z/OS

# UNIX System Services File System Interface Reference

SA22-7808-07

z/OS

# UNIX System Services File System Interface Reference

SA22-7808-07

**Note**

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 561.

**Eigth Edition, April 2006**

This edition applies to Version 1 Release 7 of z/OS (5694-A01), to Version 1 Release 7 of z/OS.e™ (5655-G52), and to all subsequent releases and modifications until otherwise indicated in new editions.

This is a major revision of SA22-7808-06.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this document, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
Your International Access Code +1+845+432-9405

IBMLink™ (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: www.ibm.com/servers/eserver/zseries/zos/webqs.html

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
- Title and order number of this document
- Page number or topic related to your comment

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

**© Copyright International Business Machines Corporation 1996, 2006. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents



© Copyright IBM Corp. 1996, 2006 

# Figures



© Copyright IBM Corp. 1996, 2006 

# Tables



© Copyright IBM Corp. 1996, 2006 

# About this document

This document describes the interfaces that are used to create physical file systems (PFSs) and virtual file system (VFS) servers that can operate with z/OS UNIX System Services (z/OS UNIX). PFSs and VFS servers might be written to extend the services provided by z/OS UNIX in the areas of device support for a file system or network access to file systems. This document also describes how to use these interfaces.

Chapter 1 is a general overview that shows how the physical file system, logical file system, and virtual file system server interact. Chapters 2 and 3 describe the physical file system interface. Chapters 4 and 5 describe the virtual file system server interface. Chapter 6 describes the Operating System Interface (OSI) callable services.

In the appendixes, you will find information about:
- System control offsets to callable services
- Mapping macros
- Callable services examples
- Interface structures for C language servers and clients
- Assembler and C-language facilities for writing a PFS in C
- Accessibility features
- Notices
- An index

## Who should use this document?

This document is intended for a specialized audience: system programmers using C or assembler language to create a physical file system (PFS) or a virtual file system (VFS) server, or to port a PFS or a VFS server to z/OS UNIX. Knowledge of POSIX or UNIX® is assumed.

Depending on the complexity of the PFS or VFS server involved, a considerable amount of MVS™ system programming knowledge might be required. Detailed information on MVS services that might be needed can be found in:
- *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*
- *z/OS MVS Programming: Authorized Assembler Services Reference ENF-IXG*
- *z/OS MVS Programming: Authorized Assembler Services Reference LLA-SDU*
- *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*
- *z/OS MVS Programming: Extended Addressability Guide*
- *z/OS MVS Programming: Authorized Assembler Services Guide*

This document should be used in conjunction with *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, and supplements information that is contained in IEEE Std 1003.1-1990 and IEEE Std 1003.1a.

## Where to find more information

Where necessary, this document references information in other documents about the elements and features of z/OS™. For complete titles and order numbers for all z/OS documents, see *z/OS Information Roadmap*.

Direct your request for copies of any IBM publication to your IBM representative or to the IBM branch office serving your locality.

© Copyright IBM Corp. 1996, 2006 

There is also a toll-free customer support number (1-800-879-2755) available Monday through Friday from 6:30 a.m. through 5:00 p.m. Mountain Time. You can use this number to:

- Order or inquire about IBM publications
- Resolve any software manufacturing or delivery concerns
- Activate the program reorder form to provide faster and more convenient ordering of software updates

## Softcopy publications

The z/OS UNIX library is available on the *z/OS Collection Kit*, SK2T-6700. This softcopy collection contains a set of z/OS and related unlicensed product documents. The CD-ROM collection includes the IBM® Library Reader™, a program that enables customers to read the softcopy documents.

Softcopy z/OS publications are also available for web-browsing and PDF versions of the z/OS publications for viewing or printing using Adobe Acrobat Reader. Visit the z/OS library at www.ibm.com/servers/eserver/zseries/zos/bkserv/.

## IBM Systems Center publications

IBM Systems Centers produce Redbooks that can be helpful in setting up and using z/OS UNIX System Services. You can order these publications through normal channels, or you can view them with a Web browser. See the IBM Redbooks site at www.ibm.com/redbooks.

These documents have not been subjected to any formal review nor have they been checked for technical accuracy, but they represent current product understanding (at the time of their publication) and provide valuable information on a wide range of z/OS UNIX topics. You must order them separately. A selected list of these documents is on the z/OS UNIX web site at http://www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1pub.html/.

## z/OS UNIX porting information

There is a *Porting Guide* on the z/OS UNIX porting page at www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1por.html.You can read the *Porting Guide* from the web or download it as a PDF file that you can view or print using Adobe Acrobat Reader. The *Porting Guide* covers a range of useful topics, including: sizing a port, setting up a porting environment, ASCII-EBCDIC issues, performance, and much more.

The porting page also features a variety of porting tips, and lists porting resources that will help you in your port.

## z/OS UNIX courses

For a current list of courses that you can take, go to www.ibm.com/services/learning/.

You can also see your IBM representative or call 1-800-IBM-TEACH (1-800-426-8322).

## z/OS UNIX home page

The z/OS UNIX home page on the World Wide Web contains technical news, customer stories, and information about tools. You can visit it at www.ibm.com/servers/eserver/zseries/zos/unix/.

Some of the tools available from the web site are ported tools, and some are home-grown tools designed for z/OS UNIX. The code works in our environment at the time we make it available, but is not officially supported. Each tool has a README file that describes the tool and lists any restrictions.

The simplest way to reach these tools is through the z/OS UNIX home page. From the home page, click on **Tools and Toys**.

The code is also available from **ftp://ftp.software.ibm.com/s390/zos/unix/** through anonymous ftp.

**Restrictions**

Because the tools are not officially supported, APARs cannot be accepted.

## z/OS UNIX customization wizard

For help with customizing z/OS UNIX, check out our Web-based wizard at www.ibm.com/servers/eserver/zseries/zos/wizards/.

This wizard builds two BPXPRM*xx* parmlib members; one with system processing parameters and one with file system statements. It also builds a batch job that does the initial RACF® security setup for z/OS UNIX. Whether you are installing z/OS UNIX for the first time or are a current user who wishes to verify settings, you can use this wizard.

The wizard also allows sysplex users to build a single BPXPRM*xx* parmlib member to define all the file systems used by sysplex members participating in a z/OS UNIX shared file system.

## Discussion list

Customers and IBM participants also discuss z/OS UNIX on the **mvs-oe discussion list**. This list is not operated or sponsored by IBM.

To subscribe to the mvs-oe discussion, send a note to:

**listserv@vm.marist.edu**

Include the following line in the body of the note, substituting your first name and last name as indicated:

```
subscribe mvs-oe first_name last_name
```

After you are subscribed, you will receive further instructions on how to use the mailing list.

## Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can use LookAt from these locations to find IBM message explanations for z/OS elements and features, z/VM®, VSE/ESA™, and Clusters for AIX® and Linux™:

- The Internet. You can access IBM message explanations directly from the LookAt Web site at http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/.

- Your z/OS TSO/E host system. You can install code on your z/OS or z/OS.e systems to access IBM message explanations using LookAt from a TSO/E command line (for example: TSO/E prompt, ISPF, or z/OS UNIX System Services).
- Your Microsoft® Windows® workstation. You can install LookAt directly from the *z/OS Collection* (SK3T-4269) or the *z/OS and Software Products DVD Collection* (SK3T-4271) and use it from the resulting Windows graphical user interface (GUI). The command prompt (also known as the DOS > command line) version can still be used from the directory in which you install the Windows version of LookAt.
- Your wireless handheld device. You can use the LookAt Mobile Edition from http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookatm.html with a handheld device that has wireless access and an Internet browser (for example: Internet Explorer for Pocket PCs, Blazer or Eudora for Palm OS, or Opera for Linux handheld devices).

You can obtain code to install LookAt on your host system or Microsoft Windows workstation from:

- A CD-ROM in the *z/OS Collection* (SK3T-4269).
- The *z/OS and Software Products DVD Collection* (SK3T-4271).
- The LookAt Web site (click **Download** and then select the platform, release, collection, and location that suit your needs). More information is available in the LOOKAT.ME files available during the download process.

## Using IBM Health Checker for z/OS

IBM Health Checker for z/OS is a z/OS component that installations can use to gather information about their system environment and system parameters to help identify potential configuration problems before they impact availability or cause outages. Individual products, z/OS components, or ISV software can provide checks that take advantage of the IBM Health Checker for z/OS framework. This book refers to checks or messages associated with this component.

For additional information about checks and about IBM Health Checker for z/OS, see *IBM Health Checker for z/OS: User's Guide*. z/OS V1R4, V1R5, and V1R6 users can obtain the IBM Health Checker for z/OS from the z/OS Downloads page at http://www.ibm.com/servers/eserver/zseries/zos/downloads/.

SDSF also provides functions to simplify the management of checks. See *z/OS SDSF Operation and Customization* for additional information.

## Finding more information about sockets

You can find more detailed information on sockets and their operations in various publications, including the following:

- *4.3BSD UNIX Operating System*, by S. J. Leffler et al.
- *z/OS XL C/C++ Programming Guide*
- *z/OS XL C/C++ Run-Time Library Reference*
- *AIX Version 4.3 Communications Programming Concepts*, SC23-4124

## Finding more information about timer units

You can find detailed information about timer units in *z/Architecture Principles of Operation*, SA22-7832.

# Summary of changes

**Summary of changes**
**for SA22-7808-07**
**z/OS Version 1 Release 7**

This document contains information previously presented in *z/OS UNIX System Services File System Interface Reference*, SA22-7808-06, which supports z/OS Version 1 Release 7.

**Changed information**

- Minor changes have been made to the following callable services:
  - "osi_copyin — Move data from a user buffer to a PFS buffer" on page 370
  - "osi_copyout — Move data from a PFS buffer to a user buffer" on page 373
  - "osi_copy64 — Move data between user and PFS buffers with 64-bit addresses" on page 376
  - "osi_kipcget — Query interprocess communications" on page 388
  - "osi_kmsgctl — Perform message queue control operations" on page 391
  - "osi_kmsgget — Create or find a message queue" on page 395
  - "osi_kmsgrcv — Receive from a message queue" on page 398
  - "osi_kmsgsnd — Send a message to a message queue" on page 402
  - "osi_uiomove — Move data between PFS buffers and buffers defined by a UIO structure" on page 426
  - "vn_bind — Bind a name to a socket" on page 125
  - "vn_setattr — Set the attributes of a file" on page 213

This document has been enabled for the following types of advanced searches in the online z/OS LibraryCenter: *examples*.

You may notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

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

**Summary of changes**
**for SA22-7808-06**
**z/OS Version 1 Release 7**

The document contains information previously presented in *z/OS UNIX System Services File System Interface Reference*, SA22-7808-05, which supports z/OS Version 1 Release 6.

**New information**

- Support has been added for the dynamic service activation capability, including a new flag, pfsi_modind, in the PFS initialization structure (BPXYPFSI).
- The BPX1VLO/BPX4VLO (v_lockctl) callable service has been enhanced and several new fields have been added to the Vlok structure (mapped by BPXYVLOK) to support the implementation of the version 4 NFS server protocols.

© Copyright IBM Corp. 1996, 2006

- The following callable services have been added to support the implementation of the version 4 NFS server protocols:
  - BPX1VCL/BPX4VCL (v_close)
  - BPX1VOP/BPX4VOP (v_open)

    A new macro, BPXYVOPN, maps the open parameters for the v_open service.

### Changed information

- Information about waiting and posting has been clarified to indicate that LFS serialization will not be dropped for writes to the stream sockets using the default socket option of exclusive write.
- Minor changes have been made to the following callable services:
  - BPX1VRD/BPX4VRD (v_readdir)
  - BPX1VRG/BPX4VRG (v_reg)
  - BPX1VRM/BPX4VRM (v_remove)
  - BPX1VRN/BPX4VRN (v_rename)
  - BPX1VRW/BPX4VRW (v_rdwr)
  - BPX1VSA/BPX4VSA (v_setattr)
- Throughout this document, the phrase 'shared HFS' has been changed to 'shared file system'.

### Deleted information

- The BPXTTOD sample assembler routine that was listed in the appendix is no longer accurate. The routine is not needed for writing PFSs and has been removed from this document.

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

## Summary of changes for SA22-7808-05 z/OS Version 1 Release 6

The document contains information previously presented in *z/OS UNIX System Services File System Interface Reference*, SA22-7808-04, which supports z/OS Version 1 Release 5.

For a list of new and changed callable services, see z/OS UNIX summary of interface changes in *z/OS Summary of Message and Interface Changes*.

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

## Summary of changes for SA22-7808-04 z/OS Version 1 Release 5

The document contains information previously presented in *z/OS UNIX System Services File System Interface Reference*, SA22-7808-03, which supports z/OS Version 1 Release 4.

### Changed information

- The v_setattr (BPX1VSA) callable service has been modified to support the use of security labels.
- An Osi field is added for improved Async I/O performance (see "Related OSI fields" on page 56).
- Chapter 2, "Physical file systems," on page 3 has been updated for MLS support.

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

## Summary of changes for SA22-7808-03 z/OS Version 1 Release 4

The document contains information previously presented in *z/OS UNIX System Services File System Interface Reference*, SA22-7808-02, which supports z/OS Version 1 Release 3.

### New information

A new section, "Considerations for Internet Protocol Version 6 (IPv6)" on page 62, is added to Chapter 2, "Physical File Systems".

### Changed information

Minor changes have been made to the vfs_mount callable service.

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

# Chapter 1. General overview

z/OS UNIX System Services (z/OS UNIX) allows you to install virtual file system servers (VFS servers) and physical file systems (PFSs).

- A **VFS server** makes requests for file system services on behalf of a client. A VFS server is similar to a POSIX program that reads and writes files, except that it uses the lower-level *VFS callable services API* instead of the POSIX C-language API.

  An example of a VFS server is the Network File System.
- **A physical file system (PFS)** controls access to data.

  PFSs receive and act upon requests to read and write files that they control. The format of these requests is defined by the *PFS interface*.

  PFSs include pipes, sockets, the Network File System client, and the following UNIX file systems: HFS, zFS, and TFS.

  Another name for a PFS is an *installable file system*.

User-written programs use the POSIX API to issue file requests. VFS servers use the *VFS callable services API* to issue file requests. These requests are routed by the *logical file system (LFS)* to the appropriate PFS through the PFS interface. See Figure 1 on page 2 for a view of this structure.

This document describes these two interfaces and discusses the things you need to know to write a VFS server or a PFS, or to port one to the z/OS UNIX environment. In order to do this, you should be a system programmer who is familiar with POSIX or UNIX.

**Porting note**

This document uses notes like this one to highlight certain points of the implementation that are particularly important to readers who are considering porting an existing UNIX-based program to z/OS UNIX.

z/OS UNIX supports the following types of files:
- Regular files
- Directories
- Symbolic links
- Character special files (for example, terminals)
- Pipes (both FIFOs and unnamed)
- Sockets

**Note:** Character special and unnamed pipe physical file systems cannot be implemented with this interface. Unnamed pipes and socket files cannot be exported by a VFS server.

## System structure

The position of the VFS server and the PFS in the structure of z/OS UNIX and the interfaces they use are illustrated in Figure 1 on page 2.

© Copyright IBM Corp. 1996, 2006

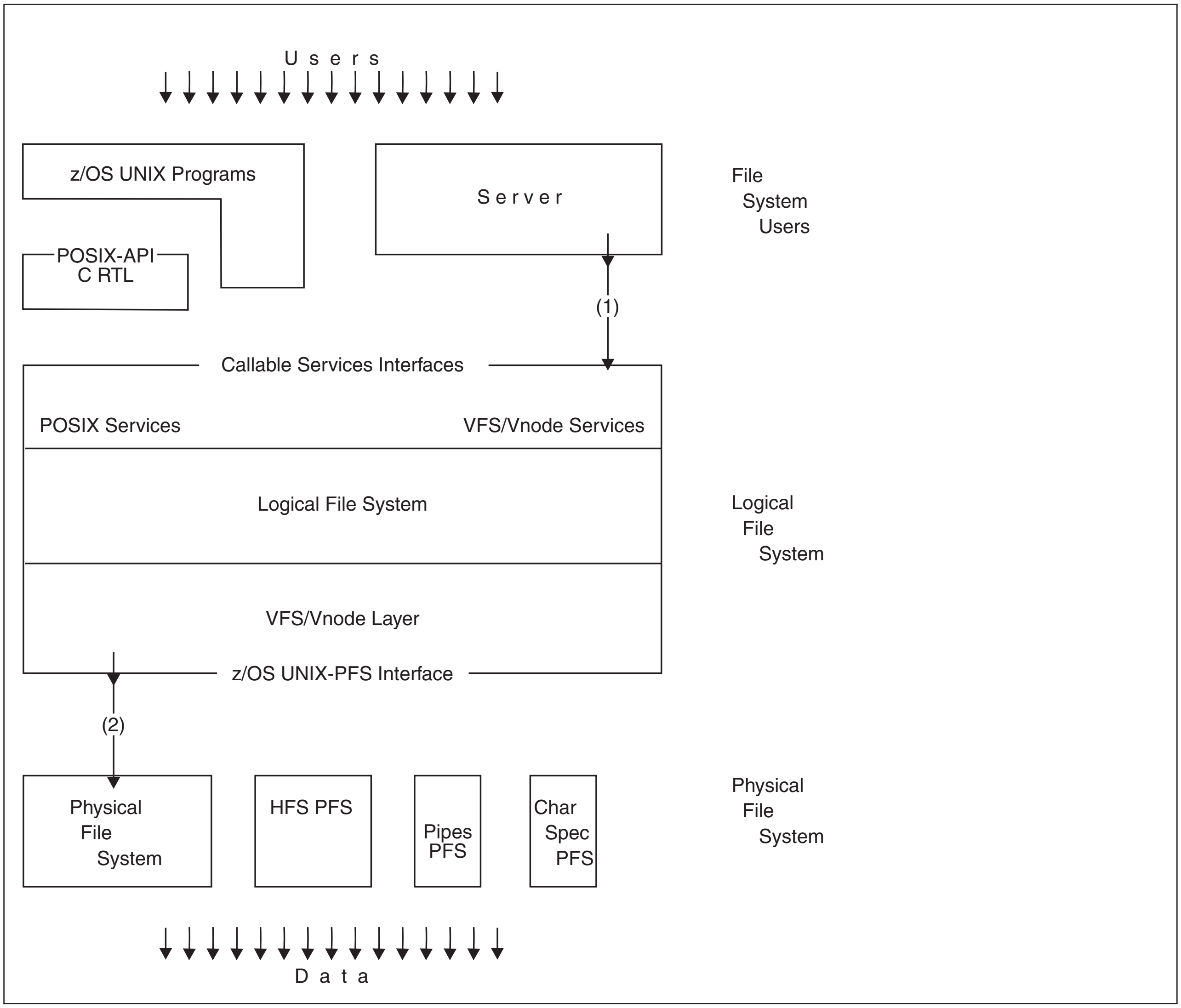


*Figure 1. VFS server and PFS structure*

(1) The VFS callable services API is used by VFS servers to call the logical file system.

(2) The logical file system calls the PFSs through the PFS interface.

# Chapter 2. Physical file systems

This chapter describes:
- How to install a physical file system (PFS)
- How a PFS is activated and deactivated
- The functions that must be provided by a PFS
- The functions that are provided for it
- Cross-memory considerations
- Considerations for writing a PFS in C
- Security considerations
- Running a PFS in a colony address space
- Considerations for Internet Protocol Version 6 (IPv6)
- PFS support for multilevel security
- PFS support for 64-bit virtual addressing

## Installing a PFS

A physical file system (PFS) is packaged as one or more MVS load modules. These load modules must be installed in an APF-authorized MVS load library. The hierarchical file system is not available when a PFS is loaded, so it cannot be installed in the file system.

The PFS must have an initialization routine whose entry point, called *PFS_Init* below, is externally known through the system link list or the STEPLIB of the OMVS cataloged procedure. If the PFS runs in a colony address space (see “Running a PFS in a colony address space” on page 14), it must be found through the system link list or a STEPLIB of the colony address space’s procedure.

A physical file system is defined to z/OS UNIX through the BPXPRMxx parmlib member you specify when you start the kernel address space (OMVS=xx). The FILESYSTYPE statement defines a single instance of a PFS.

Additional MOUNT, ROOT, SUBFILESYSTYPE, or NETWORK statements activate file system or socket support in the PFS.

```
FILESYSTYPE TYPE(file_system_type)
            ENTRYPOINT(PFS_Init)
            PARM(parameter_string)
            ASNAME(procname)
```

where:
- **TYPE** specifies a 1-to-8-character name that uniquely identifies this PFS. This name is used to route subsequent MOUNT, ROOT, SUBFILESYSTYPE, or NETWORK statements (as well as later MOUNT and PFSCTL syscalls) to the correct PFS.
- **ENTRYPOINT** specifies the name of the PFS’s initialization module. The LFS attaches the PFS_Init entry point as an MVS task. This task remains active for as long as the PFS is active. See “Activating and deactivating the PFS” on page 4 for a description of initialization processing requirements for this routine.
- **PARM** specifies a PFS-defined parameter text string that can contain any value and be up to 1024 bytes long. The meaning of this string is defined by the individual PFS. The string is passed to the PFS when the PFS_Init routine is attached.
- **ASNAME** specifies that the PFS is to run outside the kernel in a separate address space.

© Copyright IBM Corp. 1996, 2006

*procname* is the name of the procedure to be used when starting this address space, and a logical name for the address space. Each *procname* generates a different address space when it is first encountered, and each PFS with the same *procname* shares that address space. These address spaces are logical extensions of the kernel. They are referred to as *colony address spaces.*

All PFSs are activated automatically when z/OS UNIX is started, based on the FILESYSTYPE and SUBFILESYSTYPE statements in the parmlib member. This is the only way a PFS can be started.

Mounts may also be issued dynamically at a later time through a TSO/E command or a program function call. A mount is not strictly necessary, but it is required if the files that are managed by the PFS are to be visible in the file hierarchy (that is, if they are to be represented by standard pathnames). Support for mount generally implies support for the lookup operation, which is used to resolve a pathname to a file. Pipes and sockets are examples of files that are not in the hierarchy; these PFSs do not use mount.

For a discussion of mount processing, refer to “Mounting file systems” on page 27.

The ROOT statement is a special case of MOUNT. It can be issued only from parmlib, and it defines the system’s root file system.

The NETWORK statement does for a sockets PFS what MOUNT does for a data file type of PFS: It activates an address family, or domain, so that subsequent **socket()** calls are routed to that PFS to service.

For a discussion of network processing, refer to “Activating a domain” on page 43.

## Activating and deactivating the PFS

A PFS is started for each FILESYSTYPE statement in the BPXPRMxx parmlib member whenever z/OS UNIX is started. The LFS and PFS exchange information during this initialization phase. Usually the PFS does not terminate.

The same ENTRYPOINT name may be specified on two or more FILESYSTYPE statements with different TYPE operands. This causes the same PFS to be started more than once. It is up to the PFS to allow this or to detect and reject it.

## Activation flow for the PFS_Init module

The LFS builds a general file system table (GFS) for each PFS and attaches the PFS’s initialization entry point. This creates an independent MVS task, which is expected to follow these general steps:

1. Perform any PFS initialization that is necessary.
2. Load its VFS and vnode operation service routines and build their respective vector tables.

   These are the PFS routines that the LFS calls to get such services as mount, open, read, and write. The VFS and vnode operations vector tables make up the major part of the PFS interface.

   This loading may be done by link-editing the operational routines with the PFS_Init routine.
3. Save the OSI operations vector table (OSIT) address.

   The OSI operations vector table contains the addresses of LFS routines that the PFS uses to get certain services, such as those used to create vnodes.

4. Pass back to the LFS an 8-byte token that is saved by the LFS and used on all subsequent VFS and vnode operations. This token typically contains the address of the PFS's main anchor block. Its use is optional.
5. Exchange miscellaneous items of information between the LFS and PFS. Refer to "The PFSI structure" on page 6 and the PFSI structure in Appendix D for details on the specific information that is exchanged.
6. Notify the LFS that initialization has finished, by posting the initialization-complete ECB that was supplied.
7. Wait on the termination ECB, which is also supplied by the LFS. This ECB is posted by the LFS when it is time to terminate the PFS.

Each PFS is initialized synchronously and serially during z/OS UNIX initialization, so that no PFS may go into an extended wait during initialization.

**Note:** The file system is not available this early in z/OS UNIX initialization. If the PFS_Init routine needs configuration or other information from a file, it must use an MVS data set.

## PFS_Init entry interface

The PFS_Init routine receives control as the result of an MVS ATTACH in the following environment:

| | |
|---|---|
| **Authorization** | Supervisor state, PSW key 0 |
| **Dispatchable unit mode** | Task |
| **Cross memory mode** | PASN = HASN |
| **AMODE** | 31 bit |
| **ASC mode** | Primary mode |
| **Interrupt status** | Enabled for interrupts |
| **Locks** | Unlocked |
| **Control parameters** | All parameters are addressable in the primary address space |

On entry, register 1 points to a variable-length list of parameter addresses. The high-order bit of the last parameter address is turned on. For information about other entry registers, see *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN* for a description of ATTACH.

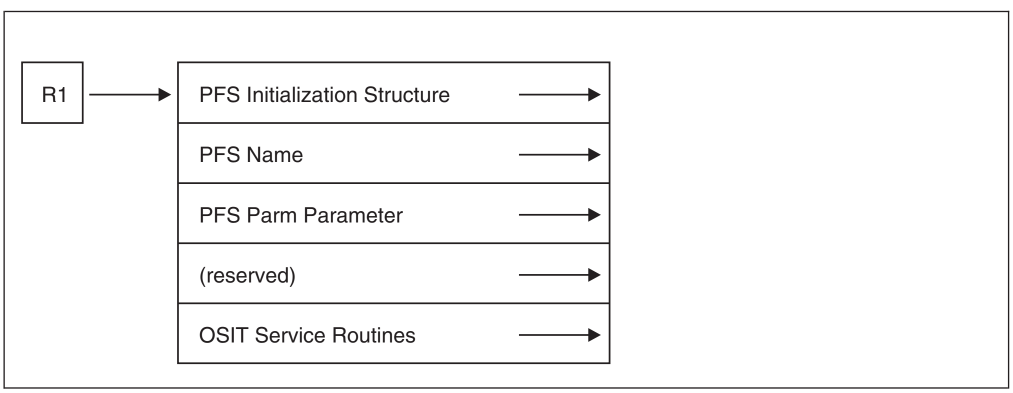


*Figure 2. PFS_Init entry parameter list*

The addresses in the parameter list point to the following parameters, each of which is described in Appendix D.

**Parameter** **Description**

**PFSI**
The PFS initialization structure. This contains information that is being passed to the PFS and fields that are to be filled in by the PFS during its initialization. See "The PFSI structure" for a description of these fields.

**PFSNAME**
An 8-byte field that contains the name of the PFS. This name was specified in either the TYPE parameter of a FILESYSTYPE parmlib statement or the NAME parameter of a SUBFILESYSTYPE parmlib statement. This name is used to identify the PFS for the **pfsctl()** function and, when applicable, for the **v_reg()** function.

**PFSPARM**
A variable-length field that contains the text string that is specified in the PARM parameter of the FILESYSTYPE statement. This is a 2-byte field that contains the length of the text string, followed by the string. If this parameter is absent, the length field is zero.

**OSIT**
The OSI service routine vector table, which provides the PFS with the addresses of the LFS service routines it needs to perform some basic functions.

See Chapter 6 for a description of the interfaces to, and functions of, each of these OSI routines.

## The PFSI structure

The PFS initialization structure (BPXYPFSI, referred to in this document as the PFSI) contains the following fields (each name is prefixed with the characters **pfsi**_):

| Field | Description |
|---|---|
| | **Supplied Fields** |
| **ver** | The version number of this PFSI. |
| **ook** | An indication that this PFS is running outside the kernel. |
| **alone** | An indication that this PFS is the only PFS running in the address space. |
| **new** | An indication that this is the first time this PFS has been initialized in the address space. |
| **romntclient** | Set on to indicate that the PFS does not support simultaneous R/O mounts from multiple systems; the LFS is responsible for making R/O file systems available for sharing in a sysplex system.<br><br>The default value is off. This indicates that the PFS supports sharing of R/O file systems in a sysplex. |
| **rwmntsysplex** | Set on to indicate that the PFS does not support simultaneous R/W mounts from multiple systems; the LFS is responsible for making R/W file systems available for sharing in a sysplex system.<br><br>The default value is off. This indicates that the PFS supports sharing of R/W file systems in a sysplex. |
| **initcompecb** | The ECB that the PFS posts when its initialization is complete. |
| **pfsecb** | The ECB that the LFS posts when z/OS UNIX is stopped. The PFS must be waiting on this ECB. |

**restart**
The address of the restart option byte. The PFS sets this byte any time during its processing, to control if and how it is to be restarted if it should terminate.

**dumpptr**
The address of dump information. This information is used by the PFS to add significant LFS areas to the dumps that are taken by the PFS.

**pfsid**
The PFS identifier that is used with osi_sleep and osi_wakeup.

**asname**
The value of the ASNAME parameter of the FILESYSTYPE statement.

**ep**
The value of the ENTRYPOINT parameter of the FILESYSTYPE statement.

**Returned Fields**

**pfsanchor**
The PFS initialization token. This token value is passed back to the PFS on every subsequent call from the LFS as part of the token_structure, which is the first parameter of every call. This field typically contains the address of the PFS's main anchor block.

**vfso**
The address of the PFS's VFS operation vector table.

**vnop**
The address of the PFS's vnode operation vector table.

**srb**
An indication that SRB mode is supported.

**asyio**
An indication that asynchronous I/O is supported.

**usethreads**
An indication that the PFS is requesting support for the osi_thread service. This field can be set only by PFSs that are running outside the kernel.

**disableLLA**
An indication that the LFS should not provide lookup lookaside function for this PFS. If there is not a strict one-to-one correspondence between the spelling of a file name in a directory and the vnode-inode pair that represents the file, the PFS should set this bit. For example, if `'/usr/dl/fl,attr=fb'` and `'/usr/dl/fl'` represent the same file in the **/usr/d1** directory, you must disable the LFS lookup lookaside function. If directories are remote and files may be removed from them remotely, the LFS's LLA cache should also be disabled.

**stayalone**
An indication that the LFS should not initialize any other PFSs in this address space. This field can be set only by PFSs that are running outside the kernel.

**immeddel**
An indication that the PFS supports deleting a removed file's data when its open count becomes zero, rather than waiting for **vn_inactive** to free the space.

**cpfs**
An indication that the PFS is written in C, and is requesting that the LFS invoke it with pre-initialized C environments.

**datoffmove**
An indication that the PFS supports DATOFF move for page read operations. For more information, see “Reading from and writing to files” on page 36.

**pfstype**
The type of the PFS. This identifies the PFS as a local file PFS, a remote file PFS, or a socket PFS.

**pipebuf**
**pathconf()** _PC_PIPE_BUF value, if applicable

**maxcanon**
**pathconf()** _PC_MAX_CANON value, if applicable

**maxinput**
**pathconf()** _PC_MAX_INPUT value, if applicable

**chownrstd**
**pathconf()** _PC_CHOWN_RESTRICTED value, if applicable

**vdisable**
**pathconf()** _PC_VDISABLE value, if applicable.

**Pathconf()** values that are not constant for all files supported by the PFS may be reported through the **vn_pathconf** operation.

**compon**
The PFS’s three-letter component (or module) prefix.

**compid**
The PFS’s five-letter component (or product) ID.

The component prefix and ID are used in dump titles for dumps that are taken by the LFS when there is an abnormal end in the PFS from which it does not recover.

**modind**
An indication that the PFS is supplying indirect addresses in the VFS and vnode operations vector tables for the various VFS and vnode operations routines.

## VFS and vnode operations vector tables

VFS and vnode operations vector tables are allocated and built by the PFS, and their addresses are returned in the PFSI. These tables may not be altered after the PFS posts the initialization-complete ECB.

Vnode operations, such as vn_open and vn_readdir, deal with file system objects. VFS operations, such as vfs_mount and vfs_statfs, deal with whole file systems or with the PFS itself.

The routine that supports each particular operation is loaded into storage by the PFS_Init routine, and the entry-point address is placed into the corresponding vector table entry. If the PFS supports the dynamic service activation capability, it must instead supply indirect addresses (that point to the actual entry-point addresses for each operation routine) in the vector table entries and set the pfsi_modind flag in the PFSI. When the LFS processes a VFS or vnode operation request, it will recognize the flag and use the address supplied in the vector table as an indirect address to locate the target operation routine.

If the PFS does not support a particular operation, the corresponding operation’s vector must contain 0. The number of operations that are placed in the table by the PFS, as determined by the returned table’s length, may be less than or equal to the number of operations that are supported by the LFS. If this value is less, the LFS treats all remaining operations as not supported, just as though the PFS had supplied 0 for those operation vectors. If the table contains more entries than the LFS expects, it is considered a serious product-level mismatch between the LFS and PFS, and the PFS is terminated.

For more information, see the description of vnoptab and vfsotab structures in Appendix D.

## Recycling a PFS externally

PFS Recycle can be driven externally by two calls to **pfsctl**. The caller must be a superuser. This is supported for kernel-resident PFSs only; for PFSs that are running in a colony address space, cancel the space to recycle the PFS.

PFS Recycle refreshes the PFS load module after service has been applied. The kernel space does not terminate; the only way to refresh a kernel-resident PFS load module is for the PFS_Init task to exit. The PFS may have its own technique to accomplish this and the PFS_Init task can exit on its own at any time. PFS Recycle restarts the PFS, or the LFS issues a WTOR and waits for a reply before restarting the PFS. Refer to “Termination considerations” on page 11 for details. These **pfsctl** commands coordinate the PFS’s termination with the LFS so that calls into the PFS can be quiesced before the PFS_Init task exits.

### PC#RecyclePFS X'8000000C'

PC#RecyclePFS X'8000000C' initiates a PFS recycle by posting the PFS’s termination ECB.

- If no argument is passed, or if the argument value is not 1, the LFS returns to the caller immediately after calls to the PFS have quiesced and the PFS has been posted to terminate. The caller and the PFS must coordinate any dependencies that they have on each other after this point, because the PFS may not have terminated when the caller regained control.
- If a fullword argument value of 1 is passed, the LFS waits for the PFS to terminate before returning to the caller.

The Return_value is 0 if the PFS is found.

Before this call the caller or PFS must ensure that:

- All current osi_waiters have been osi_posted.

  If the v_reg service has been used to register that the PFS is dependent on the caller’s process for osi_post, the LFS osi_posts the osi_waiters, just as it would if the caller’s process had terminated.
- All outstanding asyncio has been osi_scheduled.
- All internal waiters have been posted.
- No new vnode ops will be accepted by the PFS, or that no new ops will be allowed to wait or for asyncio to cue.

Before posting the PFS termination ECB, the LFS ensures that there are no more threads executing code in the PFS layer and it will permit no more VFS or vnode ops to branch into the PFS. The LFS waits for any threads that are still in the PFS layer at the time of the pfsctl call. These could include, for example, threads that were just osi_posted, but whose address space had not been swapped in yet, or that were otherwise not dispatched, so they have not had a chance to return back up to the LFS layer.

A race condition exists between this call and user threads that are branching into the PFS layer at about the same time. The PFS begins to reject these calls and the LFS waits for those rejected threads to exit from the PFS layer.

When the termination ECB is posted, the PFS cleans up and exits the PFS_Init module. This decrements the load module's use count; when that count goes to zero the load module is deleted. This assumes a PFS that was not packaged to reside in LPA.

If the second pfsctl, PC#Restart PFS, is going to be used, the PFS must have left the Restart Option Byte (pfsi_restart) at its default value or reset it to RESTART_WTOR before exiting. In this case, the normal WTOR message is not issued when the PFS terminates, and the second pfsctl takes the place of the operator reply to restart the PFS. Alternatively, the second pfsctl does not have to be used if the PFS sets the Restart Option Byte to RESTART_AUTO.

The second pfsctl can also be used without the first if the PFS exits with the Restart Option Byte set to RESTART_PFSCTL(7). This suppresses the WTOR message and causes the LFS to wait for the second pfsctl before restarting the PFS.

## PC#RestartPFS X'8000000D'

PC#RestartPFS X'8000000D' restarts the PFS by reattaching the PFS_Init module.

- If no argument is passed, or if the argument value is not 1, the LFS waits for the PFS initialization to complete before returning to the caller.
- If a fullword argument value of 1 is passed, the LFS returns to the caller immediately after posting the internal thread that does the reattach. The caller and the PFS must coordinate between themselves for the restart. This is similar to a startup during IPL.

The Return_value is 0 if the PFS was found and was awaiting this restart. The Return_value is 1 if the PFS was found but was not waiting to be restarted. This would be a normal situation immediately after an IPL, or if the caller did not recycle the PFS. If the PFS is not found the call fails.

This call can be made before the PFS has finished terminating, in which case the LFS proceeds directly to the PFS restart when it does finally terminate.

If all copies of the PFS have been recycled and the PFS load module does not reside in the LPA, the first reattach of the load module brings a fresh copy into storage.

The PFS should run through a more or less normal PFS initialization sequence with respect to the LFS. The regular sequence of returning VFS and vnode operation vectors, posting the LFS ECB, and waiting for the PFS termination ECB must be followed.

On each restart of a PFS, the previously returned value of pfsi_pfsanchor is passed into the new instance of the PFS. The PFS may use a design in which this anchor points to persistent storage so that it can reuse or reclaim resources from a prior instance.

For Socket PFSs:

- After the PFS completes its reinitialization, the LFS reissues any vfs_network calls that were originally made to set up for the address family domains that this PFS supports.

- The master socket opens with the normal sequence of events.

For File System PFSs, prior active mounts are reissued.

The PFS does not have to remember anything from one instance to the next with respect to the LFS and the LFS/PFS interfaces.

## Termination considerations

Because no "normal" termination is defined for a physical file system, there is no operator command or other interface supplied by z/OS UNIX to terminate an individual PFS. A PFS can define its own interface for this, although it cannot use the operator STOP or MODIFY commands unless it is running outside the kernel.

Usually a PFS does not stop.

There is nothing to prevent a PFS from terminating, either normally in a manner defined by the PFS, or abnormally. A PFS that is running in an address space outside the kernel terminates if that address space is terminated. If the PFS_Init program task terminates for any reason before the LFS posts the termination ECB, the LFS takes the following actions:

1. All activity to this PFS is halted. Users receive EIO or EMVSERR errors for any reference to a file that is owned by this PFS.
2. Every file system that is mounted for this PFS is logically unmounted. The PFS's vfs_umount is not called, because all activity is halted; but otherwise the file system is unmounted as it would be for an UNMOUNT FORCE command.

   File systems that are owned by other PFSs that are mounted on directories that are owned by the terminating PFS are also unmounted. These PFSs receive vfs_umount force.
3. The PFS is restarted or not depending on the setting of the restart option byte. The address of this byte is passed to the PFS in the PFSI during initialization. Its value may be adjusted by the PFS any time before it terminates.
4. If the PFS was running in an address space outside the kernel, that address space may be stopped and restarted, depending on the setting of the restart option byte.

The restart options available are:

| | |
|---|---|
| RESTART_NONE | Do not restart. |
| RESTART_AUTO | Automatic restart. |
| RESTART_WTOR | Prompt the operator before restarting. |
| RESTART_RCNONE | Stop the address space and do not restart the PFS. |
| RESTART_RCAUTO | Stop the address space and automatically restart the address space and the PFS. |
| RESTART_RCWTOR | Stop the address space and prompt the operator before restarting the address space and the PFS. |

The default restart option is RESTART_WTOR.

**Notes:**

1. If the PFS is restarted, file systems that were mounted at the time of failure are not automatically remounted, and network statements are not reprocessed. Socket file systems should specify that the PFS is not to be restarted, because NETWORK statements cannot be issued dynamically.

2. If the PFS requests that the colony address space in which it runs be stopped, the ASID for that address spaced is marked unusable.

## Cross-memory considerations

Because all of the VFS and vnode operations can be called in cross-memory mode, a PFS that must invoke MVS functions that cannot run in this mode must attach a worker task, or tasks, to accomplish these functions. A worker task is a subtask that performs non-cross memory work for PFS operations.

See “Using daemon tasks within a PFS” on page 41 for information about some services that make this task easier.

Although the PFS_Init task can be used as a worker task, if this task terminates, the PFS also terminates.

## Considerations for writing a PFS in C

A PFS can be written in System Programmer’s C. The BPXYPFSI and BPXYVFSI headers define the structures and parameters that are needed for PFSs that are written in C. A PFS that is written in C can avoid the cost of establishing a C environment each time it is invoked for a vnode or VFS function, by requesting that the LFS invoke the PFS with pre-initialized C environments. The PFS requests this at initialization by setting the pfsi_cpfs flag in the PFSI.

The PFS must not do anything that would sever addressability to the stack.

Because the PFS is running in a cross-memory environment, Language Environment® and C/C++ run-time library functions are not available. A PFS that needs to invoke these functions must attach a worker task, or tasks, to accomplish these functions.

See “Using daemon tasks within a PFS” on page 41 for information about services that make creating these worker tasks easier.

Some assembler services that may be useful are provided in Appendix E, “Assembler and C-language facilities for writing a PFS in C,” on page 553. In particular, BPXFASM must be assembled and link-edited with the PFS modules, to provide the correct @@XGET/@@XFREE routines for their C environment.

## Security responsibilities and considerations

z/OS UNIX maintains system security by verifying user identities and file access control information. A PFS is primarily concerned with file access control.

For those functions where POSIX .1 (IEEE Standard 1003.1-1990) specifies that “appropriate privilege” is required, the PFS refers to a bit that is set by the LFS to determine whether the function has appropriate privileges. For more information, see “Appropriate Privileges” in the POSIX standards.

Access control checks are based on information that is stored with each individual file, and are generally carried out on the system where the data resides.

Access control is integrated with the SAF interface to call RACF, or whichever security product is used at a particular installation.

The basic flow of file security is as follows:

1. Security information, such as the owner's UID-GID and the permission bits for a file, is kept in a 64-byte area called the file security packet (FSP), which is mapped by IRRPIFSP. The FSP is the security-related section of a file's attributes.
2. The FSP is created by a SAF call from the PFS when a file is created. Some of the information is taken from the current security environment, and some of it is passed as parameters.
3. The PFS stores the FSP with the attributes of the file.
4. When an access check is to be done, the PFS calls SAF with the type of check that is being requested, the audit_structure from the current call, and the file's FSP. SAF passes these to the security product, which extracts user information from the current security environment and compares it against the access control that is stored within the FSP. The audit_structure is used primarily for any auditing that may be necessary.

   There are many access and privilege checks defined by the POSIX standards. The detailed description of each vnode operation in Chapter 3 discusses the access checks that are expected.
5. When a file's access control information is changed, such as by **chmod()**, the PFS calls SAF with the type of change, the new values, the audit_structure from the current call, and the file's current FSP. A new version of the FSP is returned to the PFS, which then replaces the file's old FSP with the new one.
6. When a file is deleted, the PFS discards the FSP.

In the flow described above, the PFS provides some private space within the file attributes for the security product's use, ensures common access checking across all PFSs, allows for the installation of different security products, and lets the security product perform auditing or other non-POSIX processing.

The PFS is ultimately responsible for the following access checks:

- If the PFS controls the storage of its own files, it follows the flow outlined above to create, maintain, and use security information.
- If the PFS is a client getting its data from some remote repository, it sends the request to the remote system, where the access checks are performed using the osi_getcred service.
- If access is not controlled for the type of data that is supported by a particular PFS, the PFS may choose to skip these security procedures.

Some events that occur in the LFS are audited for security purposes by the vn_audit operation. For example, because relative pathnames may be audited during an access check, it is important to audit the working directory so that a full pathname can be constructed if necessary. When a user calls **chdir()** or **fchdir()**, the LFS invokes vn_audit to record the new working directory. **chroot()**, which changes the current root, is another call that causes an audit record to be created.

Refer to *z/OS Security Server RACF Callable Services* for more information about these interfaces.

"PFS support for multilevel security" on page 64 discusses PFS responsibilities and considerations for multilevel security.

## Running a PFS in a colony address space

By default, PFSs are initialized in the kernel address space. An installation may choose to run a PFS in a separate *colony address space* by specifying an ASNAME parameter on its FILESYSTYPE statement. You may want to have a PFS run in a colony address space if:

- The PFS is constrained by kernel address space resources, such as:
  - Storage
  - Data set allocations
  - Lock contention
- The PFS needs to request callable services itself, in order to:
  - Use sockets
  - Make remote procedure calls
  - Obtain POSIX file I/O

When a PFS runs in a colony address space, an extra address space is created, and each PFS operation has a slightly longer path length.

Any PFS can run in a colony address space unchanged. PFSs that are running in colony address spaces can use the osi_thread service, which is not available to PFSs that are running in the kernel address space. Any PFS that uses this service must document to its users that the PFS must be initialized in a colony address space. See “Using daemon tasks within a PFS” on page 41 for more information about the osi_thread service.

The writer of a PFS cannot assume that the PFS will run in the kernel, nor that it will run under the task that calls it.

## Overview of the PFS interface

The PFS interface is a set of protocols and calling interfaces between the logical file system (LFS) and the PFSs that are installed on z/OS UNIX. PFSs mount and unmount file systems and perform other file operations.

This section describes the services provided by the PFS routines that are called by the LFS. The services are described in terms of the requirements the PFS must meet and the expectations of the LFS. Also included are descriptions of the design that are intended to clarify the implementation of a physical file system on z/OS UNIX.

There are two types of PFSs, those that manage files and those that manage sockets:

1. File management PFSs deal with objects that have pathnames and that generally follow the semantics of POSIX files.
2. Socket PFSs deal with objects that are created by the **socket()** and **accept()** functions and that follow socket semantics.

As described in Chapter 1, the LFS is called by POSIX programs, non-POSIX z/OS UNIX programs, and VFS servers. In this document, “the caller” refers to the LFS or any of the programs that call the LFS. When the LFS is mentioned specifically, it is usually to clarify a point of the design.

This interface is a modification of the architecture that is outlined by S. R. Kleiman in the paper “Vnodes: An Architecture for Multiple File System Types in Sun UNIX”, which was published in *Proceedings: Summer Usenix Technical Conference & Exhibition* (June 1986).

**Porting note**

Some operations that are found on some UNIX systems are not called by the z/OS UNIX logical file system, and are not shown in the list in Table 1. Table 1 includes some functions that are unique to the logical file system.

## Operations summary

The following PFS operations are grouped by category and by applicability to file or socket PFSs.

*Table 1. PFS operations by PFS type and category*

| | | |
|---|---|---|
| File PFS - File System Services | VFS_MOUNT | Mount a file system |
| | VFS_UMOUNT | Unmount a file system |
| | VFS_SYNC | Synchronize a file system (synchronize all files) |
| | VFS_STATFS | Get general file system attributes |
| | VFS_VGET | Get a vnode from a file ID (FID) |
| File PFS - Directory Services | VN_LOOKUP | Look up a filename in a directory |
| | VN_READDIR | Read a directory |
| | VN_CREATE | Create a regular, FIFO, or character special file |
| | VN_MKDIR | Create a directory |
| | VN_SYMLINK | Create a symbolic or external link |
| | VN_LINK | Create a hard link to a file |
| | VN_RMDIR | Remove a directory |
| | VN_REMOVE | Remove a file |
| | VN_RENAME | Rename a file or directory |
| File PFS - File Services | VN_OPEN | Open a file |
| | VN_CLOSE | Close a file |
| | VN_READLINK | Read a symbolic link file or external link file |
| | VN_ACCESS | Perform access check |
| | VN_TRUNC | Truncate a file |
| | VN_FSYNC | Synchronize a file (save data to disk) |

*Table 1. PFS operations by PFS type and category (continued)*

| | | |
|---|---|---|
| Any PFS - File Services | VN_RDWR | Read or write |
| | VN_READWRITEV | Read or write with multiple buffers |
| | VN_GETATTR | Get attributes for a file |
| | VN_SETATTR | Set attributes of a file |
| | VN_IOCTL | Control I/O |
| | VN_AUDIT | Perform security auditing |
| | VN_SELECT | Select on a vnode |
| | VN_INACTIVE | Inactivate a vnode-inode |
| | VN_PATHCONF | Return configurable limits |
| | VN_RECOVERY | Recover from an abend for an operation in progress |
| | VFS_RECOVERY | Recover from an EOM condition for an operation in progress |
| | VFS_PFSCTL | PFS Control |
| | VFS_BATSEL | Select on a set of files/sockets |
| Sockets PFS - Address Family, or Domain, Services | VFS_NETWORK | Activate a domain |
| | VFS_SOCKET | Create socket or socketpair in a domain |
| | VFS_GETHOST | Get host ID or name |
| Sockets PFS - Socket Services | VN_ACCEPT | Accept a connection request |
| | VN_BIND | Bind a socket |
| | VN_CONNECT | Establish a connection |
| | VN_GETNAME | Get the name of the peer or socket |
| | VN_SOCKOPT | Get or set socket options |
| | VN_LISTEN | Get ready to accept connection requests |
| | VN_SNDRCV | Send or receive |
| | VN_SNDTORCVFM | Send to or receive from |
| | VN_SRMSG | Send a message or receive a message |
| | VN_SETPEER | Set a peer |
| | VN_SHUTDOWN | Shut down a socket |

The VFS-vnode vector tables returned by the PFS after its initialization contain either the direct or indirect addresses (depending on the value of the pfsi_modind flag in the PFSI) of the routines that implement the operations in the preceding list.

## LFS/PFS control block structure

In the LFS/PFS model that is used in z/OS UNIX, each active file system object is represented in the LFS and PFS by its own control blocks or structures. These are called the *vnode* and *inode*, respectively. There is a one-to-one relationship between the LFS's vnode and the PFS's inode. They effectively point to each other across the interface, although neither ever directly refers to the other's fields.

**Porting note**

Such terms as “build the inode”, as used in this document, mean “construct the in-storage representation of a file”. This does not imply anything about the file representation as it is stored on disk.

There is only one vnode-inode pair for each data object in the system, no matter how many links there are to the object (for file objects), or how many users may be accessing the object. Users who access a vnode through the LFS must be accessing the same data object through the PFS.

**Token_structure:**

A difference between the z/OS UNIX PFS interface and other implementations is that the vnode is not directly addressable by the PFS during a vnode operation. A Token_structure is presented on all calls as a vnode surrogate.

The Token_structure contains the following 8-byte PFS tokens:

- **Initialization token**, returned from the PFS_Init routine during PFS activation. This token usually contains the address of the PFS anchor block.
- **Mount token**, returned from the vfs_mount or vfs_network operation for the file system that is related to the current call. This token usually contains the address of the PFS mount block.
- **File token**, originally passed by the PFS to osi_getvnode when the file’s vnode-inode pair was created. This token usually contains the address of the PFS file block—that is, the inode.

For a vnode operation, Token_structure contains all three tokens; for a VFS operation, it contains only the initialization and mount tokens.

See the TOKSTR typedef in Appendix D for the mapping of Token_structure.

**Porting note**

The file token within Token_structure is a copy of the “private data” area in the vnode. If a PFS expects a vnode structure as an input parameter, but does not refer to any vnode fields other than the PFS’s private data pointer, the subfields within the program’s vnode structure can be rearranged so that the pointer’s offset matches that used in Token_structure. In this way, the PFS code that refers to this field will pick up the correct value when it is recompiled, and does not have to be changed.

Token_structure is transient; it lives only for the duration of a single call.

