMPLETE                       (WM_USER + 1)
#define WFS_CLOSE_COMPLETE                      (WM_USER + 2)
#define WFS_LOCK_COMPLETE                       (WM_USER + 3)
#define WFS_UNLOCK_COMPLETE                     (WM_USER + 4)
#define WFS_REGISTER_COMPLETE                   (WM_USER + 5)
#define WFS_DEREGISTER_COMPLETE                 (WM_USER + 6)
#define WFS_GETINFO_COMPLETE                    (WM_USER + 7)
#define WFS_EXECUTE_COMPLETE                    (WM_USER + 8)

#define WFS_EXECUTE_EVENT                       (WM_USER + 20)
#define WFS_SERVICE_EVENT                       (WM_USER + 21)
#define WFS_USER_EVENT                          (WM_USER + 22)
#define WFS_SYSTEM_EVENT                        (WM_USER + 23)

#define WFS_TIMER_EVENT                         (WM_USER + 100)

/****** Event Classes *****************************************************/

#define SERVICE_EVENTS                          (1)
#define USER_EVENTS                             (2)
#define SYSTEM_EVENTS                           (4)
#define EXECUTE_EVENTS                          (8)

/****** System Event IDs **************************************************/

#define WFS_SYSE_UNDELIVERABLE_MSG              (1)
#define WFS_SYSE_HARDWARE_ERROR                 (2)
#define WFS_SYSE_VERSION_ERROR                  (3)
#define WFS_SYSE_DEVICE_STATUS                  (4)
#define WFS_SYSE_APP_DISCONNECT                 (5)
#define WFS_SYSE_SOFTWARE_ERROR                 (6)
#define WFS_SYSE_USER_ERROR                     (7)
```

```
#define WFS_SYSE_LOCK_REQUESTED                 (8)
#define WFS_SYSE_FRAUD_ATTEMPT                  (9)


/****** XFS Trace Level *******************************************/

#define WFS_TRACE_API                           (0x00000001)
#define WFS_TRACE_ALL_API                       (0x00000002)
#define WFS_TRACE_SPI                           (0x00000004)
#define WFS_TRACE_ALL_SPI                       (0x00000008)
#define WFS_TRACE_MGR                           (0x00000010)

/****** XFS Error Actions *****************************************/

#define WFS_ERR_ACT_NOACTION                    (0x0000)
#define WFS_ERR_ACT_RESET                       (0x0001)
#define WFS_ERR_ACT_SWERROR                     (0x0002)
#define WFS_ERR_ACT_CONFIG                      (0x0004)
#define WFS_ERR_ACT_HWCLEAR                     (0x0008)
#define WFS_ERR_ACT_HWMAINT                     (0x0010)
#define WFS_ERR_ACT_SUSPEND                     (0x0020)

/****** XFS SNMP MIB Category Codes *********************************/
/* NOTE: To support the XFS SNMP MIB, the WFSGet[Async]Info category codes between
60000 and 60999 are reserved.*/


/****** API functions ************************************************/

HRESULT extern WINAPI WFSCancelAsyncRequest (HSERVICE hService, REQUESTID RequestID);

HRESULT extern WINAPI WFSCancelBlockingCall (DWORD dwThreadID);

HRESULT extern WINAPI WFSCleanUp ();

HRESULT extern WINAPI WFSClose (HSERVICE hService);

HRESULT extern WINAPI WFSAsyncClose (HSERVICE hService, HWND hWnd, LPREQUESTID
lpRequestID);

HRESULT extern WINAPI WFSCreateAppHandle (LPHAPP lphApp);

HRESULT extern WINAPI WFSDeregister (HSERVICE hService, DWORD dwEventClass, HWND
hWndReg);

HRESULT extern WINAPI WFSAsyncDeregister (HSERVICE hService, DWORD dwEventClass, HWND
hWndReg, HWND hWnd, LPREQUESTID lpRequestID);

HRESULT extern WINAPI WFSDestroyAppHandle (HAPP hApp);