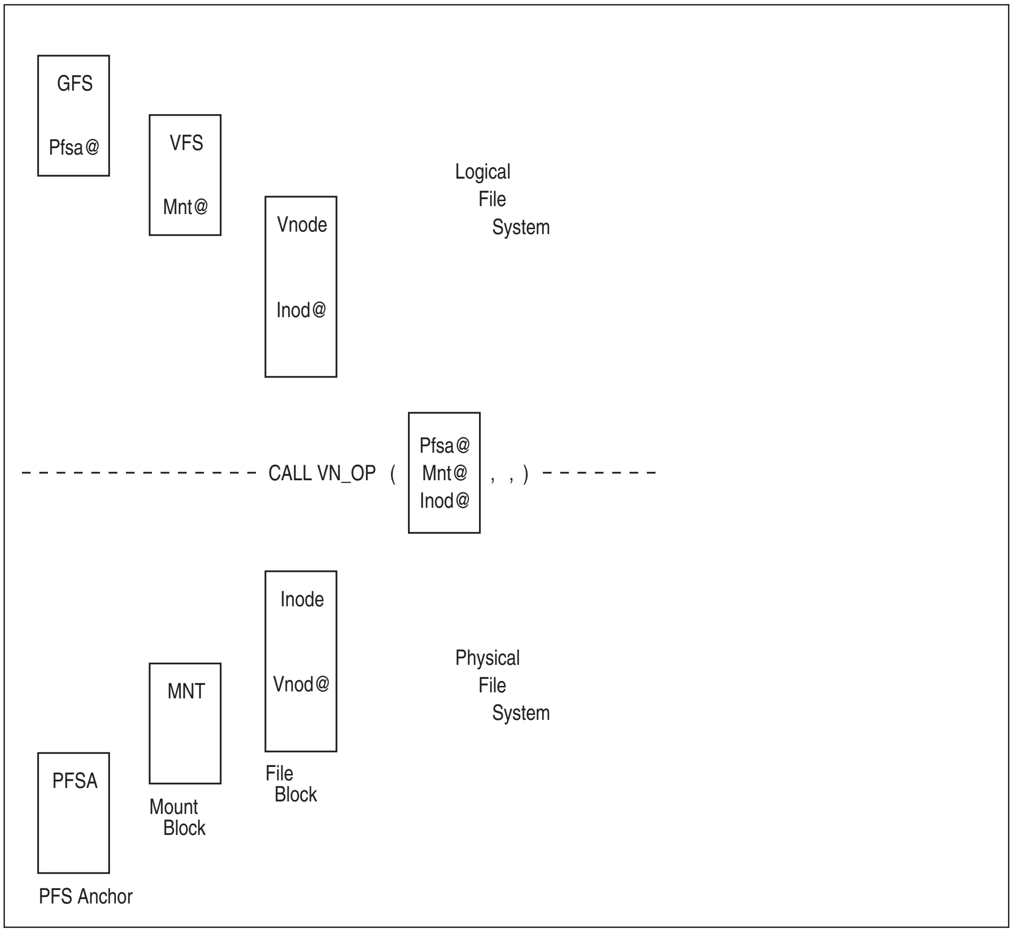


*Figure 3. The LFS/PFS control block structure*

The control block relationships described so far are illustrated in Figure 3. Reading from left to right, in the order they are created:

- The GFS-PFS_anchor pair is created at PFS initialization time and exists as long as the PFS does. Pfsa@ represents the PFS token saved by the LFS.
- The VFS-MNT pair is created during a file system mount or socket network activation, and exists until the file system is unmounted, or forever, respectively. Mnt@ represents the PFS token saved by the LFS from that operation.
- The vnode-inode pair is created during lookup and creation operations, which are explained in "Creating, referring to, and inactivating file vnodes" on page 31 and "Creating, referring to, and closing socket vnodes" on page 44.

  Each of these control blocks contains the other's token for the file object. The vnode's Inod@ token is placed in Token_structure as input for a call to the PFS, and an inode's Vnod@ token is returned by the PFS from any call that has a vnode as output.
- Token_structure contains all three PFS tokens, and spans the LFS-PFS interface as the first parameter of each call.

## Sharing files

The LFS manages user access to the vnodes. For programs that use the **open()** or **socket()** function, the LFS allocates file descriptors and manages sharing between processes and threads within a process. For VFS server programs, the LFS allocates vnode tokens, which behave somewhat like file descriptors. All programs, of any type, share the same file hierarchy.

The PFS is not aware of who is using a file or how it is being shared. To the PFS, there is only a vnode-inode pairing, and all file references come through that structure. In effect, the PFS has only one user: the LFS.

The PFS does not generally maintain any state information that would associate a sequence of calls. Successive calls to the PFS may relate to different end users, so every call is self-contained and does not depend on any information saved by the PFS from a previous call.

Files become shared when different end users open the same file, and when additional references to descriptors are created through the **fork()** and **dup()** functions.

Because the LFS maintains reference counts in its structures, it knows how many references to a given vnode are active and how many threads are currently making a call to the PFS with each reference. The PFS does not, therefore, have to be aware of how many users are accessing a given vnode-inode pair. The LFS ensures that all activity has ended and that the vnode-inode pair is no longer in use before it invokes vn_inactive to disassociate the vnode and inode.

## LFS-PFS control block integrity

To preserve the vnode-inode relationship, the LFS guarantees the following:

- On every operation, the inode, represented by the PFS's token in Token_structure, has not been inactivated.
- When the PFS is called to break the relationship (via vn_inactive at the time that a vnode is being freed), the LFS ensures that there are no other operations in progress against this vnode and, by extension, against the inode.

  There are, in fact, no operations in progress against any file that is in the same mounted file system as the file that is being inactivated. This is so that no other operation may be attempting to find or recreate the inode while it is being deleted.
- After a vn_inactive, the PFS does not receive any additional vn_ calls for that inode until the PFS creates a new vnode-inode binding for this same object as a result of a vn_lookup or vfs_vget call.

## The OSI structure

The second parameter of every call from the LFS to the PFS is the address of the operating system interface (OSI) structure. This structure contains information that is used by the OSI_operations and MVS-specific information that needs to be passed between the LFS and the PFS. It is mapped by the OSI typedef in Appendix D. The fields are described as follows:

| Field | Description |
| --- | --- |
| | **Wait-post fields** |
| **token** | Wait-post token. Set by osi_wait when it is called to set up for a wait. This token is the input to osi_post when it is called to wake up the current thread. |
| **ecb** | Address of an event control block (ECB). Set by osi_wait when it is called to set up for a wait. This is the ECB that is used by osi_wait when it is called to suspend. A program that cannot call osi_post can use this ECB with an MVS cross-memory post to wake up the current thread. However, using the MVS cross-memory post for this ECB can result in a system integrity problem. |

**ascb**
Address of the address space control block (ASCB). Set by osi_wait when it is called to set up for a wait. This ASCB address is used, along with the ECB, for an MVS cross-memory post.

**SMF accounting fields**

**diribc**
Directory I/O block count that occurred on this operation.

**readibc**
Read I/O block count that occurred on this operation.

**writeibc**
Write I/O block count that occurred on this operation.

**bytesrd**
The number of bytes that were read on this operation.

**byteswr**
The number of bytes that were written on this operation.

**Miscellaneous fields**

**rtokptr**
Address of the recovery token area. The recovery token area is set and cleared by the PFS on each operation, to provide for abnormal end and end-of-memory recovery. Refer to "Recovery considerations" on page 24 for details.

**workarea**
Address of a work area for use by the PFS. This area can be used for the dynamic, or automatic, storage necessary to run the current operation. This can save the PFS the overhead of obtaining and freeing stack storage on every call. The workarea is on a doubleword boundary.

**workarealen**
Length of the workarea. The workarea length is 3KB. This allows 2KB for routines that call the SAF Chk_Owner_Two_Files routine or the osi_uiomove service, each of which requires that a 2-KB work area be passed. The other SAF security routines require a 1-KB work area.

**pid**
The current thread's process ID (PID). This is the input to osi_signal if it is called to send a signal to the current thread's process.

**pfsid**
A PFS identifier that is used with osi_sleep and osi_wakeup.

**attr**
Address of an output file attribute buffer. Whenever this field is nonzero, the PFS should build and return a standard attribute structure for the file operated on at the end of the current operation. This is the same attribute structure that would be returned by vn_getattr.

The buffer is preset with an attribute structure header that contains the available length of the buffer.

Because this buffer may be the same area as an input attribute structure, it should not be modified until the very end of the current operation.

If the PFS does not return the file's attributes when asked, the LFS invokes vn_getattr to get them. This results in poorer performance for files that are supported by this PFS.

**fsp**
Address of an output File Security Packet (FSP). Whenever this field is nonzero, the PFS should return an fsp structure for the file operated on. This is the same fsp structure that would be returned by vn_getattr.

If the PFS does not return the file's FSP when asked, the LFS builds one. For a description of the FSP, refer to "Security responsibilities and considerations" on page 12.

**remount** A flag that indicates that the current operation is running during a remount (that is, during UNMOUNT with the REMOUNT option).

| **NotSigReg** Indicates that the calling process is not registered for signals and so should not be sent any.

## Waiting and posting

OSI_Operations are provided to the PFS to wait for internal events and to post the waiting thread when the event occurs.

Three important reasons for using the OSI wait and post services rather than native MVS WAIT and POST are:

- The OSI services allow signals to interrupt a wait.
- Users are not left hanging if z/OS UNIX or the PFS is stopped.
- The PFS is protected from any system integrity exposures that might result from the cross-memory post operation.

There are two kinds of wait, distinguished by whether or not signals are enabled during the wait:

- **Not signal-enabled:** Used to wait for internal serialization or other activities that are independent of external forces likely to take a long time. These waits should generally not be used with human interactions. Examples are: waiting for data to be read from disk, or waiting for an available output buffer from a pool that is shared by all users.
- **Signal-enabled:** Usually correspond to the blocking situations that are defined by POSIX, and often involve waiting for an end user to do something. Examples are: waiting for data to be read or written by another independent program, such as a socket session, or reading input from a terminal.

  Signals should be enabled when the end user may need to break out of an indefinite wait.

When a signal-enabled wait is entered, all serialization that was obtained by the LFS is dropped before the wait and re-obtained after it. This means that other operations may intrude on an otherwise exclusive operation. The PFS must take this into account if it uses signal-enabled waits. This does not mean that two exclusive operations will actually be running in the PFS for the same vnode-inode at the same time, but that a second operation may run while the first is blocked. When the first is resumed there may have been state changes made by the second. For writes on stream sockets, the default socket option of exclusive write will prevent the dropping of LFS serialization during single-enabled waits.

The WAITX option also allows LFS serialization to be dropped around the wait, independent of whether signals are enabled. See the next section for details on LFS serialization.

As a consequence of dropping LFS serialization, it is possible for a file system to be unmounted, with the IMMEDIATE or FORCE operands, while a task is waiting. If this happens, the wait service returns with an OSI_UNMOUNTED return code when it is posted, and the PFS must cancel the rest of the operation and return to the LFS with some care. Because it is expected that vfs_umount will have cleaned up all file-system-related resources, the current operation may have to avoid references to internal file system structures that are freed by vfs_umount.

Waits that are signal-enabled or that request the LFS to drop its serialization cannot be used on some vnode and VFS operations. The implementation notes for those operations state this.

The OSI sleep and wakeup functions are similar to wait and post, with these advantages:

- Osi_sleep
  – Does not require a separate setup call
  – Associates a Resource_id and Pfs_id with the sleeping thread
- Osi_wakeup
  – Wakes up all threads that match Resource_id and Pfs_id

**Implementation details:** The PFS implementation for waiting and posting involves the steps described here. There are two threads involved: the waiting thread and the posting thread.

1. The waiting thread is running on behalf of some VFS or vnode operation when it must wait for an event to occur. It calls osi_wait to set up for the wait, performs internal coordination to schedule the eventual wakeup, and calls osi_wait again to actually suspend the thread.
2. The posting thread may be an independent PFS task, or it may be running on behalf of some other user's VFS or vnode operation. It determines that a thread is waiting for the resource it is dealing with, and calls osi_post to wake that thread up.
3. When the waiting thread wakes up, it checks the return code from osi_wait and reacts accordingly.

| Waiting Thread | Posting Thread |
|---|---|
| • Determine that a wait is necessary.<br>`osi_wait(OSI_SETUPSIG, OSI, RC)`<br>• Create an internal wait structure that is used by the posting thread to recognize that the waiting thread is waiting.<br>• Save the osi_token in this structure.<br>• Chain the wait structure where the posting thread will find it.<br>`osi_wait(OSI_SUSPEND, OSI, RC)` | (None) |
| (None) | When an event occurs, scan the wait structures to see if anyone is waiting for this event. Unchain and free the wait structure.<br>`osi_post(saved_token, RC)`<br>If the return code is not zero, the waiting thread did not get this post and you may need to go on to the next waiting thread. |

| Waiting Thread | Posting Thread |
| --- | --- |
| Select on return code:<br>When (OSI_POSTED): proceed with what you were going to do.<br>When (OSI_SIGNALRCV): a signal has arrived (when using SETUPSIG rather than SETUP). Back out of this operation and return EINTR.<br>Otherwise: an abnormal end or unexpected error occurred. Back out of this operation and return EMVSERR.<br>End | (None) |

**Notes:**

1. This example assumes that the PFS has its own serialization around the chaining and unchaining of the wait structure.
2. A variation of the steps in this table would be to unchain and free the wait structure on the waiting thread. In this case, the posting thread marks the structure as "posted" so that another event occurrence cannot result in the same structure's being used again. Recovery is more complicated with this approach, though.
3. One also has to consider abnormal ends while waiting—for instance, the user might be canceled. In that case, control does not return to the code after the osi_wait. If the PFS supports vn_recovery, or has an ESTAE or FRR active, it gets control there and the situation can be handled as when a signal is received.
4. For abnormal ends and any return code other than OSI_POSTED, additional serialization between the waiting thread and the posting thread is necessary. In these cases the waiting thread is ending before, or even while, the posting thread is trying to wake it up.

   This is why it is important to save a copy of the osi_token from the waiting thread's OSI, rather than just the address of the waiting thread's OSI. The waiting thread's OSI storage could be gone by the time the posting thread tries to refer to it.
5. Another consideration is user address space end-of-memory, which abnormally terminates the waiting thread without activating any ESTAE or FRR. In this case, the LFS uses the OSI recovery token to invoke vfs_recovery, which gives the PFS a chance to clean up.

## LFS-PFS control block serialization

The LFS serializes use of the vnode-inode pair for each vnode operation. Writing of file data is done under an exclusive latch. Reading of file data is also done under an exclusive latch, unless shared read support has been indicated by the PFS for the file, and the read is via vn_rdwr or vn_readwritev. Shared read can be indicated in the OSI by the PFS upon return from vn_open, vn_close, vn_rdwr, vn_readwritev, vn_setattr, and vn_trunc.

Other read operations, such as vn_readdir, are done under a shared latch.

In particular, to optimize the performance of pathname resolution, only a shared latch is held on the directory that is involved in a vn_lookup operation.

**Recommendation:** Read operations that are done under a shared latch may require the PFS to update some structures; for example, to mark the access time of a file for update. The PFS is responsible for any additional serialization that is required to maintain integrity of its structures when functions are called with a shared or an exclusive latch. Often the compare and swap instruction is sufficient for this additional serialization. In order to avoid contention problems, the cross-memory local lock (CML) should not be used.

For the operations that refer to more than one vnode (vn_remove, vn_rmdir, vn_link, and vn_rename), exclusive latches are held on all the vnodes that are involved in the operation. This includes vnodes that are not explicitly passed on the interface, such as the file that is being unlinked on vn_remove.

When the PFS enters a signal-enabled wait, as described in the previous section, or when the WAITX option has been used to drop serialization around the wait, all vnode and file system latches are released before the wait and re-obtained after it. This means that other operations may be invoked from another thread for a given vnode during an exclusive operation that enters a signal-enabled wait, although there would not be two operations running at the same time, because the blocked thread re-obtains exclusive access when it wakes up.

**Note:** While any operation is active, the PFS never receives a vn_inactive call for that vnode, even if the latches are released. In cases of vn_open or vn_close processing, the LFS does not allow a close against the last active file descriptor while another thread has any operation in progress against it.

Refer to the individual operations for the level of serialization that is provided for each call.

The serialization that is provided can be changed by the PFS when the osi_getvnode service is called to create a vnode. The PFS can specify that no LFS latching be performed. If no LFS latching is specified, all discussions in this chapter about latches held on vnodes do not apply. Other LFS latches are unaffected; sigwait and waitx should still be used to drop other latches, where necessary.

## Recovery considerations

There are several recovery situations that must be handled by the PFS.

### PFS task or address space termination

As discussed in "Termination considerations" on page 11, if the PFS_Init task terminates for any reason, the LFS terminates the PFS and restarts it based on the current setting of the restart option byte. If the PFS is started in a colony address space and that address space terminates, the PFS_Init task is also terminated by MVS.

### User process and thread termination

Two possible situations are discussed here: when the process or thread is between calls to the PFS, and when it is actually running in the PFS code during a PFS interface operation.

In general, when a user process terminates normally or abnormally, the LFS closes all active file descriptors. There is nothing special about these close operations. The PFS receives a normal vn_close if all file descriptors for an open file reference happen to be closed. If forked children have not closed their inherited file descriptors, the PFS does not receive a vn_close and may never know that the user process terminated.

Individual user requests are run on dubbed tasks, but POSIX semantics assign file resources to the process. Consequently, if a user task terminates between calls to the PFS, and its process does not also terminate, the PFS is not notified.

When a VFS server address space terminates, all of its vnode tokens are released and files that were opened for the server are closed. If a vnode's reference count goes to zero, that vnode is inactivated. If this happens to remove all references to a vnode, that vnode is inactivated after a delay interval. The PFS does not receive any special notification.

## PFS abnormal ends

If the user address space or task terminates while actually running in the PFS code for a PFS interface operation, or if the PFS code itself fails, an MVS abnormal end is generated for each affected task. The MVS system then usually runs the FRR and ESTAE recovery exits.

- If the PFS does not have recovery established, the vn_recovery operation is available to allow the PFS to run its recovery processing as an exit from the LFS's ESTAE. See the description of vn_recovery and vfs_recovery that follows this list.
- If the PFS needs its own special recovery, it must establish an FRR or ESTAE on each entry from the LFS.
- If task-level recovery is bypassed by MVS, the end-of-memory (EOM) resource manager established by z/OS UNIX is run. It ensures that the PFS has a last chance to clean up by calling vfs_recovery. See the next section on vn_recovery and vfs_recovery.

**vn_recovery and vfs_recovery** are called to permit a PFS to recover resources when a user request ends abnormally, or when the user's address space enters EOM processing while a request to that PFS is active. This works as follows:

1. On every VFS and vnode operation, the LFS makes an 8-byte recovery area available to the PFS. This field is in the PFS's primary address space, not in the user's address space. Its address is in the OSI.
2. The PFS should set this field soon after entry, or when it has resources that need protection. The field is used for recovery information, or for the address of a recovery structure that is not in the user's address space.
3. The PFS clears the field on exit. The LFS also clears the field as soon as the PFS returns, as it has meaning only during a call, and presumably the area it points to is no longer valid. The PFS should clear the field so that it cannot be invoked with bad data if the user is canceled after the PFS has returned, but before the LFS can zero out the field.
4. If an abnormal end occurs and the LFS ESTAE routine finds this area to be nonzero, the area is passed to the PFS with a call to vn_recovery and cleared after this call.

   See "vn_recovery — Recover resources after an abend" on page 190 for more details.
5. If the EOM resource manager for a user address space finds this area to be nonzero, the area is passed to the PFS with a call to vfs_recovery. This can happen only for an abnormal end that bypasses normal ESTAE processing, or when an address space is canceled during ESTAE processing.

   See "vfs_recovery — Recover resources at end-of-memory" on page 94 for more details.
6. The PFS uses the information that is stored in the area during vn_recovery or vfs_recovery to clean up whatever was in progress at the time of the interruption.

The PFS can establish its own MVS dynamic resource managers if it must perform special recovery for a user or z/OS UNIX task or address space termination. This is not recommended, however, because severe performance degradation occurs if these resource managers have to be set up and removed on every operation.

## Terminating a PFS's associated separate address space

If a PFS communicates with a separate address space, that is, one unknown to z/OS UNIX, and waits for replies from that address space, users could be left waiting forever if that address space abnormally terminates while it has outstanding responsibilities to post user threads. Usually, the PFS has to remember all users that are waiting in this situation and post them from a recovery resource manager of the separate address space. This can involve extra serialization and overhead during mainline operations.

If, however, the separate address space registers with the v_reg() function, specifying the PFS that is dependent on it, and uses osi_wait and osi_post, the system remembers this information in a task-related area that does not require additional serialization or overhead during mainline operations. When the separate address space terminates, the system scans through all users looking for those in a potential wait for this address space and posts them. Thus the extra overhead is incurred only when the separate address space terminates.

## Dumping LFS data

Information that can be used by the PFS to add LFS data areas to dumps taken by the PFS is passed at initialization. Pfsi_dumpptr contains the address of an array of elements, mapped by BPXYFDUM, shown pictorially in Figure 4. These may be used to construct entries in a LISTD-type list passed to SDUMPX.

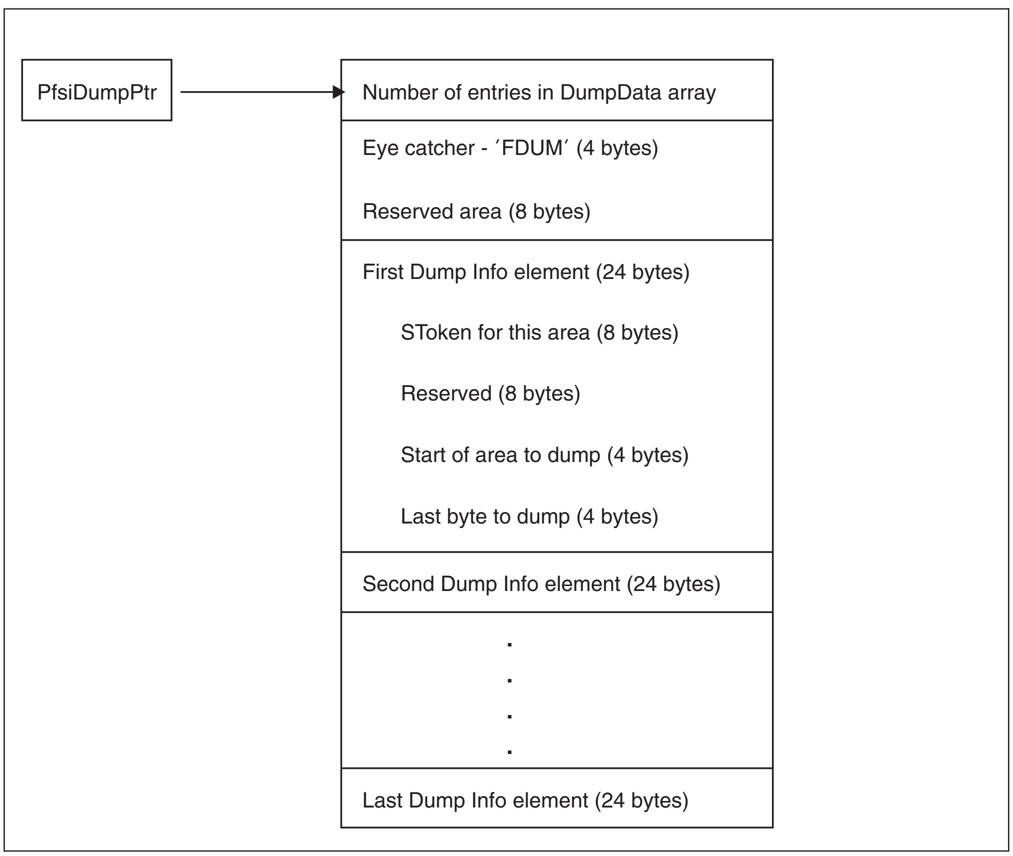


*Figure 4. Format of BPXYFDUM*

## PFS interface: File PFS protocols

## Mounting file systems

Mountable file systems are subsets of the file hierarchy that are added and deleted by mount and unmount. Each has its own root and hierarchical directory structure. One such file system serves as the root of the whole file hierarchy, and mounts are done upon the directories of other mounted file systems.

A mount may be issued from the BPXPRMxx parmlib member that is used with the start of z/OS UNIX, by a user through ISHELL, by the TSO/E MOUNT command, by automount, or by a program using the **mount()** function. The latter function is restricted to users with appropriate privileges.

Here is the syntax of a MOUNT statement, showing the parameters that are important to this discussion:

```
MOUNT FILESYSTEM(file_system_name) or DDNAME(ddname)
      TYPE(file_system_type)
      MOUNTPOINT(pathname)
      MODE(READ | RDWR)
      PARM(parameter_string)
      SETUID | NOSETUID
```

where:

- **FILESYSTEM** specifies a 1-to-44-character name, blank padded, by which this file system is to be known. It must be unique among previously mounted file systems. This is also used by some PFSs as an MVS data set name.
- **DDNAME** specifies the ddname on an ALLOCATE that is issued from the OMVS cataloged procedure. This is an alternative to the FILESYSTEM parameter for mounts that are issued from the parmlib member only. The real data set name becomes the mounted file system's name.
- **TYPE** identifies the PFS that supports this mounted file system. This operand must match the TYPE operand used on the FILESYSTYPE statement that defined the PFS.
- **MOUNTPOINT** specifies the pathname of the mount point directory within the file hierarchy where this file system is to be mounted. This item is passed to the PFS, but only for informational purposes.
- **MODE** specifies the type of access that the issuer of MOUNT has to this file system. READ is specified for read-only access, and RDWR is specified for read/write access.

  The LFS enforces this parameter to prohibit operations such as writing and creating files. The PFS must ensure that it does not update access times for read operations, or otherwise change file systems that are mounted read-only.
- **PARM** specifies a PFS-defined parameter text string. It may contain any value and be up to 1024 bytes long. The meaning of this text string is defined by the individual PFS, and the text is passed to the PFS for it to interpret and process.
- **SETUID | NOSETUID** specifies whether the SETUID and SETGID mode bits on executables in this file system are to be respected. This is enforced by z/OS UNIX; the information is passed to the PFS for informational purposes only.

See the MOUNT command description in *z/OS UNIX System Services Command Reference* for more information about the MOUNT command.

The parameters that are described above are passed to the PFS on the vfs_mount operation. The FILESYSTEM or PARM values are used by the PFS to identify the file system object that is being mounted.

During vfs_mount the PFS is expected to:

1. Ready the file system for all later processing.
2. Save the device number that has been assigned to this file system so that it can be output on vn_getattr for any file within this file system. This number corresponds to the st_dev value of POSIX.
3. Set output fields, as appropriate, in the MTAB.
4. Create an inode that represents the root of the file system.
5. Call osi_getvnode to create a vnode. The returned vnode token is saved in the inode.
6. Return the vnode token of the root to the LFS.
7. Return an 8-byte token that will be saved by the LFS and used on all subsequent VFS and vnode operations for this file system. This token is typically the address of the PFS's mount block. Its use is optional.

**Porting note**

This differs from some implementations in that vfs_root is not used to extract the vnode of the root of a just-mounted file system.

The root vnode is never explicitly inactivated. If this file system is unmounted, the vfs_umount operation implies vn_inactive for the root vnode-inode pair.

The PFS cannot use a signal-enabled wait or WAITX during MOUNT.

The LFS does not permit two mounts on a single MVS image with the same file system name. If the PFS identifies its mounted objects through the PARM parameter or by some other means, the PFS must permit or reject attempts to mount the same object more than once. If the mounted file system is on DASD, DASD file sharing must be taken into account. If the file system object is on or is using a resource that is shared by multiple systems, the PFS is responsible for managing or denying shared access.

The ROOT statement defines the system root. It is valid only from the parmlib member, and it has the same parameters as MOUNT, except that a MOUNTPOINT is not specified.

## Asynchronous mounting

The PFS may choose to complete mounting the file system asynchronously. Because latches are held by the LFS during execution of vfs_mount, it is desirable to perform the mount asynchronously if it cannot be completed immediately (perhaps because of the need to communicate with another system).

Asynchronous mount processing follows this sequence:

1. The vfs_mount service is called by the LFS as part of the mount processing described in "Mounting file systems" on page 27.
   - If the PFS decides to complete the mount asynchronously, it must indicate this to the LFS with the AsynchMount flag in the MTAB before returning to the LFS.

- If the SynchOnly flag in the MTAB is set on, the mount must be completed synchronously. The PFS must either complete it synchronously or reject it, returning EINVAL.

2. When the PFS has completed its asynchronous processing, it calls osi_mountstatus to indicate to the LFS that the mount can now be completed.
3. The LFS then calls vfs_mount a second time, from within the OMVS address space. On the second call, AsynchMount in the MTAB is turned on so that the PFS can identify this as the second mount.

   The PFS completes the mount actions described above.

After the PFS returns to the LFS from the first call to vfs_mount, the LFS may call any vfs_ operation. In particular, the PFS must be prepared to process vfs_unmount and vfs_statfs. If the PFS can determine the file attributes on the first call, it can create and return the root vnode on that call. Otherwise, it defers this until the second call. If a vnode is returned on the first call and also on the second call, it must be the same vnode each time. If the mount operation fails during the asynchronous phase, the PFS calls osi_mountstatus and reports the failure on the second vfs_mount call.

**Serialization:** During each vfs_mount, the PFS has exclusive access to the file system that is being mounted, and no access is allowed until the second vfs_mount has completed.

# Resolving pathnames

## LFS processing

Pathname resolution starts from the user's root or working directory. The LFS looks up the first component of the pathname in that directory. This often yields another directory, and the LFS looks up the second component of the name in this new directory. The LFS looks up each successive component of the name in the directory that was returned from the previous lookup, until the end of the pathname is reached.

When the LFS encounters a directory that is a mount point, it switches to the root directory of the file system that was mounted there. The next lookup is done in the mounted file system's root directory, rather than in the directory that was returned from the previous lookup. This is called *crossing mount points*; it is because of these mount points that pathname resolution has to be done one component at a time.

## PFS processing

Resolving pathnames and identifying mount points is a function of the LFS. Except for the individual vn_lookup operations that are invoked, the PFS is not involved.

# Unmounting file systems

A user can issue an unmount through ISHELL, the TSO/E UNMOUNT command, automount, or a program that is written to use the **unmount()** function. This function is restricted to users with appropriate privileges.

Here is the syntax of the TSO/E UNMOUNT command, showing the parameters that are important to this discussion:

```
UNMOUNT FILESYSTEM(file_system_name)
        NORMAL | DRAIN | RESET | IMMEDIATE | FORCE | REMOUNT(RDWR | READ)
```

where:

- **FILESYSTEM** specifies the name that was used when the file system was mounted.
- **NORMAL | DRAIN | RESET | IMMEDIATE | FORCE | REMOUNT(RDWR | READ)** specifies the type of unmount to perform.

## LFS processing

- NORMAL. The LFS checks to make sure no user is using any of the files in the file system that is to be unmounted, and passes the request to the PFS via vfs_umount. If files in this file system are being accessed, the LFS rejects the unmount request.
- DRAIN. The LFS checks to make sure that no user is accessing any of the files in the file system that is to be unmounted, and passes the request to the PFS via vfs_umount. If files in this file system are being accessed, the LFS waits until all activity has ceased, and then passes the request to the PFS.
- RESET. The LFS cancels a previous unmount drain request. The file system goes back to the normal mounted state.
- IMMEDIATE. The LFS stops further user access to the file system that is being unmounted. Any attempt to access files in this file system receives an error return code. The LFS then passes the request to the PFS via vfs_umount.

  UNMOUNT with IMMEDIATE can be used to override a previous UNMOUNT DRAIN request for a file system.
- FORCE. The LFS stops further user access to the file system that is being unmounted. Any attempt to access files in this file system receives an error return code. The LFS passes the request to the PFS via vfs_umount.

  UNMOUNT with FORCE can be used to unmount a file system even if I/O errors are being received from the underlying device.

  An IMMEDIATE unmount request must be issued before a FORCE unmount can be requested.
- REMOUNT. The LFS handles this like an IMMEDIATE unmount followed by a mount. User access is suspended while the operations are in progress. vfs_vget is used to establish the vnode/inode bindings so that the remount is not disruptive to the users.

## PFS processing

1. The PFS processes requests for UNMOUNT with the NORMAL, IMMEDIATE, and FORCE options as follows:
   - NORMAL. Synchronizes all data buffers to disk (if appropriate for this PFS). This saves all data changes to files in the file system that is being unmounted. If an I/O error occurs during this activity, the unmount request fails.
   - IMMEDIATE. Synchronizes all data buffers to disk (if appropriate for this PFS). If an I/O error occurs during this activity, the unmount request fails.
   - FORCE. Synchronizes all data buffers to disk (if appropriate for this PFS). If an I/O error occurs during this activity, the unmount proceeds anyway and data is lost.

   The difference between NORMAL and IMMEDIATE is whether the PFS is likely to find itself with any active inodes other than the one belonging to the root. The difference between IMMEDIATE and FORCE is whether the PFS continues if it encounters an I/O error while trying to synchronize data during the unmount.
2. The PFS frees any inodes that are still active, including the root inode, which is never explicitly inactivated.
3. The PFS reverses the vfs_mount and returns the file system to unready status.

**Serialization:** The whole file system is serialized under an exclusive latch at the time vfs_umount is called. No other vnode or VFS operations are running, although some may be in the PFS in a blocked state. See "LFS-PFS control block serialization" on page 23 for more about serialization and blocking.

## Creating, referring to, and inactivating file vnodes

The PFS creates vnodes by calling osi_getvnode, which is one of the OSI services in the OSIT vector table that is passed to the PFS during its initialization. The output of osi_getvnode is actually an 8-byte vnode token, but for the purposes of this discussion the vnode and the vnode token are the same, and the term *vnode* is used for both.

The first vnode for a mounted file system is created during vfs_mount processing. At this time, the PFS must create a vnode-inode pair to represent the root of the mounted file system and return the vnode token of the root. The LFS never inactivates this first vnode; it is cleaned up as part of vfs_umount processing.

Subsequent vnodes within a mounted file system are created by calls to vn_lookup, vn_create, vn_mkdir, or vfs_vget. The first three of these routines are passed a previously obtained directory vnode, represented by a token structure, and the name of a file within that directory to find or create.

The vfs_vget operation also generates vnodes directly from the file identifier (FID) of a file within a given file system. See "Exporting files to a VFS server" on page 42.

During vn_lookup the PFS must:

1. Look up the filename in the directory. If the name is not found, vn_lookup fails.
2. Find or create an inode that represents the named file. This may involve reading the file's control information from a disk when the file has not been referred to for a while.
3. For a new inode or one without a vnode (depending on PFS design), call osi_getvnode to create a vnode. The PFS's file token is passed to osi_getvnode to be saved in the vnode, and the returned vnode token is saved by the PFS in the inode.
4. Return the vnode token from the inode that represents the named file in the specified directory. The file may itself be another directory.

The creation operations of vn_create and vn_mkdir follow a similar flow. See "Creating files" on page 32 for more information. They are also invoked with a directory vnode and a name, but in these cases the file itself is created if it does not exist. vn_lookup may create an inode, but it does not create the file.

The vnode is generally used in subsequent operations, such as vn_rdwr for a file or vn_lookup and vn_create for a directory. A directory vnode may become a mount point, the current root, or the working directory of POSIX processes. None of these references to the vnode involve any processing by the PFS.

Eventually the vnode falls out of use. After all opens have been closed and all other references to the vnode have been released, the LFS marks the vnode for inactivation. If the vnode is not referred to again for some time after it is marked for inactivation, the LFS invokes vn_inactive, or vfs_inactive if the PFS supports batch inactive and actually frees the vnode. The same functions are performed by vfs_inactive and vn_inactive; vfs _inactive requires only one call to the PFS to performs these functions for multiple vnodes.

During vn_inactive the PFS must:

1. Disassociate the inode from the vnode.
2. Perform any inode cleanup desired.

   If the inode's link count is zero, it must be deleted; otherwise it is just deactivated and can be reactivated with vn_lookup.

After the call to vn_inactive, or vfs_inactive for multiple vnodes, LFS frees the vnode, unless the PFS reports a problem via a bad return code from the operation.

**Porting note**

The PFS does not free the vnode. This is a change from some implementations.

In cases in which a file is repeatedly opened and closed by a single process, the deactivation delay helps to avoid the cost of reconstructing the vnode-inode relationship, and whatever other overhead is incurred by a PFS in reactivating a file. In these cases, file caching is done by the LFS and need not be done by the PFS.

**Serialization:** The vn_lookup service is called with a shared latch held on the directory being searched. The vn_inactive service is called with an exclusive latch on the whole file system that the object belongs to.

The serialization of vn_inactive ensures that no operations are running that could possibly find, or attempt to create, the inode that is being processed by vn_inactive. This is because an exclusive latch is held on the inode's file system during vn_inactive and the LFS does not allow links across file systems, therefore no parent directory of the object that is being inactivated can be referred to while the PFS is trying to inactivate the object.

The PFS must serialize the creation of its own inodes, to ensure that a single file does not have two or more inodes. This is because the same file object may be looked up or created by more than one process concurrently. The PFS must atomically create the vnode-inode pair and associate the inode with the file object, either through a global latch or with a Compare and Swap algorithm.

To help with a Compare and Swap algorithm, a Return an Unused Vnode option is provided on osi_getvnode so that the Compare and Swap loser can free the vnode it had acquired. The vnode obtained from osi_getvnode does not represent anything until the PFS returns it to the LFS from this or another concurrent operation. The instant that the PFS associates a vnode-inode pair with an object, any vn_lookup for the same object that is running on another process must find this same vnode-inode pair.

## Creating files

File hierarchy objects are created with the vn_create, vn_mkdir, and vn_symlink calls.

The interface for all these operations includes:

- The object's parent directory vnode, as a token structure
- The object's name, as a character string

- An ATTR structure

**Serialization:** An exclusive latch is held on the parent directory vnode.

### PFS processing

During these operations the PFS must:

1. Fail the operation if the object already exists—that is, if the name is already in the directory.
2. Otherwise, create the object and add an entry to the parent directory.

   A unique nonzero *inode number* that corresponds to the st_ino value of POSIX must be assigned to this object. This value only has to be unique within this file system and at this time. It may be reused after the object is deleted. For additional information about reusing file identifiers, see "Exporting files to a VFS server" on page 42.

   A directory object should be initialized by the PFS with the "." and ".." entries. For a root, ".." refers to itself, but for any other directory ".." refers to its parent directory. These entries are not strictly required by POSIX.
3. Store at least the file's type, major number, and minor number from the passed ATTR structure with the stored attributes of the file. Whenever osi_getvnode is called, the PFS must construct and pass an ATTR structure, as would be returned by vn_getattr, so that the vnode can be built properly.
4. Call SAF to create the FSP. The user credentials and ATTR mode bits from the interface and the FSP of the parent directory are passed to SAF, so that it can construct the FSP and do any auditing that is necessary. See "Security responsibilities and considerations" on page 12.
5. Store the FSP with the rest of the attributes of the file.
6. For vn_create and vn_mkdir, build an inode-vnode pair, as it would for a vn_lookup of this object, and return the corresponding vnode token.

The PFS is responsible for link counts, which must be initialized here. The *link count* of an object is the number of directory entries within the file system that point to the object. It is reported to a caller via vn_getattr, and changed by vn_link, vn_remove, vn_rmdir, and vn_rename.

Special consideration must be made for the "." and ".." entries when creating directories. "." implies that a directory's initial link count would be two. ".." implies that a directory's parent directory's link count has to be incremented when the child directory is created and decremented when it is deleted.

vn_link creates a new node in the file hierarchy, but it does not create a new object.

The LFS does not allow the creation of links (vn_link) to a directory.

## Deleting files

File hierarchy objects are deleted with the vn_remove, vn_rmdir, and vn_rename calls. The vn_rename function causes the deletion of the new_name file when it exists.

The interface for all these operations includes the object's:

- Parent directory vnode, as a token structure
- Name, as a character string
- PFS file token

**Serialization:** An exclusive latch is obtained for the parent directory vnode and the object's vnode. For vn_rename, an exclusive latch is held on both parent directories, the old object vnode, and the new object vnode, if it exists.

### PFS processing

During these operations the PFS must:

1. Call SAF's Check Access service to verify that the caller has write permission to the parent directory. If the sticky bit (S_ISVTX) is on in the parent directory's mode, the PFS must call SAF's Check2Owners service to verify that the caller is allowed to delete or rename the object.
2. Remove the directory entry for the named object, and update the Change and Modification times for the directory.
3. Decrement the link count in the object whose name was removed.

   If a directory is being removed, it must be empty except for the "`.`" and "`..`" entries. The parent's link count is also decremented to account for the "`..`" entry in the removed directory.
4. If the object's link count goes to zero, the object itself is deleted later during vn_inactive, but the deletion is recorded for audit purposes now.

   If the object is a regular file that is not open, the space used by its data must be released now. If a regular file is still open, its data is deleted on the last vn_close. This behavior is required by POSIX.

   A POSIX-conforming PFS should set the immeddel flag in the PFSI during initialization to let the LFS know that this requirement is in force. Otherwise, the LFS must issue vn_getattr and vn_trunc during **unlink()** and **close()** in order to check the link count and free regular file data.
5. While an inode's link count and open count both are zero, the PFS may reject subsequent operations, except for vn_readdir, which would return no entries, and vn_inactive.

## Opening and closing files and first references to files

POSIX programs read and write files or read directories within an open-close bracket, whereas VFS servers do this directly from the vnodes that they have looked up or created.

The LFS inserts a single open-close bracket around the operations that are issued by a VFS server against regular files. Operations that affect a file's attributes or read a directory may or may not be preceded by an open, and a PFS has to be prepared for either case. In particular, a file's size may be changed with the truncate() function, which results in a call to vn_setattr without a preceding vn_open.

The PFS must perform two main functions to support reading and writing, both of which tend to be done only once:

1. Physically prepare to do the I/O. This may involve getting buffers ready or using lower-layer protocols for a device or access method.
2. Perform access checking.

   Note that for performance reasons, the fewest number of access checks possible should be done when a particular end user accesses a particular file.

**Serialization:** Both vn_open and vn_close are invoked under an exclusive vnode latch.

The PFS is expected to do the following:

- During vn_open:
  1. Perform access checks. This must be done here for POSIX users.
  2. Prepare for I/O, if necessary.
  3. Increment an open counter in the inode for regular files.
- During reading or writing:

  Perform access checks, if the Check Access bit is on in the UIO.
- During vn_close:
  1. Perform any I/O that is necessary, instead of deferring it to the vn_inactive call. Examples include saving the contents of data buffers to disk and updating access times. This allows I/O to be charged back to the end user, whereas I/O that is done during vn_inactive is charged to z/OS UNIX.
  2. Decrement the inode's open counter for regular files. If this goes to zero and the file's link count is zero, the file's data blocks are deleted and their space is reclaimed before the return from vn_close.

     A PFS that reclaims space on the last vn_close of a deleted file should set the immeddel bit in the PFSI during initialization, for best performance. Otherwise, the LFS issues vn_trunc unnecessarily.
  3. Perform the minimum amount of other cleanup. It is better to defer cleanup to vn_inactive processing. Even if no one is still referring to a file, which would not be apparent to the PFS, performance is better if the PFS allows LFS file caching to reuse a closed file with minimal overhead.
- During vn_inactive, or vfs_inactive if the PFS supports batch inactive:

  Perform final cleanup for the file or directory inode. This operation runs on a z/OS UNIX system task with the containing file system locked, so the PFS should accomplish this cleanup as quickly as possible. Avoid waits and I/O during this cleanup processing.

If this process is followed, the access credentials of POSIX users are checked only during their **open()** call. A VFS server that maintains state information requests access checking for the first reference by a particular end user to a particular file, but not for subsequent references. A VFS server without this state knowledge must pay the price of access checks on every reference.

The LFS builds and manages the file descriptors that are used by POSIX programs.

The vn_open-vn_close pair has the following characteristics:
- There may be many vn_opens issued for the same file or directory, and any number may be outstanding at a given time.
- The LFS may share a single vn_open with many users, because of forking or VFS server usage. This sharing is not apparent, nor is it of concern, to the PFS.
- For any vn_open that is seen by the PFS, there is a corresponding vn_close. Because there may be many vn_opens active, getting a vn_close does not mean that the file is in any sense no longer in use. The PFS does not get any indication that a particular vn_close is the "last close", so it needs to maintain an "open counter" to control the deletion of data blocks for removed regular files.
- There is no "open token" in this protocol, such as the traditional MVS DCB structure or the POSIX file descriptor. The PFS does not know for which vn_open a particular read, write, or close operation is being performed.

## Reading from and writing to files

The PFS is responsible for actually moving data that is to be read or written, and for implementing the semantics that are required by the standards supported by z/OS UNIX.

See also "Opening and closing files and first references to files" on page 34.

vn_rdwr and vn_readwritev are UIO operations, which means that:

- The UIO structure is part of the interface.
- The UIO contains the address, ALET, storage key, and address space ID of the user's buffer or buffers. It has a read/write flag to distinguish direction. For reads, it contains the length of the user's buffer or buffers. For writes, it contains the number of bytes that are to be written.
- The UIO contains the process file size limit for the file. On a write or writev request it is the responsibility of the PFS to determine when this limit has been reached or exceeded. When a write or writev request is unable to write any data without exceeding the file size limit, the PFS must set the bit in the UIO that indicates that the limit was exceeded, and set the errno to EFBIG. The PFS must also be aware of one other special value for the file size limit: If both UIO.u_fssizelimithw and UIO.u_fssizelimitlw are equal to 0, there is no file size limit set for the process.
- It is the responsibility of the PFS to maintain system integrity while moving data between the address spaces. This means that the Move With Source Key and Move With Destination Key machine instructions or the osi_copyin, osi_copyout, and osi_uiomove services must be used.
- The caller maintains file positioning for the PFS, and the current file cursor is in the UIO for every operation. This indicates the position from which the read or write is to start.

  When the O_APPEND flag is set on in the open flags parameter for a write operation, the UIO cursor is ignored by the PFS. Writing begins at the end of the file, as it is known by the PFS at the time of the write.

  The UIO cursor may reflect the last read/write operation that was seen by the PFS; it may be from a different instance of vn_open; or it may have been changed through seek operations that were issued by the user and that are not seen by the PFS.

  The PFS modifies the UIO cursor to reflect the file position after the operation.

  The UIO cursor area is 8 bytes long, to support large files. It is the responsibility of the PFS to handle file offsets greater than $2^{31}$ or to reject them. The 8-byte cursor is a doubleword signed binary integer.

During vn_rdwr and vn_readwritev the PFS must:

1. Do access checking, if the UIO check-access bit is on.
2. Move the data. During vn_rdwr, if the UIO real-page bit is on, use the DATOFF services of MVS to move the data. The ability to refer to real pages is indicated by the PFS during its initialization. If this cannot be supported, the LFS supplies an intermediate virtual page buffer.
3. Synchronize the data, if the UIO sync-on-write bit is on, and turn on the sync-done bit to notify the LFS that it was done. Otherwise, the LFS issues vn_fsync explicitly and the whole operation takes a little longer.
4. Ensure that the operation does not write beyond the process file size limit. If the starting position is already at or beyond the limit, the PFS must set the

limit-exceeded bit in the UIO and return with EFBIG. This check is done in the PFS because of the O_APPEND case, in which it is much more efficient for the PFS to verify the starting position.

5. Return the number of bytes that were transferred.
6. Modify the UIO cursor to reflect the file position after the operation.

**Serialization:** The vn_rdwr and vn_readwritev services are invoked with an exclusive latch for both reads and writes. This is to help the PFS implement the POSIX semantics that require atomic operations and immediate visibility to all other processes.

## Reading directories

To optimize directory reading, vn_readdir is designed to return as many entries as possible on each call. The C run-time library deblocks the entries for POSIX programs, to provide the sequencing that they expect.

Like vn_rdwr and vn_readwritev, vn_readdir is a UIO operation, but the interpretation of the cursor is different. Cursor technique is described in the next section. See also “Opening and closing files and first references to files” on page 34.

**Serialization:** Because the LFS obtains a shared latch for the vn_readdir operation, there may be many users reading the same directory at the same time.

The vn_readdir output buffer is mapped by the DIRENT structure, and its format is defined as follows:

- The buffer contains a variable number of variable-length directory entries. Only full entries are placed in the buffer, up to the buffer size specified, and the number of entries is returned on the interface.
- Each directory entry that is returned in the buffer has the following format:
  1. 2-byte Entry_length. This length field includes itself.
  2. 2-byte Name_length. This is the length of the following Member_name subfield.
  3. Member_name. A character field of length Name_length. This name is not null-terminated.
  4. File-system-specific data. If Entry_length equals Name_length plus 4 bytes, this subfield is not present. Whenever this field is present, it must start with the file’s inode number, st_ino, in 4 bytes.

     To be XPG-conforming, the PFS must include the file’s inode number.

     This subfield is not part of POSIX, but it is passed through to all programs to use or ignore as they wish. A non-standards-conforming program may take advantage of additional information provided by a specific PFS that it knows about.
- The entries should be packed together. The length fields are not aligned on any particular boundary.

An example of an entry for the name *abc* and inode number X'1234' is X'000B 0003 818283 00001234'.

Many applications expect entries for “`.`” and “`..`” to be returned. This is not strictly required for standards conformance.

Successive calls to vn_readdir for a particular end user must proceed through the directory from the point at which the last one left off. A call does not have to account for activity that occurred “behind” its position in the directory, nor worry about items that may be deleted from “in front” of the current position before it was reached.

The PFS does not directly maintain positioning over successive calls to vn_readdir. The 8-byte UIO cursor is used to specify the positioning within the directory.

Not all directories are implemented as simple linear files that hold an array of name entries. Two continuation techniques may be used, and these must both be supported by a PFS. These techniques are:

- **Cursor technique.** The cursor that is returned by the PFS in the UIO contains PFS-specific information that locates the next directory entry. The caller is required to preserve the UIO cursor and the entire output buffer from the last vn_readdir, and present both of these on the next vn_readdir.

  The PFS may use the cursor as an offset into a simple linear directory file, ignoring the buffer; or it may use it as an offset into the output buffer of the last entry that was returned. The latter approach can be used by a PFS with a tree-structured directory, where the previous entry name is used as a key to search for the next entry. That is, the last returned name, a 1-to-255-byte-long text string, is really the cursor for the caller’s position in the directory. To ensure data integrity, you have to use the Move With Source Key instruction or osi_copyin for the entry header, and then again for the name length.

  The cursor technique is used by the [ for POSIX-conforming functions.
- **Index technique.** The index that is set in the UIO by the caller determines which entry to start reading from. To read through the directory, the caller starts at 1 and maintains the index by adding the number of entries returned to the previous index. The caller may jump around in the directory, and there is no requirement that the next index be related to the last vn_readdir.

  This technique views the directory as a one-based array, where the first entry has an index of 1, the second entry has an index of 2, and so on.

  The index technique is used by the Network File System and by the C/C++ run-time library for XPG-conforming functions.

The UIO contains both the cursor and index fields that are used with these continuation techniques. The interpretation of these two fields is summarized in the following table:

| Index | Cursor | Action |
|---|---|---|
| 0 | 0 | Start reading from the first entry. |
| 0 | M | Use the cursor value to resume reading. |
| N | 0 | Start reading from entry N. |
| N | M | Start reading from entry N. |
| **Note:** 0=zero; N and M are nonzero values. | | |

A nonzero index overrides the cursor. When both are zero or the index is 1, reading starts from the front of the directory.

The general flow for reading a directory is:

1. On the first vn_readdir of a sequence, both fields are zero and the PFS starts at the front of the directory. The normal cursor value of the PFS and the number of entries that were placed in the buffer are returned.
2. On the next vn_readdir, the caller specifies whether the cursor technique or index technique is being used to proceed through the directory. The PFS positions itself in the directory based on the technique used, reads more entries, and returns its normal updated cursor value and the number of entries that were placed in the buffer.

   The PFS must always return an updated cursor value, even if the index technique is being used. Some callers may switch between techniques, as the C/C++ run-time library does for the **seekdir()** function.
3. In most cases, the caller continues in this way until the directory is exhausted.
4. The application can reset the directory stream to the beginning, but this action is not passed through to the PFS. The next vn_readdir simply has both cursor and index values of zero. The application can also begin reading from any desired entry.

The Move With Destination Key machine instruction or the osi_copyout or osi_uiomove services must be used to write to the user's buffer.

The end of the directory stream is indicated by the PFS in two different ways:
- A Return_value of 0 entries is returned. This must be supported by the PFS for cases in which a vn_readdir is issued and the position is already at the end of the directory.
- A null name entry is returned in the output buffer. A null name entry has an Entry_length of 4 and a Name_length of 0—for example, X'00040000'.

  This would be the last entry in the buffer, when the directory end has been encountered on a call and there are at least 4 bytes left in the buffer.

  A PFS that supports this indicator helps the caller to run faster. A small directory may be read in only one operation, because the caller can detect that a second call is unnecessary.

**Note:** POSIX allows **open()** and **read()** from a directory, but it only specifies that these operations do not fail with an error. The PFS cannot tell whether a vn_open is from an **open()** or from an **opendir()**, but **read()** results in a vn_rdwr while **readdir()** results in a vn_readdir. The PFS is free to support vn_rdwr as a traditional UNIX system would, or to just return zero bytes on every operation. The *X/Open Portability Guide, Version 4, Issue 2* allows the EISDIR error to be returned for **read()**. The LFS ensures that only reading is allowed.

## Getting and setting attributes

The PFS is responsible for storing file attributes with its files. POSIX users can read these attributes with such functions as **stat()**, and set various attributes through such functions as **chmod()**. A VFS server does the same things with **v_getattr()** and **v_setattr()**.

All of this is passed through to the PFS when the LFS calls the vn_getattr or vn_setattr service with the ATTR structure (BPXYATTR). The ATTR structure is the file attribute interface between the LFS and the PFS. It contains all the fields of the POSIX STAT structure, plus z/OS UNIX extensions that the PFS may support if it can.

A file's attributes are logically split between the security-related and non-security-related attributes. The security-related attributes are kept in the file security packet, IRRPIFSP, or FSP for short. The FSP is stored with the attributes of the file by the PFS, but it is created and changed only through SAF-defined routines. The FSP contains the file's mode bits, UID, and GID; it may also contain other information that is defined by the security product.

The FSP is the file attribute interface between the PFS and SAF. Refer to "Security responsibilities and considerations" on page 12 and "Creating files" on page 32 for more details on SAF and the FSP.

**Serialization:** The vn_getattr service is invoked with a shared vnode latch, and the vn_setattr service with an exclusive latch.

vn_getattr and vn_setattr do not require vn_open, although the file may be open for read or write at the time of these calls. Reads and writes would not be in progress at the time of the get or set.

All times in the ATTR structure are specified in POSIX format, which is "Seconds Since the Epoch" (00:00:00 January 1, 1970, Coordinated Universal Time). The PFS may keep time values internally in any format, but they must be in POSIX format across the LFS-PFS interface.

The ATTR structure's header is initialized with the ATTR's length before any get or set call.

The vn_getattr protocol is as follows:

1. All ATTR fields that are supported by the PFS are returned.
2. To account for different release levels, the PFS should zero out the area and set fields it understands only up to the minimum of the input area's length (from the ATTR length subfield) and the PFS's native ATTR length (the one it was compiled with). The input area's ATTR length subfield should be updated to reflect the amount of data that is returned or zeroed out.

   A simple way to do this is to construct a local ATTR structure and copy this, truncating it if necessary, to the input ATTR.

The vn_setattr protocol is as follows:

1. More than one attribute may be changed on a single vn_setattr call, and each settable field in the ATTR structure is conditionally and individually set. Bit flags are set by the LFS in an ATTR flag area to indicate which fields from the ATTR structure are being set.
   - In general, if a change bit is on, the PFS updates the corresponding file attribute from the value that is passed in the corresponding ATTR field.
   - **Security fields**. For each security-related field, such as mode, owner, or audit, that is being changed, there is a corresponding SAF routine that the PFS calls to actually make the changes in the FSP. This allows the security product to do permission checks and security auditing, or other necessary security-related processing.
   - **Time fields**. Two bits are defined for each time field. The first bit indicates that a change is to be made, and the second bit indicates whether to use the corresponding ATTR time field's value, or if the current time of day is to be generated and stored by the PFS.

Non-security fields may still have access control defined for them. This means that SAF is called to see if the user has permission to make the change, but the PFS does the change.
2. The PFS should ensure that either all changes or no changes are permanently recorded for a single vn_setattr call.
3. To account for different release levels, the PFS must not refer to fields beyond the input ATTR's length, as specified in its length subfield.

**Note:** To optimize performance for VFS servers, several of the vnode operations, such as vn_lookup and vn_rdwr, pass an ATTR structure pointer in the OSI structure and expect an implicit vn_getattr to be performed at the end of the current operation. If the PFS cannot support this, the LFS calls vn_getattr after the operation in question. This flow has poorer performance when accessing files owned by this PFS.

## File tags

The file tag is a file attribute that identifies the character set of the text data within a file.

It is not expected that the PFS will use file tags, but if the PFS supports its own conversion capability, it may have to take file tags into consideration now that the LFS is also doing conversions. For example, NFS Client will fail vfs_mount if both the LFS TAG() parameter and the NFS PARM(XLATE()) parameter are specified.

The following headers are used by both the PFS interface and the VFS Server functions v_getattr() and v_setattr().

```
In C header BPXYVFSI:
      The following 'SetAttr Change Flag' is added:
         BIT    at_charsetidchg :1;       /*  File Info Set       */

The following is added to the _BPX_MNTE2 form of the s_mnt struct:
      char    me_filetag[4]               /*  file tag            */

In C header BPXYPFSI:

      The following is added to the s_mtab structure:
      char    mt_tag[4];                  /*  TAG() Parameter     */
```

## Using daemon tasks within a PFS

If the PFS needs to invoke functions that cannot be performed in a cross-memory environment, it must make use of other tasks to perform these functions. To use these daemon tasks the PFS must, at a minimum:
1. Attach these tasks and
2. Communicate with them

Several services are provided to make this easier. They are:
- osi_kmsgctl
- osi_kmsgget
- osi_kmsgrcv
- osi_kmsgsnd
- osi_thread

The osi_thread service is available only to PFSs that are running in a colony address space.

The PFS can attach these tasks via the MVS ATTACH service from its initialization task, or it can use the osi_thread service. The osi_thread service attaches a task in the PFS's address space that runs in primary mode. The initial module on this task is a C Main function that fetches the module that is specified by the invoker using the C/C++ **fetch()** function, and then calls it. When called on this task, or thread, the specified module can perform a single function and return; or it can service work requests by the PFS until the PFS terminates. In the latter case, the osi_thread service is used to attach a PFS daemon task.

When attached, these tasks need to communicate with the PFS functions that are invoked by the LFS. One way these processes can communicate is through message queue functions that are provided by the osi_kmsg services in the list above. For descriptions of these services, see Chapter 6.

## Exporting files to a VFS server

For a VFS server to access files that are owned by a PFS on the same system, the following support is necessary in the PFS:

- Its file objects must be visible in the file hierarchy. This is the same as saying that the PFS supports vfs_mount and vn_lookup, as described earlier in this chapter.
- Each file must have a unique and persistent file identifier (FID). This is 8 bytes long, and is usually made up from the file's 4-byte st_ino value and a 4-byte uniquifier. The uniquifier must be constructed by the PFS if it reuses file st_ino values, so that the full 8-byte FID is unique and never reused.

  The FID must persist over PFS restarts and even full-system IPLs. A VFS server's client may access a file days after it has obtained the FID.
- The FID must be returned in all ATTR structures that are returned.
- The PFS must be able to look up a file by its FID reasonably efficiently. The vfs_vget operation must be supported to convert a FID value to a vnode-inode pairing. This is similar to vn_lookup, except that a FID within a file system is looked up, rather than a name within a directory.
- Access checking on read/write must be supported, as discussed in "Opening and closing files and first references to files" on page 34.
- vn_readdir must not require vn_open and vn_close.
- For better performance, the PFS should support:
  - Implicit vn_getattr on any operation that passes a nonzero ATTR pointer in the OSI structure.
  - Sync-on-write, when that bit is on in the UIO. (This eliminates the need for a separate call to fsync.)
  - Real-page support with DATOFF moves for memory-mapped files.

**Porting note**

The vn_fid operation is not used to convert a vnode to a FID. The combination of returning the FID in the ATTR structure and implicit vn_getattr on many operations is much faster for VFS servers.

When a VFS server's client mounts part of the file hierarchy, it really only obtains tokens to a directory and the directory's file system. It is not a mount like that performed for the MOUNT command, and the PFS does not receive a vfs_mount or any indication that it occurred. The first call from a VFS server that the PFS would see is likely to be a vfs_vget, vn_lookup, or vn_readdir.

## Select

A PFS should consider supporting the vn_select operation if data for a read-type operation may arrive asynchronously when no read has been issued; or if buffers for a write-type operation are rationed and are therefore sometimes not immediately available (require a WAIT).

The LFS answers READY for any select status requested from a PFS that does not support vn_select.

See "Select/poll processing" on page 45 for more details.

# PFS interface: Socket PFS protocols

## Activating a domain

NETWORK statements in the BPXPRMxx parmlib member that is used to start z/OS UNIX assign socket domains, or address families, to the socket PFSs.

The NETWORK syntax is:

```
NETWORK TYPE(file_system_type)
        DOMAINNAME(domain_name)
        DOMAINNUMBER(domain_number)
        MAXSOCKETS(number)
```

where:

- **TYPE** identifies the PFS that supports this domain. This operand must match the TYPE operand that is used on the FILESYSTYPE statement that defined the PFS.
- **DOMAINNAME** specifies the domain, or address family, name. The AF_UNIX and AF_INET domains are supported by IBM-supplied socket PFSs.
- **DOMAINNUMBER** specifies the numeric value of the domain that is passed by programs that call **socket()**. The values that are supported for this field are defined in **socket.h**.
- **MAXSOCKETS** specifies the maximum number of currently active sockets that are to be supported.

The parameters just described are passed to the PFS on the vfs_network operation.

During vfs_network the PFS is expected to:

1. Activate support for this domain.
2. Optionally return an 8-byte token that is saved by the LFS and used on all subsequent VFS and vnode operations. This token is typically the address of the PFS's domain block.

When a user calls **socket()**, the first parameter is a domain number. The LFS routes this request to the appropriate PFS with a call to vfs_socket.

The NETWORK statement is analogous to the MOUNT statement that is used by file-oriented PFSs.

See *z/OS MVS Initialization and Tuning Reference* and the description of the NETWORK statement of BPXPRMxx in *z/OS UNIX System Services Planning* for more information.

## Creating, referring to, and closing socket vnodes

The PFS creates vnodes by calling osi_getvnode, which is one of the OSI services in the OSIT vector table that is passed to the PFS during its initialization.

Sockets are created by user calls to **socket()** and **accept()**. The corresponding vnodes are created during vfs_socket and vn_accept, respectively. vfs_socket creates a socket, and if that socket is connected, a stream session is established to another socket that is created by vn_accept. **socketpair()** generates a special case of the vfs_socket call that creates two connected sockets. This is similar to the **pipe()** function.

During vfs_socket and vn_accept, the PFS is expected to:
1. Set up its socket support and build an inode.
2. Call osi_getvnode to create a vnode.
3. Return the vnode token that was returned by osi_getvnode.

The LFS builds the file descriptor, which is also called a socket descriptor, that is the output of the **socket()** and **accept()** functions.

Sockets do not have a name in the file hierarchy; consequently, they cannot be opened by POSIX users or exported by VFS servers.

The user program makes socket calls on the file descriptor, and the calling parameters are generally passed straight through to the PFS by the LFS.

Socket descriptors can be inherited over **fork()**, and they can be duplicated with **dup()**. The LFS manages this sharing; the PFS is not aware of how many active references to a socket there are.

Eventually the program calls **close()** for its socket descriptors. After all active references to the socket vnode-inode are closed, the LFS calls vn_close. Because sockets cannot be opened like files, the PFS receives only a single vn_close for any socket.

During vn_close, the PFS severs the user's socket session.

After the vn_close, the LFS calls vn_inactive for the final cleanup of the vnode-inode relationship.

During vn_inactive, the PFS is expected to:
1. Disassociate the inode from the vnode.
2. Perform any inode cleanup that is desired.

After the call to vn_inactive, the LFS frees the vnode unless the PFS reports a problem through a bad return code.

**Porting note**

Because sockets cannot be reused after vn_close, the PFS can combine its close and inactive processing in vn_close, and choose not to support vn_inactive. Nonsupport is not considered a failure of vn_inactive.

## Reading and writing

The five variations on read/write—vn_rdwr, vn_readwritev, vn_sndrcv, vn_sndtorcvfm, and vn_srmsg—are all UIO operations, and are described in “Reading from and writing to files” on page 36.

The UIO contains additional fields for the socket-specific buffers that are used on some of these calls.

During these read/write calls, the PFS must:

1. Move the data using Move With Source Key or Move With Destination Key, as appropriate. The osi_copyin and osi_copyout services can be used to move data areas between the user and kernel address spaces. The osi_uiomove service can be used to move data areas based on the UIO structure for vn_rdwr and vn_readwritev.
2. Return the number of bytes that were transferred.

**Serialization:** All five operations are called with an exclusive latch for writing. All five operations are called with an exclusive latch for reading, with the exception of vn_rdwr and vn_readwritev, which may be called with a shared latch for reading if the PFS has specified shared read support for the file being read. The LFS defaults to exclusive latching for both reading and writing, to help the PFS implement the POSIX semantics of atomic operations and immediate visibility to all other processes. This latching can be turned off if it is not needed by the PFS. Refer to “LFS-PFS control block serialization” on page 23 for more details.

## Getting and setting attributes

Socket descriptors are eligible for **fstat()**, so sockets can be called for vn_getattr. The PFS should consider supporting this operation and returning some information in the ATTR structure. At a minimum, you could return: the file type, permission bits of 777, the current time for the time values, the devno as passed by vfs_network, and an inode number for the socket that is unique for this socket at this point in time.

**Note:** Some programs use fdopen() with a socket descriptor, and this function does an fstat() under the covers.

Generally, a program cannot set any attributes of a socket, so the PFS does not have to support the vn_setattr operation.

## Select/poll processing

An application program calls **select()** or **poll()** with a list of file descriptors and the events that are to be waited for. The file descriptors can represent files, sockets, pipes, or terminals; they are all referred to as “files” in this discussion. The events that can be waited for are: ready for reading, ready for writing, and exceptional conditions. Because a **poll()** is converted into a **select()** call by the time the request reaches the PFS, for this discussion only select will be discussed.

There are two operations that can be called to handle the select request: vfs_batsel and vn_select. The vfs_batsel operation is useful for a performance boost; it does not have to be supported. If a PFS supports the vfs_batsel operation, a single call is made to that PFS with an array of information about its files. If a single descriptor is requested, or the PFS does not support vfs_batsel, the vn_select operation is called for the owning PFS for each file specified.

The LFS converts the file descriptors into vnodes. If the user has multiple file descriptors in the list that refer to the same file, such as after a **dup()**, or if a particular PFS owns more than one file that is present in the list, it receives a separate call for each file if the vfs_batsel operation is not supported. Otherwise, a single call is made with multiple array entries for the same file. While one user is waiting in **select()** for some files, another user may issue **select()** for some of the same files. The LFS manages the lists and the associations of users to requests. The PFS should just treat each vn_select or vfs_batsel array entry as a completely separate and independent action against the file, and be prepared for more than one **select()** to be active at a time for a file.

Select processing consists of two phases, called *Query* and *Cancel*, which are identified by a parameter on the select call. Each file may be called for both phases or just for Cancel. When a user specifies a timeout value of 0, the LFS skips the Query phase and goes right into the Cancel phase.

The LFS passes a select token to the PFS with each vn_select or vfs_batsel array element call. The select token uniquely identifies a request for both phases, and thus can be used by the PFS to correlate Queries and Cancels. This token is unique to this single instance of vn_select(Query) being called, and is not used again until after the corresponding call to vn_select(Cancel).

There is also a *PFS_work_token* available on vn_select and in each array element of vfs_batsel that can be set by the PFS to correlate Queries and Cancels.

**Note:** To simplify the discussion, only vn_select is mentioned in the next section. The only difference between vn_select and vfs_batsel is that similar processing must occur within a loop for the array elements of the vfs_batsel request.

## Query phase

In the Query phase of select processing, the LFS queries the PFSs by calling vn_select(Query) with the vnode that is represented by each file descriptor.

During vn_select(Query), the PFS must:

1. Return status information without taking any other action, if any requested event is immediately available.
2. Otherwise, save the select token (16 bytes) and the Select_Options in a select-pending structure that is chained from its inode.

The Query phase ends as soon as any PFS reports immediate status. The remaining PFSs are contacted during the Cancel phase, so the user can receive the most information available at this time.

The LFS may omit recalling the PFS for the Cancel phase if:

1. The PFS does not set any of the PFS_work_tokens, and
2. For vfs_batsel, status is returned in the array entries.

If the PFS is dependent on being recalled for Cancel whenever it has been recalled for Query, it must set a PFS_work_token to some nonzero value. For optimal performance, the PFS should not have this dependence when it is able to report immediate status to the Query request.

If no PFS reports immediate status, the LFS waits for one of the PFSs to call osi_selpost, or for the time limit to expire.

**Event occurrence:** Eventually an event occurs asynchronously within a PFS for a given file. The PFS process or thread that handles these events notices that the file has selects pending for it. Examples of such events are: data arriving for a read, buffers freeing up for a write, or sessions terminating for an exceptional condition.

When such an event occurs, the PFS is expected to do the following:

1. Scan through the select-pending structures that are chained from the inode for those that are waiting for this type of status.

   The PFS must serialize this with its own processing for Cancel; see "Cancel phase."
2. For each pending select that is satisfied:
   a. The PFS removes the select-pending structure, or marks it as "posted". The PFS must ensure that it never calls osi_selpost more than once for a particular vn_select(Query) request or select token.
   b. The osi_selpost routine is called with the select token saved during the Query phase.

The osi_selpost routine uses the select token to find the waiting process and thread and post it.

**Note:** The identity of the event that occurred is not passed to osi_selpost. This information is picked up by the LFS during the Cancel phase.

## Cancel phase

The LFS goes through the Cancel phase by invoking vn_select(Cancel) for each file descriptor when:

- One of the PFS events has occurred and osi_selpost is called
- Any PFS reported status during the Query phase
- The timeout value expires

Note that if a PFS reported status during the Query phase, the loop that was doing the queries is terminated; therefore, a cancel request may be received by a PFS even though no query was done.

During vn_select(Cancel), the PFS is expected to do the following:

1. Scan the pending-select structures that are chained from the inode for one with a matching select token. If one is found, it is removed so that osi_selpost is not invoked for that select token after the PFS returns from this vn_select(Cancel) call.

   **Note:** It is the PFS's responsibility to serialize the cancellation of a pending select with its asynchronous event handler, which may be attempting to call osi_selpost. It is critical that osi_selpost never be called for a particular select token after the PFS returns to the LFS from a call to vn_select(Cancel) for that same select token.

   It is not unusual for the PFS not to find a pending select to be canceled, as it could have been already removed by the event handler, or this PFS may not have been queried in the first place.
2. After the PFS ensures that the select is no longer pending, it checks for the requested status and returns this information to the LFS.

The LFS collects status from all of the files and reports it back to the program that called **select()**.

**Note:** Although it is rare in practice, there is nothing to stop a user from selecting and reading on the same socket from two different processes or threads. Consequently, it is technically possible that an event that is reported by select may no longer be true when the selecting program finally acts on the information. A selecting program may not act on the information, but pass it off to another process to handle. Therefore, reporting back on select does not reserve the data or buffers for the caller; it merely reports the status of the file at that time.

# Common INET sockets

## Common INET sockets PFS structure

The Common INET layer (CINET) is inserted between the LFS and a sockets PFS to allow multiple AF_INET transports to be used by a single application socket. A sockets PFS may be attached directly to the LFS when it is the only AF_INET transport on the system, or attached through the CINET layer when it is one of several. To be attached to CINET, the PFS must implement the “master socket” and support several additional ioctl command types, as described in this section. The interface to the PFS is the same in both cases. Once the additional support for CINET is written, the PFS does not have to distinguish between the two cases.

When Common INET is used, the sockets file system is initialized by the SUBFILESYSTYPE statement in the parmlib member, instead of by the FILESYSTYPE statement, which initializes the Common INET support. The operands of the SUBFILESYSTYPE statement are similar to those for the FILESYSTYPE statement.

The general model is that of a sockets PFS that is split into two pieces: a PFS layer that runs in the kernel address space, and additional programming that runs in a separate address space and that actually controls the transport interface to the network. For the purposes of this discussion, the PFS layer piece will be called the **transport driver (TD)** and the separate address space piece will be called the **transport provider (TP)**.

The transport driver is started by z/OS UNIX, as a PFS, and communicates with the transport provider through its own internal mechanisms, usually by a space switching program call (PC).

The transport provider (such as TCP/IP) is started independently, and communicates with the transport driver through the master socket.

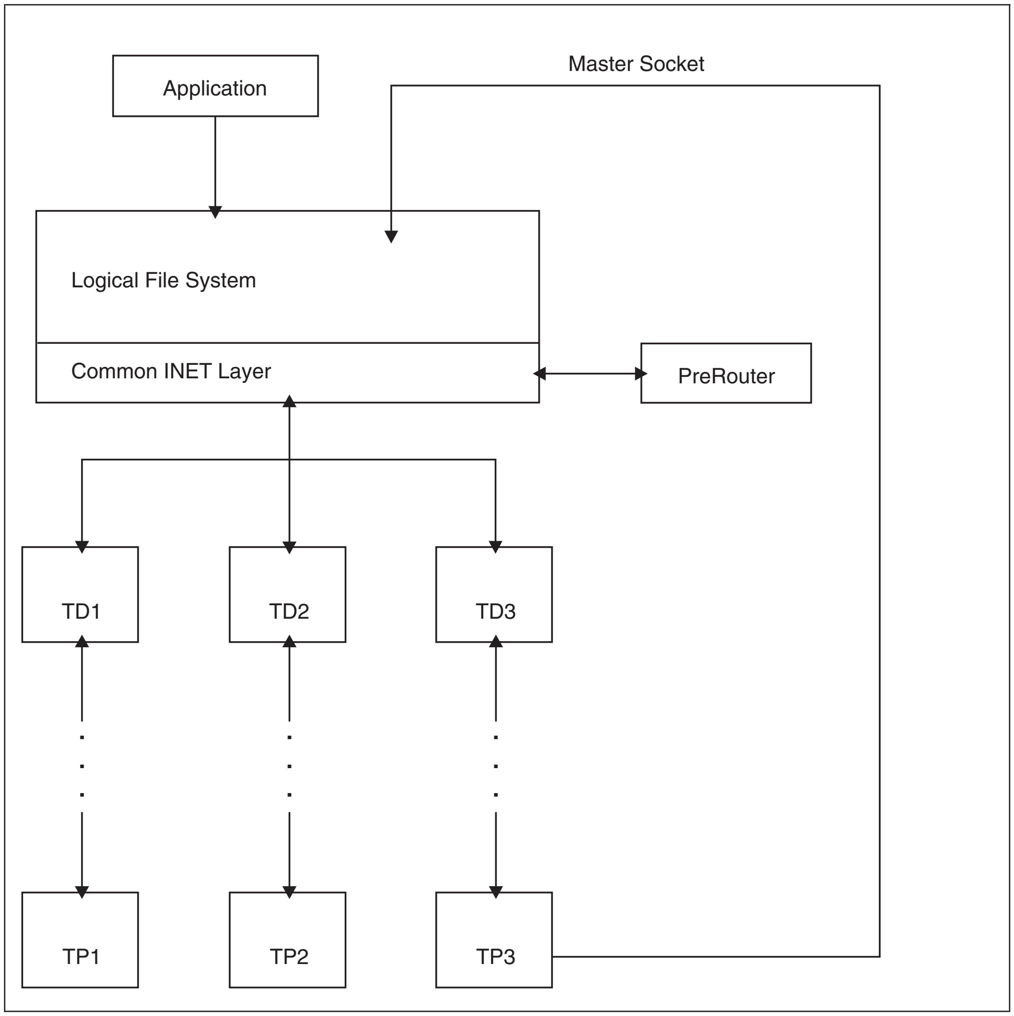


*Figure 5. Common INET sockets PFS structure*

A TD/TP that is structured entirely within the PFS in the kernel address space still has to establish the master socket and pass the minimum ioctl commands to run under the CINET layer.

## The master socket

The master socket is used to communicate between the transport provider and both the Common INET layer and its own transport driver. It is used mostly for initialization and, potentially, for later dynamic route updates. If the TP ever has to initiate a message to the TD (for instance, due to an asynchronous configuration update), it can do so over the master socket.

- The master socket is created by the transport provider with the standard socket() C function or the BPX1SOC/BPX4SOC callable service, by specifying AF_INET for the *Domain* and -1 for the *Protocol* parameters.

  This builds a session from the TP to the CINET layer.

  The TP address space must be defined to RACF as a z/OS UNIX user with a UID of 0.
- The only functions that are used with the master socket are ioctl and close.

  Most of the ioctl command codes that are used with z/OS UNIX are nonstandard, so these ioctls must be issued with the w_ioctl() C function or the BPX1IOC/BPX4IOC callable service.

The socket can be closed with either close() or BPX1CLS/BPX4CLS.

- The first thing that flows on the master socket must be an SIOCSETRTTD ioctl to connect the socket to a specific transport driver. This ioctl is also known as the left bookend, signifying the start of TD–TP initialization. On the call, the Argument_length should be specified as 8, and Argument should refer to an 8-byte area in which the TD name is filled in. For more information about the interface to ioctl, refer to “vn_ioctl — I/O control” on page 154.

  The vfs_socket request is issued at this point to the specified TD, which builds the normal socket support between the LFS and PFS, but does not propagate this session to the TP.

  The SIOCSETRTTD command is then passed on to the TD with an ioctl call.

  **Note:** The TP must know the name of its own TD in order to select it with SIOCSETRTTD. This name was specified with the NAME parameter of the SUBFILESYSTYPE statement that started the TD, and is passed to the TD when it is initialized. There are several ways to make this name known to the TP. It could be a product-specified constant value; the value could be configured into the TP through its externals; the TD could pass the name to the TP if it starts the PC session first; or the TD could store the name with the MVS Named Token Services, where the TP would retrieve it.

- Subsequent ioctls are then sent from the TP to the TD to perform product-specific initialization, as necessary. For instance, these could drive the TD to establish the PC session to the TP. These ioctl calls can specify application-defined commands, or use existing command definitions. The ioctl command values that are used must not conflict with any of the commands that are discussed here, or any that are used by the prerouter.

  These commands pass through z/OS UNIX without any interpretation.

  **Note:** If the PFS is designed to run directly attached to the LFS, it has already solved the problems of initialization between the TD and TP. This does not have to change when it is attached through CINET. Only the first and last ioctl commands discussed here are required on the master socket.

- After any product-specific initialization is finished, an IOCC#TCCE ioctl command is sent by the TP to notify CINET that this file system is ready for business. This ioctl command is also known as the right bookend, signifying the end of TD–TP initialization. For this command, no other specific data is required, so the Argument_length can be zero.

  This command is also passed on to the TD.

  At this time, the transport is considered to be active. The prerouter gathers configuration information from the transport and applications that had used the SO_EioIfNewTP socket option receive notification that a new transport is available for use. This notification is performed by failing any socket accept or receive type calls with a return code of EIO, after which the application closes that socket and opens a new socket to pick up the new transport.

  If the transport is not yet ready to accept new socket requests, the notification phase can be delayed. If the argument length for IOCC#TCCE is four bytes and the argument contains a value of one, this signifies a delay and the SO_EioIfNewTP notification phase will be skipped. The transport must later send another IOCC#TCCE ioctl command with a value of two to perform just the notification phase.

- At this point the prerouter will start its conversation with the TD–TP on a separate socket session, see “Common INET prerouting function” on page 51.

Ioctls that flow on the master socket to the TD are never passed through to the TP, because that is where they came from. Some of the ioctl commands are intended only for the Common INET layer, and these are not even passed on to the TD. However, the TD should be coded to ignore the ioctl commands that are intended for the Common INET layer, because when it is connected directly to the LFS it will receive these requests. The TP could also be configured to know how the TD is set up within z/OS UNIX and process accordingly, but this is usually not worth the extra effort and externals.

The master socket is left open for the duration of the transport provider. If this socket is closed, the prerouter assumes that the transport provider has terminated. This socket may also be needed later for dynamic route updates, and it can be used within the TD/TP recovery design. If the TP abnormally terminates, the master socket for it is closed. The TD sees this as a vn_close, at which point it can take whatever recovery actions may be necessary. Thus, a resource manager for the TP and the code to notify the TD are not necessary solely for the purpose of letting the TD know when the TP crashes.

The constants for the various ioctl commands that are used during initialization are defined in BPXYPFSI.

## Common INET prerouting function

The Common INET support allows an installation to connect up to 32 different instances of TCP/IP or other AF_INET physical file systems. Application programs that use sockets do not need to change any code to take advantage of the multiple AF_INET file systems.

Supporting multiple AF_INET physical file systems and providing a single file system image to the user means that the Common INET must perform a set of management and distribution functions that govern how a socket behaves with multiple file systems. A fundamental requirement for distributing work across multiple file systems is an understanding of the IP configurations of each file system. The IP configurations are needed to determine which file system should handle a bind() to a particular home IP address, a connect(), a sendto(), and so forth.

When the Common INET processes a socket request that requires it to select only a particular file system based on an input IP address from a user, the Common INET uses its copy of each file system's IP configuration to select the correct file system to process the user's request. Copies of the IP configurations are maintained by the Common INET internally, and are only used for "prerouting" a socket call to the correct file system. The file system that was selected performs all of the official file system functions, such as routing, once the socket request reaches the file system from the Common INET.

Each file system that is connected to the Common INET must provide a copy of its internal IP routing table. An ioctl is issued to each transport provider (TP) as part of the PFS initialization. This allows the Common INET function to query the routing tables for that file system. Once the Common INET prerouter function has successfully retrieved and stored routing information from a particular file system, message BPXF206I is issued to the hardcopy log. Message BPXF206I is also issued whenever a file system refreshes its routing table. For example, IBM's TCP/IP may refresh its routing tables as part of the OBEYFILE command. Message BPXF207I is issued to the hardcopy log whenever the Common INET deletes internal routing information for a file system. When the connection with a specific file system is severed, the Common INET routing information for that file system is deleted.

## Limitations of common INET-attached PFS IP configurations

System programmers and network administrators should be aware of the following information about the common INET prerouting function:

1. Two or more file systems may contain home IP addresses on the same network or subnetwork. However, load balancing across file systems is not done. If a user has not done a bind() to a home address, the same file system is selected for all subsequent sendto()s, even if there are other transport providers with routes to the same destination.
2. Two or more file systems may contain a route to the same destination. Again, load balancing across the file systems is not performed.
3. Metrics for network routes: If two or more transports maintain network routes to the same destination network, metric information is needed from each transport in order to correctly select the best route. For IBM's TCP/IP, this is best accomplished when each TCP/IP is running with a dynamic routing daemon (OMPROUTE). Statically defined indirect routes (routes to destinations that do not reside on a transport's directly attached links) do not provide adequate metric information to select the shortest route to a destination network when two or more transports maintain indirect routes to the network.

   In cases in which two or more file systems maintain duplicate destination network addresses and not all file systems provide metric information, selection of the file system to process a request is unpredictable. Generally speaking, the file systems with metric information are selected because of implementation details.
4. In the event that two or more file systems contain network routes with no metric information or duplicate metrics, selection of a file system to process the request is as follows:
   a. If one of the file systems with a route to the destination is the default file system as specified in the BPXPRMxx parmlib member, the default file system is selected.
   b. Otherwise, the file systems are selected in the order in which they were defined in the BPXPRMxx parmlib member.
5. Host-defined routes are always searched before network routes.
6. If a file system severs its connection, all routing information for the severed file system is deleted. If the severed file system maintained duplicate home or network routes, these routes are deleted. Subsequent requests for the duplicate routes are routed to the remaining file systems.
7. If two transport providers have connections to the same network and two applications that are running on the same MVS start communicating with each other, performance may not be optimal. If for some reason the two applications bind to different transport providers, the external network is used, rather than the Common INET local INET support. Therefore, it is suggested that applications use a method analogous to gethostid() to get the IP address of themselves and bind to the address that is returned from the gethostid(). This method ensures that the default transport provider is selected. The local INET support works only with the default transport provider.

## Initialization for an AF_INET (IPV4) transport driver

When a transport driver is being initialized, the prerouter is notified of the TD's arrival. The prerouter performs the following functions:

1. Opens a socket from the kernel address space. This is not the master socket, but a regular user socket that is initiated through the z/OS UNIX socket interface.
2. Issues an ioctl SIOCGIFCONF to get the list of home interfaces maintained by the file system instance and adds them to the home IP table.
3. After all of the home routes have been processed, issues an ioctl with the SIOCGRTTABLE function code. This gets the file system host and network routing information in a table format. The mapping for this request is found in **ioctl.h**.
4. Places the routes from the SIOCGRTTABLE in the host and network routing tables managed by the prerouter. Note that the installation can give metrics in hop counts or millisecond delays by setting the appropriate flag in the header of the SIOCGRTTABLE structure. All metrics are converted to hop counts.
5. Closes the socket. The prerouter is now initialized for the transport driver.

## Initialization for an AF_INET6 (IPV6) transport driver

When the transport driver that is being initialized is IPV6 capable, the prerouter is notified of the TD's arrival. The prerouter performs the following functions:

1. Opens a socket from the kernel address space. This is not the master socket, but a regular user socket that is initiated through the z/OS UNIX socket interface.
2. Issues an ioctl SIOCGHOMEIF6 to get the list of home IPV6 interfaces maintained by the file system instance.
3. After all of the IPV6 home routes have been processed, issues an ioctl with the SIOCGRT6TABLE function code. This gets the file system IPV6 host and network routing information in a table format.
4. Places the routes from the SIOCGRT6TABLE in the host and network routing tables managed by the prerouter. Note that IPV6 metrics are in hop counts.
5. Closes the socket. The prerouter is now initialized for the transport driver.

## Route changes

The prerouter handles BSD-style route changes for the routeD add (SIOCADDRT) and delete (SIOCDELRT) functions. When a route is added, the rt_use field is checked for a nonzero code. If rt_use is nonzero, it is assumed to be a hop count metric. Metrics can be changed by reissuing the SIOMETRIC1RT ioctl or setting the rt_use field in the SIOCADDRT to the new metric value.

Route changes can be sent to the prerouter in two ways:

- When using ioctls for add (SIOCADDRT) and delete (SIOCDELRT) functions that use z/OS UNIX sockets, z/OS UNIX automatically passes the ioctls to the prerouter and the prerouter makes the needed updates.

  **Note:** IPV6 capable stacks should use the SIOCADDRT6 and SIOCDELRT6 functions for adding and deleting IPV6 addresses.
- If a routing daemon does not use z/OS UNIX sockets, but uses a different interface to a file system, the ioctls for add (SIOCADDRT) and delete (SIOCDELRT) functions must be propagated to z/OS UNIX. The file system needs to use the add (SIOCMSADDRT) and delete (SIOCMSDELRT) functions. These are issued on the master socket and are denoted with 'MS'. z/OS UNIX needs the MS, or these functions are propagated back to the file system and there is an endless loop.

ICMP redirects are handled using the SIOCMSICMPREDIRECT ioctl.

If the file system encounters a situation where it believes that the routing information needs to be re-synchronized, the file system can issue the SIOCMSRBRTTABLE ioctl (or, for IPV6 capable stacks, the SIOCMSRBT6TABLE ioctl) on the master socket. This causes the prerouter to flush the routing information for the file system and rebuild it from scratch. If the IPV6 home information needs to be re-synchronized, SIOCMSRBHOMEIF6 should be used.

**Note:** If a user does a socket request during a rebuild, the user may or may not be able to connect with the file system. The routing table is in flux.

## SRB-mode callers

z/OS UNIX supports programs that are running on SRB dispatchable units, in addition to the more standard TCBs. This affects the PFS, as the resulting vnode operations are also running in SRB mode.

SRB mode is even more restrictive than cross-memory mode. Additional restrictions on the PFS include the following:

- There are no MVS WAITs; instead you have to use SUSPEND/RESUME. This can impact some of the internal functions of the PFS that may not be easy to modify, including task switching, lock managers, and tracing.

  **Note:** The osi_wait/osi_post services transparently support both TCB and SRB-mode callers.
- No TCB is available (Psatold=0). The TCB address is used by some programs to build identifiers, or in other algorithms.
- There is no EOT or ESTAE recovery, although you can use an FRR.

  **Note:** vn_recovery support is still available from the LFS.
- Because SRB callers do not receive POSIX signals, they cannot break out of extended waits, as they can in the EINTR cases.

  Signal-enabled osi_waits should still be set up where they are set up now, because this also indicates that the osi_wait may be interrupted for process termination.

The following OSI services are enabled for SRB-mode callers:

| | |
|---|---|
| osi_copyin | osi_sched |
| osi_copyout | osi_selpost |
| osi_copy64 | osi_uiomove |
| osi_getvnode | osi_upda |
| osi_mountstatus | osi_wait |
| osi_post | osi_wakeup |

The PFS signifies that it supports SRB-mode callers on the pfsi_srb bit that is returned during PFS initialization. The LFS inhibits SRB-mode calls to PFSs that do not support them.

All sockets-related vnode operations are potentially callable from an SRB, and in the future this may be extended to file-related operations. The PFS should therefore be made completely SRB safe.

Refer to *z/OS MVS Programming: Authorized Assembler Services Guide* for more information about SRB-mode programs.

## Asynchronous I/O processing

An asynchronous capability is provided by z/OS UNIX for socket calls that may block. These include accept, connect, select, poll, and the five pairs of read/write type functions. These services are provided asynchronously to programs through the asyncio callable service. Refer to *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for details.

Asynchronous I/O processing between the LFS and PFS is implemented with a two-pass technique using the regular vnode operations, such as vn_accept and vn_rdwr:

- Part 1, which is indicated by a bit in the Osi structure, starts with the beginning of the normal vnode operation and continues up to the point at which the PFS would call osi_wait to block. The PFS returns to the LFS instead of waiting. When the I/O can be completed, the PFS calls the osi_sched service at the point at which it would call osi_post for a blocked operation.
- Part 2, which is indicated by another bit in the Osi structure, continues from the point after which osi_wait would have been called through the end of the operation.

These two stages are covered in detail in the next sections.

## Related services

Two special osi services are used in asynchronous I/O processing:

- osi_upda, which is called during Part 1 to pass a PFS token to the LFS. Refer to “osi_upda — Update async I/O request” on page 429 for specifics.
- osi_sched, which is called to drive Part 2 when the I/O can be completed. Refer to “osi_sched — Schedule async I/O completion” on page 410 for specifics.

The vn_cancel service is a special vnode operation that is used to cancel an outstanding request. Refer to “vn_cancel — Cancel an asynchronous operation” on page 128 for specifics.

The vnode operations that can be run in two passes are:

```
vn_accept    vn_rdwr         vn_sndtorcvfm
vn_connect   vn_readwritev   vn_srmsg
             vn_sndrcv
```

## Impact on initialization

The PFS signifies that it supports asynchronous I/O on the pfsi_asyio bit that is returned during PFS initialization. To support asynchronous I/O, the PFS must also support SRB-mode callers, because Part 2 runs from an SRB, and it must support vn_cancel. The LFS inhibits asynchronous calls to PFSs that do not support them.

## Waits that are avoided

Asynchronous I/O is intended to avoid long waits only. These are blocking, indeterminate waits that usually depend on something from the network or an end user. Long waits also tend to be conditional, based on the current non-blocking mode. Short internal waits, such as lock waits for serialization, are not avoided. An example is that of a read: you can wait for a lock to look at the inbound queue, but if the queue is empty you cannot wait for the data.

## Related OSI fields

The OSI fields that are significant to this discussion are:

- osi_asy1, which signifies Part 1
- osi_asy2, which signifies Part 2
- osi_asytok, which holds the LFS's token on entry to Part 1 and the PFS's token on entry to Part 2.
- osi_ok2compimd, which indicates that the PFS may complete the operation immediately, if possible. See "Asynchronous I/O flow details" on page 58 for details.
- osi_compimd, which is returned by the PFS to indicate that it has completed the operation immediately. This is valid only if osi_ok2compimd is on.
- osi_commbuff, which indicates that Part 2 of Async I/O must not occur. Within the PFS, the changes from normal Async I/O flow are:
  1. Received data can be copied directly to the user's buffers from the PFS's inbound data handler.
  2. osi_sched is called after the data has been copied.
  3. The amount of data being returned must be supplied to osi_sched.
  4. There must be no dependence on Part 2 being called.

**Note:** The last four fields are meaningful only when osi_asy1 or osi_asy2 are on; they should not be referred to otherwise.

These fields are covered in more detail in Figure 6 on page 58.

## Canceling an operation

The LFS attempts to cancel an outstanding operation with vn_cancel. There are two types of vn_cancel: normal and forced.

- A normal vn_cancel only flows to the PFS between Part 1 and Part 2, and is used to get requests off the waiting, or blocking, queues in the PFS. If the request is not currently on a waiting queue, nothing is done. If the request is found, it is removed from the queue and failed with ECANCELED.
- A forced vn_cancel is used during process termination of the original requestor. It can be sent logically at any time, but the PFS will already have abnormally ended and gone through recovery if the request was in Part 1 or Part 2 at the time the process terminated. There is no Part 2 after a vn_cancel force, so the PFS must do any necessary cleanup during the vn_cancel.

Refer to "vn_cancel — Cancel an asynchronous operation" on page 128 for more information.

## Responsibilities for the semantics

The semantics for the asyncio function are split between the PFS and the LFS. Some of the features whose support might be ambiguous are discussed here. Refer to aio_suspend (BPX1ASP, BPX4ASP) — Wait for an asynchronous I/O request in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* while reading this section.

The LFS must handle the following:

- The aiocb structure. The interface to the PFS is through the regular vnode operations, such as vn_rdwr and vn_sndrcv.

- The returned information. The PFS should return 0 for a successful Part 1, and the normal functional values from Part 2. In particular, the LFS handles the EINPROGRESS return_code.
- Scheduling the SRB and calling the I/O completion notification. This includes calling the user exit, posting an ECB, and sending a signal.
- AioSync. This appears to the PFS as a normal synchronous operation (osi_asy1=osi_asy2=OFF).
- AioOk2CompImd, for accept and connect. The osi_ok2compimd bit is always on in the PFS for vn_accept and vn_connect, so the PFS can always complete these operations immediately without calling osi_upda or osi_sched. osi_compimd should be turned on if the PFS does happen to complete these operations immediately.
- The select and poll functions, which are already asynchronous with respect to the PFS. The PFS continues to call osi_selpost for the vfs_batsel and vn_select operations.

The PFS must handle or contribute to the support of:
- AioOk2CompImd, for reads and writes, through support for osi_ok2compimd. Even when the PFS is able to complete a read or write type of operation immediately, it must still call osi_sched whenever osi_ok2compimd=off. See “Asynchronous I/O flow details” on page 58 for details.
- AioCallB4 and deferred buffer allocation, by not requiring the presence of the user’s data buffers during Part 1, unless osi_ok2compimd=on; and by passing the length of data that is available to be received to osi_sched.
- The ECANCELED Return_code, by failing a request with that return code when the request has been removed from a waiting queue because of vn_cancel. The race condition between vn_cancel and data arrival can only be resolved by the PFS.

## Asynchronous I/O flow diagram

This diagram describes the general flow of an asynchronous operation, noting those parts of the interface that are specific to its asynchronicity, and the significant design points within the PFS that the LFS is dependent on. As it is based on a somewhat generic PFS model, it may not match any specific implementation, and a PFS may have to do some work to accommodate it. PFSs that have an associated separate address space should be able to fit this model. These design points can be met either in the kernel address space or in the associated address space.

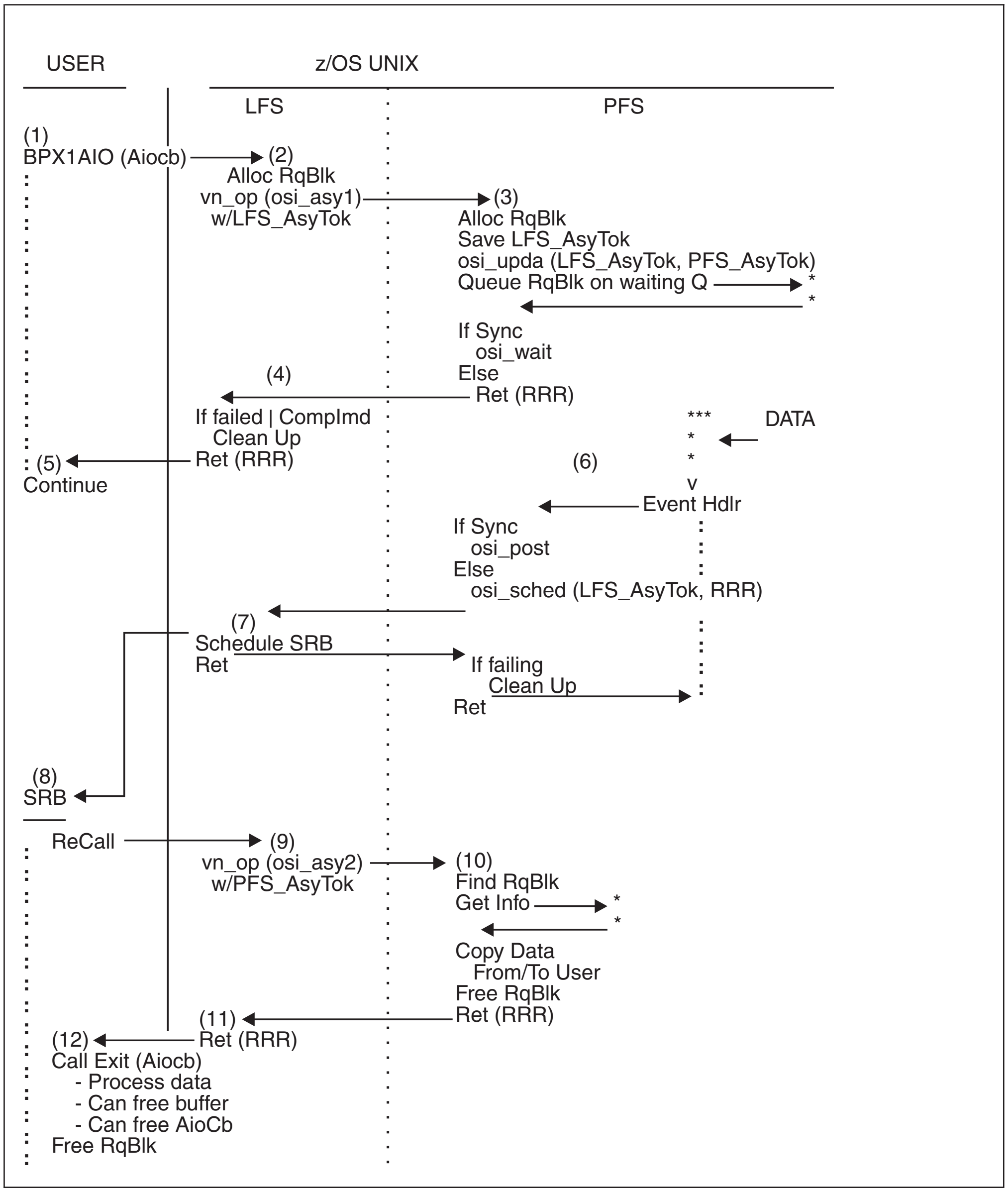


*Figure 6. Async operation flow*

## Asynchronous I/O flow details

This flow is discussed as an addition to an existing PFS design that already handles synchronous blocking and non-blocking socket operations.

1. BPX1AIO/BPX4AIO (asyncio) is called with an Aiocb structure. The Aiocb contains all the information that is needed to do the specific function.
2. The LFS builds an Async I/O Request Block (RqBlk). The PFS has signified support via the Pfsi_Asyio PFSinit output bit. The regular vnode operation for the function is invoked in the PFS with:
   - + The osi_asy1 bit turned on to indicate Async I/O Part 1.
   - + The osi_asytok field holding the LFS_AsyTok token.
3. **Part 1 in the PFS:**
   - The PFS builds its own Request Block. The LFS_AsyTok is saved for later use with osi_sched(). The PFS's PFS_AsyTok is passed back to the LFS via

osi_upda(). This identifies the request to the PFS in Part 2 and to vn_cancel. Basic preliminary parameter and state checking can be done here.

- The user's read buffers are not referenced during Part 1 unless osi_ok2compimd=ON; see the variations below. This allows the user to defer read buffer allocation to just before Part 2. The requested length for reads is available, even if the buffers are not.
- The PFS queues the request to await the desired event. This is essentially the same thing that is normally done for blocking requests. Instead of calling osi_wait(), as it would at this point for a blocking request, the PFS returns to the LFS with the Return_value, Return_code, and Reason_code (RRR) from queueing the asynchronous I/O. For a successfully queued request, the Return_value is 0, and any output from the operation is deferred until Part 2. Important PFS structures are preserved as necessary over this return and the subsequent reentry to the PFS for Part 2.

**Variations:**

- If the operation fails during Part 1, the normal path is taken and, instead of the request being queued, the failure is returned. This includes both queueing failures and failures of the function that is being requested.
- If the operation can be completed immediately and osi_ok2compimd=ON, the PFS can proceed as it would normally and complete the operation synchronously. osi_compimd is turned ON to tell the LFS that this has happened.
- If osi_ok2compimd=OFF, the PFS must make the call to osi_sched from within this vnode operation, and proceed from Part 2 as if the data were not immediately available. This bit is only OFF for read/write type operations. If the PFS does not need to be recalled for Part 2 (for instance, with a short write), it can skip the call to osi_upda. It is all right to transfer the responsibility for calling osi_sched to some other thread, making the call asynchronously and returning to the LFS, as long as you do not wait for network input.

4. The LFS returns to the caller with AioRC=EINPROGRESS; or, if it has failed or completed immediately, cleans up and returns the operation's results.
5. The original caller continues. All structures and data buffers must persist throughout the operation.
6. Event occurrence in the PFS:
   - At some point data arrives for the socket, or buffers become available, and the request can be completed.
   - The PFS notices, or responds to, this condition as it normally does. Instead of calling osi_post(), as it would at this point for a blocked request, it calls osi_sched() with the saved LFS_AsyTok to drive Part 2.
   - For read type operations, the passed Return_Value contains the length of the data that is available to be read in Part 2. This is an optional performance enhancement that some applications may take advantage of. If the length is not easily known, 0 should be passed.
   - The rest of the action happens on the SRB, because user data cannot generally be moved while it is on the thread that calls osi_post/osi_sched.

   **Variations:**

   - If the request fails asynchronously, the PFS can report this on the call to osi_sched() by passing the failing three R's. There will be no Part 2 if the passed Return_value is -1, so the PFS has to clean everything up from here.

- Alternatively, the PFS can save the results, pass success to osi_sched(), and report the failure from Part 2. This is sometimes more convenient when the event handler is in a separate address space and the PFS has resources to clean up in the kernel address space. The only time osi_sched() fails is if the passed LFS_AsyTok is no longer valid, which may represent a logic error in the PFS. osi_sched() succeeds even after the user has terminated, but the PFS sees vn_cancel instead of Part 2.

7. The LFS schedules an SRB into the user's address space and returns to the PFS. The SRB runs asynchronously to the caller of osi_sched().
8. The SRB runs in the user's address space, so that the user's data buffers can be referenced from "home" while in cross-memory mode. This also gets the user's address space swapped in if necessary. The LFS is recalled to get into the kernel address space.
9. The LFS reconstructs the original vnode request structures. The same vnode operation is invoked in the PFS as for Part 1, with:
   - + The osi_asy2 bit turned on to indicate Async I/O Part 2.
   - + The osi_asytok field holding the PFS_AsyTok value from osi_upda()

   **Variations:**

   If osi_upda was not called during Part 1, the PFS is not called for Part 2.
10. **Part 2 in the PFS:**
    - This is running on an SRB instead of the more usual TCB, and the PFS has to be able to handle this mode.
    - From the PFS_AsyTok, the PFS is able to pick up from where it left off at the end of Part 1 (3), when it returned to the LFS instead of waiting. Necessary information that is related to the completing operation is obtained in a manner similar to that in which it is obtained after coming back from osi_wait().
    - Data is moved between the user's and the PFS's buffers for read/write types of operations; or the operation is completed as appropriate.
    - The normal cross-memory environment has been recreated, with the user's buffers in home and the PFS's buffers in primary; or it is otherwise addressable as arranged by the PFS.
    - The normal move-with-key instructions are used to protect against unauthorized access to storage. The osi copy services are available.
    - For unauthorized callers in a TSO address space, the LFS has stopped the user from running authorized TSO commands while async I/O is outstanding. This avoids an obscure integrity problem, with user key storage being modified from a system SRB.
    - The PFS returns to the LFS with the results of the operation and the normal output for this particular vnode operation, such as the vnode_token from vn_accept. The operation is over at this point, as far as the PFS is concerned.

    **Variations:**

    - If the operation fails during Part 2, this is reported back. An earlier failure may have been deferred to Part 2 by the PFS.
    - For very large writes, the PFS may not want to commit all of its buffers to one caller. It may instead loop, sending smaller segments and waiting in between for more buffers. If this is the case, the PFS remains in control and does not return from Part 2 until the whole operation is complete, that is, until the remainder of the operation is synchronous and the PFS blocks as necessary, as it normally does in this loop. osi_wait is convenient here, as it accommodates SRB callers. Essentially, osi_sched() is only called when the

first set of buffers become available and the effect is to offload the work from the user's task or SRB to a system SRB. The operation is still asynchronous to the user. This ties up the SRB, but it is considered to be a situation of relatively small frequency.

- Because SRBs are not interrupted with signals, osi_waits during Part 2 normally do not return as they do in the EINTR cases. If the user's process terminates, signal-enabled osi_waits return as if they have been signaled.

11. On return to the LFS, signals are sent and unauthorized exits are queued to the user's TCB (not shown).
12. The LFS returns to the SRB.
13. On return to the SRB, authorized exits are called and ECBs are posted. When the user program is notified that the I/O has completed, either on the SRB or user's TCB, it can free the Aiocb and buffers. The operation is over, as far as the LFS is concerned, either at the end of the SRB or after an unauthorized exit has run on the user's TCB.