HRESULT extern WINAPI WFSExecute (HSERVICE hService, DWORD dwCommand, LPVOID
lpCmdData, DWORD dwTimeOut, LPWFSRESULT * lppResult);

HRESULT extern WINAPI WFSAsyncExecute (HSERVICE hService, DWORD dwCommand, LPVOID
lpCmdData, DWORD dwTimeOut, HWND hWnd, LPREQUESTID lpRequestID);

HRESULT extern WINAPI WFSFreeResult (LPWFSRESULT lpResult);

HRESULT extern WINAPI WFSGetInfo (HSERVICE hService, DWORD dwCategory, LPVOID
lpQueryDetails, DWORD dwTimeOut, LPWFSRESULT * lppResult);

HRESULT extern WINAPI WFSAsyncGetInfo (HSERVICE hService, DWORD dwCategory, LPVOID
lpQueryDetails, DWORD dwTimeOut, HWND hWnd, LPREQUESTID lpRequestID);

BOOL extern WINAPI WFSIsBlocking ();

HRESULT extern WINAPI WFSLock (HSERVICE hService, DWORD dwTimeOut, LPWFSRESULT *
lppResult);

HRESULT extern WINAPI WFSAsyncLock (HSERVICE hService, DWORD dwTimeOut, HWND hWnd,
```

```
LPREQUESTID lpRequestID);

HRESULT extern WINAPI WFSOpen (~~LPSTR~~LPCSTR lpszLogicalName, HAPP hApp, ~~LPSTR~~LPCSTR
lpszAppID, DWORD dwTraceLevel, DWORD dwTimeOut, DWORD dwSrvcVersionsRequired,
LPWFSVERSION lpSrvcVersion, LPWFSVERSION lpSPIVersion, LPHSERVICE lphService);

HRESULT extern WINAPI WFSAsyncOpen (~~LPSTR~~LPCSTR lpszLogicalName, HAPP hApp,
~~LPSTR~~LPCSTR lpszAppID, DWORD dwTraceLevel, DWORD dwTimeOut, LPHSERVICE lphService,
HWND hWnd, DWORD dwSrvcVersionsRequired, LPWFSVERSION lpSrvcVersion, LPWFSVERSION
lpSPIVersion, LPREQUESTID lpRequestID);

HRESULT extern WINAPI WFSRegister (HSERVICE hService, DWORD dwEventClass, HWND
hWndReg);

HRESULT extern WINAPI WFSAsyncRegister (HSERVICE hService, DWORD dwEventClass, HWND
hWndReg, HWND hWnd, LPREQUESTID lpRequestID);

HRESULT extern WINAPI WFSSetBlockingHook (XFSBLOCKINGHOOK lpBlockFunc,
LPXFSBLOCKINGHOOK lppPrevFunc);

HRESULT extern WINAPI WFSStartUp (DWORD dwVersionsRequired, LPWFSVERSION
lpWFSVersion);

HRESULT extern WINAPI WFSUnhookBlockingHook ();

HRESULT extern WINAPI WFSUnlock (HSERVICE hService);

HRESULT extern WINAPI WFSAsyncUnlock (HSERVICE hService, HWND hWnd, LPREQUESTID
lpRequestID);

HRESULT extern WINAPI WFMSetTraceLevel (HSERVICE hService, DWORD dwTraceLevel);


/*   restore alignment   */
#pragma pack(pop)

#ifdef __cplusplus
}       /*extern "C"*/
#endif

#endif  /* __inc_xfsapi__h */
```

~~16.~~217.2 XFSADMIN.H

```
/******************************************************************************
*                                                                             *
* xfsadmin.h    XFS-Administration and Support functions                      *
*                                                                             *
*               Version 3.~~30  (March 19 2015)~~ 40  (December 6 2019)
*
*                                                                             *
******************************************************************************/