## Colony PFS PC

A PC number is established in colony address spaces that can be used from code running in the kernel to PC into the colony. This could be used by a related PFS that runs in the kernel or by a related file exporter's glue exit.

The PC number is passed to the PFS in the pfsi_pfspc field during initialization. Using this PC involves the following:

- The colony PFS must have a PC routine that will be the target of the PCs. This routine must reside in the colony or in common storage.
- The colony PFS passes the pfsi_pfspc PC number and the address of its PC routine to the cooperating code that runs in the kernel or otherwise makes these values known to the kernel code that will use them.
- The kernel PC caller must place the colony PC routine address in Register 15 and invoke the PC instruction with the pfsi_pfspc value.
- In the colony, the real PC routine that was established by the LFS branches to the address that is in Register 15.
- The PFS's PC routine is responsible for anything that it may need, and its entry is not much different from that of a real PC routine.

  The PC is defined to be entered in the following state:

| | |
|---|---|
| **PSW key:** | 0 |
| **Authorization:** | Supervisor state |
| **AR:** | ASC mode |
| **AMODE:** | 31–bit |

**Registers on entry:**

| **Register** | **Contents** |
|---|---|
| **0-13** | As they were in the PC caller |
| **14** | A return address that can be used by the PC routine |
| **15** | The routine address as set by the PC caller |

The routine does not have to save or restore any registers or state information. This is a stacking PC.

The routine must acquire any working storage that it may need in the primary, colony, address space.

The routine must set up an FRR or ESTAE if it needs any recovery to be run in the colony address space. It will be officially running under an ARR (associated recovery routine), but there will be no recovery done by that ARR.

When it has completed, the routine may either issue a PR instruction to return back to the PC caller, or return to the address that was in Register 14 on entry; that is, issue BR 14.

- The PC caller must beware of the colony address space terminating while it is using the PC. If the colony address space terminates before the PC or during the PC routine's execution, the PC caller will abend.

# Considerations for Internet Protocol Version 6 (IPv6)

## Activating IPv6 on a system

IPv6 is activated on a system with a second NETWORK statement for DOMAINNAME(AF_INET6) with DOMAINNUMBER(19), which arrives at the PFS as a second vfs_network call. If a PFS supports IPv6, it must support both AF_INET and AF_INET6; there are no IPv6-only stacks.

To indicate support for IPv6, a PFS must:

1. Set PfsiIpv6 on during initialization, to indicate that it can receive vfs_network(AF_INET6).
2. Return successfully from that call.

An administrator can add the second NETWORK statement for AF_INET6 dynamically with SETOMVS RESET=. The stack is free to reject the vfs_network if it arrives after initialization. Generally, both vfs_network calls are passed to the PFS during OMVS startup or after a PFS recycles. The vfs_network calls for AF_INET and AF_INET6 may be in any order.

If PfsiIpv6 has not been set, or if the vfs_network for AF_INET6 is not accepted, IPv6 sockets are not opened to that stack. When an application opens an AF_INET6 socket across a Common INET configuration of both IPv6-capable and IPv4-only stacks, an AF_INET socket is opened to the IPv4-only stacks, and a certain amount of address conversion and emulation is performed by CINET for the IPv4-only stack. An IPv6-capable stack must do its own conversions and emulations for any IPv4 partners that it permits on an IPv6 socket.

## Common INET transport driver index

In a multi-stack configuration there can be duplication of interface indices. CINET inserts its transport driver index, TdIndex, into the upper halfword of all output interface indices to identify the interfaces uniquely. On input interface indices, the upper halfword is used to select a stack, and is cleared before the information is passed on to the stack. Each stack's TdIndex value is passed to it in PfsiTdIndex, but the stack does not have to do anything with the value.

For more information about the transport driver index, see the discussion of the SIOCGIFNAMEINDEX ioctl command in w_ioctl (BPX1IOC, BPX4IOC) — Control I/O in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

## ioctl used by the C/C++ Run-Time Library

The if_nameindex(), if_nametoindex(), and if_indextoname() functions use the SIOCGIFNAMEINDEX (Get Interface Name/Index Table) ioctl, which returns the

Interface Name/Index Table for a PFS. The command and output arguments are defined in the BPXYIOCC macro, and are described in the discussion of the SIOCGIFNAMEINDEX ioctl command inw_ioctl (BPX1IOC, BPX4IOC) — Control I/O in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

# ioctls used by the prerouter

The dialog between a stack and the Common INET prerouter for IPv6 is basically the same as the one for IPv4. The prerouter uses these ioctl commands, which are defined in the BPXYIOCC macro:

```
SIOCMSADDRT6     Constant('8044F604'x), /* Add IPV6 Route      */
SIOCMSDELRT6     Constant('8044F605'x), /* Delete IPV6 Route   */
SIOCGRT6TABLE    Constant('C014F606'x), /* Get IPV6 Rte Table  */
SIOCMSRBRT6TABLE Constant('8000F607'x), /* Rebuild Rte & Home  */
SIOCGHOMEIF6     Constant('C014F608'x), /* Get IPV6 HomeIf     */
SIOCMSRBHOMEIF6  Constant('8000F609'x)  /* Rebuild IPV6 HomeIf */
```

The associated argument structures are defined in the BPXYIOC6 macro.

# ioctls used by the resolver

The resolver uses two ioctl commands to get specific information from a stack. These command codes are defined in BPXYIOCC, and the associated argument structures are described as follows:

## SIOCGSRCIPADDR (obtain source IP addresses for an array of IPv6 and IPv4 destination addresses)

SIOCGSRCIPADDR obtains the associated source address (by Source Address Selection algorithm, which is part of the Default Address Selection IETF draft) for each of the IPv6 addresses passed in an array. This information is ultimately used to sort the IPv6 and IPv4 destination addresses, using the algorithm described in the Default Address Selection IETF draft for destination addresses.

**Argument**: An array of IPv6 and IPv4 destination addresses, with a total count of the addresses being passed. Upon return from the IOCTL invocation, the array structure is to include a source IP address (determined by the use of the IETF draft for Default Address Selection) for each of the array elements associated with the destination address that is being passed. This source address is determined by the stack, using the IETF draft for Default Address Selection. If a source address cannot be determined for a specific destination IP address (for example, if there is no route to the destination), a null value is placed in the array element's IP source address field (SisSrcIPaddr).

```
DCL 1 SrcIpSelect Based Bdy(Word),
      2 SisHeader,
        3 SisVersion Fixed(8),     /* Version of the IOCTL interface
                                                             */
        3 *              Char(3), /* Available                */
        3 SisNumEntries   Fixed(32), /* Number of destination
                                   addresses for which a source
                                   address must be selected      */
      2  SisIpAddrs(*),
        3 SisDestIPaddr  Char(16), /* Destination IP address. Can
                                    contain a native IPv6 address,
                                    mapped IPv4 address, or an
                                    IPv4 compatible address      */
          5 SisIpV4prefix   Char(12),  /* IP address prefix        */
            7 SisIpV4nulls    Char(10), /* Always nulls for IPv4
                                    compatible or IPv6 mapped
                                    addresses                    */
```

```
        7 SisIpV4mapped   Char(2), /* IPv6 mapped prefix       */
      5 SisV4DestIPaddr Char(4), /* IPv4 address             */
    3 SisSrcIPaddr   Char(16), /* Associated Source IP address
                                (output from IOCTL)          */
    3 SisRetcode     Fixed(32), /* Return code from attempt to
                                obtain an interface      */
    3 SisSrcAddrFlags Bit(8),  /* Source IP address flags
                                (output from IOCTL           */
      5 SisSrcDeprecated Bit(1), /*  B'1' indicates address is
                                deprecated (only applicable for
                                native IPv6 addresses        */
      5 *             Bit(7),
    3 *              Char(3); /* Available                   */

 DCL SrcIpSelect_Version Fixed(8) Constant(1);
```

### SIOCGIFVERSION (determine if an IPv4 or IPv6 interface has been configured on a TCP/IP stack)

SIOCGIFVERSION determines if a TCP/IP stack in an INET environment has a configured IPv6 or IPv4 source address. (In this case, the loopback address is not considered to be valid as a configured interface.) This information is needed so that appropriate DNS queries can be made (IPv6 address records (AAAA) vs. IPv4 address records (AA)).

**Argument**: A four-byte area containing flags that provide the following information:

```
DCL 1 IfVersionInfo Based,        /* SIOCGIFVERSION structure     */
       2 IfVerFlags Bit(16),       /* Stack flags                  */
         3 IfVerIPv6Interfaces Bit(1), /* Are there any IPv6
                                  interfaces active other than
                                  loopback                      */
         3 IfVerIPv4Interfaces Bit(1), /* Are there any IPv4
                                  interfaces active other than
                                  loopback                      */
         3 IfVerIPv6Supported  Bit(1), /* Is IPv6 supported by this
                                  stack                         */
         3 *                   Bit(13), /* Available             */
       2 *         Char(2);       /* Available                    */
```

## PFS support for multilevel security

To support multilevel security, a PFS must provide the following capabilities:

- **vn_link:**

  If a link is attempted to a character special file, and there is a security label on the file or on the directory for the new link, the vn_link call will fail with EPERM. If the ZCredSeclablActive flag is on, the following checks should be done:

  1. If zCredSeclablRequired is on and the object has no security label, the zCredROSeclabel should be used as the object security label for all subsequent checks.
  2. If the directory for the new link has a security label of SYSMULTI, no further security label checking is necessary.
  3. If the directory for the new link has no security label, or has a security label other than SYSMULTI, a check for equality must be done between the security label of the directory and the security label of the file. If the values are equal, no further security label checking is necessary.
  4. If the equality check fails, a dominance check must be made to check that the security label of the directory and the security label of the file are equivalent. The call to check security label equivalence should look like this:

     ```
     RACROUTE REQUEST=DIRAUTH,RSECLABEL=(x),TYPE=EQUALMAC,USERSECLABEL=(y)
     ```

where x and y are registers that contain the addresses for the security labels.

- **vn_readdir:**

  If the ZCredSeclablActive flag is set, the following checks should be done:

  1. If zCredSeclablRequired is on and the directory has no security label, the zCredROSeclabel should be used as the object security label for all subsequent checks.
  2. If the directory has a security label of SYSMULTI, a dominance for read should be made between the user's security label and the security label of each entry in the directory. The user's security label is passed in the ZCredSeclabel field. If the security label of the directory entry is SYSMULTI or SYSLOW, the dominance check can be bypassed. If the dominance check fails, the directory entry should be excluded from the output buffer. The dominance check should look like this:

     ```
     RACROUTE REQUEST=DIRAUTH,RSECLABEL=(x),ACCESS=READ,USERSECLABEL=(y)
     ```

     where x and y are registers that contain the addresses for the security labels.

  **Notes:**

  1. The PFS may cache object security labels to avoid rechecking for labels that have already passed the dominance check. A good cache is likely to result in a single check for each unique security label per readdir call.
  2. No indication will be returned from the PFS if some names were excluded from the output buffer.
  3. Discrepancy between the apparent number of entries in a directory and the number that can be read is acceptable.
  4. The LFS will not filter names based on security label when it does a readdir2 for a PFS that does not support security labels. Any PFS that supports security labels must also support readdir2.
  5. When the index method is used to read a directory, the meaning of the index is not the relative name in the directory, but the relative name that the user can access. For example, if the request is to return entries beginning with entry 10, the PFS must start at the first entry and verify dominance on each name until the 10th name that the user is permitted to see is found, and start returning names that can be seen from that point.

- **vn_readlink:**

  If the zCredSeclablActive flag is set, the following checks should be done:

  1. If zCredSeclablRequired is on and the directory has no security label, the zCredROSeclabel should be used as the object security label for all subsequent checks. If this flag is on, and the resulting object security label continues to be null because no value was provided by zCredROSeclabel, vn_readlink should return with a failure of EACCES.
  2. A dominance check should be performed between the user's security label and the security label of the symbolic link. The user's security label is passed in the zCredSeclabel field. If the security label of the directory entry is SYSMULTI or SYSLOW, the dominance check can be bypassed. If the dominance check fails, the vn_readlink should return with a failure of EPERM. The dominance check should look like this:

     ```
     RACROUTE REQUEST=DIRAUTH,RSECLABEL=(x),ACCESS=READ,USERSECLABEL=(y)
     ```

     where x and y are registers that contain the addresses for the security labels.

- **vn_setattr:**

If the AttrSeclabelChg flag is set, a call to the SAF callable service IRRSSB00 should be made to set the security label for the file. The new security label is passed in the zCredSeclabel field, which is passed to SAF. The PFS does not have to access the new or the old security label.

## PFS support for 64-bit virtual addressing

The entry environment and parameters for the vnode and VFS operations are the same for 31-bit and 64-bit addressing. The PFS is always entered in AMODE 31, with a 31-bit parameter list address in R1 that points to a parameter list of 31-bit addresses. All calling parameters are below the 2-gigabyte line, although some of these parameters may contain 64-bit addresses of areas that are above the 2-gigabyte line.

The main consideration for 64-bit addressing is the user data buffers, which may require 64-bit addressing in the UIO, IOV, and MSGH structures. In general, the other user parameters are copied into the kernel below the 2-gigabyte line, and these copies are passed to the PFS.

The data length parameter for read and write-type operations with 64-bit addressing remains 31 bits long.

## Levels of support for 64-bit virtual addressing

From the point of view of the LFS, there are three levels of PFS support for 64-bit virtual addressing: None, 64-bit supporting, and 64-bit exploiting.

- **None:**

  The PFS has no understanding of 64-bit addresses. The LFS copies 64-bit addressable user data to an internal 31-bit addressable buffer before it invokes the PFS for write-type operations, and vice versa for reads.
- **64-bit supporting:**

  The PFS can handle 64-bit user virtual addresses, or it makes use of the OSI services that can. It does not itself use buffers above the 2-gigabyte line or run in AMODE 64, at least not to the knowledge of the LFS.
- **64-bit exploiting:**

  The PFS supports 64-bit user virtual addresses. It may run in AMODE 64 and have its own data buffers, or even autodata, above the 2-gigabyte line. Some considerations for these PFSs are:
  - Unless otherwise specified, the OSI service routines expect to be called in AMODE 31, with a 31-bit parameter list address and 31-bit parameter addresses. The calling interface may have to be manually constructed below the 2-gigabyte line.
  - The SAF (RACF) services do not support 64-bit callers or addresses.
  - MVS WAIT and POST services do not support ECBs above the 2-gigabyte line.

**Recommendation:** A PFS should be at least 64-bit supporting, in order to avoid the extra LFS data move that is otherwise required for high user buffers.

## Indicating support for 64-bit virtual addressing

A PFS indicates support for 64-bit user virtual addressing during initialization with:

```
pfsi_addr64    Indicates the PFS supports 64-bit user virtual addresses in
               the UIO, IOV, and MSGH structures.   PfsiAddr64  in PL/X.
```

A user indicates 64-bit addressing to the PFS with the following fields and structures:

```
u_addr64       Indicates that this UIO, and any associated IOV and/or MSGH
               when present, uses 64-bit addresses.   FuioAddr64 in PL/X.
u_buff64vaddr  A 64-bit field that contains the virtual address of the area
               being passed.   FuioBuff64VAddr in PL/X.
```

The IOV and MSGH structures have corresponding 64-bit formats, IOV64 and MSGH64.

When an application program in AMODE 64 calls a z/OS UNIX service, 64-bit user addressing is assumed and is used by the LFS. This does not necessarily mean that the 64-bit address values are actually greater than 2 gigabytes. Most 64-bit addresses will come from C programs that have been compiled with LP64, which makes all longs and pointers 64 bits by default, regardless of whether the program's heap is above the 2-gigabyte line.

### osi_copy64 routine

The OSI routine osi_copy64 ("osi_copy64 — Move data between user and PFS buffers with 64-bit addresses" on page 376) helps a PFS deal with 64-bit addresses. It takes 64-bit virtual addresses and operates in much the same way as osi_copyin and osi_copyout. osi_copy64 is a high-performance routine that does not PC into the kernel. It handles 31- or 64-bit user and PFS buffer addresses for AMODE 31 or AMODE 64 PFS callers.

## Minimum 64-bit support

The minimum needed by a PFS to be 64-bit supporting is:

- If the only data moves to or from the user address space are done with osi_uiomove, the PFS just needs to set pfsi_addr64 during initialization.
- If osi_copyin or osi_copyout are used for user buffers, the PFS must check the FuioAddr64 flag at each of these calls, and use osi_copy64 or osi_uiomove whenever this flag is on.
- If the PFS does its own MVCSKs and MVCDKs, it must check the FuioAddr64 flag at each of these locations and handle moves with 64-bit addresses; or call osi_copy64 or osi_uiomove at these points. Doing your own moves is, of course, fastest.

## Specific considerations for vnode operations

The following vnode operations contain parameters that may contain 64-bit addresses or point to structures that contain 64-bit addresses. Each of these operations has **Fuio** as an input parameter, which may point to a 64-bit user buffer:

- vn_rdwr
- vn_readdir
- vn_readlink
- vn_sndrcv
- vn_sendtorcvfrom
- vn_readwritev—the IO vectors passed may be in an IOV or an IOV64 structure.
- vn_srmsg—the message header passed may be an MSGH or an MSGH64 structure.

**Notes:**

1. MSGH64 and IOV64 are always used together.
2. Whenever FuioAddr64 is on (and FuioRealPage is off):
   - FuioBuff64Vaddr points to a buffer, an IOV64, or an MSGH64.
   - A MSGH64 always points to an IOV64.

## Expanded 64-bit time values

As part of the POSIX standards for 64-bit computing, known as LP64 (64-bit Longs and Pointers), the **time_t** data type for file times is expanded to 64 bits in z/OS V1R6. The current signed 31-bit data type will go negative in 2038. Because the 390 system clock will wrap in 2042, there is an issue for PFSs that store time in STCK format.

The z/Architecture™ has a 128-bit STCKE that adds one byte to the left of the current 8-byte format; that is, it has five bytes of "seconds", and goes to about the year 36765. An 8-byte POSIX time value goes far beyond that. A 9-byte time field, or the left 8 bytes of the new STCKE, would hold any real times, and an 8-byte POSIX format field would hold anything that could be set by a user.

### C/C++ Run-Time Library support

C/C++ Run-Time Library supports old 31-bit programs and new LP64 programs with a stat structure that contains 4-byte and 8-byte time fields for all five file time values: the POSIX atime, mtime, ctime; and the z/OS UNIX reference time and create time. The old fields could not be expanded in place without changing the offset of all the following fields; new fields were therefore added to the end. When a C program is compiled without LP64, the stat structure is generated with the POSIX names (such as st_atime) on the 4-byte fields; and when it is compiled with LP64, those names coincide with the new 8-byte fields. The unused fields in each compile have dummy names that would not be referenced by the average C program.

There are two separate run-time libraries, compiled from the same source with and without LP64, so that even the RTL will not reference both field types at the same time.

### PFS support

The kernel supports 31-bit and 64-bit programs with the same routines. The PL/X stat structure, BPXYSTAT, has both fields generated; the new fields have new names. BPXYATTR ("BPXYATTR — Map file attributes for v_ system calls" on page 445) also has five new 8-byte time fields:

```
3 AttrEndVer1     Char(0),     /* +A0--- End of Version 1 --- @D2C*/

 3 AttrStat4 ,                 /* +A0 Fourth part of the stat @DAA*/
   5 AttrLP64 ,                /* +A0 LP64 Versions           @DAA*/
     7 AttrAtime64     Char(8),     /*+A8 Access Time          @DAA*/
     7 AttrMtime64     Char(8),     /*+B0 Data Mod Time        @DAA*/
     7 AttrCtime64     Char(8),     /*+B8 Medadata Change Time @DAA*/
     7 AttrCreateTime64  Char(8),   /*+C0 File Creation Time   @DAA*/
     7 AttrRefTime64   Char(8),     /*+C8 Reference Time       @DAA*/
     7   *             Char(8),     /*+A0 May be AttrIno64     @DAA*/

   5 *              Char(16),  /* +D0 Reserved (1st consider  @DAA
                                    space at +5C,+8D,+94)  @DAA*/
 3 AttrEndVer2 Char(0),        /* +E0 End of Version 2       @DAA*/
```

The associated 4- and 8-byte fields will usually contain the same values, until some time in the year 2038.

The C ATTR structure in BPXYVFSI exactly matches the PL/X Attr:

```
                           /* +A0 --- End Ver 1 ---   @P5A*/
 char    at_atime64[8];     /* Large Time Fields       @P5A*/
 char    at_mtime64[8];                              /*@P5A*/
 char    at_ctime64[8];                              /*@P5A*/
 char    at_createtime64[8];                         /*@P5A*/
 char    at_reftime64[8];                            /*@P5A*/
```

```
 char    at_rsvd4[8];                               /*@P5A*/
 char    at_rsvd5[16];                              /*@P5A*/
                            /* +E0 --- End Ver 2 ---   @P5A*/
```

| PFSs must return both sets of time fields in all output ATTRs. This includes vn_getattr, any osi_attrs, and ReadDirPlus (part of "v_readdir (BPX1VRD, BPX4VRD) — Read entries from a directory" on page 326). The LFS always passes to the PFSs an ATTR that is large enough to hold the 8-byte times (at least of length Attr#Ver2Len). The **stat()** function is performance-sensitive, because it is called so often by programs in the field, and it is faster for the PFSs to set the five extra fields than for the LFS to check to see if it has been done, and then copy the 4-byte values to the 8-byte fields.

| PFSs that support vn_setattr, or setting times at all, must accept 8-byte time values. The AttrLP64Times bit in BPXYATTR indicates that the time value is being passed in the 8-byte fields. Most of these 8-byte time values will still be less than 2 gigaseconds, but they are being passed by LP64 programs. An LP64 program may try to utime() beyond 2 gigaseconds.

PFSs that use BPXXCTME should use the new syntax for large time values. The BPXXCTME macro converts to and from the extended STCKE TOD format with the optional EXTENDED keyword:

```
?BPXXCTME INPUT(TOD|SSE)
    TOD(8ByteArea|16ByteArea)
    SSE(WordArea|DWordArea)
    MICSEC(WordArea)
    EXTENDED(8<,4>|16<,4>) (optional)
```

INPUT indicates the input field, and TOD is a doubleword-aligned 8- or 16-character field containing the input TOD or the converted value. SSE is a word-aligned 4-byte character field or doubleword-aligned 8-byte character field containing the input SSE or the converted value. Table 2 shows the TOD and SSE fields with the EXTENDED keyword:

*Table 2. TOD and SSE fields with the EXTENDED keyword*

| EXTENDED | TOD | SSE |
|---|---|---|
| Keyword is omitted | Bytes 1 through 8 of the STCK format | A 4-byte character field |
| EXTENDED(8) | Bytes 1 through 8 of the STCKE format | An 8-byte field |
| EXTENDED(16) | Bytes 1 through 16 of the STCKE format | An 8-byte field |
| EXTENDED(16,4) | Bytes 1 through 16 of the STCKE format | A 4-byte field |

# Chapter 3. PFS operations descriptions

This chapter describes each PFS operation, which are arranged in alphabetic order. The C language prototypes and definitions for these operations can be found in Appendix D, "Interface structures for C language servers and clients," on page 503. Assembler definitions are in Appendix B, "Mapping macros," on page 443.

## Environment for PFS operations

Each PFS operation (vfs_ and vn_ functions) operates in the following environment:

### Environment at entry

| | |
|---|---|
| **Authorization:** | Supervisor state, PSW key 0 |
| **Dispatchable unit mode:** | Task or SRB, if the PFS has indicated that it supports SRB-mode callers. You cannot assume that vfs or vn routines receive control under the same dispatchable unit as the requestor of the related callable service. For example, **unmount()** and **sync()** do not. |
| **Cross memory mode:** | Any |
| **AMODE:** | 31-bit |
| **ASC mode:** | Primary mode |
| **Interrupt status:** | Enabled for interrupts |
| **Locks:** | Unlocked |
| **Control parameters:** | All parameters are in key 0 storage in the primary address space. They are not fetch protected. |

### Registers at entry

The contents of the registers on entry to this operation are:

| **Register** | **Contents** |
|---|---|
| **0** | Undefined |
| **1** | Parameter list address |
| **2-12** | Undefined |
| **13** | Save area address, of a 136-byte save area |
| **14** | Return address |
| **15** | Entry address |
| **AR0-15** | Undefined |

### Environment at exit

Upon return from this operation, the entry environment must be restored.

### Registers at exit

Upon return from this operation, the register contents must be:

| **Register** | **Contents** |
|---|---|
| **2-13** | Restored from the entry values |
| **0,1,14,15** | Undefined |
| **AR0-15** | Untouched or restored from the entry values |

© Copyright IBM Corp. 1996, 2006

## C header files

The C header files that are referred to in this section (such as **stat.h**) can be found in *z/OS XL C/C++ Run-Time Library Reference*.

# vfs_batsel — Select/poll on a batch of vnodes

## Function

The vfs_batsel operation monitors activity on a batch of vnodes (multiple vnodes) to see if they are ready for reading or writing, or if they have an exceptional condition pending. The vnodes can be for a socket, pipe, regular, or pseudoterminal file.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vfs_batsel  (Token_structure,
             OSI_structure,
             Audit_structure,
             Reserved_1,
             Function,
             Batch-Select_Structure
             Reserved_2,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) being operated on. It contains the PFS’s initialization token and mount token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Reserved_1**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The value 0. This parameter is reserved to maintain consistency with the vn_select operation interface.

**Function**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that specifies whether this is a batch-select query or a batch-select cancel request, and whether it is a poll or a select request. The values for this field are defined in the BPXYPFSI header file (see Appendix D).

Function specifies the type of select that is being requested:

- Query (SEL_BATSELQ or SEL_BATPOLLQ): The PFS should perform the following for query:
  1. Check each of the files in the Batch-Select_Structure to see if any of the specified events for a file can be satisfied immediately. If so, the BSIC Response fields for those files are updated, and the status for any one of them is returned in the Return_value parameter.
  2. If there is no immediate status to report for any file in the Batch-Select_Structure, the PFS records that a select is pending for each of the files and sets up to invoke osi_selpost later, when one of the selected events has occurred. The PFS returns a value of 0 in Return_value after it has performed its internal processing to set up select pending for each of the files.

     The occurrence of an event and the subsequent invocation of osi_selpost happen asynchronously on another thread or MVS task.
- Cancel (SEL_BATSELC or SEL_BATPOLLC): The PFS performs the following for cancel:
  1. If there is a pending select recorded for a file with the same SelectToken that was specified on a previous query, it must be canceled in such a way that osi_selpost is not invoked.
  2. Check each of the files that are specified in the Batch-Select_Structure to see if any of the specified events can be immediately satisfied. If at least one file has status, that status is returned in the Return_value parameter, and the status for each of the selected files is returned in the BSIC Response fields for those files. If a file does not have status, a 0 is returned in the BSIC Response field for that file. If none of the files have status, 0 is returned in the Return_value parameter.

**Batch-Select_Structure**
Returned parameter

| | |
|---|---|
| **Type:** | BSIC |
| **Length:** | Calculated: A BSIC header plus one BSIC entry for each selected file. |

An area that contains information about the selected files and events. It specifies which files and events are being selected, a SelectToken for each file, a response area for status, and work area pointers for use by the PFS. This area is mapped by the BSIC typedef in the BPXYPFSI header file (see Appendix D). The events that can be selected for select requests are:

- **SEL_READ:** A read that is issued against this file will not block.
- **SEL_WRITE:** A write that is issued against this file will not block.
- **SEL_XCEPT:** An exceptional condition, as defined by the particular PFS, has occurred. This could happen when a socket connection becomes inoperative because of network problems, or when the other end of the socket is closed.

For poll requests, the events that can be selected are documented in other manuals (for instance, *z/OS XL C/C++ Run-Time Library Reference*). The mapping for these fields is defined in the BPXYPFSI header file (see Appendix D).

For reading and writing, an error condition that would cause the read or write to fail means that the operation will not block and therefore the file is ready for that operation.

If one or more of the selected events are ready for any of the selected files, the PFS immediately returns the status for one of the files in the Return_value parameter, using the same bit mapping that is used in the BSIC Response field.

**Reserved_2**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The value `0`. This parameter is reserved, to maintain consistency with the vn_select operation interface.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vfs_batsel service returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. This causes the whole **select()** or **poll()** request, as made by the application program, to fail. The Return_code and Reason_code values are passed back to the application program. |
| **0** | There is no status for any of the files in the Batch-Select_Structure, and the operation was successful.<br>• For query (**SEL_BATSELQ** or **SEL_BATPOLLQ**): The PFS is set up to invoke osi_selpost when the requested event occurs.<br>• For cancel (**SEL_BATSELC** or **SEL_BATPOLLC**): The PFS has canceled the request to invoke osi_selpost, or it was |

never set up to do so. The PFS will not invoke osi_selpost after returning from this call.

**Greater than 0**
Status is being returned in the Batch-Select_Structure. The returned status in this parameter has the same format as the BSIC Response field.

- For query (**SEL_BATSELQ** or **SEL_BATPOLLQ**): The operation is complete and the PFS will not invoke osi_selpost for this request.
- For cancel (**SEL_BATSELC** or **SEL_BATPOLLC**): The PFS has canceled the request to invoke osi_selpost if it had been recorded.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_batsel operation stores the return code. The vfs_batsel operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_batsel operation stores the reason code. The vfs_batsel operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vfs_batsel processing

The vfs_batsel operation is identical to the vn_select operation, except that a batch of files (multiple files) are selected using the Batch-Select_Structure, instead of only one. For information on vn_select, refer to “Select/poll processing” on page 45.

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications mentioned in “Finding more information about sockets” on page xvi for the select function.

### Specific processing notes

– On the query request, the PFS should save the BSIC SelectToken for each file passed in the Batch-Select_Structure. This token is used both during the cancel request (to delete the request) and when an event occurs that the LFS should be informed of through the osi_selpost function.

– The PFS can use the BSIC entry workptr field in the Batch-Select_Structure to save information about each file during a query request. It can also use the BSIC header workptr field to save information about the entire query (such as an address where it has stored information about this request) so that it can be found during a cancel request. The data is used to correlate the cancel request with its matching query request. This provides an alternative to scanning the PFS control blocks for matching SelectToken values.

**Serialization provided by the LFS**: None

**Security calls to be made by the PFS**: None.

## Related services

- "vn_select — Select or poll on a vnode" on page 204

# vfs_gethost — Get the socket host ID or name

## Function

The vfs_gethost operation gets the ID or the name of the socket host.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vfs_gethost (Token_structure,
             OSI_structure,
             Audit_structure
             Name_length,
             Name,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS's initialization token and mount token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Name_length**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of the name. If this value is zero, the request is for the host ID. Otherwise, this is the length of the buffer to hold the name. On return, for host name, this field contains the length of the name plus one for the null.

**Name**
Returned parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Name_length |

An area that contains the name on return, if the host name was requested. This name must be null-terminated by the PFS.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_gethost operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful (for getting the host name). |
| **Greater than 0** | The operation was successful (for getting the host ID) and is the identifier of the current host. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_gethost operation stores the return code. The vfs_gethost operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_gethost operation stores the reason code. The vfs_gethost operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vfs_gethost processing**

For more information on the semantics of this operation, refer to the publications mentioned in "Finding more information about sockets" on page xvi for the **gethostid()** and **gethostname()** functions.

**Specific processing notes**

The PFS determines whether to get the host name or host ID depending on Name_length. A zero length indicates a **gethostid()** request.

**Serialization provided by the LFS**

The vfs_gethost operation is invoked with an exclusive latch held on the domain of the PFS.

**Security calls to be made by the PFS**: None.

# vfs_inactive — Batch inactivate vnodes

## Function

The vfs_inactive disassociates multiple vnodes from the PFS's related inodes.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vfs_inactive (Token_structure,
              OSI_structure,
              Audit_structure,
              InactBuffer_structure,
              InactBuffer_length,
              Return_value,
              Return_code,
              Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

**Type:** TOKSTR
**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS's initialization token, and mount token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**InactBuffer_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | IAB |
| **Length:** | Calculated: An IAB header plus one IAB entry for each selected vnode. |

The InactBuffer_structure contains information about the vfs and the vnodes that are to be made inactive. This area is mapped by the IAB typedef in the BPXYPFSI header file (Appendix D).

This structure contains the following fields:

**Server_devno**
A fullword that contains the device number of this vfs.

Each **Server_devno** is followed by an array of records containing the following information:

| | |
|---|---|
| **Vnode_pointer** | A pointer to the vnode. |
| **Pfs_token** | An eight-byte area that contains the pfs token for this vnode. |
| **Server_Vnode** | A pointer to the server's vnode. |
| **Return_Value** | A fullword in which the vfs_inactive operation returns the results of the operation for the vnode. A nonzero value indicates that the operation was not successful. |

**InactBuffer_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that supplies the length of the InactBuffer_structure.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vfs_inactive service returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_inactive service stores the return code. The vfs_inactive service returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vfs_inactive service should support the following error value:

| Return_code | Explanation |
|---|---|
| EIO | An I/O error occurred while accessing the file. |

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vfs_inactive service stores the reason code. The vfs_inactive service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

# Implementation notes

**Overview of vfs_inactive processing**

“Creating, referring to, and inactivating file vnodes” on page 31 provides an overview of file inactivate processing.

**Specific processing notes**

– The Return_value for each vnode that is being made inactive is returned in the InactBuf_structure while the results of the vfs_inactive service is provided in the returned parameters.

– If a transient error, such as an I/O error, is encountered, the Return_value should be set to −1. In this case, the request is retried later.

– If a permanent error that prevents the specified file or directory from being used is encountered, Return_value should be set to zero. In this case, all references to the file or directory are removed from the LFS and the request is not retried. The PFS must not issue a signal-enabled wait during inactivate processing. “Waiting and posting” on page 21 provides an overview of wait and post processing.

– If a file’s link count is zero, but its open count is not zero, the PFS should ignore the open count and delete the file’s data along with the file. This might happen, for example, when an address space is canceled right after vn_open finishes in the PFS, but before the LFS regains control.

**Serialization provided by the LFS**

The vfs_inactive operation is invoked with an exclusive latch held on the file system containing the vnode.

**Security calls to be made by the PFS**: None.

# Related services

- “osi_wait — Wait for an event to occur” on page 431
- “vn_inactive — Inactivate a vnode” on page 151

# vfs_mount — Mount a file system

## Function

The vfs_mount operation activates a file system and returns the root directory vnode_token.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vfs_mount   (Token_structure,
             OSI_structure,
             Audit_structure,
             Mount_table,
             Vnode_token,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS's initialization token and mount token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Mount_table**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the MTAB.mtab_hdr.cblen field |

An area that is used to pass the file system name, mount options, and PFS-specific parameters to the vfs_mount operation. This area is mapped by the MTAB typedef in the BPXYPFSI header file (see Appendix D).

**Vnode_token**
Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

An area in which the vfs_mount service returns the vnode_token for the root directory of the mounted file system.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_mount service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| −1 | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| 0 | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_mount service stores the return code. The vfs_mount service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vfs_mount operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EEXIST | A file system with the same name has already been mounted. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_mount service stores the reason code. The vfs_mount service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vfs_mount processing**

"Mounting file systems" on page 27 provides an overview of file system mount processing.

**Specific processing notes**

– The PFS is responsible for the following fields:

**token_structure.ts_mount**
The PFS should fill in this field with a token that it can use to locate the PFS structures that are associated with the mounted file system. On subsequent calls for files within this file system, the token_structure value contains the token set here by the PFS.

**MTAB.mt_filesys or MTAB.mt_ddname**
On entry to the PFS, the field MTAB.mt_filesys contains either the blank padded file system name or nulls. On a successful return, if this field is not nulls and it represents an MVS data set name, the field MTAB.mt_ddname should be filled in by the PFS with the dynamically allocated ddname.

If the field MTAB.mt_filesys is nulls on entry to the PFS, the field MTAB.mt_ddname contains the ddname of an allocated MVS data set for the file system. On a successful return, the field MTAB.mt_filesys should be filled in by the PFS with the MVS data set name that is specified on the DD statement.

If every file in this file system has the same values, the PFS is responsible for filling in the MTAB with the following pathconf values (see the IEEE POSIX 1003.1 specification for further details):

| | |
|---|---|
| **MTAB.mt_linkmax** | LINK_MAX |
| **MTAB.mt_namemax** | NAME_MAX |
| **MTAB.mt_notrunc** | POSIX_NO_TRUNC |
| **MTAB.mt_chownrstd** | POSIX_CHOWN_RESTRICTED |

Alternatively, the PFS may meet this responsibility by supporting vn_pathconf.

– The PFS must not issue a signal-enabled wait under the thread invoking vfs_mount.

– "Waiting and posting" on page 21 provides an overview of wait and post processing.

– If the mount is to be completed asynchronously:

  - The PFS must set MTAB.mt_asynchmount on before returning to the LFS. The LFS in turn sets MTAB.mt_asynchmount on before calling the PFS for the second call to vfs_mount.
  - When the mount operation has completed, the PFS indicates this to the LFS by calling osi_mountstatus.

- The vnode_token must be returned on at least one of the calls to vfs_mount. However, if the PFS chooses to return a nonzero vnode_token on each call, it must be the same token.
- If asynchronous mount processing in the PFS fails, the PFS should call osi_mountstatus to drive the second call to vfs_mount. When called by the LFS to complete the mount, the PFS should then return the error to the LFS, which deletes all references to the incompletely mounted file system. No call to vfs_umount results.
- If MTAB.mt_synchonly is set on in the Mount_table, vfs_mount must either complete the mount synchronously or reject the request, returning EINVAL. MTAB.mt_synchonly is always set on for the system root and for mounts that result from MOUNT statements in BPXPRMxx that specify DDNAME.
- Vfs operations, such as vfs_umount and vfs_statfs, may need to be handled during an asynchronous mount.

– It is not necessary for the PFS to perform security checking during mount processing, because the LFS has already performed all necessary checking.

– The PFS returns an aggregate name, if it has one, from the vfs_mount operation. If mt_aggnameptr is not zero, it points to mt_aggname, which is a 45-byte area where the PFS can put the aggregate name. If the PFS may run on an earlier release, it should test for mt_hdr.cblen > 0x80 before it tests mt_aggnameptr. If read-only mounts of file systems with the same aggregate name should be function shipped to the owning system rather than locally mounted, mt_aggattachrw should be turned on. If subsequent recovery of this mount should not attempt to attach the aggregate before issuing the vfs_mount, mt_agghfscomp should be turned on.

**Serialization provided by the LFS**

The vfs_mount operation is invoked with an exclusive latch held on the file system, to ensure that no other operations are attempted upon the file system being mounted. In addition, the LFS ensures that all vfs_mount and vfs_umount calls are serialized.

**Note:** However, if the mount is asynchronous, there is a time between the start and the end of the mount in which the latch is not held.

**Security calls to be made by the PFS**: None.

## Related services


- “vfs_unmount — Unmount a file system” on page 106
- “vn_pathconf — Determine configurable pathname values” on page 173
- “osi_getvnode — Get or return a vnode” on page 385
- “osi_ctl — Pass control information to the kernel” on page 379
- “osi_wait — Wait for an event to occur” on page 431

# vfs_network — Define a socket domain to the PFS

## Function

The vfs_network operation is called as a result of the NETWORK statement in the BPXBPRMxx parmlib member that is used to start z/OS UNIX. It defines information about a socket domain to the PFS that is supporting it.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vfs_network (Token_structure,
             OSI_structure,
             Audit_structure,
             Network_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS's initialization token and mount token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Network_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | NETW |
| **Length:** | Specified by netw.netw_hdr.cblen |

The Network_structure is an area, built during initialization, that contains the information that is included on the NETWORK statement—the socket domain name and number and the maximum number of sockets. This area is mapped by the NETW typedef in the BPXYPFSI header file (see Appendix D).

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_network operation returns the results of the operation as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_network operations stores the return code. The vfs_network operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vfs_network operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EAFNOSUPPORT | The address family that was specified in the Network_structure is not supported by this PFS. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_network operation stores the reason code. The vfs_network operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vfs_network processing

For information concerning the vfs_network call, refer to “Activating a domain” on page 43.

**Specific processing notes**

The PFS should ensure that it does not do any blocking waits during its processing.

The PFS is responsible for returning two fields set so that they can be used for subsequent processing. These fields are:

**NETW.nt_localremote**
An indication of whether the communication done by this PFS is local or remote. Turn the bit on to indicate remote communication.

**TOKSTR.ts_mount**
The 8-byte token that is returned by the PFS and used on all subsequent calls to this PFS. This token is used by the PFS to locate the PFS structures that are associated with this network.

**Serialization provided by the LFS**

The logical file system ensures that only one vfs_network statement is processed at a time. Further, the PFS does not receive any socket requests specifying this domain until the vfs_network operation completes.

**Security calls to be made by the PFS**: None.

# vfs_pfsctl — PFS control

## Function

The vfs_pfsctl operation passes control information to the PFS.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vfs_pfsctl  (Token_structure,
             OSI_structure,
             Audit_structure,
             Command,
             User_IO_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure contains the PFS’s initialization token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Command**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The command indicates the function that is to be performed by the PFS.

**User_IO_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the UIO.u_hdr.cblen field |

An area that is to be used by the vfs_pfsctl service to determine the buffer address, length, storage key, and other attributes of the argument that is passed by the caller of pfsctl (BPX1PCT). This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D).

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_pfsctl operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |
| **0 or greater** | Can be used by the PFS to return the length of the information that is being returned in a modified argument buffer. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_pfsctl operation stores the return code. The vfs_pfsctl operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vfs_pfsctl operation should support at least the following error values when the situation applies:

| **Return_code** | **Explanation** |
|---|---|
| EMVSPARM | The command or argument parameters are incorrect. |
| EFAULT | The address of the argument buffer is incorrect, or the user is not authorized to read or write to that location. |
| EINTR | The service was interrupted by a signal. |
| EPERM | Permission was denied. The calling program does not have sufficient authority for the service that was requested. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_pfsctl operation stores the reason code. The vfs_pfsctl operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS product.

## Implementation notes

**Overview of vfs_pfsctl processing**

This function is like vn_ioctl, except that the data is directed to the PFS itself rather than to, or for, a particular file.

A program can communicate with the PFS through the pfsctl (BPX1PCT) callable service, which is converted by the LFS into vfs_pfsctl. An example of this would be a program that is provided with a particular PFS product that displays performance statistics for that PFS.

You should avoid passing addresses with this service, and instead include all data in the buffer.

Negative command values are reserved for use by the LFS.

Command values of less than 0x40000000 are considered to be authorized functions, and a privilege check is made. See "Security calls to be made by the PFS".

For more information, see *z/OS DFSMS Using Data Sets*.

**Specific processing notes**

The token_structure of this operation contains only the initialization token.

The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the argument buffer |
| **UIO.u_buffalet** | Specifies the ALET of the argument buffer |
| **UIO.u_count** | Specifies the length of the argument buffer |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_key** | Specifies the storage key of the argument buffer |

**Serialization provided by the LFS**: None.

**Security calls to be made by the PFS**

None expected by the LFS.

When the command value is less than 0x40000000, the LFS calls SAF's Check Privilege callable service to determine if the caller has appropriate privileges before it invokes the PFS with vfs_pfsctl. The results of this call are passed to the PFS using the osi_privileged bit.

If the osi_privileged bit is *on*, the user has appropriate privileges. If the PFS wishes to restrict this function or certain command values, it can check this bit.

## Related services

None.

# vfs_recovery — Recover resources at end-of-memory

## Function

The vfs_recovery operation permits a PFS to recover resources when a user address space enters end-of-memory processing while a request to that PFS is active.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vfs_recovery  (Token_structure,
               OSI_structure,
               Audit_structure,
               Recovery_area,
               Return_value,
               Return_code,
               Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS's initialization token and mount token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Recovery_area**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | 8 bytes |

A copy of the Recovery_area that was filled in by the PFS during the operation that was interrupted. This area is mapped by osirtoken in BPXYPFSI (see Appendix D).

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_recovery operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_recovery operation stores the return code. The vfs_recovery operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_recovery operation stores the reason code. The vfs_recovery operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

# Implementation notes

### Overview of vfs_recovery processing

“Recovery considerations” on page 24 provides an overview of recovery processing, and discusses the flow for vfs_recovery in particular.

### Specific processing notes

When an active request to the PFS is interrupted in a situation where normal ESTAE processing is bypassed by MVS, the PFS may have resources, such as storage and locks, that are left in a state that will cause problems for other users.

To allow the PFS a chance to clean up if this should happen, a Recovery_area is passed on every operation, through the osi_rtokptr pointer in the OSI_structure, where the PFS can record its resources or store a pointer to a recovery block. Any information that is stored in this area by the PFS during an operation is passed back to the PFS via the Recovery_area parameter of vfs_recovery if the operation is interrupted by end-of-memory for the user address space.

The OSI Work Area and the Pre-initialized C Environment Stack, if used, are still addressable and left as they were at the time of the abend. These areas can be used to hold a recovery block whose address is placed in the Recovery_area. The vfs_recovery operation is invoked with its own areas like any other operation.

Refer also to "vn_recovery — Recover resources after an abend" on page 190, which is the operation that is invoked during normal ESTAE processing.

There is no EOM recovery for the vfs_recovery operation itself. The operation is invoked with osi_rtokptr pointing to a new recovery area that can be used for standard PFS abend recovery; that is, with vn_recovery.

The PFS is not called if the file system has been unmounted between the original vnode operation and the running of the EOM resource manager. This can only happen if the user was in a signal-enabled wait at the time the address space was terminated. It is expected that the PFS has cleaned up all its file-system-related resources during vfs_umount.

See also the OSI and osirtoken structures in Appendix D.

The state of any file-level objects that may have been involved with the interrupted operation is unknown at the time vfs_recovery is invoked.

**Serialization provided by the LFS**

The vfs_recovery operation is invoked with a shared latch held on the file system represented by the token_structure.

Any file-level objects that may have been involved with the interrupted operation are not serialized.

**Security calls to be made by the PFS**: None.

# vfs_socket — Create a socket or a socket pair

## Function

The vfs_socket operation creates one socket or two related sockets.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vfs_socket  (Token_structure,
             OSI_structure,
             Audit_structure,
             Domain,
             Type,
             Protocol,
             Array_dimension,
             Vnode_token_array,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS’s initialization token and mount token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Domain**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains a number that represents the address family the socket is to be created for. The values defined for this field are mapped by **socket.h**.

**Type**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains a number that represents the socket type. The values defined for this field are mapped by **socket.h**.

**Protocol**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains a number that represents the protocol to be used with the socket.

**Array_dimension**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that specifies the number of Vnode_tokens to get. The allowable values for this field are 1 (for the socket call) and 2 (for the socketpair call).

**Vnode_token_array**
Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 16 bytes |

A two-element array that contains the one or two Vnode_tokens obtained.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_socket operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_socket operation stores the return code. The vfs_socket operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vfs_socket operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EAFNOSUPPORT | The address family that is specified by Domain is not supported by this PFS. |
| EINVAL | The socket type that was specified is not supported; or the Array_dimension that was specified is incorrect. If the PFS does not support the **socketpair()** call, an Array_dimension of 2 is incorrect. |
| EPROTONOSUPPORT | The protocol that was specified is not supported. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_socket operation stores the reason code. The vfs_socket operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vfs_socket processing**

For more information on the semantics of this operation, refer to the publications that are mentioned in “Finding more information about sockets” on page xvi for the **socket()** and **socketpair()** functions. Also refer to “Creating, referring to, and closing socket vnodes” on page 44 for general information on sockets.

**Specific processing notes**

If the PFS does not support socketpair(), the LFS simulates this function by creating and connecting two separate sockets. This is done in response to a Return_Code of EINVAL when Array_dimension is two.

**Serialization provided by the LFS**

The vfs_socket operation is invoked with a shared latch held on the domain of the PFS.

**Security calls to be made by the PFS**: None.

## Related services

- “vn_close — Close a file or socket” on page 132

# vfs_statfs — Get the file system status

## Function

The vfs_statfs operation returns status information about a mounted file system.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vfs_statfs  (Token_structure,
             OSI_structure,
             Audit_structure,
             Fsattr_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS's initialization token and mount token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Fsattr_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | FSATTR |
| **Length:** | Specified by FSATTR.fs_hdr.cblen |

An area in which the vfs_statfs operation returns the file system status information. This area is mapped by the FSATTR typedef in the BPXYVFSI header file (see Appendix D).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_statfs operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_statfs operation stores the return code. The vfs_statfs operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_statfs operation stores the reason code. The vfs_statfs operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vfs_statfs processing

The vfs_statfs operation returns information about the status of the file system.

To account for different release levels, the PFS should zero out the FSATTR area and set fields it understands only up to the smaller of:

– the input area's length, from the FSATTR length subfield

– the PFS's native FSATTR length (the one with which it was compiled)

The input area's FSATTR length subfield should be updated to reflect the amount of data that is returned, or zeroed out. The PFS must not refer to fields beyond the input FSATTR's length, as specified in its length subfield.

**Specific processing notes**

– The value that is returned in FSATTR.fs_hdr.cblen must match the amount of valid data that is returned in the Fsattr_structure.

– When a Return_Value of 0 is returned, the PFS is responsible for returning valid data in at least the following fields in the FSATTR:
  - FSATTR.fs_blocksize
  - FSATTR.fs_totalspace
  - FSATTR.fs_usedspace
  - FSATTR.fs_freespace

– vfs_statfs may be called before the mount process completes for a file system that is being mounted asynchronously. If the PFS is unable to provide valid data, the PFS must return a Return_value of -1, along with a Return_code of EAGAIN.

**Serialization provided by the LFS**

The vfs_statfs operation is invoked with a shared latch held on the mounted file system.

**Security calls to be made by the PFS**: None.

# vfs_sync — Harden all file data for a file system

## Function

The vfs_sync operation writes to disk (or otherwise stabilizes) all changed data in a buffer cache for files in a mounted file system.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vfs_sync    (Token_structure,
             OSI_structure,
             Audit_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) being operated on. It contains the PFS's initialization token and mount token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Return_value**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword where the vfs_sync service returns the results of the operation as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vfs_sync service stores the return code. The vfs_sync service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vfs_sync service should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EROFS | The file system is mounted read-only. |

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vfs_sync service stores the reason code. The vfs_sync service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS product.

## Implementation notes

### Overview of vfs_sync processing

vfs_sync writes to non-volatile storage (usually disk) all modified data for each file in the mounted file system that is indicated by the mount token in the input token_structure. The PFS can use the synch daemon to synchronize modified data at regular intervals, by specifying the desired interval in the MTAB during the mount operation.

A PFS could perform vfs_sync processing asynchronously, although this is not recommended. The osi_usersync flag in the OSI can be set to indicate to the PFS that the vfs_sync request is the result of a user request, rather than a timer pop. If this bit is set, the PFS must complete vfs_sync processing before it returns from the call.

To allow for timer-driven cleanup, vfs_sync is called for readonly file systems also.

**Specific processing notes**

Data should be completely hardened before vfs_sync returns to its caller.

**Serialization provided by the LFS**

The vfs_sync operation is invoked with an exclusive latch held on the mounted file system.

**Security calls to be made by the PFS**: None.

## Related services

- "vn_fsync — Harden file data" on page 142

# vfs_unmount — Unmount a file system

## Function

The vfs_unmount operation unmounts a file system and inactivates the root vnode.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vfs_umount  (Token_structure,
             OSI_structure,
             Audit_structure,
             Unmount_options,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS’s initialization token and mount token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6, “OSI services,” on page 367 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Unmount_options**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

An area that is used to pass the options that are to be used to unmount the file system that is specified in Token_structure. The values for this parameter are defined in the **stat.h** header. For a description of this header, see *z/OS XL C/C++ Run-Time Library Reference*.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_unmount service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_unmount service stores the return code. The vfs_unmount service returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vfs_unmount operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EIO | An I/O error occurred while the file system was being unmounted. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vfs_unmount service stores the reason code. The vfs_unmount service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vfs_unmount processing

“Unmounting file systems” on page 29 provides an overview of file system unmount processing.

**Specific processing notes**

The PFS cannot issue a signal-enabled wait during unmount processing. "Waiting and posting" on page 21 provides an overview of wait and post processing.

It is not necessary for the PFS to perform security checking during unmount processing, because the LFS has already performed all necessary checking.

A file system that is being mounted asynchronously may be unmounted before the mount process completes. Consequently, if the PFS returns only the vnode_token on the second call to vfs_mount, vfs_unmount must be capable of successfully unmounting a file system without reference to its inode token.

If vfs_umount is being invoked for a remount (MT_REMOUNT or OSI_REMOUNT), the PFS receives a vfs_mount for the same file system as soon as the vfs_umount completes. This is followed by vfs_vgets to recreate the vnode-inode pairs that were active at the time of the unmount operation. If a file was open at the time of the remount, the vnode's open counter is reestablished through calls to vn_open.