#ifndef __INC_XFSADMIN__H
#define __INC_XFSADMIN__H

#ifdef __cplusplus
extern "C" {
#endif

#include    <xfsapi.h>

/*   be aware of alignment   */
#pragma pack(push,1)


/* values of ulFlags used for WFMAllocateBuffer */

#define WFS_MEM_SHARE                       0x00000001
#define WFS_MEM_ZEROINIT                    0x00000002

/****** Support Functions *****************************************************/

HRESULT extern WINAPI WFMAllocateBuffer (ULONG ulSize, ULONG ulFlags, LPVOID *
lppvData);

HRESULT extern WINAPI WFMAllocateMore (ULONG ulSize, LPVOID lpvOriginal, LPVOID *
lppvData);

HRESULT extern WINAPI WFMFreeBuffer (LPVOID lpvData);

HRESULT extern WINAPI WFMGetTraceLevel (HSERVICE hService, LPDWORD lpdwTraceLevel);

HRESULT extern WINAPI WFMKillTimer (WORD wTimerID);

HRESULT extern WINAPI WFMOutputTraceData (~~LPSTR~~LPCSTR lpszData);

HRESULT extern WINAPI WFMReleaseDLL (HPROVIDER hProvider);

HRESULT extern WINAPI WFMSetTimer (HWND hWnd, LPVOID lpContext, DWORD dwTimeVal,
LPWORD lpwTimerID);


/*   restore alignment   */
#pragma pack(pop)

#ifdef __cplusplus
}       /*extern "C"*/
#endif

#endif    /* __INC_XFSADMIN__H */
```

~~16.3~~17.3 XFSCONF.H

```
/******************************************************************************
*                                                                             *
* xfsconf.h      XFS - definitions for the Configuration functions            *
*                                                                             *
*                Version 3.~~30  (March 19 2015)~~ 40  (December 6 2019)
*
*                                                                             *
******************************************************************************/

#ifndef __INC_XFSCONF__H
#define __INC_XFSCONF__H

#ifdef __cplusplus
extern "C" {
#endif

/******* Common ***************************************************************/

#include    <xfsapi.h>

/*   be aware of alignment   */
#pragma pack(push,1)

// following HKEY and PHKEY are already defined in WINREG.H
// so definition has been removed
// typedef HANDLE  HKEY;
// typedef HANDLE * PHKEY;

/******* Values of hKey *******************************************************/

#define     WFS_CFG_HKEY_XFS_ROOT               ((HKEY)1)
#define     WFS_CFG_HKEY_MACHINE_XFS_ROOT       ((HKEY)2)
#define     WFS_CFG_HKEY_USER_DEFAULT_XFS_ROOT  ((HKEY)3)
#define     WFS_CFG_CURRENT_USER_XFS_ROOT       ((HKEY)4)
// The following values are added for backwards compatibility reasons
#define     WFS_CFG_MACHINE_XFS_ROOT            WFS_CFG_HKEY_MACHINE_XFS_ROOT
#define     WFS_CFG_USER_DEFAULT_XFS_ROOT       WFS_CFG_HKEY_USER_DEFAULT_XFS_ROOT

/******* Values of lpdwDisposition ********************************************/

#define     WFS_CFG_CREATED_NEW_KEY             (0)
#define     WFS_CFG_OPENED_EXISTING_KEY         (1)

/******* Configuration Functions **********************************************/

HRESULT extern  WINAPI  WFMCloseKey (HKEY hKey);

HRESULT extern  WINAPI  WFMCreateKey (HKEY hKey, ~~LPSTR~~LPCSTR lpszSubKey, PHKEY
phkResult, LPDWORD lpdwDisposition);

HRESULT extern  WINAPI  WFMDeleteKey (HKEY hKey, ~~LPSTR~~LPCSTR lpszSubKey);

HRESULT extern  WINAPI  WFMDeleteValue (HKEY hKey, ~~LPSTR~~LPCSTR lpszValue );

HRESULT extern  WINAPI  WFMEnumKey (HKEY hKey, DWORD iSubKey, LPSTR lpszName, LPDWORD
lpcchName, PFILETIME lpftLastWrite);