The PFS does not have to do anything special for remount; however, for performance reasons, it may want to maintain some resources at vfs_umount in anticipation of reusing them for the next vfs_mount. Socket or RPC sessions are examples of resources that might be worth maintaining.

If the PFS cannot support remount, it should reject the vfs_umount request. One reason for not supporting remount is that the PFS would not complete the following vfs_mount synchronously.

**Serialization provided by the LFS**

The vfs_unmount operation is invoked with an exclusive latch held on the file system, to ensure that no other operations are attempted upon the file system that is being unmounted. In addition, the LFS ensures that all mount and unmount operations are serialized.

**Security calls to be made by the PFS**: None.

## Related services

- "osi_wait — Wait for an event to occur" on page 431
- "vfs_mount — Mount a file system" on page 84

# vfs_vget — Convert a file identifier to a vnode Token

## Function

The vfs_vget operation returns a vnode token for the file or directory that is represented by the input file identifier (FID).

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vfs_vget    (Token_structure,
             OSI_structure,
             Audit_structure,
             File_identifier,
             Vnode_token,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file system (VFS) that is being operated on. It contains the PFS’s initialization token and mount token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**File_identifier**
Supplied parameter

| | |
|---|---|
| **Type:** | FID |
| **Length:** | 8 bytes |

The name of an 8-byte area containing the file identifier of the file or directory for which a vnode token is to be returned. This area is mapped by the FID typedef in the BPXYVFSI header file (see Appendix D).

**Vnode_token**
Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

Vnode_token is used to return the vnode token that corresponds to the input FID.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vfs_vget service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. This causes the vfs_vget request to fail. The Return_code and Reason_Code are returned to the caller. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vfs_vget service stores the return code. The vfs_vget service returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vfs_vget service should support at least the following error values:

| Return_code | Explanation |
|---|---|
| ENOENT | The file indicated by the File_identifier does not exist in the mounted file system that is indicated by token_structure |
| EIO | An input/output error occurred while attempting to access data pertaining to the file indicated by the File_identifier. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vfs_vget service stores the reason code. The vfs_vget service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vfs_vget processing**

Given a file identifier as input, vfs_vget returns a vnode token that refers to the file. The file identifier uniquely identifies a file in a particular mounted file system. Its validity persists across mounting and unmounting of the file system, as well as z/OS UNIX re-IPLS. This distinguishes the file identifier from the vnode token, which relates to a file in active use, and whose validity persists only until the token is released via vn_inactive. The FID for a file is created by the PFS and returned in the ATTR structure, which is mapped by typedef ATTR in the BPXYVFSI header file (see Appendix D) by vn_getattr.

**Specific processing notes**

File identifier zero is taken to refer to the root of the mounted file system.

**Serialization provided by the LFS**

The vfs_vget operation is invoked with a shared latch held on the mounted file system.

**Security calls to be made by the PFS**: None.

## Related services

- "vn_getattr — Get the attributes of a file" on page 145

# vn_accept — Accept a socket connection request

## Function

The vn_accept operation accepts a connection request for a socket server from a socket client. It returns a new socket descriptor.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_accept   (Token_structure,
             OSI_structure,
             Audit_structure,
             Sockaddr_length,
             Sockaddr,
             Open_flags,
             Vnode_token,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Sockaddr_length**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that supplies the length of the Sockaddr buffer and returns the length of the Sockaddr structure that is returned.

**Sockaddr**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | SOCK |
| **Length:** | Specified by Sockaddr_length |

A structure that varies depending on the address family type. On return, it contains the address that was used for this operation. For an example of this mapping for AF_INET, see **in.h**.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that contains the bits that are associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**Vnode_token**
Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

An area in which a token that represents the newly created socket is returned.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_accept operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_accept operation stores the return code. The vn_accept operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_accept operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EINTR | The request was interrupted by a signal. |
| EINVAL | An incorrect request, such as a socket for which a listen has not been issued (that is, a server), was received. |
| EWOULDBLOCK | The operation would have required a blocking wait, and this socket was marked as nonblocking. |

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword where the vn_accept operation stores the reason code. The vn_accept operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_accept processing**

– For more information on vn_accept, refer to "Creating, referring to, and closing socket vnodes" on page 44. For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications mentioned in "Finding more information about sockets" on page xvi for the accept function.

– The vn_accept service can be used from a multithreaded server, that is, a server with several threads simultaneously calling accept() on the same socket. The PFS must handle queuing for vn_accept requests on the same socket that are waiting to be satisfied. When a connection arrives it is given to one of the waiting vn_accept requestors. All the server threads are expected to be equal; their requests may be satisfied in any order.

**Serialization provided by the LFS**

The vn_accept operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

## Related services

- "vn_listen — Listen on a socket" on page 160

# vn_access — Check access to a file or directory

## Function

The vn_access operation checks whether the calling process has the requested access permission to the specified file or directory.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_access   (Token_structure,
             OSI_structure,
             Audit_structure,
             Access_intent,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Access_intent**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

An input structure passed through to the SAF Check Access callable service by the vn_access operation. The values for this parameter are defined in **unistd.h.**

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_access service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_access service stores the return code. The vn_access service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_access operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EACCES | The caller does not have the requested access to the specified file or directory. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_access service stores the reason code. The vn_access service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

# Implementation notes

### Overview of vn_access processing

“Security responsibilities and considerations” on page 12 provides an overview of file access checking.

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **access()** function in the POSIX.1 standard (IEEE Std 1003.1-1990).

**Specific processing notes**

The PFS should provide reason codes that distinguish between the SAF reason codes:

– User is not authorized to access the file.

– Input that is not valid.

**Serialization provided by the LFS**

The vn_access operation is invoked with a shared latch held on the vnode.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF's Check Access callable service to check that the user has the requested access to the file or directory.

# vn_anr — Accept a socket connection and read the first block of data

## Function

The vn_anr operation accepts a connection request for a socket server from a socket client, and reads the first block of data.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_anr      (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             Acp_token,
             User_IO_structure
             Anr_addrs
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that supplies the bits associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**Acp_token**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

An area that is used in one of two ways:

- The LFS passes the PFS’s token for a reusable socket.
- The LFS passes a value of 0, and the PFS returns the Vnode token for a new accepted socket.

**User_IO_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the buffer parameters for the receive operation that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See “Specific processing notes” for details on how the fields in this structure are processed.

**Anr_addrs**

Supplied parameter

| | |
|---|---|
| **Type:** | struct anr_addrs |
| **Length:** | sizeof(anr_addrs) |

A structure that describes the remote and local socket addresses. This structure contains the following fields:

| **Field** | **Description** |
|---|---|
| **Remote_sockaddr_length** | A fullword that supplies the length of the Remote_sockaddr buffer that is pointed to by Remote_sockaddr_ptr. On return, this parameter contains the length of the socket address that was put in the Remote_sockaddr buffer.<br><br>If the value of Remote_sockaddr_length is 0, the Remote_sockaddr is not to be returned. |
| **Remote_sockaddr_ptr** | A pointer to the Remote_sockaddr buffer. On return, this buffer contains the socket address of the remote socket that has just connected. |

| | |
|---|---|
| **Local_sockaddr_length** | A fullword that supplies the length of the Local_sockaddr buffer that is pointed to by Local_sockaddr_ptr.<br>On return, this parameter contains the length of the socket address that was put in the Local_sockaddr buffer.<br>If this value is 0, the Local_sockaddr is not to be returned. |
| **Local_sockaddr_ptr** | A pointer to the Local_sockaddr buffer. On return, this buffer contains the socket address of the new local socket that was just created. |

**Return_value**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_anr operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was either not successful or, when Return_code is EINTRNODATA, partially successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful; the value represents the number of bytes that were transferred. |

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_anr operation stores the return code. The vn_anr operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_anr operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EFAULT | The address of one of the buffers is not in addressable storage. |
| EINTR | A signal arrived before a connection was assigned to this request. |
| EINTRNODATA | A signal arrived after a connection was assigned to this request, but before any data had arrived. The connection has been established. The result of this call is equivalent to a successful vn_accept.<br>This condition does not occur in a PFS that does not assign arrived connections to a vn_anr request until some data has also arrived. |
| EINVAL | An incorrect parameter was specified. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword where the vn_anr operation stores the reason code. The vn_anr operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_anr processing**:

The vn_anr operation is a functional combination of the vn_accept and vn_rdwr operations, in that an inbound connection is accepted to create a new socket and the first block of data is read on that socket. The output is the new connected socket and the data.

The vn_anr operation is generated from an application call to the accept_and_recv callable service (BPX1ANR). The accept_and_recv callable service is designed to work with the send_file service (BPX1SF) to provide an efficient file transfer capability for connection-oriented servers with short connection times and high connection rates. See accept_and_recv (BPX1ANR, BPX4ANR) — Accept a connection and receive the first block of data in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for more information on how this is used.

The vn_anr operation is intended to be used from a multithreaded server, that is, a server with several threads simultaneously calling accept_and_recv() on the same socket. The PFS must handle queuing for vn_anr requests on the same socket that are waiting to be satisfied. When a connection and its first data have arrived, the connection and data are given to one of the waiting vn_anr requesters. All of the server threads are expected to be equal, and their requests may be satisfied in any order. In particular, LIFO order would reduce the serialization necessary to manage the requester queue.

The PFS does not complete the vn_anr operation until the first data has arrived on the new connection or a signal arrives for this thread. The listening socket must be in blocking mode; this requirement is enforced by the LFS.

When socket reuse is supported by the PFS, the Acp_token parameter is used to pass the PFS's token for the socket that is being reused. When reuse is not supported, or when a reusable socket is not supplied by the application, the Acp_token parameter is used to return the vnode token of the new socket that is created. In this case, the input Acp_token is 0, and the output Acp_token is basically the same as the Vnode_token parameter of the vn_accept operation.

A PFS that does not support socket reuse does not have to be coded to reject vn_anr requests that attempt to reuse a socket. A reusable socket is one that has been closed by a prior write-type operation that specified the REUSE flag. If the PFS does not honor the REUSE flag, it is assumed that the PFS does not support reuse, and the socket is closed in the normal way. Consequently, the Acp_token parameter would be 0 on a subsequent vn_anr request.

Because the vn_anr operation is a combined operation, it can be interrupted between the connection arrival and the data arrival. If the PFS irrevocably associates a new connection to a vn_anr request before any data has arrived and is subsequently interrupted by a signal, it may return the connection via Acp_token, and set a Return_value of −1 and a Return_code of EINTRNODATA. It is strongly recommended that the PFS not assign connections to vn_anr requests until data has arrived, because doing so ties up a server's worker threads while the PFS is waiting for the data to arrive.If an application uses both accept() and accept_and_recv() calls on the same socket from several threads at the same time, the results are allowed to be unpredictable. Depending on PFS design and timing, the vn_accept and vn_anr calls may be satisfied in any order. Because it is not recommended that connections be assigned to vn_anr requests until the first data has arrived, it is possible that vn_accept requests could consume all arriving connections.

### Specific processing notes

The following UIO fields are provided by the LFS:

**UIO.u_hdr.cbid**
Contains UIO_ID (from the BPXYVFSI header file).

**UIO.u_hdr.cblen**
Specifies the length of the UIO.

**UIO.u_buffadr**
Specifies the address of the user's buffer.

**UIO.u_count**
Specifies the size of the user's buffer.If this value is `0`, no read is done, and vn_anr is functionally equivalent to vn_accept. In this case, the rest of the UIO fields should be ignored.

**UIO.u_asid**
Specifies the ASID of the user.

**UIO.u_rw**
Set to `0`, specifying a read request.

**UIO.u_key**
Specifies the storage key of the caller.

The Remote_sockaddr, Local_sockaddr, and data buffer are all optional.

### Serialization provided by the LFS

The vn_anr operation is invoked with an exclusive latch held on the listening vnode if latching is requested by this PFS.

### Security calls to be made by the LFS

None.

## Related services

- "vn_listen — Listen on a socket" on page 160

# vn_audit — Audit an action

## Function

The vn_audit operation audits the action that is indicated by the audit_structure.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_audit    (Token_structure,
             OSI_structure,
             Audit_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter
**Type:** TOKSTR
**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter
**Type:** OSI
**Length:**
Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter
**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Return_value**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vn_audit operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vn_audit operation stores the return code. The vn_audit operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vn_audit operation stores the reason code. The vn_audit operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_audit processing**

– The vn_audit operation calls the SAF Audit interface to write an audit record.
– The Audit_structure contains a code that identifies the function that is being audited, defined in IRRPAFC.

**Serialization provided by the LFS**

The vn_audit operation is invoked with a shared latch held on the vnode of the file.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF’s Audit callable service to write the audit record.

# vn_bind — Bind a name to a socket

## Function

The vn_bind operation associates a name with a socket.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_bind     (Token_structure,
             OSI_structure,
             Audit_structure,
             Sockaddr_length,
             Sockaddr,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6, “OSI services,” on page 367 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Sockaddr_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of sockaddr.

**Sockaddr**
Supplied parameter

| | |
|---|---|
| **Type:** | SOCK |
| **Length:** | Specified by Sockaddr_length |

A structure that varies depending on the address family type. It contains the address that is to be used for this operation. For an example of this mapping for AF_INET, see **in.h**.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_bind operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_bind operation stores the return code. The vn_bind operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_bind operation should support at least the following error values:

| **Return_code** | **Explanation** |
|---|---|
| EAFNOSUPPORT | The address family that was specified is not supported. |
| EINVAL | The length of the name is either too short or negative. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword where the vn_bind operation stores the reason code. The vn_bind operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_bind processing

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in “Finding more information about sockets” on page xvi for the **bind()** function.

### Specific processing notes

An “unbind” flag can be passed in the first word of the system data area of the Token_structure, ts_sysdl. A value of 1 in this word indicates that the socket should be reset to an unbound state, if possible, when a **bind()** has succeeded on some transports but failed on others. If the socket can be reset to an unbound state, then a subsequent vn_bind call with a different Sockaddr might be successful. If the socket cannot be unbound, the call will be rejected. All other parameters are the same as on a successful vn_bind call. There is no external application interface for this function; it is used internally by the Common INET (CINET) layer so that CINET can try to place an application socket back into a state where another call to **bind()** may succeed.

### Serialization provided by the LFS

The vn_bind operation is invoked with an exclusive latch held on the vnode of the socket.

### Security calls to be made by the PFS

When a program specifies a port value less than 1024 decimal, the PFS must call SAF’s Check Privilege function to verify that the caller has the authority to do so.

# vn_cancel — Cancel an asynchronous operation

## Function

The vn_cancel operation cancels the wait for an asynchronous operation to complete, or cancels the remaining portion of an operation after the I/O completion has been scheduled.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_cancel  (Token_structure,
            OSI_structure,
            Audit_structure,
            VnCan_Flags,
            PFS_AsyTok,
            LFS_AsyTok,
            Return_value,
            Return_code,
            Reason_code)
```

## Parameters

**Token_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6, "OSI services," on page 367 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503.

**Audit_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**VnCan_Flags**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | 4 bytes |

Control flags for this cancelation. Refer to the vncanflags structure in BPXYPFSI.

- **vncanforce:** This flag specifies whether a normal or forced cancelation is being requested:

  *0*– **Normal Cancel**.
  Only the wait for completion is being canceled; otherwise the operation is to proceed normally.

  If the PFS finds the request on a waiting queue, it is to be removed from the queue and completed with a return code of ECANCELED. That is, osi_sched should be called, and the normal flow for a failed request should be followed. Note that if it is the PFS's custom to handle asynchronous failures in Part 2, it may call osi_sched with success and return ECANCELED from the Part 2 call.

  If the PFS does not find the request on a waiting queue, it should take no action whatsoever. The request is completing, or has completed, normally and should not be interrupted.

  *1*– **Forced Cancel and Cleanup**.
  Part 2 is not run for this operation, usually because the user's process is terminating. The PFS should remove the request from any waiting queues, and discard all buffers and other resources that were allocated to this request. Regardless of whether the request was found on the waiting queues, the PFS must clean up the request if it is still active.

**PFS_AsyTok**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | 8 bytes |

A copy of the PFS's Asynchronous I/O Request Token, which identifies the request that is being canceled.

This is the token that was originally passed by the PFS to the LFS via a call to osi_upda during Part 1 of the asynchronous operation. This is also the same token that is passed in osi_asytok on Part 2 of an asynchronous operation to identify the request to the PFS.

**LFS_AsyTok**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | 8 bytes |

A copy of the LFS's Asynchronous I/O Request Token, which was originally passed to the PFS in the osi_asytok field on Part 1 of the request that is being canceled.

This token has presumably been saved by the PFS in its request structure during Part 1, since it is needed for osi_sched, and can be used to validate the PFS request structure. The PFS's original request structure must be validated on vn_cancel because the original operation might have finished by the time the vn_cancel reaches the PFS, and therefore its request structure might have been already freed or reused for another operation. Once cancel is started for a request, the LFS does not reuse its token until after the cancel has completed.

The PFS may also, of course, perform this validation on its own and ignore the LFS_AsyTok if it is so designed. A request structure could be validated, for instance, with a structure sequence number that is included within the PFS_AsyTok, or by running a chain of active request blocks.

See also the notes below.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_cancel operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **0** | The request was found. |
| **−1** | The request was not found. |

Generally, vn_cancel is not called after osi_sched has been called, but there is a race condition between these two acts and so this Return_value is really not very definitive. See the notes below.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_cancel operation stores the return code. The vn_cancel operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_cancel operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EINVAL | The PFS_AsyTok is not valid. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_cancel operation stores the reason code. The vn_cancel operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_cancel processing**

“Asynchronous I/O processing” on page 55 provides an overview of asynchronous I/O, and discusses the flows that are related to vn_cancel.

**Specific processing notes**

– Vn_cancel is for one specific request only.
– Only requests that originally had OsiAsy1=ON are potentially cancelable.
– The normal cancel only “pushes through” the original request, and does not attempt to abort it if it is not blocked.
– Vn_cancel is not an asynchronous operation in the sense of the OsiAsy1 and OsiAsy2 bits. It is also not normally a blocking operation. If the original request is found on a waiting queue it may be removed, and osi_sched() called, on another thread while vn_cancel returns to the LFS.
– Vn_cancel must contend with situations in which a thread may be calling osi_sched, or an SRB may be running Part 2 of the original request. This can be a problem in either case, if the PFS is about to free up the structures that are related to the original request and the PFS_AsyTok. Hopefully, the original request structure can be validated or not used directly, in order to avoid introducing additional serialization points into the main line path just to deal with a potential cancel. For instance, for a normal cancel, only requests that are found on a waiting chain need be referenced directly, and for cancel force some cleanup may be able to be deferred to vn_close.

  Technically, though, because of fork() and inherited descriptors, vn_cancel(Force) might not soon be followed by vn_close. However, it would be rather rare for an application in this position to carry on. The results of the application would be unpredictable because of timing; and at a minimum it would have to expect data loss, since the termination could just as easily have occurred on entry to its I/O Completion exit.

  Vn_cancel(Force) is a result of process termination; therefore, any requests that were still in the PFS have gone through recovery and generally have been handled, as they would be for any abnormal end situation.

  - Part 1 requests run on the user’s TCB or SRB, and these are abnormally ended before vn_cancel is issued.
  - For process termination in general, new SRBs are not permitted to start Part 2, but old SRBs are allowed to finish. Osi_wait(), though, returns as if interrupted with a signal, in an attempt to keep these SRBs from blocking. If the user address space goes to memterm, nothing is able to run, so Part 2 can be abnormally ended for that reason. If the PFS issues its own MVS suspend during Part 2, it can also be abnormally ended by the system.

**Serialization provided by the LFS**

The vn_cancel operation is invoked with an exclusive vnode latch.

Additional serialization is provided even when the PFS is not using vnode latching.

1. The vn_cancel operation is not invoked while the request it is canceling is still in the PFS during Part 1 of the operation.
2. Vn_close is not invoked while vn_cancel is in progress.
3. If a user process terminates before osi_upda is called, vn_cancel is not called, since the LFS does not have the PFS’s token to pass.
4. The LFS serializes vn_cancel with the potentially simultaneously occurring end of Part 2 on the SRB, so the PFS does not have to in any sense ″wait″ within vn_cancel for Part 2.

**Security calls to be made by the PFS**: None.

# vn_close — Close a file or socket

## Function

The vn_close operation closes a file or socket.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_close    (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**

Supplied parameter

**Type:** TOKSTR

**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**

Supplied and returned parameter

**Type:** OSI

**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**

Supplied parameter

**Type:** CRED

**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Bit |
| **Length:** | Fullword |

A fullword containing the open flags that are associated with this file. These flags are defined by **fcntl.h.**

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_close operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_close operation stores the return code. The vn_close operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_close operation stores the reason code. The vn_close operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_close processing

See “Opening and closing files and first references to files” on page 34 for a discussion of close processing.

See “Creating, referring to, and closing socket vnodes” on page 44 for a discussion of relevant socket processing.

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **close()** function in the POSIX .1 standard (IEEE Std 1003.1-1990).

### Specific processing notes

1. The Return_value parameter is preset to -1 before the PFS is called. If the PFS program checks or ends abnormally during the vn_close operation and the abend is percolated back to the LFS, the LFS uses the Return_value to determine what to do next. If the Return_value is still -1, the PFS is recalled with vn_close; otherwise it is not. Therefore, just before the PFS reaches a point at which it would rather not be recalled if it should end abnormally, it should zero out the Return_value.
2. If the PFS supports vn_recovery, and vn_recovery returns control information to direct the outcome of the original call, the rule above is overridden. That is, vn_close is not recalled if it appears that vn_recovery has handled the problem, regardless of the value of Return_value.
3. Although the Return_value, Return_code, and Reason_code values are returned to the caller, the operation always succeeds in that the user's file descriptor is freed and the vnode's open counter is decremented, regardless of the Return_value.
4. If vn_inactive is not supported by the PFS, the LFS will free its vnode after the vn_close returns. If vn_inactive is supported, the LFS keeps the vnode for a few minutes and then invokes vn_inactive, at which time the vnode is freed.

   For sockets PFSs, the total number of vnodes in use is used to enforce the MAXSOCKETS limit. Thus, for sockets PFSs that use vn_inactive, it is possible for a heavily loaded system to reach its MAXSOCKETS limit—even though not that many sockets are open—because of closed vnodes that have not yet been inactivated.

   Refer to "Creating, referring to, and closing socket vnodes" on page 44 for more information on socket close and inactivation.

**Serialization provided by the LFS**

The vn_close operation is invoked with an exclusive latch held on the vnode of the file. Shared read support for the file that is being closed can be modified in the OSI by the PFS upon returning from the vn_close operation.

**Security calls to be made by the PFS**: None.

## Related services


- "vn_open — Open a file" on page 170
- "vfs_socket — Create a socket or a socket pair" on page 97

# vn_connect — Connect to a socket

## Function

The vn_connect operation connects to a socket. The socket can be either a stream socket or a datagram socket. The connection is done for stream sockets by a client; a bind and a listen request must have preceded this request at the server.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_connect  (Token_structure,
             OSI_structure,
             Audit_structure,
             Sockaddr_length,
             Sockaddr,
             Open_flags,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for the mapping of this structure.

**Sockaddr_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of sockaddr.

**Sockaddr**
Supplied parameter

| | |
|---|---|
| **Type:** | SOCK |
| **Length:** | Specified by Sockaddr_length |

A structure that varies depending on the address family type. It contains the address that is to be used for this operation. For an example of this mapping for AF_INET, see **in.h**.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that contains the bits that are associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_connect operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_connect operation stores the return code. The vn_connect operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_connect operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| ECONNREFUSED | The connection request was rejected. |
| EINTR | The request was interrupted by a signal. |
| EINVAL | The length of the name specified was too short, or negative. |
| EISCONN | The socket is already connected. |
| ENOAFSUPPORT | The PFS does not support this address family. |
| EOPNOTSUPP | The socket that was specified is a server; a listen has been done. |
| EPROTOTYPE | The request is for an incorrect socket type. |
| EWOULDBLOCK | The operation would have required a blocking wait, and this socket was marked as nonblocking. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_connect operation stores the reason code. The vn_connect operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_connect processing**

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in “Finding more information about sockets” on page xvi for the connect function.

**Specific processing notes**

The **connect()** function performs a different action for each of the following types of initiating sockets:

– If the initiating socket is SOCK_DGRAM, the **connect()** function establishes the peer address. The peer address identifies the socket to which all datagrams are sent on subsequent **send()** functions. No connections are made by this **connect()** function.

– If the initiating socket is SOCK_STREAM, the **connect()** function attempts to make a connection to the socket that is specified by the Sockaddr parameter.

**Serialization provided by the LFS**

The vn_connect operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

## Related services

- “vn_listen — Listen on a socket” on page 160
- “vn_accept — Accept a socket connection request” on page 112
- “vn_bind — Bind a name to a socket” on page 125

# vn_create — Create a new file

## Function

The vn_create operation creates a new file using the file type and attributes that are provided by the caller.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_create   (Token_structure,
             OSI_structure,
             Audit_structure,
             Name_length,
             Name,
             Attribute_structure,
             Vnode_token,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter
**Type:** TOKSTR
**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter
**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503.

**Audit_structure**
Supplied parameter
**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of Name. The name is between 1 and 255 bytes long.

**Name**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Name_length |

An area, of length Name_length, that contains the name of the file that is to be created. This name is not null-terminated.

**Attribute_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | ATTR |
| **Length:** | Specified by ATTR.at_hdr.cblen. |

An area that is to be used by the vn_create operation to set the attributes of the file that is to be created. This area is mapped by typedef ATTR in the BPXYVFSI header file (see Appendix D).

**Vnode_token**
Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

An area in which the vn_create operation returns the vnode token that is created.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_create operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_create operation stores the return code. The vn_create operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_create operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EACCES | The caller does not have write permission for the parent directory. |
| EEXIST | A file with the same name already exists. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_create operation stores the reason code. The vn_create operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_create processing**

“Creating files” on page 32 provides an overview of file creation processing.

**Specific processing notes**

– The token structure that is passed on input represents the directory in which the file is created.

– The following attribute_structure fields are provided by the LFS:

| | |
|---|---|
| **ATTR.at_hdr.cbid** | Contains Attr_ID (from the BPXYVFSI header file) |
| **ATTR.at_hdr.cblen** | Specifies the length of the attribute_structure |
| **ATTR.at_mode** | Specifies the file type and permission bits. See the ATTR typedef in Appendix D for the mapping of this field.<br><br>The user’s file creation mask, umask() value, has already been applied to the permission bits. |
| **ATTR.at_major** | Specifies the major number for character-special files. This is provided only when the file type is character-special. |
| **ATTR.at_minor** | Specifies the minor number for character-special files. This is provided only when the file type is character-special. |

– If the file that is named in the Name parameter already exists, the vn_create operation returns a return code of EEXIST, and the output vnode_token is optional.

**Serialization provided by the LFS**

The vn_create operation is invoked with an exclusive latch held on the vnode of the parent directory.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF's Check Access callable service to verify that the user has write permission to the directory. The PFS is also expected to invoke SAF's Make FSP callable service to create a file security packet.

## Related services


- "osi_getvnode — Get or return a vnode" on page 385
- "vn_remove — Remove a link to a file" on page 194

# vn_fsync — Harden file data

## Function

The vn_fsync operation writes to disk (or otherwise stabilizes) all changed data in a file.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_fsync    (Token_structure,
             OSI_structure,
             Audit_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_fsync service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_fsync service stores the return code. The vn_fsync service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_fsync service should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EINVAL | The operation is not possible for the specified file. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_fsync service stores the reason code. The vn_fsync service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_fsync processing

For the file token in the token_structure, vn_fsync must write all modified data that is not yet placed in nonvolatile storage to such a medium.

### Specific processing notes

- Data should be completely hardened before vn_fsync returns to its caller.
- For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **fsync()** function in the POSIX .1a standard (IEEE Std 1003.1a), draft 7.

### Serialization provided by the LFS

The vn_fsync operation is invoked with an exclusive latch held on the vnode of the file.

**Security calls to be made by the PFS**: None.

## Related services

- "vfs_sync — Harden all file data for a file system" on page 103

# vn_getattr — Get the attributes of a file

## Function

The vn_getattr operation gets the attributes of a file.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_getattr  (Token_structure,
             OSI_structure,
             Audit_structure,
             Attribute_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter
**Type:** TOKSTR
**Length:**
Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token.

Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter
**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter
**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Attribute_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | ATTR |
| **Length:** | Specified by ATTR.at_hdr.cblen. |

An area used by the vn_getattr operation to return the file attributes for the file that is specified by the vnode token. Before a call to vn_getattr, Attribute_structure must be initialized with the ID and length fields set correctly and the unused fields set to zero. This area is mapped by typedef ATTR in the BPXYVFSI header file (see Appendix D).

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vn_getattr service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_getattr service stores the return code. The vn_getattr service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_getattr service stores the reason code. The vn_getattr service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_getattr processing

vn_getattr is used to read file attributes, as described in “Getting and setting attributes” on page 39.

### Specific processing notes

– The input attribute_structure length may not match the length that is supported by the PFS. The PFS must return the minimum of:
  - Input ATTR.at_hdr.cblen

- The attribute_structure length that is supported by this release of the PFS

  The returned value in ATTR.at_hdr.cblen must match the size returned.

– Time-related fields that are marked for update must be updated before the attributes are returned.

**Serialization provided by the LFS**

The vn_getattr operation is invoked with a shared latch held on the vnode of the directory.

**Security calls to be made by the PFS**: None.

## Related services

- “vn_setattr — Set the attributes of a file” on page 213

# vn_getname — Get the peer or socket name

## Function

The vn_getname operation gets the peer name or the socket name.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_getname  (Token_structure,
             OSI_structure,
             Audit_structure,
             Name_type,
             Sockaddr_length,
             Sockaddr,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Name_type**
Supplied parameter
**Type:** Integer
**Length:** Fullword

A fullword that specifies whether to get the peer name or the socket name. The values for this field are defined in the BPXYPFSI header file (see Appendix D).

**Sockaddr_length**
Supplied and returned parameter
**Type:** Integer
**Length:**
Fullword

A fullword that supplies the length of the Sockaddr buffer, and returns the length of the Sockaddr structure that is returned.

**Sockaddr**
Supplied and returned parameter
**Type:** SOCK
**Length:**
Specified by Sockaddr_length

A structure that varies depending on the address family type. On return, it contains the address that was used for this operation. For an example of this mapping for AF_INET, see **in.h**.

**Return_value**
Returned parameter
**Type:** Integer
**Length:**
Fullword

A fullword in which the vn_getname operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

A fullword in which the vn_getname operation stores the return code. The vn_getname operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_getname operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EINVAL | The length of the name that was specified is too short. |
| ENOTCONN | The socket is not connected for a getpeername request. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_getname operation stores the reason code. The vn_getname operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_getname processing**

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in “Finding more information about sockets” on page xvi for the getpeername and getsockname functions.

**Serialization provided by the LFS**

The vn_getname operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

# vn_inactive — Inactivate a vnode

## Function

The vn_inactive disassociates a vnode from the PFS's related inode.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_inactive  (Token_structure,
              OSI_structure,
              Audit_structure,
              Return_value,
              Return_code,
              Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vn_inactive service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_inactive service stores the return code. The vn_inactive service returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_inactive service should support the following error value:

| Return_code | Explanation |
|---|---|
| EIO | An I/O error occurred while accessing the file. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_inactive service stores the reason code. The vn_inactive service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_inactive processing

“Creating, referring to, and inactivating file vnodes” on page 31 provides an overview of file inactivate processing.

### Specific processing notes

– If a transient error, such as an I/O error, is encountered, the Return_value should be set to −1. In this case, the request is retried later.

– If a permanent error that prevents the specified file or directory from being used is encountered, Return_value should be set to zero. In this case, all references to the file or directory are removed from the LFS and the request is not retried. The PFS must not issue a signal-enabled wait during inactivate processing. “Waiting and posting” on page 21 provides an overview of wait and post processing.

– If a file’s link count is zero, but its open count is not zero, the PFS should ignore the open count and delete the file’s data along with the file. This might happen, for example, when an address space is canceled right after vn_open finishes in the PFS, but before the LFS regains control.

**Serialization provided by the LFS**

The vn_inactive operation is invoked with an exclusive latch held on the file system containing the vnode.

**Security calls to be made by the PFS**: None.

## Related services


- “osi_wait — Wait for an event to occur” on page 431
- “vfs_inactive — Batch inactivate vnodes” on page 81

# vn_ioctl — I/O control

## Function

The vn_ioctl operation conveys a command for a file or device driver. The specific commands that are supported are defined by the PFS.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_ioctl    (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             Command,
             Argument_length,
             Argument,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

**Type:** TOKSTR
**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

An area that contains the open options that are associated with the file. These flags are defined in **fcntl.h**.

**Command**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The command indicates the function that is to be performed by the PFS. The values that are defined in **ioctl.h** are for regular calls. The special values for sockets initialization are defined in BPXYPFSI (see Appendix D).

**Argument_length**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

Argument_length contains the length of the argument.

**Argument**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Defined by the PFS or the Device Driver |
| **Length:** | Specified by Argument_length |

Argument is the buffer that is to be processed by the PFS. It may contain input data to be processed, data placed in it by the PFS or device driver, or both.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_ioctl service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_ioctl service stores the return code. The vn_ioctl service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_ioctl service should support at least the following error value:

| Return_code | Explanation |
|---|---|
| ENODEV | The requested function is not supported by the PFS. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_ioctl service stores the reason code. The vn_ioctl service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_ioctl processing**

vn_ioctl provides a vehicle by which a PFS may provide functions not described by the POSIX standard.

**Specific processing notes**

- The PFS could use vn_ioctl to support unique file operations.
- vn_ioctl could be used to allow direct access to devices that are controlled by the PFS. You should avoid passing addresses with this service (using *argument*), and instead include all data in the buffer.
- The maximum Argument_length that is supported by the LFS is 1024 bytes.
- Refer to “Common INET sockets” on page 48 for information on the commands that a PFS must support in order to be an AF_INET socket PFS.
- Open_flags are all zero when vn_ioctl is the result of the w_pioctl (BPX1PIO) function, since the file being operated on has not been opened. The PFS may want to include a special access check in this case.
- For those cases in which user data addresses are passed in the argument, the user’s storage key is passed to the PFS. This key should be used with MVCSK/MVCDK or osi_copyin/osi_copyout to reference the user data areas.

  The key is passed in the first word of the system data area of the Token_structure, ts_sysd1, with a format of X'PPPP020K', where K is the four-bit key value. When ts_sysd1 is all zeroes, keys are not passed.

  The first two bytes of ts_sysdl, when byte 3 is X'02', are the first and third bytes of the user’s PSW, which are the bytes that contain the user’s AMODE and Supervisor State bits.

  This information is passed in ts_sysdl for all instances of program ioctl() calls, but some internal uses of vn_ioctl, mostly for FIONBIO, do not do so. These cases do not contain addresses in the argument.

**Serialization provided by the LFS**

The vn_ioctl operation is invoked with an exclusive latch held on the vnode of the file.

**Security calls to be made by the PFS**

The PFS may choose to invoke SAF’s Check Access callable service to verify that the user has write permission to the file or device.

# vn_link — Create a link to a file

## Function

The vn_link operation creates a link to the file that is specified by Token_structure in the directory that is specified by Directory_token_structure. The link is a new name that identifies an existing file. The new name does not replace the old one, but provides an additional way to refer to the file.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_link     (Token_structure,
             OSI_structure,
             Audit_structure,
             Link_name_length,
             Link_name,
             Directory_token_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

See "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Link_name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of Link_name. The name can be between 1 and 255 bytes long.

**Link_name**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Link_name_length |

An area, of length Link_name_length, that contains the new name by which the file is to be known. This name contains no nulls.

**Directory_token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKStr.ts_hdr.cblen. |

The Directory_token_structure represents the vnode of the directory that is to contain Link_name.

This area is mapped by the TOKSTR typedef in the BPXYPFSI header file (see Appendix D) for details.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_link service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_link service stores the return code. The vn_link service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_link service should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EEXIST | A file with the same name already exists. |
| ENAMETOOLONG | The length of Link_name exceeds the length that is supported by this PFS. |
| EROFS | The file system is mounted read-only. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_link service stores the reason code. The vn_link service returns Reason_code only if Return_value is –1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_link processing**

vn_link must create an entry in the directory that is specified by Directory_token_structure, pointing to the file that is specified by Token_structure.

**Specific processing notes**

– If the link is created successfully, the operation increments the link count of the file. The link count shows how many links to a file exist. (If the link is not created successfully, the link count is not incremented.)

– The LFS does not permit links to directories.

– If the link is created successfully, the change time of the linked-to file is updated, as are the change and modification times of the directory that contains Link_name, that is, the directory that holds the link.

– For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **link()** function in the POSIX .1 standard (IEEE Std 1003.1-1990).

**Serialization provided by the LFS**

The vn_link operation is invoked with an exclusive latch held on the vnodes of the directory and the file.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF's Check Access callable service to verify that the user has *any* access to the file, and has *write* access to the directory.

For a discussion of vn_link processing in a multilevel security environment, see "PFS support for multilevel security" on page 64.

## Related services

• "vn_remove — Remove a link to a file" on page 194
• "vn_rename — Rename a file or directory" on page 197

# vn_listen — Listen on a socket

## Function

The vn_listen operation identifies the socket as a server and establishes the maximum number of incoming connection requests that can be queued.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_listen   (Token_structure,
             OSI_structure,
             Audit_structure,
             Backlog,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Backlog**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that specifies the maximum number of connection requests that can be queued.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_listen operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_listen operation stores the return code. The vn_listen operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_listen operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EINVAL | Either a bind has not been issued on this socket; a listen was already done; or this socket has been connected. |
| EOPNOTSUPP | Listen is valid only for stream sockets. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_listen operation stores the reason code. The vn_listen operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

# Implementation notes

### Overview of vn_listen processing

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in “Finding more information about sockets” on page xvi for the listen function.

**Specific processing notes**: None.

**Serialization provided by the LFS**

The vn_listen operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

## Related services

- “vn_bind — Bind a name to a socket” on page 125

# vn_lookup — Look up a file or directory

## Function

The vn_lookup searches the directory that is represented by token_structure for the file or directory whose name is supplied.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_lookup   (Token_structure,
             OSI_structure,
             Audit_structure,
             Name_length,
             Name,
             Vnode_token,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Name_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of Name. The name is between 1 and 255 bytes long.

**Name**

Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Name_length |

An area, of length Name_length, that contains the name of the file or directory that is to be searched for. This name is not null-terminated.

**Vnode_token**

Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

An area in which the vn_lookup operation returns the vnode token of the file or directory that is supplied in the name parameter.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_lookup operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_lookup operation stores the return code. The vn_lookup operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_lookup operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EACCES | The caller does not have search permission for the parent directory. |
| ENAMETOOLONG | The Name_length that was supplied is greater than the maximum name length that is supported by this PFS. |

| Return_code | Explanation |
|---|---|
| ENOENT | The file or directory does not exist in the parent directory. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_lookup operation stores the reason code. The vn_lookup operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_lookup processing**

Lookup processing is described in “Creating, referring to, and inactivating file vnodes” on page 31.

**Specific processing notes**

– The token structure that is passed on input represents the directory that is searched for the input name.

– If the file or directory that is named in the Name parameter does not exist in the parent directory, the vn_lookup operation returns a failing return code, and no vnode_token is returned.

**Serialization provided by the LFS**

The vn_lookup operation is invoked with a shared latch held on the vnode of the parent directory.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF’s Check Access callable service to verify that the user has search permission to the directory.

## Related services

- “osi_getvnode — Get or return a vnode” on page 385

# vn_mkdir — Create a directory

## Function

The vn_mkdir operation creates a directory using the attributes that are provided by the caller.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_mkdir    (Token_structure,
             OSI_structure,
             Audit_structure,
             Name_length,
             Name,
             File_attribute_structure,
             Vnode_token,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

**Type:** TOKSTR
**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Name_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of the directory name that is to be created. The name can be between 1 and 255 bytes long.

**Name**

Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Name_length |

An area, of length Name_length, that contains the name of the directory that is to be created. This name contains no nulls.

**File_Attribute_Structure**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the ATTR.attr_hdr.cblen field |

An area that contains the attributes of the directory that is to be created. This area is mapped by the ATTR typedef in the BPXYVFSI header file (see Appendix D). See "Specific processing notes" for details on how the fields in this structure are processed.

**Vnode_token**

Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

An area in which the vn_mkdir service returns the vnode_token for the new directory.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_mkdir service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| −1 | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1 |
| 0 | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_mkdir service stores the return code. The vn_mkdir service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_mkdir service should support the following error values:

| Return_code | Explanation |
|---|---|
| EACCES | The caller does not have write authority for the parent directory. |
| EEXIST | A directory with the same name already exists. |
| ENOENT | The parent directory has been marked for deletion. |
| ENAMETOOLONG | The length of the name is greater than the maximum supported length. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_mkdir service stores the reason code. The vn_mkdir service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_mkdir processing

"Creating, referring to, and inactivating file vnodes" on page 31 provides an overview of directory creation processing.

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **mkdir()**function in the POSIX .1 standard (IEEE Std 1003.1-1990).

### Specific processing notes

– The token structure that is passed on input represents the parent directory in which the new directory is created.

– The following ATTR fields are provided by the LFS:

| | |
|---|---|
| **Attr.at_hdr.cbid** | Contains Attr_ID (from the BPXYVFSI header file) |
| **Attr.attr_hdr.cblen** | Specifies the length of the File_Attribute_Structure |
| **ATTR.at_mode** | Specifies the directory permission bits. See Appendix D for the mapping of this field. |

– If the directory that is named in the Name parameter already exists, the vn_mkdir service returns a return code of EEXIST, and the output vnode_token is optional.

### Serialization provided by the LFS

The vn_mkdir operation is invoked with an exclusive latch held on the vnode of the parent directory.

### Security calls to be made by the PFS

The PFS is expected to invoke SAF's Check Access callable service to verify that the user has write permission to the directory. The PFS is also expected to invoke SAF's Make FSP callable service to create a file security packet.

## Related services


- “osi_getvnode — Get or return a vnode” on page 385
- “vn_remove — Remove a link to a file” on page 194

# vn_open — Open a file

## Function

The vn_open operation opens a file.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_open     (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Bit |
| **Length:** | Fullword |

A fullword containing the binary flags that describe how the file is to be opened. These flags are defined by **fcntl.h**.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword where the vn_open operation returns the results of the operation as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_open operation stores the return code. The vn_open operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_open operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EACCES | The caller does not have permission for the requested (read or write) access. |
| ENOENT | The file does not exist. |
| EROFS | An attempt was made to open a file for write in a read-only file system. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword where the vn_open operation stores the reason code. The vn_open operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_open processing

See “Opening and closing files and first references to files” on page 34 for a discussion of open processing.

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **open()** function in the POSIX .1 standard (IEEE Std 1003.1-1990).

**Specific processing notes**

– The O_RDONLY and O_WRONLY bits in the Open_flags control whether the SAF Check Access callable service is called for a read, write, or read and write access check.

– When the O_EXEC flag is ON in the Open_flags, the SAF Check Access call must be made with a check for execute permission rather than read or write permission. This bit is an z/OS UNIX extension that is defined in Appendix D.

– When the O_TRUNC flag is ON in the Open_flags the PFS must truncate the file to zero length.

– The LFS implements the semantics of the O_CREAT and O_EXCL flags.

– The Open_flags will be remembered by the LFS and passed to the PFS on all read/write type operations that are related to this open. The O_APPEND and O_NONBLOCK flags, for instance, are processed by the PFS during those read/write operations from the flags passed to it at that time. The O_SYNC flag is transferred by the LFS to the UIO.u_sync flag for all read/write type operations so that this function can be processed by the PFS the same way for both POSIX and NFS users.

**Serialization provided by the LFS**

The vn_open operation is invoked with an exclusive latch held on the vnode of the file. Shared read support for the file being opened may be modified in the OSI by the PFS upon returning from the vn_open operation.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF’s Check Access callable service to check that the user has permission for the requested (read, write, or execute) access.

## Related services

- “vn_close — Close a file or socket” on page 132

# vn_pathconf — Determine configurable pathname values

## Function

The vn_pathconf operation returns the current value of a configurable limit or option that is associated with a file or directory.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_pathconf (Token_structure,
             OSI_structure,
             Audit_structure,
             Option,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Option**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The option parameter contains a value that indicates which configurable limit or option is returned in Return_value. These values are defined in **unistd.h** and are:

| Variable Returned | Description |
|---|---|
| _PC_CHOWN_RESTRICTED | Change ownership function is restricted to a process with appropriate privileges, and to changing the group ID (GID) of a file only to the effective group ID of the process or to one of its supplementary group IDs. |
| _PC_LINK_MAX | Maximum value of a file's link count. |
| _PC_MAX_CANON | Maximum number of bytes in a terminal canonical input line. |
| _PC_MAX_INPUT | Minimum number of bytes for which space is to be available in a terminal input queue; therefore, the maximum number of bytes a portable application may require to be typed as input before it reads them. |
| _PC_NAME_MAX | Maximum number of bytes in a filename (not a string length; count excludes a terminating null). |
| _PC_NO_TRUNC | Pathname components longer than 255 bytes generate an error. |
| _PC_PATH_MAX | Maximum number of bytes in a pathname (not a string length; count excludes a terminating null). |
| _PC_PIPE_BUF | Maximum number of bytes that can be written atomically when writing to a pipe. |
| _PC_VDISABLE | Terminal special characters that are maintained by the system can be disabled using this character value. |

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vn_pathconf operation returns the current value of the pathname variable that corresponds to Name specified, or −1 if not successful.

If the named pathname variable does not have a limit for the specified file, Return_value is set to −1 and Return_code and Reason_code remain unchanged.

If _PC_CHOWN_RESTRICTED is specified for Option, and _POSIX_CHOWN_RESTRICTED is active, Return_value is set to 1.

If _PC_CHOWN_RESTRICTED is specified for Option, and _POSIX_CHOWN_RESTRICTED is not active, Return_value is set to 0.

If _PC_NO_TRUNC is specified for Option, and _POSIX_NO_TRUNC is active, Return_value is set to 1.

If _PC_NO_TRUNC is specified for Option, and _POSIX_NO_TRUNC is not active, Return_value is set to 0.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_pathconf operation stores the return code. The vn_pathconf operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_pathconf operation stores the reason code. The vn_pathconf operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Specific processing notes**

– If the PFS does not have a limit for the specified option, Return_value is set to -1, but Return_code and Reason_code are unchanged. A Return_value of -1 in this case represents a limit of infinity (or no limit) for the requested option.

– The vn_pathconf operation is not invoked by the LFS if the PATH_MAX option is specified. The LFS value for PATH_MAX, 1023, is returned.

– If the PC_NAME_MAX option is specified, the LFS compares its value to the PFS value, and returns the minimum.

**Serialization provided by the LFS**

The vn_pathconf operation is invoked with a shared latch held on the vnode.

**Security calls to be made by the PFS**: None.

# vn_rdwr — Read or write a file

## Function

The vn_rdwr operation reads data from or writes data to a file or a socket.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_rdwr     (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             User_IO_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

An area that contains the options that are to be used when reading from or writing to the file or socket. This area is mapped by **fcntl.h.** See *z/OS XL C/C++ Run-Time Library Reference* for a description of this header.

**User_IO_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the UIO.u_hdr.cblen field |

An area to be used by the vn_rdwr service to determine the buffer address, length, storage key, and other attributes of the read or write request. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See the description of the vn_readwritev service ("Specific processing notes") for details on how the fields in this structure are processed.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_rdwr service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0 or greater** | The operation was successful; the value represents the number of bytes that were transferred. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_rdwr service stores the return code. The vn_rdwr service returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_rdwr service should support the following error values:

| Return_code | Explanation |
|---|---|
| EFAULT | A buffer address that was not valid was passed. |
| EINTR | The request was interrupted by a signal. |
| EACCES | The caller does not have the requested (read or write) access to the file. |

| Return_code | Explanation |
|---|---|
| EFBIG | Writing to the specified file would exceed the file size limit for the process, or the maximum file size that is supported by the physical file system. |
| EIO | An I/O error occurred while the file was being accessed. |
| EWOULDBLOCK | The request was made of a non-blocking descriptor, and a block was needed to satisfy the request. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_rdwr service stores the reason code. The vn_rdwr service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_rdwr processing**

“Reading from and writing to files” on page 36 provides an overview of file read and write processing.

**Specific processing notes**

– The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the caller’s input/output buffer |
| **UIO.u_buffalet** | Specifies the ALET of the caller’s input/output buffer |
| **UIO.u_offseth** | Specifies the upper word of a doubleword value that contains the offset into the file. The updated value for this field is returned by the PFS as a result of the vn_rdwr operation. |
| **UIO.u_offset** | Specifies the lower word of a doubleword value that contains the offset into the file. The updated value for this field is returned by the PFS as a result of the vn_rdwr operation. |
| **UIO.u_count** | Specifies the number of bytes that are to be read or written |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_rw** | Specifies whether the request is a read (0) or a write (1) |
| **UIO.u_key** | Specifies the storage key of the caller’s input/output buffer |
| **UIO.u_fssizelimithw** | Specifies the high word of the file size limit for the process |

**UIO.u_fssizelimitlw** Specifies the low word of the file size limit for the process

**UIO.u_sync** Specifies that the file is to be written to disk before the PFS returns. The PFS sets UIO.u_syncd to indicate that this has been done.

**UIO.u_chkacc** Specifies that access checking is to be performed

**UIO.u_realpage** Specifies that a real storage address is being passed. This flag is used only if the PFS reported during initialization that it supports DATOFF moves.

**PFS limit processing**

The UIO contains the process file size limit for the file. This is a doubleword value that is contained in UIO.u_fssizelimithw and UIO.u_fssizelimitlw. When a write request is unable to write any data before exceeding the file size limit, the PFS must set the UIO.u_limitex bit on, in addition to setting a Return_code of EFBIG. This includes detecting the special case in which the UIO.u_fssizelimithw is equal to UIO_NONEWFILES, which prohibits the expansion of existing files.

(Note that for vn_setattr, the LFS handles file size limit checking.)

The PFS must also be aware of one other special value for the file size limit. If both UIO.u_fssizelimithw and UIO.u_fssizelimitlw are equal to 0, there is no file size limit set for the process.

**Serialization provided by the LFS**

The vn_rdwr operation is invoked with an exclusive latch held on the vnode, unless the VnodSharedRead flag indicates that shared read is supported, in which case a shared latch is held on the vnode.

Shared read support for the file that is being read from or written to may be modified in the OSI by the PFS upon returning from the vn_rdwr operation.

**Security calls to be made by the PFS**

If u_chkacc is on in the user_IO_structure, the PFS is expected to invoke SAF's Check Access callable service to check that the user has permission to read from or write to the file. This check should be based on the access intent that is specified by u_rw.

The PFS is expected to invoke SAF's Clear Setid callable service whenever a write is done to a file with the S_GID or S_UID options. System overhead can be significantly reduced by setting an internal flag in the Inode to indicate that Clear Setid has been called, so that subsequent calls can be avoided. This flag would be cleared whenever the file's mode is changed via vn_setattr. In other words, Clear Setid should only be called once on the first write after the file's mode is changed or its Inode is created in storage.

# vn_readdir — Read directory entries

## Function

The vn_readdir operation reads entries from the directory that is represented by the input Token_structure, and returns as many entries as will fit in the caller's buffer.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_readdir  (Token_structure,
             OSI_structure,
             Audit_structure,
             User_IO_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**User_IO_structure**
Supplied and returned parameter
**Type:** UIO
**Length:**
Specified by UIO.u_hdr.cblen.

An area containing the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See “Specific processing notes” for details on how the fields in this structure are processed.

**Return_value**
Returned parameter
**Type:** Integer
**Length:** Fullword

A fullword in which the vn_readdir operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful, and there are no more directory entries to be read. No entries are returned. |
| **0 or greater** | The operation was successful; the value represents the number of directory entries that are returned. |

**Return_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

A fullword in which the vn_readdir operation stores the return code. The vn_readdir operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_readdir operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EACCES | The caller does not have search permission for the directory. |
| EFAULT | A buffer address that was specified is not in addressable storage. |
| EINVAL | There was a parameter error, such as an input buffer that is too small for any entries. |

**Reason_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

A fullword in which the vn_readdir operation stores the reason code. The vn_readdir operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS product.

## Implementation notes

**Overview of vn_readdir processing**

"Reading directories" on page 37 provides an overview of readdir operation.

**Specific processing notes**

– The token structure that is passed on input represents the directory that is to be read.

– The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the caller's buffer |
| **UIO.u_alet** | Specifies the ALET of the caller's buffer |
| **UIO.u_offseth** | Specifies the high-order word of the cursor |
| **UIO.u_offset** | Specifies the low-order word of the cursor |
| **UIO.u_count** | Specifies the maximum number of bytes that can be written to the caller's buffer |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_key** | Specifies the storage key of the caller's buffer |
| **UIO.u_rdindex** | Specifies the **readdir** index field |

– The following UIO fields must be set by the PFS:
  - **UIO.u_offseth**
  - **UIO.u_offset**

– The PFS is expected to write directory entries into the caller's buffer. These directory entries are mapped by the DIRENT and DIREXT typedefs in the BPXYVFSI header file (see Appendix D).

– For more information on the semantics of this operation for a POSIX-conforming PFS, see readdir (BPX1RDD, BPX4RDD) — Read an entry from a directory in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

**Serialization provided by the LFS**

The vn_readdir operation is invoked with a shared latch held on the vnode of the directory.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF's Check Access callable service to verify that the user has read permission to the directory.

## Related services

- "vn_open — Open a file" on page 170

# vn_readlink — Read a symbolic link

## Function

The vn_readlink operation reads the symbolic link file that is represented by Token_structure, and returns the contents in the buffer that is described by User_IO_structure.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_readlink (Token_structure,
             OSI_structure,
             Audit_structure,
             User_IO_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**User_IO_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_readlink service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0 or greater** | The operation was successful and represents the number of bytes that were transferred. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_readlink service stores the return code. The vn_readlink service returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_readlink service should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EFAULT | The buffer address that was specified in the input user_IO_structure is not in addressable storage. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_readlink service stores the reason code. The vn_readlink service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_readlink processing

"Reading from and writing to files" on page 36 provides an overview of file read and write processing.

The vn_readlink operation reads a symbolic link file. A symbolic link file contains the pathname or external name that was specified when the symbolic link was created.

**Specific processing notes**

– The token structure that is passed on input represents the symbolic link that is to be read.

– The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the caller's buffer |
| **UIO.u_alet** | Specifies the ALET of the caller's buffer |
| **UIO.u_count** | Specifies the maximum number of bytes that can be written to the caller's buffer |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_key** | Specifies the storage key of the caller's buffer |

– If the buffer that is supplied to vn_readlink is too small to contain the contents of the symbolic link, the value should be truncated to the length of the buffer (UIO.u_count).

– There is no difference in vn_readlink processing for symbolic and external links.

– Refer to the **readlink()** function in the POSIX .1a standard (IEEE Std 1003.1a), draft 7, for more information on the semantics of this operation for a POSIX-conforming PFS.

**Serialization provided by the LFS**

The vn_readlink operation is invoked with a shared latch held on the vnode of the directory.

**Security calls to be made by the PFS**: None.

## Related services

- "vn_symlink — Create a symbolic link" on page 238

# vn_readwritev — Read or write using a set of buffers for data

## Function

The vn_readwritev operation reads or writes on a file or socket, using a set of buffers to hold the data that is read or written.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_readwritev (Token_structure,
               OSI_structure,
               Audit_structure,
               Open_flags,
               User_IO_structure,
               Return_value,
               Return_code,
               Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that contains the bits that are associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**User_IO_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See “Specific processing notes” for details on how the fields in this structure are processed.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_readwritev operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0 or greater** | The operation was successful; the value represents the number of bytes that were transferred. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_readwritev operation stores the return code. The vn_readwritev operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_readwritev operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EINVAL | Either a negative number of bytes was requested, or this socket has been shut down. |
| EFAULT | A buffer address that was specified is not in addressable storage. |
| EFBIG | Writing to the specified file would exceed the file size limit for the process or the maximum file size that is supported by the physical file system. |

| Return_code | Explanation |
| --- | --- |
| EWOULDBLOCK | The operation would have required a blocking wait, and this socket was marked as nonblocking. |

**Reason_code**
Returned parameter

| | |
| --- | --- |
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_readwritev operation stores the reason code. The vn_readwritev operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_readwritev processing**

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in "Finding more information about sockets" on page xvi for the readv and writev functions.

**Specific processing notes**

– The following UIO fields are provided by the LFS:

| | |
| --- | --- |
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the caller's iov structure. The iov structure is mapped in **uio.h**. |
| **UIO.u_buffalet** | Specifies the ALET of the caller's iov structure |
| **UIO.u_offseth** | Specifies the upper word of a doubleword value that contains the offset into the file. The updated value for this field is returned by the PFS as a result of the vn_readwritev operation. |
| **UIO.u_offset** | Specifies the lower word of a doubleword value that contains the offset into the file. The updated value for this field is returned by the PFS as a result of the vn_readwritev operation. |
| **UIO.u_count** | Specifies the number of elements in the IOV array |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_rw** | Specifies whether the request is a read (0) or a write (1) |
| **UIO.u_key** | Specifies the storage key of the caller's buffer |
| **UIO.u_iovbufalet** | Specifies the ALET of the iov's buffers. All of the iov buffers must use the same ALET. |

**UIO.u_fssizelimithw** Specifies the high word of the file size limit for the process.

**UIO.u_fssizelimitlw** Specifies the low word of the file size limit for the process.

Also refer to "Reading from and writing to files" on page 36 for details on how reads and writes are done by the file system.

The UIO contains fields that may point to a 64-bit addressable user buffer. When FuioAddr64 is on (and FuioRealPage is off), FuioBuff64Vaddr points to a buffer, an IOV64, or an MSGH64.

**PFS Limit Processing**

The UIO contains the process file size limit for the file. This is a doubleword value that is contained in UIO.u_fssizelimithw and UIO.u_fssizelimitlw. When a write request is unable to write any data before exceeding the file size limit, the PFS must set the UIO.u_limitex bit on, in addition to setting a Return_code of EFBIG. This includes detecting the special case in which the UIO.u_fssizelimithw is equal to UIO_NONEWFILES, which prohibits the expansion of existing files.

(Note that for vn_setattr, the LFS handles file size limit checking.)

The PFS must also be aware of one other special value for the file size limit. If both UIO.u_fssizelimithw and UIO.u_fssizelimitlw are equal to 0, there is no file size limit set for the process.

**Serialization provided by the LFS**

The vn_readwritev operation is invoked with an exclusive latch held on the vnode of the file or socket, unless the VnodSharedRead flag indicates that shared read is supported, in which case a shared latch is held on the vnode.

Shared read support for the file that is being read from or written to can be modified in the OSI by the PFS upon returning from the vn_readwritev operation.

**Security calls to be made by the PFS**

If the check access bit is set and this PFS does access checking, the PFS is expected to invoke SAF's Check Access callable service to verify that the user has permission to read from or write to the file.

The PFS is expected to invoke SAF's Clear Setid callable service whenever a write is done to a file with the S_GID or S_UID options. System overhead can be significantly reduced by setting an internal flag in the Inode to indicate that Clear Setid has been called, so that subsequent calls can be avoided. This flag would be cleared whenever the file's mode was changed via vn_setattr. In other words, Clear Setid should only be called once on the first write after the file's mode is changed or its Inode is created in storage.

# vn_recovery — Recover resources after an abend

## Function

The vn_recovery operation permits a PFS to recover resources when an abnormal end occurs while a request to that PFS is active.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_recovery  (Token_structure,
              OSI_structure,
              Audit_structure,
              Recovery_area,
              Return_value,
              Return_code,
              Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

**Type:** TOKSTR
**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Recovery_area**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | 8 bytes |

A copy of the `Recovery_area` that was filled in by the PFS during the operation that was interrupted. This area is mapped by OSIRTOKEN (see Appendix D, "Interface structures for C language servers and clients," on page 503).

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_recovery operation returns control information to direct the outcome of the recovery processing, as follows:

| Return_value | Meaning |
|---|---|
| **VNR_NODUMP** | Suppress the system dump that is normally taken. |
| **VNR_RETSUCCESS** | Report success to the user. In this case, the value in the Return_Code parameter is passed back to the user as the return value of the original function. |
| **VNR_RETERRNO** | Report failure to the user. In this case, the values in the Return_Code and Reason_Code parameters are passed back to the user as the return and reason codes for the original function. The return value that is passed back for the original function is `−1`. |

Dump suppression may be requested with either success or failure reports; that is, with values of VNR_NODUMP+VNR_RETSUCCESS or VNR_NODUMP+VNR_RETERRNO, respectively.

If a Return_value is not returned by the PFS, a system dump is attempted and the original function fails with generic return and reason codes. The Return_values listed above are defined in BPXYPFSI (see Appendix D, "Interface structures for C language servers and clients," on page 503.)

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_recovery operation stores the return code. The vn_recovery operation returns Return_code with the Return_value that was returned, as explained above.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_recovery operation stores the reason code. The vn_recovery operation returns Reason_code with the Return_value that was returned, as explained above.

## Implementation notes

**Overview of vn_recovery processing**

“Recovery considerations” on page 24 provides an overview of recovery processing, and discusses the flow for vn_recovery in particular.

When an active request to the PFS is interrupted by an abnormal end, the PFS may have resources, such as storage and locks, that are left in a state that will cause problems for other users. This operation is defined to give the PFS a chance to clean up these resources if an abnormal end should occur.

This operation is designed for a PFS that does not have its own ESTAE or FRR protection. When a PFS has its own recovery, it normally handle abnormal ends before returning or percolating back to the LFS.

**Specific processing notes**

– An 8-byte Recovery_area is passed on every VFS and vnode operation, through the osi_rtokptr pointer in the OSI_structure, in which the PFS can record its resources or store a pointer to a recovery block. Any information that is stored in this area by the PFS during an operation is passed back to the PFS via the Recovery_area parameter of vn_recovery if the operation is interrupted by an abnormal end.

  The SDWA address is also passed to the PFS, for diagnostic purposes. This address is stored 16 bytes after the 8-byte Recovery_area. (Refer to the osirtokenx structure in Appendix D, “Interface structures for C language servers and clients,” on page 503.) The PFS must test this address for zero before using it, because the system is not always able to obtain an SDWA during recovery.

– The OSI work area and the preinitialized C Environment Stack (if used) are still addressable, and left as they were at the time of the abnormal end. These areas can be used to hold a recovery block whose address is placed in the Recovery_area. Vn_recovery is invoked with its own separate areas.

– The PFS is not called if the Recovery_area that is pointed to by osi_rtokptr is zero at the time of the abnormal end.

– The PFS is not called if the file system has been unmounted. A file system can be unmounted between the original vnode operation and vn_recovery in the following scenario:
  1. An operation goes into a signal enabled wait.
  2. The file system is unmounted with the IMMEDIATE operand.
  3. The waiting user is canceled.

  The PFS is expected to have cleaned up all its file-system-related resources during vfs_umount.

– This Recovery_area is the same one that is used by the vfs_recovery operation for user EOM recovery. The difference between these operations is that if the LFS’s ESTAE runs, it calls the PFS with vn_recovery from the same home address space and task that the original operation was invoked from. If the LFS’s ESTAE is bypassed by MVS, the LFS’s user address space EOM resource manager calls the PFS with vfs_recovery. This call is from a different task and home address space than the original call, and the original home address space no longer exists.

– Vfs_recovery is not called after vn_recovery has been called, unless vn_recovery is interrupted by a sudden end-of-memory condition for the user's address space. An example of this would be a program check in the PFS that was followed almost immediately by an operator force of the user. Another example would be if the PFS's vn_recovery routine were to get into a deadlock or extended wait, and the operator had to force the user off.

– Special care must be taken with vn_recovery, because the Token_structure may not always contain a file-level token. This is because vn_recovery is used for abend recovery of all the VFS and vnode operations. If a VFS operation is interrupted, the Token_structure on the vn_recovery call does not contain a file token; and if vfs_pfsctl is interrupted, the Token_structure contains only the PFS's initialization token.
– No recovery of any type is supplied for the vn_recovery operation itself. The operation is invoked with Osi_rtokptr pointing to a new recovery area, but this is only to allow the PFS to use common entry code that may depend on having a valid address in this field.

  See the OSI and osirtoken structures in Appendix D.
– The state of any file system and file objects that may have been involved with the interrupted operation is the same as at the time of the interruption.

**Serialization provided by the LFS**

The vn_recovery operation is invoked with the same serialization that was held at the time of the abnormal end.

**Security calls to be made by the PFS**: None.

# vn_remove — Remove a link to a file

## Function

The vn_remove service removes a link to a file. The input Name can identify a file, a link-name of a file, or a symbolic link.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_remove   (Token_structure,
             OSI_structure,
             Audit_structure,
             Name_length,
             Name,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of Name. The name is between 1 and 255 bytes long.

**Name**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Name_length |

An area, of length Name_length, that contains the name of the link that is to be deleted. This name contains no nulls.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_remove service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| −1 | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| 0 | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_remove service stores the return code. The vn_remove service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_remove service should support at least the following error values:

| Return_code | Explanation |
|---|---|
| ENAMETOOLONG | The value of Name_length exceeds the length that is supported by this PFS. |
| ENOENT | Name is marked for deletion. |
| EROFS | The file system is mounted read-only. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_remove service stores the reason code. The vn_remove service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_remove processing**

"Deleting files" on page 33 provides an overview of file deletion processing.

**Specific processing notes**

– The system data fields of the Token_structure contain the PFS's file token for the file that is being removed.

– If the name that is specified refers to a symbolic link, the symbolic link file that is named by Name should be deleted.

– If the link name is successfully removed from the directory, and the link count becomes zero, the deletion of the file is recorded for audit purposes. The actual deletion of the file object, and the inode, is done when the vnode is inactivated.

  If a regular file is not open when its link count goes to zero, the space that is occupied by its data should be freed for reuse before the return from vn_remove.

  If a regular file is still open when the link count goes to zero, its contents are not deleted at this point, but remain accessible until the open count goes to zero.

– When the vn_remove service is successful in removing a directory entry and decrementing the link count, even if the link count is not zero, it must return control to the caller with Return_value set to 0. It must update the change and modification times for the parent directory, and the change time for the file itself (unless the file is deleted).

– For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **unlink()** function in the POSIX .1 standard (IEEE Std 1003.1-1990).

**Serialization provided by the LFS**

The vn_remove operation is invoked with an exclusive latch held on the vnode of the file that is to be removed, and on the directory that contains that file name.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF's Check Access callable service to verify that the user has write permission to the directory, and the Audit callable service to record the deletion of the file.

SAF's Check2Owners service is called whenever the sticky bit is on in the parent directory.

## Related services


- "vn_create — Create a new file" on page 138
- "vn_link — Create a link to a file" on page 157
- "vn_rmdir — Remove a directory" on page 201
- "vn_symlink — Create a symbolic link" on page 238

# vn_rename — Rename a file or directory

## Function

The vn_rename renames a file or directory.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_rename   (Token_structure,
             OSI_structure,
             Audit_structure,
             Name_length,
             Name,
             New_name_length,
             New_name,
             New_token_structure
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of Name. The name is between 1 and 255 bytes long.

**Name**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Name_length |

An area, of length Name_length, that contains the file or directory name that is to be renamed. This name is not null-terminated.

**New_name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of New_name. The name is between 1 and 255 bytes long.

**New_name**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by New_name_length |

An area, of length New_name_length, that contains the file or directory name to which the file or directory is to be renamed. This name is not null-terminated.

**New_token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the structure's TOKSTR.ts_hdr.cblen field. |

New_token_structure represents the vnode of the directory that contains New_name.

Refer to "LFS/PFS control block structure" on page 16 for a discussion of the use of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D for its mapping.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_rename operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **–1** | The operation was not successful. The |

| | |
|---|---|
| | Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_rename operation stores the return code. The vn_rename operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_rename operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EACCES | The caller does not have write permission for one or both of the parent directories. |
| EBUSY | The New_name could not be deleted, or the named file or directory could not be renamed because the PFS considers it to be in use. |
| EISDIR | An attempt was made to rename a file to a directory. |
| ENAMETOOLONG | The length of one of the names supplied was greater than the maximum supported name length for this PFS. |
| ENOENT | Name was not found. |
| ENOTEMPTY | New_name specified an existing directory that was not empty. |
| ENOTDIR | Token_structure did not represent a directory, or an attempt was made to rename a directory to a file. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_rename operation stores the reason code. The vn_rename operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_rename processing

The PFS renames a file or directory that is specified by Name in the directory that is represented by Token_structure to the name that is specified by New_name in the directory that is represented by New_token_structure.

“Deleting files” on page 33 provides an overview of file deletion processing.

### Specific processing notes

– The system data fields of the Token_structure contain the PFS’s file token for the file that is being renamed. The system data fields of the New_Token_structure contain the PFS’s file token for the file that is named by New_name, if it exists.

– If a directory entry does not already exist for New_name, the PFS creates it. If a directory entry for New_name already exists, the file or directory that is represented by this entry is deleted, as described for vn_remove or vn_rmdir, as appropriate.

  If New_name is an existing directory that is not empty, the PFS returns a Return_value of –1 and an Return Code of ENOTEMPTY.

  If the rename is successful, the directory entry for the old name is deleted.

– The names that are passed to the PFS cannot be "`.`" or "`..`".

– For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **rename()** function in the POSIX .1 standard (IEEE Std 1003.1-1990).

**Serialization provided by the LFS**

The PFS is invoked with an exclusive latch for all of the vnodes involved in this operation. These include:

– The old parent directory
– The new parent directory
– The file or directory that is specified by Name
– If it already exists, the file or directory that is specified by New_name

**Security calls to be made by the PFS**

The PFS is expected to verify that the calling process has write permission for the directories that contain Name and New_name by calling SAF's Check Access callable service. If Name and New_name are themselves directories, the caller does not need write permission to these directories, only to the parent directories.

SAF's Check2Owners service is called whenever the sticky bit is on in the parent directory.

If the file that was previously known by New_name is deleted, invoke SAF's Audit callable service to record the deletion of the file.

## Related services


- "vn_remove — Remove a link to a file" on page 194
- "vn_rmdir — Remove a directory" on page 201

# vn_rmdir — Remove a directory

## Function

The vn_rmdir operation removes a directory. The directory must be empty.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_rmdir    (Token_structure,
             OSI_structure,
             Audit_structure,
             Directory_name_length,
             Directory_name,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter
**Type:** TOKSTR
**Length:** Specified by TOKSTR.ts_hdr.cblen.

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter
**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter
**Type:** CRED
**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Directory_name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that contains the length of Directory_name. The name is between 1 and 255 bytes long.

**Directory_name**
Supplied parameter

| | |
|---|---|
| **Type:** | String |
| **Length:** | Specified by Directory_name_length |

An area, of length Directory_name_length, that contains the name of the directory that is to be deleted. This name contains no nulls.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_rmdir service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| −1 | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| 0 | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_rmdir service stores the return code. The vn_rmdir service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_rmdir service should support at least the following error values:

| Return_code | Explanation |
|---|---|
| ENAMETOOLONG | The value of Directory_name_length exceeds the length that is supported by this PFS. |
| ENOENT | The directory name is marked for deletion. |
| ENOTEMPTY | The directory contains entries other than . and .. |
| EROFS | The file system is mounted read-only. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_rmdir service stores the reason code. The vn_rmdir service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_rmdir processing**

“Deleting files” on page 33 provides an overview of file deletion processing.

**Specific processing notes**

– The system data fields of the Token_structure contain the PFS’s file token for the file that is being removed.

– The directory that is specified by Directory_name must be empty except for the “.” and “..” entries.

– If the directory is successfully removed, the change and modification times for the parent directory must be updated.

– The deletion of the directory is recorded for audit purposes now, but the actual deletion of the object and the inode is done when the vnode is inactivated.

– Vn_readdir of a removed directory returns zero entries.

– New files must not be be created under a directory that is removed.

– For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the **rmdir()** function in the POSIX .1 standard (IEEE Std 1003.1-1990).

**Serialization provided by the LFS**

The vn_rmdir operation is invoked with an exclusive latch held on the vnode of the directory name that is to be removed, and on the directory that contains that directory name.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF’s Check Access callable service to verify that the user has write permission to the directory, and invoke the audit callable service to record the deletion of the directory file.

SAF’s Check2Owners service is called whenever the sticky bit is on in the parent directory.

## Related services

- “vn_remove — Remove a link to a file” on page 194
- “vn_mkdir — Create a directory” on page 166

# vn_select — Select or poll on a vnode

## Function

The vn_select operation monitors activity on a vnode to see if it is ready for reading or writing, or if it has an exceptional condition pending. The vnode can be for a socket, a pipe, a regular file, or a pseudoterminal file.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_select   (Token_structure,
             OSI_structure,
             Audit_structure,
             Select_token,
             Function,
             Select_option,
             Pfs_work_token,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Select_token**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 16 Bytes |

An area that the PFS copies into its own storage and later uses to tell the LFS that a selected event has occurred for this vnode.

This token is unique among all active vn_selects on the system, and can be used to correlate a query request (SEL_QUERY or SEL_POLLQUERY) with its corresponding cancel request (SEL_CANCEL or SEL_POLLCANCEL).

**Function**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that specifies whether this is a query or a cancel request, and whether the request is for select or poll. The values for this field are defined in the BPXYPFSI header file (see Appendix D). The Function parameter specifies the subfunction that is being requested:

*Table 3. vn_select subfunctions*

| Function | Description |
|---|---|
| SEL_QUERY, SEL_POLLQUERY | The PFS should perform the following:<br>1. Check the events that are specified in Select_option to see if any of them can be immediately satisfied. If so, this status is returned in the Return_Value parameter.<br>2. If there is no immediate status to report, the PFS records that a select is pending on this file and sets up to invoke osi_selpost later, when one of the selected events has occurred. The PFS returns a value of `0` in Return_Value after it has performed its internal processing to set up for select pending.<br>The occurrence of the event and the subsequent invocation of osi_selpost happen asynchronously on another thread or MVS task. |

*Table 3. vn_select subfunctions (continued)*

| Function | Description |
| --- | --- |
| SEL_CANCEL, SEL_POLLCANCEL | The PFS performs the following:<br>1. If there is a pending select/poll recorded for a prior query with the same Select_token, it must be canceled in such a way that osi_selpost is not invoked.<br>2. Check the events that are specified in Select_option to see if any of them can be immediately satisfied. If so, this status is returned in the Return_Value parameter. |

**Select_option**

Supplied parameter

**Type:** Integer

**Length:** Fullword

A fullword that contains the bits that describe the options that are requested for this vnode. The values for this field are defined in the BPXYPFSI header file (see Appendix D).

Select_option indicates the conditions or events that are being checked for. If this is a select request, the conditions are:

| Option | Description |
| --- | --- |
| **SEL_READ** | A read that is issued against this file will not block. |
| **SEL_WRITE** | A write that is issued against this file will not block. |
| **SEL_XCEPT** | An exceptional condition, as defined by the particular PFS, has occurred. Typically this could occur because a socket connection has become inoperative because of network problems, or the other end of the socket has been closed. |

For reading and writing, an error condition that would cause the read or write to fail means that the operation will not block, and therefore the file is ready for that operation.

If one or more of the selected conditions are ready, the PFS returns the information in the Return_Value parameter immediately, using the same bit mapping to indicate which conditions are ready.

The conditions that can be specified for poll are explained in other documents (for instance, *z/OS XL C/C++ Language Reference*). The mapping for these fields is defined in the BPXYPFSI header file (see Appendix D).

**Pfs_work_token**

Supplied or returned parameter

**Type:** Token

**Length:** Fullword

A fullword that is returned on a query request and passed on a subsequent cancel request. This allows the LFS to store data that the PFS will need on the cancel request, if any is needed.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the vn_select service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. This causes the whole **select()** or **poll()** request, as made by the application program, to fail. The Return_code and Reason_Code values are passed back to the application program. |
| **0** | There is no status, and the operation was successful.<br>• For query ( **SEL_QUERY** or **SEL_POLLQUERY**):<br>The PFS is set up to invoke osi_selpost when the requested event occurs.<br>• For cancel ( **SEL_CANCEL** or **SEL_POLLCANCEL**):<br>The PFS has canceled the request to invoke osi_selpost, or it has never been set up to do so. The PFS does not invoke osi_selpost after returning from this call. |
| **Greater than 0** | There is status being returned in this parameter. The returned status has the same format as the Select_option parameter.<br>• For query ( **SEL_QUERY** or **SEL_POLLQUERY**):<br>The operation is complete and the PFS will not invoke osi_selpost for this request.<br>• For cancel ( **SEL_CANCEL** or **SEL_POLLCANCEL**):<br>The PFS has canceled the request to invoke osi_selpost if it had been recorded, or it has never been set up to do so. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_select operation stores the return code. The vn_select operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**

Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vn_select operation stores the reason code. The vn_select operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_select processing**

For information on vn_select, refer to "Select/poll processing" on page 45.

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in "Finding more information about sockets" on page xvi for the **select()** function.

**Specific processing notes**

– The PFS should save the Select_token that is passed on the query request. This token is used both during the cancel request (to delete the request), and when an event occurs that the LFS should be informed of through the osi_selpost function.

– The PFS can use the Pfs_work_token parameter on a query request to return data (such as an address where it has stored information about this request), so that it can be found during a cancel request. The data is used to correlate the cancel request with its matching query request. This provides an alternative to scanning the PFS control blocks for a matching Select_token value.

– If the session being selected becomes inoperative, the PFS must fail the operation with a Return_code of EIO. For sockets, this is critical to Common Inet processing so that a stack can be removed from a socket during the internal vn_select that is done to implement blocking reads and accepts. For application select() calls, the LFS will convert EIO from vn_select to ready status for the descriptor so that the application receives the EIO notification on the specific descriptor to which it applies.

**Serialization provided by the LFS**

The vn_select operation is invoked with an exclusive latch held on the vnode of the file.

**Security calls to be made by the PFS**: None.

# vn_sendtorcvfm — Send to or receive from a socket

## Function

The vn_sendtorcvfm operation sends datagrams to or receives datagrams from a socket. The socket can be connected or unconnected.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_sendtorcvfm (Token_structure,
                OSI_structure,
                Audit_structure,
                Open_flags,
                User_IO_structure,
                Flags,
                Sockaddr_length,
                Sockaddr,
                Return_value,
                Return_code,
                Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that contains the bits that are associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**User_IO_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See "Specific processing notes" for details on how the fields in this structure are processed.

**Flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that indicates special processing requests. The defined values for this field are mapped by **socket.h**.

**Sockaddr_length**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that supplies the length of the Sockaddr buffer and returns the length of the Sockaddr structure that is returned.

**Sockaddr**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by Sockaddr_length |

A structure that varies depending on the address family type. It contains the address that is to be used for this operation. For an example of this mapping for AF_INET, see **in.h**.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_sendtorcvfm operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **–1** | The operation was not successful. The |

| | |
|---|---|
| | Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0 or greater** | The operation was successful. The value represents the number of bytes that were transferred. |

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_sendtorcvfm operation stores the return code. The vn_sendtorcvfm operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_sendtorcvfm operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EFAULT | A buffer address that was specified was not in addressable storage. |
| EINVAL | The length that was specified was incorrect. |
| EWOULDBLOCK | The operation would have required a blocking wait, and this socket was marked as nonblocking. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_sendtorcvfm operation stores the reason code. The vn_sendtorcvfm operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_sendtorcvfm processing

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in "Finding more information about sockets" on page xvi for the **recvfrom()** and **sendto()** functions.

### Specific processing notes

– The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the caller's buffer |
| **UIO.u_buffalet** | Specifies the ALET of the caller's buffer |
| **UIO.u_count** | Specifies the maximum number of bytes that can be written to the caller's buffer |
| **UIO.u_asid** | Specifies the ASID of the caller |

| | |
|---|---|
| **UIO.u_rw** | Specifies whether the request is a read (0) or a write (1) |
| **UIO.u_key** | Specifies the storage key of the caller's buffer |

– The UIO contains fields that may point to a 64-bit addressable user buffer. When FuioAddr64 is on (and FuioRealPage is off), FuioBuff64Vaddr points to a buffer, an IOV64, or an MSGH64.

**Serialization provided by the LFS**

The vn_sendtorcvfm operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

# vn_setattr — Set the attributes of a file

## Function

The vn_setattr operation sets the attributes of a file.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_setattr  (Token_structure,
             OSI_structure,
             Audit_structure,
             attribute_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**attribute_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | ATTR |
| **Length:** | Specified by ATTR.at_hdr.cblen. |

An area that contains the file attributes that are to be set for the file that is specified by the vnode token. This area is mapped by typedef ATTR in the BPXYVFSI header file (see Appendix D).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_setattr operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_setattr operation stores the return code. The vn_setattr operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_setattr operation should support at least the following error values:

| **Return_code** | **Explanation** |
|---|---|
| EACCES | The caller does not have SAF authority to:<br>• Set the access time or modification time to current time<br>• Truncate the file |
| EPERM | The caller does not have SAF authority to:<br>• Change the mode<br>• Change the owner<br>• Change general attribute bits<br>• Set a time field to a value (not the current time)<br>• Set the change time or reference time to the current time<br>• Change the auditing flags<br>• Change the file format<br>• Set the security label; or there is already a security label associated with the file |
| EROFS | The file system is mounted read-only. |
| ENOSPC | The file system is out of space. |

| Return_code | Explanation |
|---|---|
| EINVAL | Incorrect input parameter, such as a negative time value, an incorrect mode field, or an incorrect UID-GID. |

**Reason_code**
Returned parameter
**Type:** Integer
**Character set:**
N/A
**Length:**
Fullword

A fullword in which the vn_setattr operation stores the reason code. The vn_setattr operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_setattr processing

vn_setattr is used to set file attributes, as described in “Getting and setting attributes” on page 39.

### Specific processing notes

*Table 4. attribute_structure input fields*

| Set Flags | Attribute Fields Input | Description |
|---|---|---|
| at_modechg | at_mode | Set the mode according to the value in at_mode |
| at_ownchg | at_uid<br>at_gid | Set the owner user ID (UID) and group ID (GID) to the values specified in at_uid and at_gid |
| at_setgen | at_genvalue<br>at_genmask | Only the bits corresponding to the bits set ON in the at_genmask are set to the value (ON or OFF) in at_genvalue; other bits are unchanged |
| at_trunc | at_sizeh<br>at_size | Truncate the file size to the number of bytes specified by the doubleword at_sizeh and at_size |
| at_atimechg | at_atime | Set the access time of the file to the value specified in at_atime |
| at_atimechg and at_atimeTOD | None | Set the access time of the file to the current time |
| at_mtimechg | at_mtime | Set the modification time of the file to the value specified in at_mtime |
| at_mtimechg and at_mtimeTOD | None | Set the modification time of the file to the current time |
| at_aauditchg | at_aaudit | Set the security auditor’s auditing flags to the value specified in at_aaudit |
| at_uauditchg | at_uaudit | Set the user’s auditing flags to the value specified in at_uaudit |

*Table 4. attribute_structure input fields (continued)*

| Set Flags | Attribute Fields Input | Description |
|---|---|---|
| at_ctimechg | at_ctime | Set the change time of the file to the value specified in at_ctime |
| at_ctimechg and at_ctimeTOD | None | Set the change time of the file to the current time |
| at_reftimechg | at_reftime | Set the reference time of the file to the value specified in at_reftime |
| at_reftimechg and at_refTOD | None | Set the reference time of the file to the current time |
| at_filefmtchg | at_filefmt | Set the file format of the file to the value in at_filefmt |
| at_seclabelchg | at_seclabel | Set the initial security label of the file to the value in at_seclabel |

1. In addition to the attribute fields that are specified according to Table 4 on page 215, the following ATTR header fields are provided by the caller:

   **ATTR.at_hdr.cbid** Contains ATTR

   **ATTR.at_hdr.cblen** Specifies the length of attribute_structure.
2. Multiple attributes can be changed on a single vn_setattr call. The PFS should ensure that either all supported changes or no changes are permanently recorded for a single vn_setattr call.
3. Changing mode (at_modechg = ON):
   - SAF's Change File Mode callable service is called to perform the necessary security checks and to actually make the change to the mode field in the FSP.
   - The at_mode field is mapped by **modes.h**.

   **Notes:**

   a. The file type, which is contained within at_mode, is not changed by the vn_setattr operation.
   b. Files that are open when the vn_setattr service is called retain the access permission they had when the file was opened.
4. Changing owner (at_ownchg = ON):
   - SAF's Change Owner and Group callable service is called to perform the necessary security checks and to actually make the change to the owner and/or group fields in the FSP.

   **Note:** When the UID or GID value is set to -1, the original value remains unchanged.
5. Changing general attribute bits (at_setgen = ON):
   - SAF's Check Access callable service is called for Write access before the PFS changes the file's general attribute bits.
   - For each bit ON in the genmask, the corresponding bit in the file's attributes is set to the value (ON or OFF) from the corresponding genvalue field.
6. Truncating a file (at_trunc = ON):
   - SAF's Check Access is called for write access before the PFS changes the file's size.

- The truncation of a file changes the file size to the doubleword value that is represented by at_sizeh and at_size, beginning from the first byte of the file.
  - If the file is larger than the specified file size, the data from the specified size to the original end of the file is removed. Full blocks are returned to the file system to be used again.
  - If the file is shorter than the specified size, bytes between the old and new lengths are read as zeros.
- When the file size is changed, the PFS calls SAF's Clear Setid callable service.

**Notes:**

a. The LFS handles enforcing file size limits for vn_setattr.

b. The **truncate()** function requires write permission to the file, whereas **ftruncate()** requires that the file be open for writing. The LFS handles this difference by calling vn_setattr for the former and vn_trunc for the latter when the file is open for writing.

7. Changing time fields (atime, mtime, ctime, and reftime):
   - All time fields in the ATTR are in POSIX format.
   - Each time field is controlled by a pair of bits: the *chg* bit and the *TOD* bit, as listed in Table 4 on page 215.
     - The *chg* bit (for instance, at_atimechg) indicates that the corresponding time field is to be changed.
     - The *TOD* bit (for instance, at_atimeTOD) indicates whether the change is to an explicitly specified time (bit is off) or to the current time of day (bit is on).
   - For a time change using an explicit time value, the SAF check file owner service is called to verify that the caller is the file owner or has appropriate privileges before the PFS changes the corresponding file time field.
   - For a time change using the current time of day, the SAF check access service is called to check for write permission. If that fails, the SAF check file owner service is called. The time change is permitted if the caller has write permission, is the file owner, or has appropriate privileges.
8. Changing auditor audit flags (at_aauditchg = ON) or user audit flags (at_uauditchg = ON):
   - SAF's Change Audit Options callable service is called to perform the necessary security checks and to actually make the change to the corresponding audit field in the FSP.
9. Changing file format (at_filefmtchg = ON):
   - SAF's Check File Owner is called before the PFS saves the new file format value.
10. When any attribute field is changed successfully, the file's change time is also updated.
11. Changing the security label (ATTSECLABELCHG=ON):
    - For the security label to be changed, the user must have RACF SPECIAL authorization and appropriate privileges (see Authorization in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*), and no security label must currently exist on the file. Only an initial security label can be set. An existing security label cannot be changed. The function will successfully set the security label if the

SECLABEL class is active. If the SECLABEL class is not active, the request will return successfully, but the security label will not be set.
- You can invoke the SAF IRRSSB00 callable service to set the security label.

**Serialization provided by the LFS**

The vn_setattr operation is invoked with an exclusive latch held on the vnode. Shared read support can be modified by the PFS in the OSI upon return from the vn_setattr operation.

**Security calls to be made by the PFS**

Refer to the notes above for the security calls that are made for the various file attributes.

## Related services

- “vn_getattr — Get the attributes of a file” on page 145

# vn_setpeer — Set a socket's peer address

## Function

The vn_setpeer operation presets the peer address that is associated with a socket. This causes all datagrams that are sent using the specified socket to be sent to the address that is specified here. Only datagrams that are sent from the specified address are received.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_setpeer  (Token_structure,
             OSI_structure,
             Audit_structure,
             Sockaddr_length,
             Sockaddr,
             Option,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Sockaddr_length**

Supplied and returned parameter

**Type:** Integer
**Length:** Fullword

A fullword that supplies the length of the Sockaddr buffer and returns the length of the Sockaddr structure that is returned.

**Sockaddr**

Supplied and returned parameter

**Type:** Structure
**Length:** Specified by Sockaddr_length

A structure that varies depending on the address family type. It contains the address that is to be used for this operation. For an example of this mapping for AF_INET, see **in.h**.

**Option**

Supplied parameter

**Type:** Integer
**Length:** Fullword

A fullword that specifies the option of the vn_setpeer operation to use. These values are mapped by **socket.h**.

**Return_value**

Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vn_setpeer operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vn_setpeer operation stores the return code. The vn_setpeer operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_setpeer operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EINVAL | The address length that was specified is not the size of a valid address for the specified address family. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_setpeer operation stores the reason code. The vn_setpeer operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_setpeer processing**

The vn_setpeer call is new for POSIX 1003.12, and is not currently supported by any of the socket PFSs.

**Specific processing notes**

– Calling setpeer() with the option set to SO_SET causes all datagrams that are sent through this socket to be sent to the address that is specified by sockaddr. Only datagrams that originate from sockaddr are received.

– Calling setpeer() with the option set to SO_SET on the passive end of a virtual circuit before calling listen() or connect() causes an error. Calling connect() and specifying a destination address with setpeer causes an error. Calling setpeer() after a connection is established is an error.

– The result of calling setpeer() with the option set to SO_SET on an endpoint that has already had the destination address preset causes an error if the underlying protocol does not support multiple peer addresses for a given endpoint.

**Serialization provided by the LFS**

The vn_setpeer operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

## Related services

None.

# vn_shutdown — Shut down a socket

## Function

The vn_shutdown operation shuts down all or part of a duplex socket connection.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_shutdown (Token_structure,
             OSI_structure,
             Audit_structure,
             How,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**How**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The How parameter explains the condition of the shutdown request. The values that can be specified are:

| Value | Condition |
|---|---|
| **SHUT_RD** | Shutdown reads from this socket. |
| **SHUT_WR** | Shutdown writes to this socket. |
| **SHUT_RDWR** | Shutdown reads to and writes from this socket. |

These values are defined in **socket.h**.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_shutdown operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_shutdown operation stores the return code. The vn_shutdown operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_shutdown operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EINVAL | The How argument was not valid. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword where the vn_shutdown operation stores the reason code. The vn_shutdown operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Specific processing notes**

The How parameter comes directly from the **shutdown()** system call. The LFS does not check this parameter.

**Serialization provided by the LFS**

The vn_shutdown operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

## Related services

- “vfs_socket — Create a socket or a socket pair” on page 97
- “vn_close — Close a file or socket” on page 132

# vn_sndrcv — Send to or receive from a socket

## Function

The vn_sndrcv operation sends datagrams to or receives datagrams from a socket. The socket must be connected.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_sndrcv   (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             User_IO_structure,
             Flags,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that contains the bits that are associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**User_IO_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See “Specific processing notes” for details on how the fields in this structure are processed.

**Flags**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that indicates special processing requests. The defined values for this field are mapped by **socket.h.**

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_sndrcv operation returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **–1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0 or greater** | The operation was successful. The value represents the number of bytes that were transferred. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_sndrcv operation stores the return code. The vn_sndrcv operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_sndrcv operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EFAULT | A buffer address that was specified is not in addressable storage. |
| EINVAL | An incorrect parameter was specified. |
| EWOULDBLOCK | The operation would have required a blocking wait, and this socket was marked as nonblocking. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_sndrcv operation stores the reason code. The vn_sndrcv operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_sndrcv processing**

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in "Finding more information about sockets" on page xvi for the recv and send functions.

**Specific processing notes**

The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the caller's buffer |
| **UIO.u_buffalet** | Specifies the ALET of the caller's buffer |
| **UIO.u_count** | Specifies the maximum number of bytes that can be written to or read from the caller's buffer |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_rw** | Specifies whether the request is a read (0) or a write (1) |
| **UIO.u_key** | Specifies the storage key of the caller's buffer |

The UIO contains fields that may point to a 64-bit addressable user buffer. When FuioAddr64 is on (and FuioRealPage is off), FuioBuff64Vaddr points to a buffer, an IOV64, or an MSGH64.

**Serialization provided by the LFS**

The vn_sndrcv operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

# vn_sockopt — Get or set socket options

## Function

The vn_sockopt operation gets or sets options that are associated with a socket.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_sockopt  (Token_structure,
             OSI_structure,
             Audit_structure,
             Direction,
             Level,
             Option,
             Option_data_length,
             Option_data,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Direction**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The Direction parameter specifies whether the socket options are to be set or returned to the requester. The values for this parameter are defined in the BPXYPFSI header file (see Appendix D) and are as follows:

| Value | Meaning |
|---|---|
| **GET_SOCKOPT** | Get the current socket options |
| **SET_SOCKOPT** | Change the socket options |
| **SET_IBMSOCKOPT** | Change SetIBMsockopt options |

**Level**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that specifies the protocol level. This area is mapped by **socket.h**.

**Option**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword that specifies the option that is to be set or retrieved. This area is mapped by **socket.h**.

**Option_data_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The Option_data_length is a fullword that describes the length of the Option_data parameter.

**Option_data**

Supplied parameter

| | |
|---|---|
| **Type:** | Defined by the Option |
| **Length:** | Specified by Option_data_length |

For most options, this is either a zero or nonzero, depending on whether the option is disabled or enabled.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_sockopt operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|

| | |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_sockopt operation stores the return code. The vn_sockopt operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The vn_sockopt operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| ENOPROTOOPT | The level that was specified is an incorrect protocol. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

A fullword in which the vn_sockopt operation stores the reason code. The vn_sockopt operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_sockopt processing

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in “Finding more information about sockets” on page xvi for the getsockopt and setsockopt functions.

### Serialization provided by the LFS

The vn_sockopt operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

# vn_srmsg — Send messages to or receive messages from a socket

## Function

The vn_srmsg operation sends or receives messages on a socket. Message headers are used for the reading or writing operation. The socket can be either connected or unconnected.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_srmsg    (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             User_IO_structure,
             Flags,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that contains the bits that are associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**User_IO_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See "Specific processing notes" for details on how the fields in this structure are processed.

**Flags**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that indicates special processing requests. The defined values for this field are mapped by **socket.h**.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_srmsg operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **–1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0 or greater** | The operation was successful. The value represents the number of bytes that were transferred. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_srmsg operation stores the return code. The vn_srmsg operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_srmsg operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EFAULT | The address of one of the buffers is not in addressable storage. |
| EINVAL | An incorrect parameter was specified. |
| EWOULDBLOCK | A socket that has been defined as nonblocking cannot complete its operation without blocking. |

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the vn_srmsg operation stores the reason code. The vn_srmsg operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_srmsg processing

The vn_srmsg call can be used by connected or nonconnected sockets.

For more information on the semantics of this operation for a POSIX-conforming PFS, refer to the publications that are mentioned in "Finding more information about sockets" on page xvi for the **recvmsg()** and **sendmsg()** functions.

### Specific processing notes

– The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of the caller's message header |
| **UIO.u_buffalet** | Specifies the ALET of the caller's message header |
| **UIO.u_count** | Specifies the length of the message header |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_rw** | Specifies whether the request is a read (0) or a write (1) |
| **UIO.u_key** | Specifies the storage key of the caller's buffer |
| **UIO.u_iovalet** | Specifies the ALET of the iov |
| **UIO.u_iovbufalet** | Specifies the ALET of the iov's buffers. All buffers must use the same ALET. |

– The UIO contains fields that may point to a 64-bit addressable user buffer. When FuioAddr64 is on (and FuioRealPage is off), FuioBuff64Vaddr points to a buffer, an IOV64, or an MSGH64.

– The message header is defined in **socket.h**.

– The iov structure is defined in **uio.h**.

### Serialization provided by the LFS

The vn_srmsg operation is invoked with an exclusive latch held on the vnode of the socket.

**Security calls to be made by the PFS**: None.

# vn_srx — Send or receive CSM buffers

## Function

The vn_srx operation sends or receives data using CSM (Communications Storage Manager) buffers.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

## Input parameter format

```
vn_srx      (Token_structure,
             OSI_structure,
             Audit_structure,
             Open_flags,
             User_IO_structure,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS’s initialization token, mount token, and file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Open_flags**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fullword |

A fullword that contains the bits that are associated with the socket. The defined values for this field are mapped by **fcntl.h**.

**User_IO_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D, “Interface structures for C language servers and clients,” on page 503). See “Specific processing notes” for information about how the fields in this structure are processed.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_srx operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0 or greater** | The operation was successful. The value represents the number of bytes that were transferred. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_srx operation stores the return code. The vn_srx operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_srx operation should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EFAULT | A buffer address that was specified is not in addressable storage. |
| EINVAL | An incorrect parameter was specified. |
| EWOULDBLOCK | A socket that has been defined as nonblocking cannot complete its operation without blocking. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_srx operation stores the reason code. The vn_srx operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_srx processing**

The Communications Storage Manager (CSM) provides a facility that allows programs to avoid data moves on a communications sessions by transferring buffer ownership instead of copying the buffer contents. See *z/OS Communications Server: CSM Guide* for more information about CSM.

The controlling parameters of the vn_srx operation are passed in a msghdrx structure, which is pointed to from the UIO. Included in the msghdrx is a pointer to an array of structures, each of which points to a data buffer that is obtained from CSM. For more information about the msghdrx structure and the semantics of this operation, see srx_np (BPX1SRX, BPX4SRX) — Send or receive CSM buffers on a socket in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

The vn_srx call can be used on either connected or unconnected sockets.

**Specific processing notes**

– The following UIO fields are provided by the LFS:

| | |
|---|---|
| **UIO.u_hdr.cbid** | Contains UIO_ID (from the BPXYVFSI header file) |
| **UIO.u_hdr.cblen** | Specifies the length of the user_IO_structure |
| **UIO.u_buffaddr** | Specifies the address of a primary address space copy of the caller's msghdrx structure |
| **UIO.u_buffalet** | Specifies the ALET, 0, of the msghdrx structure |
| **UIO.u_count** | Specifies the length of the msghdrx structure that is being passed |
| **UIO.u_asid** | Specifies the ASID of the caller |
| **UIO.u_rw** | Specifies whether the request is a read (0) or a write (1) |
| **UIO.u_key** | Specifies the storage key of the caller |

– The msghdrx structure is defined in **bpxysrxh.h**.

– The user's msghdrx is copied into the kernel by the LFS, and this copy is passed to the PFS. This kernel msghdrx, with any changes that are made by the PFS, is copied back to the user after the operation.

– The use of Msghdrx_length=0 in BPX1SRX to determine support for this operation is handled by the LFS, and not passed down to the PFS.

**Serialization provided by the LFS**

The vn_srx operation is invoked with an exclusive latch held on the vnode.

**Security calls to be made by the PFS**: None.

# vn_symlink — Create a symbolic link

## Function

The vn_symlink operation creates a symbolic link to a pathname or an external name. A file that is named Link_name, of type "symbolic link", is created within the directory that is represented by Token_structure. The content of the symbolic link file is the pathname or external name that is specified in Pathname.

## Environment on entry and exit

See "Environment for PFS operations" on page 71.

## Input parameter format

```
vn_symlink  (Token_structure,
             OSI_structure,
             Audit_structure,
             Pathname_length,
             Pathname,
             attribute_structure,
             Link_name_length,
             Link_name,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Token_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to "LFS/PFS control block structure" on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, "Interface structures for C language servers and clients," on page 503 for its mapping.

**OSI_structure**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see "The OSI structure" on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | CRED |

**Length:** Specified by CRED.cred_hdr.cblen.

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to "Security responsibilities and considerations" on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**Pathname_length**
Supplied parameter
**Type:** Integer
**Length:** Fullword

A fullword that contains the length of Pathname. The Pathname can be up to 1023 bytes long.

**Pathname**
Supplied parameter
**Type:** Character string
**Length:** Specified by the Pathname_length parameter

An area that contains the pathname or external name for which a symbolic link is to be created.

A pathname can begin with or without a slash:

- If the pathname begins with a slash, it is an *absolute* pathname; the slash refers to the root directory, and the search for the file starts at the root directory.
- If the pathname does not begin with a slash, it is a *relative* pathname, and the search for the file starts at the parent directory of the symbolic link file.

A pathname contains no nulls.

An external name is the name of an object outside of the hierarchical file system. It may contain nulls.

**attribute_structure**
Supplied parameter
**Type:** ATTR
**Length:** Specified by ATTR.at_hdr.cblen.

An area that contains the file attributes that are to be set for the symbolic link being created. This area is mapped by typedef ATTR in the BPXYVFSI header file (see Appendix D).

**Link_name_length**
Supplied parameter
**Type:** Integer
**Length:** Fullword

A fullword that contains the length of Link_name. The Link_name can be up to 255 bytes long.

**Link_name**
Supplied parameter
**Type:** Character string
**Length:** Specified by Link_name_length parameter

An area that contains the symbolic link that is being created. Link_name contains no nulls.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_symlink service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_symlink service stores the return code. The vn_symlink service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_symlink service should support at least the following error values:

| Return_code | Explanation |
|---|---|
| EACCES | The calling process does not have permission to write in the directory that was specified. |
| EEXIST | Link_name already exists. |
| ENAMETOOLONG | Link_name is longer than is supported by the PFS. |
| ENOENT | The parent directory has been marked for deletion. |
| ENOSPC | The file system is out of space. |
| ENOSYS | The PFS does not support storing external links. |
| EROFS | Token_structure specifies a directory on a read-only file system. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_symlink service stores the reason code. The vn_symlink service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

**Overview of vn_symlink processing**

“Creating files” on page 32 provides an overview of symbolic link creation.

**Specific processing notes**

– The Token_structure that is passed on input represents the directory in which the symbolic link is to be created.

– The following attribute_structure fields are provided by the LFS:

| | |
|---|---|
| **ATTR.at_hdr.cbid** | Contains Attr_ID (from the BPXYVFSI header file) |
| **ATTR.at_hdr.cbLen** | Specifies the length of attribute_structure |
| **ATTR.at_genvalue** | When ((at_genvalue & S_IFEXTL) == S_IFEXTL) is true, the pathname is an external link. |

– An external link is a symbolic link with an extra file attribute bit stored by the PFS. The distinction between a normal symbolic link and an external link is only apparent in the attribute structures that are returned by the PFS for the link file. There is no difference in the way vn_readlink is processed, for example.

  If the PFS cannot store this external link bit, it must fail the vn_symlink request with ENOSYS.

– If the file that is named in the Name parameter already exists, the vn_symlink operation returns a failing return code.

– Refer to the **symlink()** function in the POSIX .1a standard (IEEE Std 1003.1a), draft 7, for more information on the semantics of this operation for a POSIX-conforming PFS.

**Serialization provided by the LFS**

The vn_symlink operation is invoked with an exclusive latch held on the vnode of the directory.

**Security calls to be made by the PFS**

The PFS is expected to invoke SAF's Check Access callable service to check that the user has write permission to the directory.

## Related services


- "vn_readlink — Read a symbolic link" on page 183
- "vn_link — Create a link to a file" on page 157
- "vn_remove — Remove a link to a file" on page 194

# vn_trunc — Truncate a file

## Function

The vn_trunc operation changes the length of an open file.

## Environment on entry and exit

See “Environment for PFS operations” on page 71.