HRESULT extern  WINAPI  WFMEnumValue (HKEY hKey, DWORD iValue, LPSTR lpszValue,
LPDWORD lpcchValue, LPSTR lpszData, LPDWORD lpcchData);

HRESULT extern  WINAPI  WFMOpenKey (HKEY hKey, ~~LPSTR~~LPCSTR lpszSubKey, PHKEY
phkResult);

HRESULT extern  WINAPI  WFMQueryValue (HKEY hKey, ~~LPSTR~~LPCSTR lpszValueName, LPSTR
lpszData, LPDWORD lpcchData);

HRESULT extern  WINAPI  WFMSetValue (HKEY hKey, ~~LPSTR~~LPCSTR lpszValueName, LPSTR
```

```
lpszData, DWORD cchData);


/*   restore alignment   */
#pragma pack(pop)

#ifdef __cplusplus
}       /*extern "C"*/
#endif

#endif  /* __INC_XFSCONF__H */
```

~~16.~~17.4 XFSSPI.H

```
/******************************************************************************
*                                                                             *
* xfsspi.h      XFS - SPI functions, types, and definitions                   *
*                                                                             *
*               Version 3.~~30  (March 19 2015)~~ 40  (December 6 2019)
*
*                                                                             *
******************************************************************************/

#ifndef __inc_xfsspi__h
#define __inc_xfsspi__h

#ifdef __cplusplus
extern "C" {
#endif

#include <xfsapi.h>

typedef HANDLE HPROVIDER;

#include <xfsconf.h>
#include <xfsadmin.h>

/*   be aware of alignment   */
#pragma pack(push,1)

/****** SPI functions ********************************************************/

HRESULT extern WINAPI WFPCancelAsyncRequest (HSERVICE hService, REQUESTID RequestID);

HRESULT extern WINAPI WFPClose (HSERVICE hService, HWND hWnd, REQUESTID ReqID);

HRESULT extern WINAPI WFPDeregister (HSERVICE hService, DWORD dwEventClass, HWND
hWndReg, HWND hWnd, REQUESTID ReqID);

HRESULT extern WINAPI WFPExecute (HSERVICE hService, DWORD dwCommand, LPVOID
lpCmdData, DWORD dwTimeOut, HWND hWnd, REQUESTID ReqID);

HRESULT extern WINAPI WFPGetInfo (HSERVICE hService, DWORD dwCategory, LPVOID
lpQueryDetails, DWORD dwTimeOut, HWND hWnd, REQUESTID ReqID);

HRESULT extern WINAPI WFPLock (HSERVICE hService, DWORD dwTimeOut, HWND hWnd,
REQUESTID ReqID);

HRESULT extern WINAPI WFPOpen (HSERVICE hService, ~~LPSTR~~LPCSTR lpszLogicalName, HAPP
hApp, ~~LPSTR~~LPCSTR lpszAppID, DWORD dwTraceLevel, DWORD dwTimeOut, HWND hWnd, REQUESTID
ReqID, HPROVIDER hProvider, DWORD dwSPIVersionsRequired, LPWFSVERSION lpSPIVersion,
DWORD dwSrvcVersionsRequired, LPWFSVERSION lpSrvcVersion);

HRESULT extern WINAPI WFPRegister (HSERVICE hService,  DWORD dwEventClass, HWND
hWndReg, HWND hWnd, REQUESTID ReqID);

HRESULT extern WINAPI WFPSetTraceLevel (HSERVICE hService, DWORD dwTraceLevel);

HRESULT extern WINAPI WFPUnloadService ();

HRESULT extern WINAPI WFPUnlock (HSERVICE hService, HWND hWnd, REQUESTID ReqID);


/*   restore alignment   */
#pragma pack(pop)

#ifdef __cplusplus
}    /*extern "C"*/
#endif

#endif  /* __inc_xfsspi__h */
```