```
vn_trunc     (Token_structure,
              OSI_structure,
              Audit_structure,
              File_length,
              Return_value,
              Return_code,
              Reason_code)
```

## Parameters

**Token_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | TOKSTR |
| **Length:** | Specified by TOKSTR.ts_hdr.cblen. |

The Token_structure represents the file (vnode) that is being operated on. It contains the PFS's initialization token, mount token, and the file token. Refer to “LFS/PFS control block structure” on page 16 for a discussion of this structure, and to the TOKSTR typedef in BPXYPFSI in Appendix D, “Interface structures for C language servers and clients,” on page 503 for its mapping.

**OSI_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | OSI |
| **Length:** | Specified by OSI.osi_hdr.cblen. |

The OSI_structure contains information that is used by the OSI operations that may be called by the PFS. See Chapter 6 for more information.

It also contains MVS-specific information that needs to be passed to the PFS, including SMF accounting fields, a work area, a recovery area, and an optional pointer to an output ATTR structure. For more details on the OSI structure, see “The OSI structure” on page 19.

This area is mapped by the OSI typedef in BPXYPFSI in Appendix D.

**Audit_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | CRED |
| **Length:** | Specified by CRED.cred_hdr.cblen. |

The Audit_structure contains information that is used by the security product for access checks and auditing. It is passed to most SAF routines that are invoked by the PFS.

Refer to “Security responsibilities and considerations” on page 12 for a discussion of security processing, and to the CRED typedef in BPXYPFSI in Appendix D for the mapping of this structure.

**File_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Doubleword |

A doubleword that contains the number of bytes to which the file size is to be set. Only positive values are passed by the caller.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_trunc operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_Code values must be filled in by the PFS when Return_value is −1. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_trunc operation stores the return code. The vn_trunc operation returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The vn_trunc operation should support at least the following error value:

| Return_code | Explanation |
|---|---|
| EROFS | The file is on a read-only file system. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the vn_trunc operation stores the reason code. The vn_trunc operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. These reason codes are documented by the PFS.

## Implementation notes

### Overview of vn_trunc processing

The vn_trunc changes the file size to File_length bytes.

### Specific processing notes

– The difference between vn_trunc and vn_setattr(truncate) is that vn_trunc is called for **ftruncate()**, and therefore does not do a security check. vn_setattr(truncate) is called for **truncate()** and must do a security check.

– When a file is truncated, all data from File_length to the original end of the file must be removed.

Full blocks are returned to the file system so that they can be used again, and the file size must be changed to the lesser of File_length or the current length of the file.

– When the file is expanded, its length is changed to File_length and unwritten bytes read between the old end-of-file and the new end-of-file are returned as zeros.
– The LFS ensures that the file is a regular file, open for writing if necessary, and that the File_length is not negative.
– When the file size is changed successfully, the PFS calls SAF's Clear Setid callable service.
– The LFS enforces any file size limits that may be in effect.
– Refer to the **ftruncate()** function in the POSIX .1a standard (IEEE Std 1003.1a), draft 7, for more information on the semantics of this operation for a POSIX-conforming PFS.

**Serialization provided by the LFS**

The vn_trunc operation is invoked with an exclusive latch held on the vnode of the directory.

**Security calls to be made by the PFS**

Clear Setid.

## Related services

- "vn_open — Open a file" on page 170

# Chapter 4. VFS servers

A VFS server is a program that registers as a VFS server with z/OS UNIX by calling the **v_reg()** function. There is no special system definition required to become a VFS server.

VFS servers must have appropriate privileges, which are defined as *superuser authority*. For more information on appropriate privileges, see Authorization in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*. This chapter describes:

- How to install a virtual file system (VFS) server
- How a VFS server is activated and deactivated
- The functions that must be provided by a VFS server
- The functions that are provided for it
- Security considerations

In this document, a VFS server is a program that uses the VFS callable services API to access objects in the z/OS UNIX file hierarchy.

This is not to be confused with DCE or other types of servers. For example, consider a file transfer program that moves files between z/OS UNIX and a workstation. If this program uses the **open(), read(),** and **write()** functions to access the files, it is certainly a "file server", but it is not the subject of this chapter. On the other hand, if this same program uses the **v_get()** and **v_rdwr()** functions, it is the type of server discussed here. Such a program could be written as a DCE server or as a set of LU 6.2 transactions, independent of which interface is used to access the files. So there is no strict relationship between a DCE server and a VFS server.

The VFS callable services API is designed to meet the requirements of an NFS- or DFS™-style server, but it is not limited to those applications. The main difference between the POSIX API and the VFS callable services API is that POSIX programs refer to files by pathnames and VFS servers refer to them by file identifiers (FIDs). VFS servers do their own pathname resolution to convert a pathname into a FID, and later use the FID to access the file. The FID is designed to be part of the NFS file handle that the Network File System returns to its clients. A file handle always refers to the same file. A pathname, on the other hand, may refer to different files over time, because of rename, remove/re-create, or symbolic link changes.

## Installation

A VFS server may be installed in the hierarchical file system or in standard MVS load libraries. The choice depends on how the VFS server is activated.

## Activation and deactivation

Because any program with appropriate privileges can become a VFS server by calling the **v_reg()** function, VFS servers can be activated in all the ways that a program can be run on MVS. They may be independent address spaces with their own START catalogued procedure; they can run as batch programs; or they can be shell processes that are run in the background or started through **/etc/init**. A VFS server can even be a command or program that is invoked directly by a user and run in the foreground of that user's process.

© Copyright IBM Corp. 1996, 2006

Once a program successfully calls **v_reg()**, it is registered as a VFS server with z/OS UNIX and dubbed, if it has not already been dubbed. After a server is registered, appropriate privileges are not needed for subsequent v_ functions.

Server registration is not inherited across **fork()** or **spawn()**.

A VFS server, like any other program, can use the standard file and socket APIs of z/OS UNIX, along with other MVS APIs. The VFS server aspects of the program have to do only with its use of the VFS callable services API.

## Termination considerations

There is no service provided to unregister with z/OS UNIX. If and when a VFS server's process terminates, z/OS UNIX removes its registration.

A VFS server can, however, release itself from all z/OS UNIX associations by calling undub (BPX1MPC), which also removes its registration as a VFS server.

When z/OS UNIX removes a VFS server's registration, all of the z/OS UNIX resources that are allocated to that VFS server are freed.

## Security responsibilities and considerations

The security structure of z/OS UNIX consists of two parts: the user's identity and the file's access control information. A VFS server is primarily concerned with the user's identity.

As a z/OS UNIX "superuser," a VFS server has free access to all z/OS UNIX resources. Consequently, it is the VFS server's responsibility to make sure that everything it does on behalf of a particular end user is done under the authority of that end user.

For a VFS server that is directly invoked by a local user, such as by a command, the simplest thing to do is to require that the invoker be a superuser. If the VFS server runs as a setuid program or is a more traditional client/server type of server, the rest of this section applies.

It is expected that a VFS server will assume the identity of its end user while making calls to z/OS UNIX services. This consists of several steps:

1. End users must be defined to both MVS and z/OS UNIX. They will have both an MVS user ID and a z/OS UNIX UID-GID pair.
2. The VFS server must know the MVS user ID of the end user.
3. The VFS server invokes SAF services to set up a security environment based on that MVS user ID.

   `RACROUTE REQUEST=VERIFY,ENVIR=CREATE` is used to initialize the MVS part of the security environment, and Init_USP is used to add the z/OS UNIX information.

   For acceptable performance, a VFS server should maintain enough state information so that it could save this security environment for a given end user and not have to re-create it on every request.
4. Before calling z/OS UNIX services for an end user, the VFS server updates its address space or task to assume the security environment that was set up by RACROUTE and Init_USP, by storing the ACEE from RACROUTE in the security environment field of the Task Control Block (TCBSENV).

   If this is a read or write function, the VFS server must decide whether file access checking is to be performed by the system. If the VFS server maintains enough state information to recognize the first reference by a particular end user to a particular file object, it can limit the overhead of access checking to that first reference. Otherwise, every read or write must be access checked. Other types of calls are unconditionally access-checked if access control is defined for the call.

After the call, or sequence of calls, for that end user, the VFS server reverts to its own security environment or sets up for the next end user.

5. When an end user is finished using the VFS server, the VFS server invokes `RACROUTE REQUEST=VERIFY,ENVIR=DELETE` to free the security environment.

Access control checks are performed by the PFSs that own the data. These checks are based on information that is associated with each individual file. The VFS server does not control these access checks except for read and write operations.

For more information about these interfaces, refer to *z/OS Security Server RACF Callable Services*.

## VFS server considerations for 64-bit addressing

For a server that is entirely 31-bit, no changes are required.

For v_op calls in AMODE 31:

- A server may set FuioAddr64 and use 64-bit addressing within the UIO to address its own buffers for the v_rdwr, v_readdir, and v_readlink operations.
- The UIO itself and all the calling parameters must be 31-bit addressable.

For v_op calls in AMODE 64:

- The server must set FuioAddr64 appropriately to indicate whether a 31-bit or a 64-bit buffer address is being passed.
- Register 1 and the parameter list must all be 64-bit addresses; the parameters themselves may be above or below 2 gigabytes.
- BPX1 callers must use the BPX4 entry names.

## Using the VFS callable services application programming interface

The VFS callable services API separates a VFS server from the logical file system (LFS) of z/OS UNIX. It is a set of protocols and callable services that deal with accessing objects in the file hierarchy.

This section describes the services that are provided to a VFS server and the requirements and responsibilities that are placed on a VFS server.

As described in Chapter 1, a VFS server is just one of many users of the file system. File requests that are made through the various APIs that are supported by z/OS UNIX are routed by the LFS to the PFS that owns or controls the file that is being referred to. The PFS cannot tell what kind of program originated these requests.

## Operations summary

The VFS callable services API contains the following functions:

*Table 5. VFS callable sevices API functions*

| Function | Description |
| --- | --- |
| v_access | Check access permissions |
| v_close | Close a file |
| v_create | Create a regular, FIFO, or character special file |
| v_export | Export a file system |
| v_fstatfs | Get file system attributes |
| v_get | Get a vnode from a file ID (FID) |
| v_getattr | Get attributes for a file |
| v_link | Create a hard link to a file |
| v_lockctl | Control locks |
| v_lookup | Look up a filename |
| v_mkdir | Create a directory |
| v_open | Open or create a file |
| v_rdwr | Read or write to a file |
| v_readdir | Read a directory |
| v_readlink | Read a symbolic link or external link file |
| v_reg | Register a process with the file system |
| v_rel | Release a vnode |
| v_remove | Remove a file |
| v_rename | Rename a file or directory |
| v_rmdir | Remove a directory |
| v_rpn | Resolve a pathname to a file system and a file |
| v_setattr | Set attributes of a file |
| v_symlink | Create a symbolic or external link |

## VFS server – LFS control block structure

Files are contained within mounted file systems, and the collection of all the files in all the mounted file systems forms the z/OS UNIX file hierarchy.

The LFS structures for files and file systems are not directly addressable by a VFS server. Consequently, files and file systems are abstracted somewhat on the VFS callable services API.

A file is represented to a VFS server by a vnode token with the following characteristics:

- A vnode token is similar to a POSIX file descriptor, in that it is the main input to all calls that refer to the file.
- Vnode tokens are obtained most often from **v_get()** and **v_lookup()**, but also from **v_rpn(), v_create(),** and **v_mkdir()**
- Vnode tokens are not inherited across **fork()**.

- Vnode tokens are released with **v_rel()**. All vnode tokens that are obtained must eventually be released. After **v_rel()** is called, any subsequent call with the same vnode token fails.
- A single vnode token may be cached by the VFS server and shared among many end users. A single vnode token can be used by several tasks at the same time, but **v_rel()** is mutually exclusive with all other operations.
- Many different vnode tokens can be obtained for the same file.
- A vnode token that has not been released is always valid for a call in the sense that the VFS server program will not abnormally end from using it.

  Files that are deleted are still accessible with existing vnode tokens. This is the same behavior that is expected for POSIX file descriptors that have not been closed. If the underlying real file system is unmounted with IMMEDIATE or FORCE, however, calls that are made with vnode tokens for files in that file system fail with an error code.

A file system is represented to a VFS server by a VFS token with the following characteristics:

- The VFS token represents a virtual file system (VFS). With NFS, for example, this corresponds to a client mount. The directory that is mounted becomes the root of this VFS.
- A VFS is a subset of some real mounted file system. VFS servers do not refer to the mounted file system directly.
- VFS tokens are obtained from **v_rpn()**, and they are never released.
- VFS tokens are used only with the **v_get()** function, which converts a file ID within a given VFS into a vnode token.
- All VFS tokens for VFSs that are contained within the same real mounted file system are the same.
- VFS tokens remain valid for as long as the underlying real file system is mounted.

  After the underlying file system is unmounted, **v_get()** with the prior VFS token fails with an error code. This remains true even if the real file system is remounted.

## Registration

A VFS server must register with z/OS UNIX by calling the **v_reg()** service.

**v_reg()** checks that the VFS server has appropriate privileges (is a superuser), and sets up support for the VFS callable services API.

The input to **v_reg()** is contained in the NREG structure and includes the name by which the VFS server is to be known.

A DFS-style file exporter also includes the name of an exit program that the LFS is to call before and after every vnode operation for files that are being exported.

Refer to Appendix D for a description of the information that is passed during registration.

## Mounting and unmounting

Servers do not physically mount file systems. NFS-style servers connect to the file hierarchy at the directory that their client has *NFS-mounted*, and they access only those files that are in these NFS-mounted directories or lower in the hierarchy within the same physically-mounted file system.

DFS-style servers export whole mounted file systems. They connect to the file hierarchy at the root directory of those file systems.

The Resolve Path Name service, **v_rpn()**, is called to implement an *NFS mount*. The input is the directory pathname, as sent by the client. The primary output is a VFS token for the file system that the directory belongs to and the file ID (FID) of the directory. These represent a VFS and its root directory. With this information the VFS server can access any file in the same file system at or below that directory in the hierarchy.

The export service, **v_export()**, is called by file exporters. Its input is a file system name and its output is the same as it is for **v_rpn().**

If several directories in the same real file system are mounted by NFS clients, the VFS server receives the same VFS token for each **v_rpn()** that is issued during those NFS mounts. This fact is not significant to the VFS server, which associates each VFS token that is obtained with the NFS mount that was performed; there should not be any concern for the physical mount structure that underlies the file hierarchy.

The pathname that is passed to **v_rpn()** may be a regular file; in fact, determining whether it is a file or a directory may be the sole objective of the operation. Usually, though, the pathname refers to a directory that serves as a base from which other files are accessed. This access involves pathname resolution, which is explained in the next section.

When a client NFS unmounts the directory, the VFS server can release whatever information it is maintaining about the mount. This includes releasing any cached vnode tokens. The VFS server does not have to inform z/OS UNIX or release the VFS token in any way.

When a file exporter is finished with a file system it calls **v_export()** to unexport it.

## Overview of NFS processing

To understand how the VFS callable services API is used, you need to understand the typical sequence of operations for a network file system (NFS) server.

There are three major interactions between an NFS client and its NFS server:
1. Mounting a pathname
2. Resolving the pathname of a file or directory
3. Accessing an individual file or directory

### Mounting a pathname

Initially, an end user at an NFS client mounts the pathname of a directory that resides at the VFS server's system onto some mount point directory at the client. These mounts are often done automatically during the initialization of the user's workstation. The VFS server object that is mounted may be a regular file, rather than a directory, in which case information in "Resolving the pathname of a file or directory" on page 251 does not apply. This section describes only mounting a directory at the VFS server. This directory is referred to as the "initial directory."

The flow for an "NFS mount" is as follows:

1. The initial directory pathname is sent to the VFS server through the Mount remote procedure call (RPC).
2. **v_rpn()** is called by the VFS server to resolve the pathname from the RPC into: a VFS token for the pathname object's file system; a vnode token for the object itself; and the file ID (FID) of the object.
3. The VFS server builds a structure to represent and remember this mount operation.

   A unique "mount key" is constructed and saved in the structure. This may be, for example, an index number into a mount table array or a time stamp. It is used later to find the mount structure.

   The VFS token is saved in the mount structure.
4. An NFS file handle is constructed from the FID, mount key, and other control information that is specific to this VFS server.
5. Either the vnode token of the object is cached, or **v_rel()** is called to release it.
6. The file handle of the object is returned to the client.

After this exchange, the client has a file handle for the initial directory that was mounted. This file handle is saved and associated with the local mount point. All end user references to files at or below the local mount point now refer to files in the VFS server's file hierarchy that are at or below the initial directory.

## Resolving the pathname of a file or directory

Subsequently, the client's user refers to a specific file by pathname, and the pathname is resolved locally, component by component, until an NFS mount point is reached.

The client then continues with the following process:

1. The lookup RPC is called with the initial directory file handle, which was saved with the NFS mount point, and the next name component of the pathname, which is the name after the mount point name.
2. The VFS server uses the mount key from the file handle to find the related mount RPC structure where the VFS token from **v_rpn()** was saved.
3. **v_get()** is called with that VFS token and the FID from the file handle. This call returns the vnode token for the directory that is represented by the file handle. If the vnode token had been cached, this step could be skipped.
4. **v_lookup()** is called with that directory vnode token and the component name from the RPC. This call returns the named object's vnode token, FID, and attributes.
5. An NFS file handle for the named object is constructed from its FID, the mount key, and other control information that is specific to this VFS server.
6. **v_rel()** is called to release the directory vnode token.
7. **v_rel()** is called to release the named object's vnode token.
8. The file handle and attributes of the object are returned to the client.
9. At the client the file handle represents the named object that was just looked up. The object's pathname is equal to that part of the original pathname that has been resolved so far. From the attributes that are returned, the client can tell what type of file the object is:
   - If it is a symbolic link, the readlink RPC is called to retrieve the link's contents.
   - If it is a directory, and there are more name components of the pathname to be resolved, the client moves on to the next name component and calls the lookup RPC with that name and the file handle that was just returned.

10. The VFS server continues with step 2 on page 251, and this loop continues iteratively through each name component of the remaining pathname string.

**Note:** This processing does not generally cross real mount points at the server. If a particular directory encountered during these lookups has been mounted on, lookups in that directory return files from that directory, not from the directory that was mounted over it. As a consequence, all files that are obtained from a given initial directory are in the same real mounted file system. This also means that an NFS client's view of the file hierarchy is different from that of a local user. NFS clients can see "underneath" real mount points that are reachable from the directories they have NFS-mounted. This is usually of no consequence, because most mount-point directories are empty. Refer to "v_lookup (BPX1VLK, BPX4VLK) — Look up a file or directory" on page 303 for a way to override this behavior.

After a pathname has been fully resolved to the file handle of an object in the VFS server's file hierarchy, the client can use that handle on later RPC requests to perform a specific function against that object. For example:

- If the user program does an **open()** and **read()** on a file, the client resolves the open's pathname and uses the file handle to satisfy the read by issuing a read RPC.
- For a **mkdir()**, the pathname is resolved up to the last name component, yielding the file handle of the parent directory in which the new directory is to be defined. The make_dir RPC is then called with this file handle and the last name component of the original pathname.
- For a **stat()**, the pathname is resolved to its end, and the file handle is used on a get_attributes RPC.
- The lookups and readlinks that are involved with pathname resolution itself are also examples of the use of a file handle for specific operations against the directory that is represented by the handle.

## Accessing an individual file or directory

After an object's file handle is available, the flow for a functional request is as follows:

1. The functional RPC is called with the object's file handle and other parameters that are specific to this function.
2. The VFS server uses the mount key from the file handle to find the related mount RPC structure in which the VFS token from **v_rpn()** was saved.
3. **v_get()** is called with that VFS token and the FID from the file handle. This returns the vnode token for the object that is represented by the file handle.
4. The appropriate VFS callable services API function is called to perform the operation that is requested by the RPC. The parameters of the call include the object's vnode token, from step 3, and the other parameters that are specific to this function.
5. **v_rel()** is called to release the object's vnode token.
6. The data or results of the function are returned to the client.

So long as the client has cached a file handle, the pathname resolution process does not have to be repeated, and files and directories can be immediately accessed by their handle. In particular, this simpler flow would be used for all reads and writes against an open file, since the client can save the file handle with the open structures.

**Notes:**

1. If the VFS server keeps enough state information, the **v_get()-v_rel()** pairs can be skipped by caching the vnode token that is used on a sequence of inbound RPC requests. Because NFS clients do not inform their servers when they are finished with a file handle, a server that is caching vnode tokens must eventually clean them up by calling **v_rel()**, after an inactivity timeout or with some other reclamation algorithm.
2. **v_rpn()** is the only VFS callable services API function that takes a pathname for the file it acts upon.

## Capabilities and restrictions for Version 4 NFS server processing in a sysplex environment

Starting with z/OS V1R7, z/OS UNIX supports Version 4 NFS server protocols. This support includes new **v_open()** and **v_close()** callable services, including support for file sharing semantics (share reservations), and enhanced lock control interfaces and functionality (provided by the **v_lockctl()** callable service).

The following capabilities and restrictions apply to Version 4 NFS server processing in a sysplex environment:

- To open a file with share reservations, the file must be owned by a system at the z/OS V1R7 level or higher. The following applies to files that are owned by remote systems:
  - If a file is owned by a remote system that supports share reservations, they will be enforced at the owning system for all open requests within the sysplex. At the owning system, an open request from a down-level remote system behaves just like a local open request.
  - If a file is owned by a remote system that does not support share reservations, the **v_open()** fails with return code EOPNOTSUPP, reason code JrNoShrsAtOwner. Move the file system to a sysplex member that supports share reservations.
- A file system that has active share reservations on any of its files can be moved to another system that supports share reservations and those share reservations will move with the files and continue to be enforced at the new owning system.

  A file system cannot be moved to a down-level system while there are active share reservations on any file in that file system. Any attempt to do so will fail with return code EINVAL, reason code JrCantMoveShares. Either move the file system to a sysplex member that does support share reservations, stop the NFS client applications that are holding share reservations on the files, or wait for those applications to complete.
- When share reservations exist on files that are owned by a remote system and that system crashes, the following occurs:
  - If the file system is taken over by another system that supports share reservations, the reservations will be reestablished and enforced at the new owning system.
  - If the file system is taken over by another system that does not support share reservations, the share reservations can no longer be enforced. The open tokens for the affected files will be invalidated and subsequent operations with those open tokens will be rejected with return code EIO, reason code JrShrsLost. Move the file system to a sysplex member that supports share reservations; the files can then be reopened as they were before.

  **Note:** You can use the AUTOMOVE parameter on the MOUNT command to restrict such takeovers only to systems that support share reservations.

- When a file system is owned by a remote system that does not support the Version 4 NFS server protocols, the following restrictions apply:
  - Enhanced blocker information is not available when a byte range lock request cannot be granted. In such a case, the output BRLM_Rangelock structure will contain zeros.
  - The new purge locks interfaces are not supported unless the masks map to the old functionality—that is, all clients and threads or all threads at a client. TID subsetting cannot be used.
  - The UnLoadLocks function is not supported.

## NFS file handles

As mentioned before, the VFS callable services API is designed to be used with NFS, and NFS uses file handles to represent files. Two advantages of NFS file handles over pathnames are that they are a smaller fixed length (usually 32 bytes long), and that they always refer to the same file object even if that object is renamed or if it is deleted and the pathname reused for another object. In the latter case, references to the file handle fail, but this is the desired result.

An NFS file handle contains two pieces of information that are needed to convert the handle back to a file. These are the file system in which the file resides and its identifier (FID) within that file system. The FID values, which are generated by PFSs that own the data, are unique within a file system, persistent, and never reused. File systems, however, do not have a persistent and dedicated identifier that can be used in an NFS file handle.

An NFS client expects file handles to be valid for as long as the corresponding VFS server object exists. To support their validity over system or VFS server restarts, the VFS server must maintain a disk file, or database, that retains some information about the NFS mounts that have been performed. With this database, the VFS server can create unique and persistent file system identifiers to be placed in the file handles along with the file's FID. This file system identifier was called a "mount key" in the previous section, and the following process makes it unique and persistent:

1. On each mount RPC, a unique "mount key" is generated. This can be, for example, an index into a mount table or a time stamp.

   The mount key can be reused after the client issues an unmount RPC. Presumably the client will not be using old file handles from directories that it has unmounted.

   The initial directory pathname from the RPC and the mount key are saved on disk. The file system name and directory FID are also saved.

   A mount structure is built to hold the mount key and VFS token. With the mount key the VFS server is able to find the mount structure and extract the VFS token.
2. Each file handle that is constructed contains the file's FID and the mount key for the mount RPC under which the file resides.
3. Each time the VFS server is started, it reads the mount file and rebuilds the corresponding mount structures with their saved mount keys.

   **v_rpn()** is called, with the saved pathname, to get a new VFS token, which is saved in the new mount structure.

   At this point the VFS server has re-created the mount state it had before the system was restarted, and it can field inbound RPCs and process their file handles.

4. With an old file handle the VFS server can find the new mount structure, since the mount key has not changed and the new VFS token is used on the subsequent call to **v_get()**.

A mount RPC refers to a specific initial directory, which, when the RPC arrived, was known by the pathname that is included with the RPC. That specific directory can be renamed or deleted and the pathname reused for another directory. If this happens, the **v_rpn()** that is issued by a VFS server after it restarts yields the VFS token and FID of a different directory. In this case, the same file handle used by a client before and after the VFS server restart refers to two different objects!

To help detect this situation, **v_rpn()** returns additional information about the real mounted file system that the initial directory belongs to. This includes the FILESYSTEM name used on the real mount command. By saving this name and the FID of the initial directory, along with the pathname and mount key, the VFS server can validate the output of **v_rpn()** after a restart.

After a restart **v_rpn()**, the old and new FIDs are compared to catch situations in which the pathname has been reused within the same real file system. The old and new FILESYSTEM names are compared in order to catch instances in which the pathname was reused across real file systems and happens to refer to an object with the same FID within the new file system. Getting the same FID is not so uncommon; because FIDs are usually generated sequentially, the local root of every real file system, for example, tends to have the same FID.

This scheme requires that the FILESYSTEM name not be reused for another file system, but this is somewhat easier to control. Generally, mount commands are issued only from the BPXPRMxx parmlib member that was used to start z/OS UNIX, or by a small set of people with special authorization. For HFS file systems, also, the FILESYSTEM name is the name of an MVS data set. Controls can be placed over who is able to rename or delete these data sets, and they cannot be renamed or deleted by anyone while they are mounted.

## DFS-style file exporters

The main difference between a DFS-style server, called a file exporter here, and an NFS-style server is that a file exporter controls both local and remote access to the file systems that it exports. It does this through the use of an exit program that is specified at the time the exporter registers with **v_reg().**

A file exporter exports entire mounted file systems with the **v_export()** function. Usually the exporter is set up with a list of file systems that it is to export, and these are exported during initialization.

An exported file system is made known to the network in general. End users at DFS-style clients access all network files through a single “DFS” mount point on their system. The clients call a name server to find files that they are interested in, and so they are not affected when the files are moved. This differs from an NFS-style client, whose user individually mounts directories from each remote system on particular local mount points. The location of the directory, and thus the files under it, is specified at mount time, and so cannot be changed without changing the mount at each client.

For vnode operations that do not originate with the file exporter itself, an exporter exit program is used to synchronize file changes. The exporter exit program is invoked before and after every vnode operation that is called for files within an exported file system. The exit program communicates with the file exporter to coordinate file sharing between local users and remote clients. In effect, the exit program is serving as a “DFS client” for all the local users of the exported file system. Only tokens that grant permission to continue with the vnode operation are transferred via the exit, and not file data. In this way the exit and file exporter ensure that when a local program reads a file it will see all changes that may have been made to this file by remote clients.

The general flow is:

1. The exit is loaded and called for initialization when **v_reg()** is called.
2. **V_export()** is called by the file exporter to identify the file systems that are being exported. **V_export()** has the same output as **v_rpn()**, and the file exporter proceeds to access local data in the same way that NFS-style servers do.
3. The exit program is called before and after every vnode operation for an exported file system that does not originate from the file exporter.

   The exit program can communicate with the file exporter address space through its own internal mechanisms, if necessary. Significant performance degradation is possible for exported file systems if the exit and exporter are not designed to minimize this communication.

   The OSI services are available to the exit program.

   The exit can cause the vnode operation to be rejected, with return and reason codes that are passed back to the caller.
4. The **osi_ctl()** service is available for asynchronous communication from the file exporter address space to the exit program.
5. The exit program is also called when a file system is unexported and when the file exporter terminates. In the latter case the exit program is also deleted.

The interface between the LFS and the exporter exit is the GXPL structure. Refer to Appendix D, “Interface structures for C language servers and clients,” on page 503 for the structure itself and the C prototype of the interface.

The exit program receives control in the kernel address space and in the following environment:

| | |
|---|---|
| **Authorization:** | Supervisor state, PSW key 0 |
| **Dispatchable unit mode:** | Task |
| **Cross memory mode:** | Any |
| **AMODE:** | 31-bit |
| **ASC mode:** | Primary mode |
| **Interrupt status:** | Enabled for interrupts |
| **Locks:** | Unlocked |
| **Control parameters:** | All parameters are in key 0 storage in the primary address space. The parameters are not fetch protected. |

### Registers at Entry

The contents of the registers on entry to the exit are:

| **Register** | **Contents** |
|---|---|
| **0** | Undefined |
| **1** | Parameter list address. The list contains one item that is the address of the Gxpl structure. |
| **2-12** | Undefined |

| | |
|---|---|
| **13** | Save area address, of a 136-byte save area. |
| **14** | Return address |
| **15** | Entry address |
| **AR0-15** | Undefined |

**Environment at Exit**

Upon return from the exit, the entry environment must be restored.

**Registers at Exit**

Upon return from the exit, the register contents must be:

| **Register** | **Contents** |
|---|---|
| **2-13** | Restored from the entry values |
| **0,1,14,15** | Undefined |
| **AR0-15** | Untouched or restored from the entry values. |

## Reading and writing

When reading and writing to files, the VFS server is responsible for maintaining file position and for having access checks performed.

Each call to **v_rdwr()** must specify:

- The file offset from which the operation is to start. This differs from the POSIX API, where the LFS maintains a file cursor.
- Whether security access checks are to be performed. If the VFS server maintains sufficient state information to associate a sequence of reads and writes from the same end user, it can limit these access checks to that end user's first reference, thus improving performance.

Additionally, the VFS server may request a "sync on write", which forces the current write, and all previously written data, to be saved to disk before **v_rdwr()** returns.

## Reading directories

To optimize directory reading, **v_readdir()** is designed to return as many entries as possible on each call.

The VFS server must maintain directory positioning if more than one call must be made to read an entire directory, and this section describes positioning:

The **v_readdir()** output buffer is mapped by the DIRENT structure, and its format is defined as follows:

- The buffer contains a variable number of variable-length directory entries. Only full entries are placed in the buffer, up to the buffer size specified, and the number of entries is returned on the interface.
- Each directory entry that is returned in the buffer has the following format:
  1. 2-byte Entry_length. This length field includes itself.
  2. 2-byte Name_length, which is the length of the following Member_name subfield.
  3. Member_name. A character field of length Name_length. This name is not null-terminated.
  4. File-system-specific data. If (Name_length + 4) = Entry_length, this field is not present. Whenever the field is present, however, it starts with the file's serial

number, st_ino, in 4 bytes. This field is not part of POSIX, but it is supported for special-use programs that are dealing with particular file systems that they know about.

- The entries can be packed together, and the length fields are not aligned on any particular boundary.

An example of an entry for the name *abc* would be X'0007 0003 818283' or X'000B 0003 818283 00001234' with a file serial number of X'1234' also returned.

Entries for "`.`" and "`..`" may or may not be returned by the PFS that owns the directory.

In order for successive calls to **v_readdir()** to proceed through a directory from the point at which the last one left off, the VFS server must specify the directory position at which the operation is to start. There are two different ways this can be done:

- **Cursor technique**. The cursor that is returned in the UIO contains PFS-specific information that locates the next directory entry. The VFS server is required to preserve the UIO cursor and the entire output buffer from the last **v_readdir()**, and present both of these on the next **v_readdir()**.

  The PFS may use the cursor as an offset into a simple linear directory file, ignoring the buffer; or it may use it as an offset into the previous output buffer of the last entry returned. The latter approach is used by a PFS with a tree-structured directory, where the previous entry name is used as a key to search for the next entry. That is, the last returned name, a 1-to-255-byte-long text string, is really the "cursor" for the directory position.
- **Index technique**. The index that is set in the UIO by the VFS server determines which entry to start reading from. To read through a directory, the VFS server starts at one and maintains the index by adding the number of entries that are returned to the previous index. The directory is treated as a one-based array, where the first entry has index **1**, the second entry has index **2**, and so on.

  This technique is slower than the cursor technique, but it is useful when a VFS server does not maintain state information from one call to the next. The index can be passed back to the client, who must return it with the next request to continue reading the same directory for a particular end user.

The UIO contains both the cursor and the index fields that are used with these continuation techniques. The interpretation of these two fields is summarized in the following table:

| Index | Cursor | Action |
|---|---|---|
| 0 | 0 | Start reading from the first entry. |
| 0 | M | Use the cursor value to resume reading. |
| N | 0 | Start reading from entry N. |
| N | M | Start reading from entry N. |
| **Note:** 0=zero; N and M are nonzero values. | | |

A nonzero index overrides the cursor; when both are zero, reading starts from the front of the directory.

The end of the directory stream is indicated in two different ways:

- A Return_value of 0 entries is returned. This happens when the previous **v_readdir()** exhausted the directory.
- A null name entry is returned as the last entry in the output buffer. A null name entry has an Entry_length of 4 and a Name_length of 0—that is, X'00040000'.

  This happens when the current **v_readdir()** exhausts the directory and there are at least 4 bytes left in the output buffer.

## Getting and setting attributes

A file's attributes are returned by the **v_getattr()** function. Many of the other VFS callable services API functions also return file attributes as a performance enhancement, since attributes are often requested in conjunction with those functions.

A file's attributes are changed with the **v_setattr()** function. A set of "change bits" are used on this interface, and the VFS server specifies exactly which attributes are being updated, along with the new values for those attributes.

## Comparing the VFS server and PFS interfaces

Certain traditional VFS or vnode functions are missing from the VFS callable services API. In particular, the set of functions in the VFS callable services API does not match the set of file-related operations in the PFS interface.

Some of these missing functions are not generally used by an NFS-style VFS server, and some of them are implemented in other ways, as explained in the following list.

**truncate**
: A file can be truncated with **v_setattr()**, specifying the desired file size.

**sync**
: A file can be synchronized, or saved to disk, with **v_rdwr()**. Specify write, a length of 0, and sync-on-write.

**open or close**
: NFS-style VFS servers do not use these operations. To maintain the performance characteristics of an open-close protocol, the VFS server can limit access checks to an end user's first reference to a particular file.

**inactivate**
: **v_rel()** is functionally equivalent for a VFS server to the vn_inactive operation for a PFS.

**mount or unmount**
: The **v_rpn()** function implements an NFS-style mount, and these are not explicitly unmounted.

**vfs_fid**
: A file's FID is part of the ATTR structure, so it can be obtained with the **v_getattr()** function. The ATTR is returned on the operations where a FID would usually be needed, so a VFS server generally does not have to explicitly convert vnodes into FIDs.

**vfs_root**
: An NFS-style server does not do real mounts, so it does not need to find the root of a real mounted file system. **v_rpn()** returns the root of a VFS server's VFS.

**check access**
: A VFS server does not explicitly check to see if its end user has permission to access a file; instead, it

assumes the user's identity and makes the file reference under that authority.

# Chapter 5. VFS callable services application programming interface

This chapter describes the syntax of each of the VFS callable services. The services are arranged in alphabetic order. Sample invocations of each service are in Appendix C.

## Syntax conventions for the VFS callable services

A callable service is a programming interface that uses the CALL macro to access system services. To code a callable service, code the CALL macro followed by the name of the callable service and a parameter list. A syntax diagram for a callable service follows.

```
CALL Service_name,(Parm_1,
                   Parm_2,
                   .
                   .
                   Return_value,
                   Return_code,
                   Reason_code)
```

This format does not show the assembler column dependence (columns 1, 10, 16, and 72) or parameter list options (VL and MF). The exact syntax is shown in the examples in Appendix C.

When you code a callable service:

- You must code all the parameters in the parameter list, because parameters are positional in a callable service interface. That is, the function of each parameter is determined by its position with respect to the other parameters in the list. Omitting a parameter, therefore, assigns the omitted parameter's function to the next parameter in the list.
- You must place values explicitly into all supplied parameters, because callable services do not set defaults.

## Elements of callable services syntax

The following paragraphs describe the standard elements that are contained in the callable services reference pages in this document.

### CALL

CALL is the assembler macro that transfers control and passes a parameter list.

### Service_name

The name that assembler understands is the name of a module in the form BPX1xxx, where *xxx* is a three-character symbol unique to the service. AMODE 64 callers use the form BPX4xxx.

Modules are invoked in one of the following ways:

- A program can load a module first and then branch to the address where it was loaded.
- When you are link-editing a program, you can link to the linkage stub. The program can issue a call.

© Copyright IBM Corp. 1996, 2006

- You can include in the code the system control offset to the callable service. See Appendix A for information on how to use this linkage.

### Parm parameters

Parm_1, Parm_2, and so on are placeholders for variables that may be part of a service's syntax.

### Return_value

The Return_value parameter is a common parameter for many callable services. It indicates the success or failure of the service. If the callable service fails, it returns `−1` in Return_value. For most successful calls to z/OS UNIX services, the return value is set to `0`. If the request is not successful, `−1` is returned.

### Return_code

The Return_code parameter is referred to as the *errno* in the POSIX C interface. The Return_code is returned only if the service fails.

In the callable service description, some of the possible return codes are listed for services that have return codes. The return codes are described in each service if they help describe its function.

Reason codes are listed with the return codes they describe.

The return codes and their descriptions are found in *z/OS UNIX System Services Messages and Codes*.

Some Return_code values may occur for any callable service: the ones that are unique to z/OS UNIX. They are not always listed under each callable service. See *z/OS UNIX System Services Messages and Codes* for a description of these return codes.

### Reason_code

The Reason_code parameter usually accompanies the Return_code value when the callable service fails. It further defines the return code. Reason codes do not have a POSIX equivalent.

*z/OS UNIX System Services Messages and Codes* lists all the reason codes with their descriptions, both alphabetically by name and numerically by value. The value is the lower half of the reason code.

## Other subjects related to callable services

See Invocation details for callable services in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*for a discussion of other subjects related to callable services, such as:
- How to invoke them
- Their linkage conventions
- Reentrant versus nonreentrant coding
- Environmental restrictions
- Abnormal end conditions
- Authorization

## Considerations for servers written in C

The BPXYVFSI header file in Appendix D, “Interface structures for C language servers and clients,” on page 503 contains prototypes and linkage macros for all the callable services in this section. With this header, you can call each service using the v_name that is shown in the title, and you will not have to linkedit your program with the linkage stubs.

This header also contains definitions for all structures, parameters, and constants that are used on the interface.

The calling parameters are the same for C and assembler, but the call format follows C syntax. For example, the call statement for creating a file would look like this:

```
v_create(directory_vnode_token, &oss, name_length, name, sizeof(ATTR),
        attribute_structure, &file_vnode_token, &return_value, &return_code,;
        &reason_code);
```

# v_access (BPX1VAC, BPX4VAC) — Check file accessibility

## Function

The v_access service verifies that the caller has the requested access permissions to the object that is represented by Vnode_token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VAC): | 31-bit |
| AMODE (BPX4VAC): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VAC,(Vnode_token,
              OSS,
              Mode,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VAC with the same parameters.

## Parameters

**Vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the file or directory.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Mode**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the permissions to be checked. This area is mapped by the BPXYMODE macro (see “BPXYMODE — Map the mode constants of the file services” on page 466).

The Read, Write, and Execute permissions that are to be checked are set in the Owner permission bits of the Mode (the S_IRUSR, S_IWUSR and S_IXUSR bits).

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_access service returns the results of the access check.

When the request is successful, the permission bits that correspond to the caller’s allowed access for each of the input mode bits are returned here. This is in the same format as the input Mode parameter, and is therefore a subset of the input Mode bits.

If the request is not successful, −1 is returned.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_access service stores the return code. The v_access service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_access service can return the following value in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; something other than the Owner’s permission bits were set. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_access service stores the reason code. The v_access service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. This service is similar to the access() function, but the return of information is handled differently, as follows:

| Return_value | Meaning |
|---|---|
| **0** | Access is denied for all of the bits that were on in the Mode parameter. |
| **Greater then zero** | The permissible access is represented by the non-zero bits that are returned here. |

| | |
|---|---|
| **–1** | The service has failed for some reason other then an access failure. |

2. The caller's real UID and real GID are used to check for the access that is requested.
3. All access is allowed to symbolic link files, regardless of the file's mode setting. This does not imply anything about whether access to the file that is pointed to by the symbolic link would be granted.
4. The setting of the AttrLP64times bit in the BPXYATT structure, and not the AMODE of the caller, determines whether 4-byte or 8-byte time fields are used.

## Related services

- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333

## Characteristics and restrictions

A process must be registered as a server before it can invoke the v_access service; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

# v_close (BPX1VCL, BPX4VCL) — Close a file

## Function

The v_close service closes a previous open created by v_open. This frees the open token and removes all state information associated with the v_open.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VCL): | 31-bit |
| AMODE (BPX4VCL): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VCL,(Vnode_token,
              OSS,
              Open_Token,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VCL with the same parameters.

## Parameters

**Vnode_token**
Supplied parameter
**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area that contains a vnode token that represents the file that was previously opened by v_open.

**OSS**
Supplied and returned parameter
**Type:** Structure
**Length:** OSS#LENGTH (from the BPXYOSS macro)

The name of an area that contains operating system specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Open_Token**
Supplied parameter
**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area that holds the open token returned by a prior call to v_open.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_close service returns 0 if the request is successful, or −1 if it is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_close service stores the return code. The v_close service returns Return_code only if the Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_open service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, the vnode token has been released or one of the token parameters does not contain a valid token value. |
| ESTALE | The open token is stale or already closed. |
| EAGAIN | The open token is currently in use by another thread in this process. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_close service stores the reason code. The v_close service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The v_close service frees the open token represented by Open_Token and releases all state information associated with it. This includes share reservations and byte range locks associated with the open instance.
2. Byte range locks are not associated with open tokens that are created with OPEN_NLM_SHR, so v_close will not release these. They must be explicitly released with the v_lockctl service.
3. In accordance with POSIX rules, when v_close releases byte range locks on a file, all locks owned by the open owner are also released—even those obtained by this open owner using other open tokens. Also, for any lock owner who is not the open owner but who is specified on a v_lockctl call using this open token, all of the locks on the file that are owned by that lock owner will be released.
4. When v_close releases pending asynchronous byte range locks, the request completion signal will be sent and the lock request will complete with an ECANCELED error.

**Note:** There is a race condition with the lock request completing normally just before the v_close is issued and, in this case, the lock request will successfully complete but the lock will have been released. This is similar to the case where one thread obtains a byte range lock on a file and another thread closes that file before the first thread has had a chance to use the lock.

5. If any other thread is currently issuing a call (such as v_rdwr) using the same open token that v_close is attempting to close, the v_close will fail with an EAGAIN error.
6. The v_rel service implicitly calls v_close for any open token that is associated with the vnode token that is being released.

## Related services

- “v_open (BPX1VOP, BPX4VOP) — Open or create a file” on page 311
- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333
- “v_rel (BPX1VRL, BPX4VRL) — Release a vnode token” on page 337

## Characteristics and restrictions

A process must be registered as a server before the v_open service is permitted; see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

# v_create (BPX1VCR, BPX4VCR) — Create a file

## Function

The v_create service creates a new file in the directory that is represented by Directory_vnode_token. The file can be a regular, FIFO, or character special file. The input Attr is used to define the attributes of the new file. A token that represents the new file is returned in File_vnode_token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VCR): | 31-bit |
| AMODE (BPX4VCR): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VCR,(Directory_vnode_token,
              OSS,
              Name_length,
              Name,
              Attr_length,
              Attr,
              File_vnode_token,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VCR with the same parameters.

## Parameters

**Directory_vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory in which the v_create service creates the new file that is named in the Name parameter.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Name_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the filename that is to be created. The name can be up to 255 bytes long.

**Name**

Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Name_length parameter |

The name of an area, of length Name_length, that contains the filename that is to be created. It must not contain null characters (X'00').

**Attr_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the area that is passed in the Attr parameter. To determine the value of Attr_length, use the ATTR structure (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**Attr**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area, of length Attr_length, that is to be used by the v_create service to set the attributes of the file that is to be created. The attributes of the file that is created are also returned in this area. This area is mapped by the ATTR structure (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**File_vnode_token**

Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area in which the v_create service returns a Vnode_token of the file created.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword where the v_create service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_create service stores the return code. The v_create service returns Return_code only if Return_value is `−1`. See *z/OS*

*UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_create service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | The calling process does not have permission to write in the directory that was specified. |
| EEXIST | The named file already exists. |
| EFBIG | The file size limit for the process is set to zero, which means files cannot be created. |
| EINVAL | Parameter error; for example, a supplied area was too small.<br><br>The following reason codes can accompany the return code: JRSmallAttr, JRInvalidAttr, JrNoName, JrNullInPath, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EMFILE | The maximum number of vnode tokens have been created. |
| ENAMETOOLONG | The name is longer than 255 characters. |
| ENFILE | An error occurred while storage was being obtained for a vnode token. |
| ENOTDIR | The supplied token did not represent a directory. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |
| EROFS | The Directory_vnode_token is a file on a read-only file system. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_create service stores the reason code. The v_create service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The following ATTR fields are provided by the caller:

   **Attr.at_hdr.cbid** — Contains Attr#ID (from the ATTR structure).

   **AttrLen** — Specifies the length of the ATTR structure.

   **AttrMode** — Specifies the file mode permission bits. See “BPXYMODE — Map the mode constants of the file services” on page 466 for the mapping of this field.

   **AttrType** — Specifies the file type: regular, FIFO, or character special. See “BPXYFTYP — File type definitions” on page 451 for the mapping of this field.

   **AttrMajorNumber** — Specifies the major number for character special files.

| | |
|---|---|
| **AttrMinorNumber** | Specifies the minor number for character special files. |
| **AttrCVerSet** | Indicates whether the Creation Verifier (AttrCVer) is present. |
| **AttrCVer** | Specifies the Creation Verifier for the file. When the AttrCVerSet bit is on and the create is successful, the PFS saves the Creation Verifier, and the server can retrieve it with v_lookup. The Creation Verifier allows the server to determine whether a v_create that returns EEXIST should be considered successful or not. If AttrCVerSet is on, AttrCVer is returned, and the server can compare the file's Creation Verifier with the input Creation Verifier on the v_create. If they are the same, it considers the v_create successful; that is, it is a duplicate of an earlier successful request. |

Other fields in the ATTR area should be set to zeros.

2. If the file that is named in the Name parameter already exists, the v_create service returns a failing return code, and no File_vnode_token is returned.
3. Vnode tokens that are returned by the v_create service are not inherited across a fork callable service.
4. The caller is responsible for freeing vnode tokens that are returned by the v_create service by calling to the v_rel service when they are no longer needed.
5. If the file size limit for the process is set to zero, files cannot be created and file creation fails with EFBIG.
6. The value set by **umask()** for the process does not affect the setting of the mode permission bits.

## Related services

- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_rel (BPX1VRL, BPX4VRL) — Release a vnode token" on page 337

## Characteristics and restrictions

A process must be registered as a server before the v_create service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VCR, BPX4VCR (v_create) example" on page 479.

# v_export (BPX1VEX, BPX4VEX) — Export a file system

## Function

The v_export service controls whether a file system is being exported by the server that makes this call.

Both local and remote access to this file system are controlled by the server while it is being exported.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VEX): | 31-bit |
| AMODE (BPX4VEX): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VEX,(OSS,
              Function,
              File_system_name,
              VFS_token,
              Vnode_token,
              Mnte_length,
              Mnte,
              Attr_length,
              Attr,
              Vol_Handle,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VEX with the same parameters.

## Parameters

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Function**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |

**Length:** Fullword

The name of a fullword that contains the function to perform:

1. Export the file system. This activates the server's control over the file system.
2. Unexport the file system. This deactivates the server's control over the file system.

**File_system_name**

Supplied parameter

**Type:** Character string

**Length:** 44 bytes

The name of a 44-character field that identifies the file system that is to be exported or unexported. The name must be left-justified and padded with blanks.

This is the name that is specified on the mount of the file system. It is an MVS data set name in uppercase letters without surrounding quotation marks.

**VFS_token**

Returned parameter

**Type:** Token

**Length:** 8 bytes

The name of an 8-byte area in which the v_export service returns the VFS token of the file system.

**Vnode_token**

Returned parameter

**Type:** Token

**Length:** 8 bytes

The name of an 8-byte area in which the v_export service returns a vnode token of the root of the file system.

**Mnte_length**

Supplied parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword that contains the length of the area that is to be passed in the Mnte parameter.

The length of this area must be large enough to contain a mount entry header (MnteH) and one mount entry (Mnte). These fields are mapped by the BPXYMNTE macro (see "BPXYMNTE — Map response and element structure of w_getmnte" on page 463).

**Mnte**

Returned parameter

**Type:** Structure

**Length:** Specified by the Mnte_length parameter

The name of an area, of length Mnte_length, in which the v_export service returns information about the file system. This area is mapped by the BPXYMNTE macro (see "BPXYMNTE — Map response and element structure of w_getmnte" on page 463).

**Attr_length**

Supplied parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword that contains the length of the area that is to be passed in the Attr parameter. To determine the value of Attr_length, use the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**Attr**

Returned parameter

**Type:** Structure

**Length:** Specified by the Attr_length parameter

The name of an area, of length Attr_length, in which the v_export service returns the file attribute structure for the root. This area is mapped by the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**Vol_Handle**

Supplied parameter

**Type:** Token

**Length:** 16 bytes

The name of a 16-byte area that is to be associated with the exported file system and passed to the exporter exit with each call that is related to this file system.

This parameter is not interpreted by the LFS.

**Return_value**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_export service returns 0 if the request is successful, or −1 if it is not successful.

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_export service stores the return code. The v_export service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_export service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
| --- | --- |
| EINVAL | Parameter error; for example, the file system that is to be exported or unexported is not mounted or is a sysplex client; or one of the supplied areas was too small. The following reason codes can accompany the return code: JrFileSysNotThere, JrBadEntryCode, JrSmallAttr, JrSmallMnte, JrInvalidOSS, JRCantExpClient. |
| EBUSY | The file system that is to be unexported is not exported by this server. |
| EIO | The file system is being unmounted (JrQuiescing). |
| EAGAIN | The file system has been quiesced (JrQuiesced), or is being asynchronously mounted (JrAsynchMount). |

| Return_code | Explanation |
|---|---|
| EALREADY | The file system that is to be exported is already being exported; or the file system that is to be unexported is not currently exported. |
| EMFILE | The maximum number of vnode tokens have been created. |
| ENFILE | An error occurred while storage was being obtained for a vnode token. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a file exporter. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_export service stores the reason code. The v_export service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. Vnode tokens that are returned by the v_export service are not inherited across a fork callable service.
2. VFS tokens that are returned by the v_export service are inherited across a fork callable service.
3. The caller is responsible for freeing the vnode token that is returned by the v_export service, by calling the v_rel() service when it is no longer needed.
4. The caller must be registered as a server of type *file exporter.*

   Refer to "DFS-style file exporters" on page 255 for more information on file exporters.
5. The v_export service is used to gain access to the file system for the server, and is similar to v_rpn() in this respect.

   V_export(), though, also activates the server's control over local access through use of the exporter exit that is specified on v_reg(). V_export() acts against a whole mounted file system, while v_rpn() acts against the files underneath arbitrary directories.
6. The file system is quiesced before it is exported or unexported, and new requests against the file system are suspended while it is being quiesced. If there is a lot of activity against this file system, the v_export request may take some time to complete, and may cause noticeable pauses for the users.
7. The mount point pathname is not returned in the Mnte structure that is returned by v_export.
8. On a call to unexport a file system, the VFS_token, Vnode_token, Mnte, Attr, and Vol_Handle parameters are not significant, though they are syntactically required for the call. The Mnte_length and Attr_length fields may be specified as 0, in this case.
9. The exporter exit is called during an unexport to notify it about this event.
10. When a file system is shared within a sysplex, it can only be exported from the sysplex server for that file system. Once a file system has been exported at the file system's sysplex server it cannot be moved within the sysplex until it is unexported. Attempts to v_export a sysplex client file system are rejected with

EINVAL/JrCantExpClient, and attempts to chmount(move) an already exported file system are rejected with EINVAL/JRIsExported.

## Related services

- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333
- “v_rel (BPX1VRL, BPX4VRL) — Release a vnode token” on page 337

## Characteristics and restrictions

A process must be registered as a file exporter before the v_export service is permitted; see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

# v_fstatfs (BPX1VSF, BPX4VSF) — Return file system status

## Function

The v_fstatfs service returns file system status for the file system that contains the file or directory that is represented by the supplied Vnode_token parameter.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VSF): | 31-bit |
| AMODE (BPX4VSF): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VSF,(Vnode_token,
              OSS,
              FsAttr_length,
              FsAttr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VSF with the same parameters.

## Parameters

**Vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents a file or directory that is contained in the file system for which status is being requested.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**FsAttr_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |

**Length:** Fullword

The name of a fullword that contains the length of the area that is passed in the FsAttr parameter. To determine the value of FsAttr_length, use the BPXYSSTF macro (see "BPXYSSTF — Map the response structure for file system status" on page 471).

**FsAttr**

Returned parameter

**Type:** Structure

**Length:** Specified by the FsAttr_length parameter

The name of an area, of length FsAttr_length, in which the v_fstatfs service returns file system status information. This area is mapped by the BPXYSSTF macro (see "BPXYSSTF — Map the response structure for file system status" on page 471).

**Return_value**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_fstatfs service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_fstatfs service stores the return code. The v_fstatfs service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_fstatfs service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallFsAttr , JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_fstatfs service stores the reason code. The v_fstatfs service returns a Reason_code only if Return_value is `−1`. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The supplied FsAttr structure must be at least SSTF#MINLEN (from the BPXYSSTF macro) bytes in length. The length of the structure is SSTF#LENGTH.

2. The input FsAttr structure length may not match the length that is supported by the file system. The file system returns the size that represents the amount of valid data in SSTFLEN.

## Related services

- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333

## Characteristics and restrictions

A process must be registered as a server before the v_fstatfs service is permitted; see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

## Examples

For an example using this callable service, see “BPX1VSF, BPX4VSF (v_fstatfs) example” on page 480.

# v_get (BPX1VGT, BPX4VGT) — Convert an FID to a vnode Token

## Function

The v_get service returns a vnode token for the file or directory that is represented by the input FID within the mounted file system that is represented by the input VFS token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VGT): | 31-bit |
| AMODE (BPX4VGT): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VGT,(VFS_token,
              OSS,
              FID,
              Vnode_token,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VGT with the same parameters.

## Parameters

**VFS_token**

Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains the VFS token for the mounted file system that contains the file or directory that is specified by the FID parameter. This token is obtained from the v_rpn callable service.

**OSS**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**FID**

Supplied parameter

**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area that contains the file identifier of the file or directory for which a vnode token is to be returned. The FID for a file is contained in the attribute structure for the file in the AttrFid field; the ATTR structure describes the attribute structure.

**Vnode_token**

Returned parameter

**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area in which the v_get service returns a vnode token of the file or directory that is supplied in the FID parameter. The token is used to identify the file or directory to other callable services.

**Return_value**

Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the v_get service returns 0 if the request completes successfully (the file or directory exists), or −1 if the request is not successful.

**Return_code**

Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the v_get service stores the return code. The v_get service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_get service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, the VFS token parameter is obsolete.The following reason codes can accompany the return code: JRStaleVFSTok, JRInvalidOSS. |
| EMFILE | The maximum number of vnode tokens have been created. |
| ENFILE | An error occurred obtaining storage for a vnode token. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**

Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the v_get service stores the reason code. The v_get service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The FID (file identifier) uniquely identifies a file in a particular mounted file system. For files associated with a physical DASD resource, the FID validly persists across mounting and unmounting of the file system, as well as z/OS UNIX re-IPLS. This distinguishes the FID from the vnode token, which relates to a file in active use, and whose validity persists only until the token is released via the v_rel callable service. Note that automount-managed directories are virtual, and the FID is unique only as long as the directory is being referenced.

   A server application uses v_get to convert a FID to a vnode token when it is preparing to use a file, because the Vnode token identifies the file to the other VFS callable services.
2. The FID for a file is returned in the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445), by such services as v_rpn and v_lookup.
3. vnode tokens that are returned by the v_get service are not inherited across a fork callable service.
4. The caller is responsible for freeing vnode tokens that are returned by the v_get service by calling to the v_rel service when they are no longer needed.

## Related services

- "v_create (BPX1VCR, BPX4VCR) — Create a file" on page 270
- "v_getattr (BPX1VGA, BPX4VGA) — Get the attributes of a file" on page 285
- "v_lookup (BPX1VLK, BPX4VLK) — Look up a file or directory" on page 303
- "v_mkdir (BPX1VMK, BPX4VMK) — Create a directory" on page 307
- "v_rdwr (BPX1VRW, BPX4VRW) — Read from and write to a file" on page 322
- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_rel (BPX1VRL, BPX4VRL) — Release a vnode token" on page 337
- "v_rpn (BPX1VRP, BPX4VRP)) — Resolve a pathname" on page 350
- "v_setattr (BPX1VSA, BPX4VSA) — Set the attributes of a file" on page 354

## Characteristics and restrictions

A process must be registered as a server before the v_get service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VGT, BPX4VGT (v_get) example" on page 481.

# v_getattr (BPX1VGA, BPX4VGA) — Get the attributes of a file

## Function

The v_getattr service gets the attributes of the file that is represented by Vnode_token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VGA): | 31-bit |
| AMODE (BPX4VGA): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VGA,(Vnode_token,
              OSS,
              Attr_length,
              Attr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VGA with the same parameters.

## Parameters

**Vnode_token**

Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the file.

**OSS**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Attr_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of Attr. To determine the value of Attr_length, use the BPXYATTR macro (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**Attr**

Returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area, of length Attr_length, in which the v_getattr service returns the file attribute structure for the file that is specified by the vnode token. This area is mapped by the BPXYATTR macro (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_getattr service returns 0 if the request is successful, or −1 if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_getattr service stores the return code. The v_getattr service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_getattr service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallAttr, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_getattr service stores the reason code. The v_getattr service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. All time fields in the Attr area are in POSIX format.
2. The setting of the AttrLP64times bit in the BPXYATT structure, and not the AMODE of the caller, determines whether 4-byte or 8-byte time fields are used.

3. The File Mode field in the Attr area is mapped by the BPXYMODE macro (see "BPXYMODE — Map the mode constants of the file services" on page 466). For information on the values for file type, see "BPXYFTYP — File type definitions" on page 451.

## Related services

- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333

## Characteristics and restrictions

A process must be registered as a server before the v_getattr service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VGA, BPX4VGA (v_getattr) example" on page 482.

# v_link (BPX1VLN, BPX4VLN) — Create a link to a file

## Function

The v_link service creates a link (Link_name) to the file that is specified by File_vnode_token in the directory that is specified by Directory_vnode_token. The link is a new name that identifies an existing file. The new name does not replace the old one, but provides an additional way to refer to the file. To rename an existing file, see "v_rename (BPX1VRN, BPX4VRN) — Rename a file or directory" on page 343.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VLN): | 31-bit |
| AMODE (BPX4VLN): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VLN,(File_vnode_token,
              OSS,
              Link_name_length,
              Link_name,
              Directory_vnode_token,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VLN with the same parameters.

## Parameters

**File_vnode_token**

Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the file to which a link is to be established.

**OSS**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see "BPXYOSS — Map operating system specific information" on page 469).

**Link_name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of Link_name. The name can be up to 255 bytes long.

**Link_name**
Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Link_name_length parameter |

The name of an area, of length Link_name_length, that contains the name by which the file is to be known. It must not contain null characters (X'00').

**Directory_vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory from which the v_link service is to create the link that is supplied in the Link_name parameter.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_link service returns `0` if the request completes successfully, or `−1` if the request is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_link service stores the return code. The v_link service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_link service can return one of the following values in the Return_code parameter:

| **Return_code** | **Explanation** |
|---|---|
| EACCES | The process did not have appropriate permissions to create the link. Possible reasons include:<br>• The process had no write permission for the directory that is intended to contain the link.<br>• The process had no permission to access the file that is specified by File_vnode_token. |
| EEXIST | A file, directory, or symbolic link named Link_name already exists. |
| EINVAL | Parameter error; for example, one of the vnode tokens is stale. The following reason codes can accompany the return code: JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS, JRNoName, JRNullInPath. |

| Return_code | Explanation |
|---|---|
| EMLINK | The file that is specified by File_vnode_token already has its maximum number of links. The maximum number is LINK_MAX. The value of LINK_MAX can be determined through pathconf (BPX1PCF) or fpathconf (BPX1FPC). |
| ENAMETOOLONG | Link_name_length exceeds 255 characters. |
| ENOSPC | The directory that is intended to contain the link cannot be extended to contain another entry. |
| ENOTDIR | Directory_vnode_token does not specify a directory. The following reason code can accompany the return code: JRTokNotDir. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server; or the File_vnode_token specifies a directory. The following reason codes can accompany the return code: JRNotRegisteredServer, JRTokDir. |
| EROFS | Creating the link would require writing on a read-only file system. The following reason code can accompany the return code: JRLnkROFileSet. |
| EXDEV | The file that is specified by File_vnode_token and Directory_vnode_token are on different file systems. The following reason code can accompany the return code: JRLnkAcrossFileSets. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_link service stores the reason code. The v_link service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. BPX1VLN creates a link named Link_name to an existing file that is specified by File_vnode_name. This provides an alternate pathname for the existing file; the file may be accessed by the old name or the new name. The link may be stored under the same directory as the original file, or under a different directory on the same file system.
2. If the link is created successfully, the service routine increments the link count of the file. The link count shows how many links to a file exist. (If the link is not created successfully, the link count is not incremented.)
3. Links are not allowed to directories.
4. If the link is created successfully, the change time of the linked-to file is updated, as are the change and modification times of the directory that contains Link_name (that is, the directory that holds the link).

## Related services


- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333
- “v_rel (BPX1VRL, BPX4VRL) — Release a vnode token” on page 337
- “v_remove (BPX1VRM, BPX4VRM) — Remove a link to a file” on page 339
- “v_rename (BPX1VRN, BPX4VRN) — Rename a file or directory” on page 343

## Characteristics and restrictions

A process must be registered as a server before the v_link service is permitted, See “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

## Examples

For an example using this callable service, see “BPX1VLN, BPX4VLN (v_link) example” on page 483.

# v_lockctl (BPX1VLO, BPX4VLO) — Lock a file

## Function

The v_lockctl service controls advisory byte-range locks on a file.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VLO): | 31-bit |
| AMODE (BPX4VLO): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VLO,(OSS,
              Command,
              Vlock_length,
              Vlock,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VLO with the same parameters.

## Parameters

**OSS**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Command**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains one of the integer values that is mapped in the BPXYVLOK macro and indicates the action that is to be performed. For the list of commands, see “BPXYVLOK — Map the interface block for v_lockctl” on page 474.

**Vlock_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |

**Length:** Fullword

The name of a fullword that contains the length of Vlock. To determine the value of Vlock_length, use the BPXYVLOK macro (see “BPXYVLOK — Map the interface block for v_lockctl” on page 474).

**Vlock**
Supplied and returned parameter

**Type:** Structure
**Length:** Specified by the Vlock_length parameter

The name of an area that contains the lock request information. This area is mapped by the BPXYVLOK macro (see “BPXYVLOK — Map the interface block for v_lockctl” on page 474).

**Return_value**
Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the v_lockctl service returns 0 if the request is successful, or −1 if it is not successful.

**Return_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the v_lockctl service stores the return code. The v_lockctl service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_lockctl service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EAGAIN | The Lock command was requested, but the lock conflicts with a lock on an overlapping part of the file that is already set by another locker. |
| EDEADLK | The Lockwait command was requested, but the potential for deadlock was detected. The following reason codes can accompany the return code: JRBrlmDeadLockDetected, JRBrlmPromotePending, JRBrlmAlreadyWaiting, JRBrlmUnlockWhileWait. |
| EINTR | A LockWait request was interrupted by a signal. |
| EINVAL | Parameter error. The following reason codes can accompany the return code: JRBadEntryCode, JRInvalidVlok, JRInvalidServerPid, JRNoLockerToken, JRBrlmLockerNotRegistered, JRBrlmBadLType, JRBrlmObjectMissing, JRBrlmInvalidRange, JRBrlmBadL_Whence. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a lock server. |
| ENOENT | The LockCancel command was requested, but an exactly matching lock request was not found on the object’s waiting queue. |

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the v_lockctl service stores the reason code. The v_lockctl service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The v_lockctl service locks out other cooperating lockers from part of a file, so that the locker can read or write to that part of the file without interference from others.

> **Important note**
>
> All locks are advisory only. Client and local processes can use locks to inform each other that they want to protect parts of a file, but locks do not prevent I/O on the locked parts. A process that has appropriate permissions on a file can perform whatever I/O it chooses, regardless of the locks that are set. Therefore, file locking is only a convention, and it works only when all processes respect the convention.

2. **Registering as a locker (Vlok#RegLocker):** Each locker must be registered before it issues any lock requests. On a **Vlok#RegLocker** command, the following Vlock fields are provided by the caller:

   **VlokID**
   The Vlok#ID (from the BPXYVLOK macro).

   **VlokLen**
   The length of the Vlock structure.

   **VlokServerPID**
   The process ID of the lock server. If 0 is specified, the caller's PID is used.

   **VlokClientPID**
   A server-generated process ID that uniquely identifies the client within this server PID.

   Other fields in the Vlock area should be set to zeros.

   The following Vlock field is returned to the caller:

   **VlokLockerTok**
   A token to identify the locker on subsequent lock requests.

3. On a **Vlok#Query, Vlok#Lock, Vlok#LockWait,** or **Vlok#Unlock** command, the following Vlock fields are provided by the caller:

   **VlokID**
   The Vlok#ID (from the BPXYVLOK macro).

   **VlokLen**
   The length of the Vlock structure.

   **VlokLockerTok**
   The locker.

   **VlokClientTID**
   The client's thread ID.

   **VlokObjClass**
   The file object class. The possible classes are defined in the BPXYVLOK macro; see "BPXYVLOK — Map the interface block for v_lockctl" on page 474.

   **VlokObjID**
   The file object uniquely within the class. For an HFS file, VlokObjID contains the device number and FID of the file.

   **VlokObjTok**
   A token that was returned on the previous lock

request for this object. This field is optional, but will improve performance for multiple lock requests.

**VlokBrlk**
Lock information describing the byte-range. This area is mapped by BPXYBRLK (see note 5). The following BPXYBRLK fields must be provided:

| Command | Required fields |
|---|---|
| Vlok#Query | l_type, l_whence, l_start, l_len |
| Vlok#Lock | l_type, l_whence, l_start, l_len |
| Vlok#LockWait | l_type, l_whence, l_start, l_len |
| Vlok#Unlock | l_whence, l_start, l_len |

**VlokVnToken**
(Optional) The vnode token for a UNIX file system object. The use of this optional parameter can improve the performance of any operation that specifies a file system object. Additionally, for the **Vlok#UnLoadLocks** function, this also indicates that share reservations for the file are to be appended to the unloaded byte range locks (see note 11 for more information).

Other fields in the Vlock area should be set to zeros.

The following Vlock fields are returned to the caller:

**VlokObjTok**
A token to identify the object on a subsequent lock request.

**VlokBrlk**
On Query, lock information that describes a lock that would prevent the proposed lock from being set.

4. On a **Vlok#Lock**, **Vlok#LockWait**, and **Vlok#LockAsy** command, the caller can pass an open token in the OSS by providing the following field:

   **OssOpenToken**
   Contains an open token with which the byte range lock should be associated

   For open tokens other than those created with OPEN_NLM_SHR, the lock owner becomes associated with the open token. Thus, when a **v_close()** is issued using that open token, all byte range locks on this file that were obtained by this lock owner will be released.

5. Locking operations are controlled with a structure that is mapped by BPXYBRLK. This structure is needed whether the request is for setting a lock, releasing a lock, or querying a particular byte range for a lock. The following is a description of the BPXYBRLK structure:
   - The **l_type field** specifies the type of lock that is to be set or queried. (l_type is not used on Unlock.) Valid values for l_type are as follows

     **Type** — **Description**

     **F_RDLCK**
     A *read lock*. Specified as a halfword integer

| | |
|---|---|
| | value of 1, this is also known as a *shared lock*. This type of lock specifies that the locker can read the locked part of the file, and other lockers cannot write on that part of the file in the meantime. A locker can change a held write lock, or any part of it, to a read lock, thereby making it available for other lockers to read. Multiple lockers can have read locks on the same part of a file simultaneously. |
| **F_WRLCK** | A *write lock*. Specified as a halfword integer value of 2, this is also known as an *exclusive lock*. This type of lock indicates that the locker can write on the locked part of the file, without interference from other lockers. If one locker puts a write lock on part of a file, no other locker can establish a read lock or write lock on that same part of the file. A locker cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file, unless that locker is the only owner of that overlapping read lock. In such a case, the read lock on the overlapping section is replaced by the write lock that is being requested. |
| **F_UNLCK** | Returned on a Query, when there are no locks that would prevent the proposed lock operation from completing successfully. Specified as a halfword integer value of 3. |

- The **l_whence field** specifies how the byte-range offset is to be found within the file. The only valid value for l_whence is SEEK_SET, which stands for the start of the file, and is specified as a halfword integer value of 0.
- The **l_start field** identifies the part of the file that is to be locked, unlocked, or queried. The part of the file that is affected by the lock begins at this offset from the start of the file. For example, if l_start is the value 10, a Lock request attempts to set a lock beginning 10 bytes past the start of the file.

  **Note:** Although you cannot request a byte range that begins or extends beyond the beginning of the file, you can request a byte range that starts or extends beyond the end of the file.
- The **l_len field** gives the size of the locked part of the file, in bytes. The value that is specified for l_len may be negative. If l_len is positive, the area that is affected begins at l_start and ends at l_start + l_len-1. If l_len is negative, the area that is affected starts at l_start+l_len and ends at l_start-1. If l_len is zero, the locked part of the file begins at the position that is specified by l_whence and l_start, and extends to the end of the file.
- The **l_pid field** identifies the ClientProcessID of the locker that holds the lock found on a Query request, if one was found.

6. **Obtaining locks (Vlok#Lock and Vlok#LockWait):** Locks can be set by specifying a **Vlok#Lock** as the Command parameter. If the lock cannot be obtained, a return value of −1 is returned, along with an appropriate return code and reason code.

Locks can also be set by specifying **Vlok#LockWait** as the Command parameter. If the lock cannot be obtained because another process has a lock on all or part of the requested range, the LockWait request waits until the specified range becomes free and the request can be completed.

If a signal interrupts a call to the v_lockctl service while it is waiting in a LockWait operation, the function returns with a return value of −1, and a return code of EINTR.

LockWait operations have the potential for encountering deadlocks. This happens when locker A is waiting for locker B to unlock a region, and B is waiting for A to unlock a different region. If the system detects that a LockWait request might cause a deadlock, the v_lockctl service returns with a return value of −1 and a return code of EDEADLK.

7. **Asynchronous locking:**
   - *Obtaining an asynchronous lock (Vlok#LockAsy):* The Vlok#LockAsy command parameter is used to request an asynchronous lock. The lock request is either satisfied immediately or is queued for asynchronous completion. The v_lockctl call will not block. The caller should expect to receive the asynchronous lock completion through the **sigtimedwait()** or **sigwaitinfo()** interfaces. These provide an event queue for lock completions based on queued signals and is the same as that used with asynchronous I/O completions. The caller can specify the signal number and signal value to pass back on the asynchronous completion.

     The Vlock structure is set up just as it would be for the Vlok#LockWait function with the addition of a caller-supplied Aiocb structure that specifies the signal information and holds the results of the completed asynchronous request. The new fields in the Vlock structure for this function are:

     **VlokAiocbLen** Length of the Aiocb structure

     **VlokAiocb** Address of the Aiocb structure

     The Aiocb must remain valid for the life of an asynchronous request and its use is similar to that for an aio_read call. The following Aiocb fields are provided by the caller:

     **aio_sigevent.sigev_signo** The signal number

     **aio_sigevent.sigev_value** An application-specific data value to be passed with the signal

     **aio_exitdata** An application data area (not touched by the system)

     The rest of the Aiocb should be zeroed out.

     The following Aiocb fields are returned to the caller:

     **aio_rv** The return value

     **aio_rc** The return code

     **aio_rsn** The reason code

     The Return_value from **v_lockctl()** indicates the outcome of the call, as follows:

     **+1** The lock will be granted asynchronously.

     **0** The lock was granted immediately.

     **–1** The lock request failed as indicated by the accompanying return code and reason code.

     When the Return_value from **v_lockctl()** is +1, the final result of the lock request is determined when the completion signal is pulled from the signal queue using **sigtimedwait()** or **sigwaitinfo()**. At that point, the aio_rv field will contain `0` if the lock was granted or `−1` (with accompanying values in aio_rc and aio_rsn) if the lock was not granted. Generally, a request will only fail asynchronously if it is canceled.

When the Return_value from **v_lockctl()** is `0` or `−1`, the request has immediately succeeded or failed, respectively, and no signal is sent.

As with any asynchronous operation, the request may complete before the v_lockctl() call returns to the caller.

A lock owner may only have one outstanding lock request at a time on any particular range. This includes pending asynchronous requests and blocked synchronous requests. In other words, waiting locks for the same owner cannot intersect. Similarly, unlock requests may not be issued for any range that intersects with a pending lock request from the same lock owner.

- *Canceling an asynchronous lock request (Vlok#LockCancel):* To cancel a specific, outstanding asynchronous lock request, call the v_lockctl service with a command parameter of **Vlok#LockCancel** and a Vlock structure that contains all the information from the original Vlok#LockAsy request: object, owner, Brlk information, and Aiocb.

  You must use the same Aiocb on both the original Vlok#LockAsy request and theVlok#LockCancel request and the Aiocb must not have been modified between the two calls. When the Vlok#LockAsy request returns with a return value of 1, an asynchronous request token is also returned in the Aiocb and that token must be present on any subsequent call to cancel the lock request.

  An asynchronous lock request can only be canceled if it is still waiting for the lock to be granted. When a pending request is successfully canceled, the Return_value from **v_lockctl()** will be `0` and a lock completion signal will be sent with an aio_rc of ECANCELED. When an exact match for the request is not found on the object's waiting queue, the Return_value from **v_lockctl()** will be `−1` and the Return_code will be ENOENT.

  There is a race condition between a pending lock being canceled and its being granted, so there is always a chance that the call to cancel the lock request will fail because the successful lock completion signal has already been sent. Note, too, that at the time the v_lockctl call to cancel the lock request returns to the caller, the completion signal (either for the lock being granted or for its being canceled) may still be on the application's signal queue. Therefore, the application must handle the coordination between the caller of the cancel request and the handler of the completion signal.
- Refer to note 16 for the effects of a purge request on asynchronous locks.
- *Effects of changing file system ownership in a sysplex:* If the ownership of a file system is changed within a sysplex environment (for instance, by using the chmount shell command), pending asynchronous locks will be lost. This special situation is indicated by a lock failure of the original request with an aio_rc of EAGAIN and a lower half-word value in aio_rsn of `0x0607` (the value of the JrOwnerMoved reason code). The v_lockctl call must be issued again to request the asynchronous lock from the new owner. At such time, the lock may be immediately granted or it may again enter a pending state.

8. **Determining lock status (Vlok#Query):** A process can determine locking information about a file by using **Vlok#Query** as the Command parameter. The VlokBrlk structure should describe a lock operation that the caller would like to perform. When the v_lockctl service returns, the structure is modified to describe the first lock found that would prevent the proposed lock operation from completing successfully.

If a lock is found that would prevent the proposed lock from being set, the Query request returns a modified structure whose l_whence value is always SEEK_SET, whose l_start value gives the offset of the locked portion from the beginning of the file, whose l_len value is set to the length of the locked portion of the file, and whose l_pid value is set to the ClientProcessID of the locker that is holding the lock. If there are no locks that would prevent the proposed lock operation from completing successfully, the returned structure is modified to have an l_type of F_UNLCK, but otherwise it remains unchanged.

9. **Multiple lock requests:** A locker can have several locks on a file simultaneously, but can have only one type of lock set on any given byte. Therefore, if a locker sets a new lock on part of a file that it had previously locked, the locker has only one lock on that part of the file, and the lock type is the one that was given by the most recent locking operation.

10. **Returning blocker information:** A request to the v_lockctl service that cannot be granted can return information about the lock that is blocking the request from being granted. The blocking lock shares at least part of the range that was requested and may be from a granted lock range or a waiting lock request. The returned information is in the form of a BRLM_RangeLock structure, defined in IGWLBINT for PL/X and in BPXYVFSI for C.

    The caller requests the return of blocker information by specifying in **VlokBlockingLock** the address of an area in primary storage where the output BRLM_RangeLock may be placed. **VlokBlkLockLen** specifies the length of this output area. The storage for the output area is assumed to be in the caller's key.

    Blocker information can be returned in the following cases:
    - A Vlok#Lock or Vlok#LockWait request fails with a return code of EAGAIN or EDEADLK
    - A Vlok#Query request finds a blocking lock
    - A Vlok#LockAsy request returns with a Return_value of +1

    The output BRLM_RangeLock area (or, at a minimum, the server PID in the first word) should be zeroed out before the call to the v_lockctl service. If the contents are changed upon completion of the call, then information about a blocking lock was returned. Note that the blocking lock was blocking this request when the v_lockctl call was issued but is subject to change at any time.

11. **Query all locks for an object (Vlok#UnLoadLocks):** The Vlok#UnLoadLocks request provides an interface to the BRLM UnloadLocks function and also obtains the share reservations for file system objects.

    The information is returned as a chain of BRLM_UnloadLocksList structures, each of which contains control information and an array of (Object, Rangelock) pairs, each of which describe one locked range or share reservation. The storage for the chain of structures is obtained in the caller's primary address space, is in the caller's key, and is owned by the caller's TCB. Each structure in the chain must be freed by the caller using the MVS storage release service. The unloaded lock list segments may be of different lengths so the ull_length field must be used when the storage is released. These structures are defined in IGWLBINT for PL/X and in BPXYVFSI for C.

    The following Vlock fields are provided by the caller:

    **VlokObject**
    The class and ID of the object

    **VlokUllSubPool**
    An MVS storage subpool number for the areas to be obtained. For unauthorized callers, this number must be between 1 and 127.

**VlokUllRetWaiters** When set to 1, all locks are returned, including waiting locks, pending asynchronous locks, and held locks. When set to 0, only held locks are returned. Waiting locks are identified by the RlWaiting flag in the BRLM_Rangelock structure.

**VlokVnToken** (Optional) A vnode token for the VlokObject. Also indicates that the object's share reservations should be appended to the byte range locks that are returned. This must be the same file as identified by the VlokObject.

The following Vlock field is returned to the caller:

**VlokUllOutListPtr** The address of the first member of the output chain of BRLM_UnloadLocksList structures, or zero.

Zero or more BRLM_UnloadLocksList structures will be produced by BRLM. For file system objects when a vnode token is supplied, the unloaded locks will be followed by zero or more BRLM_UnloadLocksList structures for the share reservations. Share reservations may be placed in the unused slots of the last BRLM structure. The BRLM_UnloadLocksList structures may have varying numbers of locks returned in their array section so the ull_count field must be used to step through the arrays. The Return_value will contain the total number of locks and share reservations that were returned.

For each byte range lock, the rl_access field will be set to the type of lock: rl_shared, rl_excl, or rl_shr2excl.

For each share reservation, the rl_access field will be set to rl_openmodes. The rl_openacc and rl_opendeny fields will be set to the current Shr_Access and Shr_Deny modes, respectively, for that open. (Refer to "v_open (BPX1VOP, BPX4VOP) — Open or create a file" on page 311 for more information about these modes.)

12. **Releasing locks (Vlok#Unlock):** When an Vlok#Unlock request is made to unlock a byte region of a file, all locks that are held by that locker within the specified region are released. In other words, each byte that is specified on an Unlock request is freed from any lock that is held against it by the requesting locker.
13. Locks are not inherited by a child process that is created with the fork service.
14. **Effects of close and process termination:** All locks (those that are owned, pending, or waiting) for a given lock owner on a specific file will be released if any of the owner's open tokens for that file are closed with a v_close call. This includes any open token that was opened by this lock owner or one that was opened by a different lock owner but was subsequently used by this lock owner on a v_lockctl call. Owned locks are unlocked; pending and waiting locks are canceled. (This does not apply to open tokens created with OPEN_NLM_SHR.)

    If the registered server process terminates, all locks that are associated with this process are unlocked or canceled. Since the process is terminating, lock completion signals will not be delivered.
15. If the lock server terminates, all locks are released.
16. **Purging locks (Vlok#Purge):** The Vlok#Purge command releases all locks on all files that are held by a locker or a group of lockers. This is primarily a pass

through to BRLM. It will purge all types of byte range locks: held locks, waiting locks, or pending asynchronous locks. It does not affect share reservations or open tokens.

The purge interface is implemented using two bit masks that are logically ANDed with the object ID and owner ID, respectively, of each lock before they are compared with the passed arguments. The algorithm is as follows:

```
if ( (PassedObject == (LockObject & PassedObjectMask))
      && (PassedOwner == (LockOwner & PassedOwnerMask)) )
  { The lock will be purged. }
```

This purge function is enhanced and extended from the previously existing v_lockctl purge function. The following Vlock fields are provided by the caller:

**VlokObject**
The object's 16-byte identifier

**VlokServerPID**
The process ID of the lock server whose locks are to be released.

**VlokClientPID**
A server-generated process ID that uniquely identifies the client whose locks are to be released. If binary ones are specified, all locks for all clients of the specified server are released.

**VlokClientTID**
The client's thread ID for which locks are to be released. If binary ones are specified, all locks for the specified client and server are released.

**VlokPgMasks**
Points to a pair of 16-byte bit masks for the object and owner, respectively. These are defined as VlokObjectMask and VlokOwnerMask.

**VlokPgMaskslen**
Specifies the length of the bit mask pair being passed, which is 32

The three subfields of the lock owner ID (VlokServerPID, VlokClientPID, VlokClientTID) are considered to be a single concatenated 16-byte field with respect to the owner mask. Since VlokServerPID is automatically set to the server's PID by the LFS, the first four bytes of the owner mask will be set to all ones so that the server may only purge locks that it has obtained.

Other fields in the Vlock area should be set to zeros.

- *Purging locks held on an object by a server:* The following Vlock fields are provided by the caller:

  **VlokObject**
  The 16-byte identifier of a specific object

  **VlokObjectMask**
  All X'FF', for matches on just the specific object

  **VlokLocker**
  All zeroes

  **VlokClientTID**
  All zeroes

  **VlokLockerMask**
  All zeroes, for matches on any owner with the same server PID

- *Purging locks held by a client user:* The following Vlock fields are provided by the caller:

  **VlokObject**
  Zero

  **VlokObjectMask**
  All zeroes, for matches on every object

**VlokClientPID** — The appropriate client PID

**VlokClientTID** — The appropriate client TID, TID subset (padded with zeroes), or all zeroes

**VlokLockerMask** — X'FF', left-justified for a length matching the appropriate subset of the 16-byte owner ID, and then padded with X'00'. For instance:
- 16 bytes of X'FF' for exactly one lock owner
- 12 bytes of X'FF' for, perhaps, all processes for a specific user at a specific client
- 8 bytes of X'FF' for all client TIDs for a given client PID

- *Effects of purge on asynchronous locks:* If a set of locks being purged includes pending asynchronous locks, those lock requests will be canceled.

  If a set of asynchronous lock requests are purged, the application will not be able to immediately tell which pending requests have been canceled and which had been granted and then were unlocked. When the call to purge returns to the caller, the lock completion signals will have all been sent but they may still be on the signal queue. The application can coordinate the purge operation with the signal handler after the purge completes by calling **sigqueue()** with a special signal number or value to flush the queue of these lock completion signals. If only a single thread handles the signal queue, then the appearance of this flush signal will indicate that all of the successful and ECANCELED signals have arrived and have been processed.

17. Each locker should be unregistered when it has finished issuing lock requests. On a Vlok#UnregLocker command, the following Vlock field is provided by the caller:

    **VlokID** — Vlok#ID (from the BPXYVLOK macro)

    **VlokLen** — The length of the Vlock structure

    **VlokLockerTok** — A token to identify the locker to unregister

    Other fields in the Vlock area should be set to zeros.

## Related services

- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333

## Characteristics and restrictions

A process must be registered as a lock server before the v_lockctl service is permitted; see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

## Examples

For an example using this callable service, see “BPX1VLO, BPX4VLO (v_lockctl) example” on page 484.

# v_lookup (BPX1VLK, BPX4VLK) — Look up a file or directory

## Function

The v_lookup service accepts a vnode token that represents a directory and a name that identifies a file. The directory is searched for this file, and if it is found, a vnode token for this file and its file attributes are returned. The file vnode token that is returned must be supplied by the server on all subsequent VFS callable services that are related to this file.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VLK): | 31-bit |
| AMODE (BPX4VLK): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VLK,(Directory_vnode_token,
              OSS,
              Name_length,
              Name,
              Attr_length,
              Attr,
              File_vnode_token,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VLK with the same parameters.

## Parameters

**Directory_vnode_token**

Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory in which the v_lookup service searches for the file that is supplied in the Name parameter.

**OSS**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Name_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the filename that is to be searched for. The name can be up to 255 bytes long.

**Name**

Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Name_length parameter |

The name of an area, of length Name_length, that contains the filename to be searched for. It must not contain null characters (X'00').

**Attr_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the area that is passed in the Attr parameter. To determine the value of Attr_length, use the ATTR structure (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**Attr**

Returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area, of length Attr_length, in which the v_lookup service returns the file attribute structure for the file that is supplied in the Name parameter. This area is mapped by the ATTR structure (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

The file attributes information is returned only if the file is found.

**File_vnode_token**

Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area in which the v_lookup service returns a vnode token of the file that is supplied in the Name parameter.

The token is returned only if the file is found.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_lookup service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_lookup service stores the return code. The v_lookup service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_lookup service can return one of the following values in the Return_code parameter:

| **Return_code** | **Explanation** |
|---|---|
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallAttr, JRNoName, JrNullInPath, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EMFILE | The maximum number of vnode tokens have been created. |
| ENAMETOOLONG | The name is longer than 255 characters. |
| ENFILE | An error occurred while storage was being obtained for a vnode token. |
| ENOENT | Name was not found. |
| ENOTDIR | The supplied token did not represent a directory. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_lookup service stores the reason code. The v_lookup service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. Vnode tokens that are returned by the v_lookup service are not inherited across a fork callable service.
2. The caller is responsible for freeing vnode tokens that are returned by the v_lookup service, by calling to the v_rel service when they are no longer needed.
3. Local mount points are not crossed unless the OssXmtpt bit is set in the input OSS structure. When that bit is on and the name looked up turns out to be a mount point directory, the root directory of the file system that is mounted there is returned instead of the named directory. This is called “crossing down the mount point tree”. When the specified name is “..” and the specified directory is a local root, the parent directory of the underlying mount point is returned instead of the parent of the specified directory. This is called “crossing up the mount point tree”.

   In these situations, the OssXmtpt bit is left on and the VFS_Token of the crossed into file system is returned in the AttrCharSetID field of the returned ATTR structure. If a mount point is not encountered, the OssXmtpt bit is turned off.

## Related services

- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333
- “v_rel (BPX1VRL, BPX4VRL) — Release a vnode token” on page 337

## Characteristics and restrictions

A process must be registered as a server before the v_lookup service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VLK, BPX4VLK (v_lookup) example" on page 485.

# v_mkdir (BPX1VMK, BPX4VMK) — Create a directory

## Function

The v_mkdir service creates a new empty directory in the directory that is represented by Directory_vnode_token. The input Attr is used to define the attributes of the new directory. A token that represents the new directory is returned in the New_directory_vnode_token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VMK): | 31-bit |
| AMODE (BPX4VMK): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VMK,(Directory_vnode_token,
              OSS,
              Name_length,
              Name,
              Attr_length,
              Attr,
              New_directory_vnode_token,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VMK with the same parameters.

## Parameters

**Directory_vnode_token**

Supplied parameter

**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area that contains a vnode token that represents the directory in which the v_mkdir service creates the new directory that is named in the Name parameter.

**OSS**

Supplied and returned parameter

**Type:** Structure
**Length:** OSS#LENGTH (from the BPXYOSS macro)

The name of an area that contains operating system specific parameters. This area is mapped by the BPXYOSS macro (see "BPXYOSS — Map operating system specific information" on page 469).

**Name_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the directory name that is to be created. The name can be up to 255 bytes long. It must not contain null characters (X'00').

**Name**

Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Name_length parameter |

The name of an area, of length Name_length, that contains the directory name that is to be created. It must not contain null characters (X'00').

**Attr_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the area that is passed in the Attr parameter. To determine the value of Attr_length, use the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**Attr**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area, of length Attr_length, that is to be used by the v_mkdir service to set the attributes of the directory that is to be created. The attributes of the directory that is created are also returned in this area. This area is mapped by the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**New_directory_vnode_token**

Returned parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area in which the v_mkdir service returns a vnode token of the directory that is created.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_mkdir service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_mkdir service stores the return code. The v_mkdir service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_mkdir service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | The calling process does not have permission to update the directory that was specified. |
| EEXIST | The directory named already exists. |
| EFBIG | The file size limit for the process is set to zero, which means directories cannot be created. |
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallAttr, JRInvalidAttr, JrNoName, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EMFILE | The maximum number of vnode tokens have been created. |
| ENAMETOOLONG | The name is longer than 255 characters. |
| ENFILE | An error occurred while storage was being obtained for a vnode token. |
| ENOTDIR | The supplied token did not represent a directory. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |
| EROFS | Directory_vnode_token specifies a directory on a read-only file system. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_mkdir service stores the reason code. The v_mkdir service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The following Attr fields are provided by the caller:

   **AttrID** Contains Attr#ID (from the ATTR structure).

   **AttrLen** Specifies the length of the Attr structure.

   **AttrMode** Specifies directory mode permission bits. See "BPXYMODE — Map the mode constants of the file services" on page 466 for the mapping of this field.

   Other fields should be initialized to zero.
2. If the directory that is named in the Name parameter already exists, the v_mkdir service returns a failing return code, and no New_directory_vnode_token is returned.
3. Vnode tokens that are returned by the v_mkdir service are not inherited across a fork callable service.

4. The caller is responsible for freeing vnode tokens that are returned by the v_mkdir service, by calling to the v_rel service when they are no longer needed.
5. If the file size limit for the process is set to zero, directories cannot be created and directory creation fails with EFBIG.
6. The value set by **umask()** for the process does not affect the setting of the mode permission bits.
7. The setting of the AttrLP64times bit in the BPXYATT structure, and not the AMODE of the caller, determines whether 4-byte or 8-byte time fields are used.

## Related services

- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333
- “v_rel (BPX1VRL, BPX4VRL) — Release a vnode token” on page 337

## Characteristics and restrictions

A process must be registered as a server before the v_mkdir service is permitted; see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

## Examples

For an example using this callable service, see “BPX1VMK, BPX4VMK (v_mkdir) example” on page 486.

# v_open (BPX1VOP, BPX4VOP) — Open or create a file

## Function

The v_open service opens an existing file or creates and opens a new file. To open an existing file, the file's vnode token is passed. To create a new file, a directory vnode token is passed along with the name of the file to be created in that directory.

The v_open service can also be used to establish share reservations on the file. A file is opened for a particular type of access (reading, writing, or both) and a share reservation can be specified to prohibit any other conflicting access while the file is open. A v_open will fail if an existing share reservation prohibits the desired access or if the file is already open in an access mode that this v_open is trying to prohibit.

An open token is returned which represents the share reservations established by the v_open call. The open token is used on subsequent v_rdwr and v_setattr calls to show that they are being done within a share reservation owned by the caller and with v_lockctl to associate byte range locks with a particular open.

The share reservations made here can be upgraded or downgraded with another call to v_open. They are relinquished with v_close, which removes all state information associated with the v_open.

A file vnode token is returned when a file is opened by name or a new file is created. This token is used on subsequent VFS callable services that are related to this file and the token is eventually released with the v_rel service.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VOP): | 31-bit |
| AMODE (BPX4VOP): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VOP,(Vnode_token,
              OSS,
              Open_Parms_length,
              Open_Parms,
              FileName_length,
              FileName,
              CreateParm_length,
              CreateParm,
              OutputAttr_length,
              OutputAttr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VOP with the same parameters.

## Parameters

**Vnode_token**
Supplied parameter
**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area that contains a vnode token that represents the file being opened or the directory in which a new file is to be created.

**OSS**
Supplied and returned parameter
**Type:** Structure
**Length:** OSS#LENGTH (from the BPXYOSS macro)

The name of an area that contains operating system specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Open_Parms_length**
Supplied parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword that contains the length of the Open_Parms parameter.

**Open_Parms**
Supplied parameter
**Type:** Structure
**Length:** Specified by Open_Parms_length parameter

The name of an area that contains additional parameters for this open request. Refer to the usage notes for a description of these parameters. This area is mapped by the BPXYVOPN macro (see “BPXYVOPN — Map the open parameters structure for v_open” on page 476).

**FileName_length**
Supplied parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword that contains the length of the FileName parameter. The name can be up to 255 bytes long.

**FileName**
Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by FileName_length parameter |

The name of an area (of length FileName_length) that contains the name of the file to be created. The file name must not contain null characters (X'00').

**CreateParm_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the area that is passed in the CreateParm parameter.

**CreateParm**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the CreateParm_length parameter |

The name of an area whose content depends on the type of create request, as follows:

- For OPEN_CREATE_EXCLUSIVE, an 8-byte creation verifier is passed.
- For OPEN_CREATE_GUARDED and OPEN_CREATE_UNCHECKED, an attr structure is passed which contains the attributes to be assigned to the new file. The set of attributes can include any valid, writable attribute for regular files. Refer to "v_setattr (BPX1VSA, BPX4VSA) — Set the attributes of a file" on page 354 for the format of this attr structure and for setting file attributes.

Refer to the usage notes for more information on the three types of file creation.

**OutputAttr_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the area that is passed in the OutputAttr parameter, or `0` if no output attributes are desired.

**OutputAttr**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the OutputAttr_length parameter |

The name of an optional area where the system will return the attributes of the file to be opened. If no output attributes are desired, specify `0` for the preceding OutputAttr_length parameter. See "BPXYATTR — Map file attributes for v_ system calls" on page 445 for a mapping of the file attributes structure.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_open service stores the return code. The v_open service returns Return_code only if the Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_open service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EBUSY | The file is currently open in a way that conflicts with the share reservation that is being requested. The following reason codes can accompany the return code:<br>JrAccessConflict — The file is already open with access that this open is trying to deny.<br>JrShrConflict — This open conflicts with a share reservation that has denied the intended access. |
| EEXIST | The file to be created with the GUARDED or EXCLUSIVE creation protocols already exists. |
| EINVAL | Parameter error; for example, a supplied area was too small or the Vnode_token is stale. The following reason codes can accompany the return code:<br>JrUpgradeSet — The access or share mode of an OPEN_UPGRADE is not a superset of the current value.<br>JrDowngradeSet — The access or share mode of an OPEN_DOWNGRADE is not a subset of the current value.<br>JrInvAccess — The access mode is 0 or greater than 3. |
| EOPNOTSUPP | The socket or file is not a type that supports the requested function. The following reason code can accompany the return code:<br>JrNoShrsAtOwner — Share reservations are requested but the file is owned by a system that does not support shares. |
| ESTALE | The open token is not (or is no longer) valid. |
| EACCES | The user is not authorized either to create a file in this directory or to open the specified existing file. |
| EROFS | The define or open cannot be done on a read-only file system. |
| EISDIR | An open request is being attempted on a directory. |
| EFAULT | A bad parameter address was specified. |
| EMFILE | The maximum number of vnode tokens or open tokens has been created. The following reason codes can accompany the return code:<br>JRTokenMax — The maximum number of vnode tokens has been allocated for this process.<br>JROpenTokMax — The maximum number of open tokens has been allocated for this process. |

| Return_code | Explanation |
|---|---|
| ENAMETOOLONG | The name is longer than 255 characters. |
| ENFILE | An error occurred in obtaining storage for a vnode token. |
| ENOTDIR | The supplied directory token does not represent a directory. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_open service stores the reason code. The v_open service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. An output open token returned by v_open is generally freed by calling v_close. It is also freed if the vnode token with which it is associated is freed by a call to v_rel or if the process terminates.
2. The v_close service releases the share reservations made by this and subsequent calls to v_open with this open token. It also releases any byte range locks associated with this open token by v_lockctl.
3. An output vnode token returned by v_open is generally freed by calling v_rel. It is also freed if the process terminates.
4. Vnode tokens and open tokens returned by the v_open service are not inherited across a call to the fork service.
5. All calls to v_open that refer to an existing file may be rejected if the specified access intent or share reservations conflict with the current state of existing opens on that file. See the descriptions of the Shr_Access and Shr_Deny parameters in note 7 for more information.
6. The total number of open tokens that a process can acquire is limited by the MaxVnTok value that is established when the server registers with v_reg. The limit applies separately to the number of vnode tokens and the number of open tokens, not to the sum of the two.
7. The Open_Parms structure contains the following additional parameters:
   - *Open_type* — specifies the type of open being requested. All of the following open types may establish share reservations on the file.

     **OPEN_FILE** — Open an existing file. The Vnode_token parameter specifies the file to open.

     **OPEN_CREATE_UNCHECKED** — Create a new file with the unchecked create protocol. The Vnode_token parameter specifies a directory and the FileName parameter specifies the name of the file to create in that directory.

     **OPEN_CREATE_GUARDED** — Create a new file with the guarded create protocol. The Vnode_token parameter specifies a directory and the FileName parameter specifies the name of the file to create in that directory.

**OPEN_CREATE_EXCLUSIVE** — Create a new file with the exclusive create protocol. The Vnode_token parameter specifies a directory and the FileName parameter specifies the name of the file to create in that directory.

**OPEN_NLM_SHR** — Only establish share reservations on a file. The Vnode_token parameter specifies the file. This open type differs from the preceding ones in the following ways:

- The file is not actually opened to the PFS that manages the file. Normal access checking is still performed for the specified *Shr_Access* mode. However, because the file is not open to the PFS, file data is not protected from deletion if the file is removed.
- Byte range locks are not associated with NLM_SHR open tokens and, thus, are not released by a v_close call for this open token. To implement an NLM unshare, call v_close with the open token that was returned by this call to v_open.
- The share reservations that are established here are only advisory with regard to any read and write operations that are performed without an open token. See "v_rdwr (BPX1VRW, BPX4VRW) — Read from and write to a file" on page 322 for details.

**OPEN_UPGRADE** — Upgrade the access intent and share reservations that are associated with a prior open operation. The Vnode_token parameter specifies the file that was opened and the Open_token parameter contains the token that was returned by that open. The Shr_Access and Shr_Deny parameters contain the new settings to be associated with this open token. The new settings consist of the results of applying the upgrade settings to the current settings and, thus, must form a superset of the settings currently in effect for this open token.

**OPEN_DOWNGRADE** — Downgrade the access intent and share reservations that are associated with a prior open operation. The Vnode_token parameter specifies the file that was opened and the Open_token parameter contains the open token that was returned by that open. The Shr_Access and Shr_Deny parameters contain the new settings to be associated with this open token. The new settings consist of the results of applying the downgrade settings to the current settings and, thus, must form a subset of the settings currently in effect for this open token.

- *Open_Owner* — specifies a structure that contains the (server PID, client PID, thread ID) triplet that identifies the individual owner of the share reservations established here. This structure is mapped by VlokOwner in the BPXYVLOK macro and by the LOCKOWNER structure in the BPXYVFSI C header.

  **Note:** The first word is reserved and is set by the system to the server's PID.
- *Shr_Access* — specifies the access intent for this open request, as follows:

  **ACC_READ** — Access intent is read

  **ACC_WRITE** — Access intent is write

  **ACC_BOTH** — Access intent is read and write

  This v_open will be rejected with return code EBUSY, reason code JrShrConflict, if the access intent conflicts with an existing share reservation. A value is required for this parameter (must not be zero).
- *Shr_Deny* — specifies the share reservations for this open request. Share reservations specify the type of access intent that will be prohibited on subsequent open or v_open attempts for this file while this open is in effect.

This will also inhibit conflicting read and write operations that are performed without an open token. The following share reservations are valid:

**DENY_NONE** — No access is denied.

**DENY_READ** — Read access is denied. Attempts to open this file for read will be rejected.

**DENY_WRITE** — Write access is denied. Attempts to open this file for write will be rejected.

**DENY_BOTH** — Read and write access is denied. Any attempts to open this file will be rejected.

This v_open will be rejected with return code EBUSY, reason code JrShrConflict, if the file is already open for an access intent that this v_open is trying to deny. Share reservations that attempt to deny reading or writing for files in a read-only file system will be accepted but will not be enforced.

**Note:** A file system may not be remounted from read-write mode to read-only mode or vice-versa while there are active share reservations on any file in that file system.

- *Open_token* — specifies an 8-byte token that identifies a particular open instance.
  - For OPEN_UPGRADE and OPEN_DOWNGRADE open types, the open token of a prior v_open call is passed by the caller.
  - For all other open types, if the call is successful, the v_open service returns an open token that represents this open on subsequent calls to VFS callable services, in particular v_rdwr and v_lockctl.

  The open token is put into the OSS of v_rdwr and v_setattr (size change) when those operations are performed within an open context. Read and write operations that are performed within an open context do not need to be validated against the share reservations of other opens. See “v_rdwr (BPX1VRW, BPX4VRW) — Read from and write to a file” on page 322 for details.
- *Output_File_vnode_token* — specifies an 8-byte token that identifies the particular file that was just opened by name. The v_open service returns an output vnode token for successful calls that specify one of the OPEN_CREATE_*xxxxx* open types. This is the same token as that which would be returned by the v_lookup and v_create services.

8. Several v_open parameters are optional or differ in value depending on the setting of the Open_type parameter. 6 summarizes the parameters that vary by open type.

*Table 6. Summary of v_open parameters that vary by open type*

| If Open_type is... | Vnode_token specifies a... | FileName required? | CreateParm specifies a... | An Open_token is... | Output_File_vnode_token returned? |
|---|---|---|---|---|---|
| OPEN_FILE | file | | | returned | |
| OPEN_CREATE_UNCHECKED | directory | yes | attr structure | returned | yes |
| OPEN_CREATE_GUARDED | directory | yes | attr structure | returned | yes |
| OPEN_CREATE_EXCLUSIVE | directory | yes | creation verifier | returned | yes |
| OPEN_NLM_SHR | file | | | returned | |
| OPEN_UPGRADE | file | | | supplied | |
| OPEN_DOWNGRADE | file | | | supplied | |

9. There are three creation protocols available, as follows:
   a. **OPEN_CREATE_UNCHECKED** — indicates that the file should be created if a file by that name does not already exist or if encountering an existing regular file by that name is not to be considered an error. The v_open service indicates a successful return value in either case. If the name is in use by something other than a regular file, the v_open call fails with an EEXIST return code.

      For this type of create, the CreateParm parameter specifies the initial set of attributes for the file. The set of attributes can include any valid, writable attribute for regular files. Refer to "v_setattr (BPX1VSA, BPX4VSA) — Set the attributes of a file" on page 354 for the format and protocols for setting file attributes. When an unchecked create encounters an existing file, the attributes specified by CreateParm are ignored, *except* that if a file size of zero is specified, the existing file will be truncated.
   b. **OPEN_CREATE_GUARDED** — indicates that v_open should fail with an EEXIST return code if it encounters any existing file by the same name. If no object with the same name exists, the request proceeds as described for OPEN_CREATE_UNCHECKED.
   c. **OPEN_CREATE_EXCLUSIVE** — indicates that the CreateParm parameter contains an 8-byte creation verifier that will be used to ensure the exclusive creation of the file. If the file does not exist, it will be created and the verifier will be stored with the file. No attributes are provided on this call since the PFS may use an attribute of the target object to temporarily store the verifier. The verifier is reliable until the first time v_setattr is called or the file is used in any other way. There is no way to tell if an existing attribute is used (or which one is used) to temporarily store the verifier.

      If the file exists, the v_open call fails with an EEXIST return code. The server reacts to an EEXIST failure by calling v_lookup to fetch the attributes of the existing file. If those attributes contain a creation verifier that matches the creation verifier that was passed by the client, then the existing file must have been created by a prior transmission of this create request, so this request is deemed successful. Otherwise, the existing object is something different and the client's request fails.

## Related services


- "v_close (BPX1VCL, BPX4VCL) — Close a file" on page 267
- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_rel (BPX1VRL, BPX4VRL) — Release a vnode token" on page 337


## Characteristics and restrictions

A process must be registered as a server before the v_open service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

# v_pathconf (BPX1VPC, BPX4VPC) — Get pathconf information for a directory or file

## Function

The v_pathconf service accepts a vnode token that represents a file or a directory and returns the current values of options that are associated with that file or directory in the output PCFG.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VPC): | 31-bit |
| AMODE (BPX4VPC): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VPC,(Vnode_token,
              OSS,
              PCFG_length,
              PCFG,
              Attr_length,
              Attr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VPC with the same parameters.

## Parameters

**Vnode_token**

Supplied parameter

**Type:** Token

**Length:** 8 bytes

The name of an 8-byte area that contains a vnode token that represents the directory or file for which to obtain pathconf information.

**OSS**

Supplied and returned parameter

**Type:** Structure

**Length:** OSS#LENGTH (from the BPXYOSS macro)

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro; see “BPXYOSS — Map operating system specific information” on page 469.

**PCFG_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the PCFG parameter; see “BPXYPCF — Map pathconf values” on page 470.

**PCFG**
Returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the PCFG_length parameter. |

The name of an area in which the pathconf information is to be returned. This area is mapped by the BPXYPCF macro.

**Attr_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the Attr parameter.

**Attr**
Returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area in which the attributes of the file or directory are to be returned. This area is mapped by the BPXYATTR macro.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_pathconf service returns the length of the output PCFG if the request is successful, or −1 if it is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_pathconf service stores the return code. The v_pathconf service returns Return_code only if Return_value is −1. The v_pathconf service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallAttr, JrInvalidAttr, JRBuffLenInvalid, JrVTokenFreed, JrWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |

**Length:** Fullword

The name of a fullword in which the v_pathconf service stores the reason code. The v_pathconf service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value.

## Usage notes

The buffer contents that are returned by the v_pathconf service are mapped by the BPXYPCF macro.

## Related services

## Characteristics and restrictions

A process must be registered as a server before the v_pathconf service is permitted.

## Examples

For an example using this callable services, see "BPX1VPC, BPX4VPC (v_pathconf) example" on page 487.

# v_rdwr (BPX1VRW, BPX4VRW) — Read from and write to a file

## Function

The v_rdwr service accepts a vnode token that represents a file and reads data from or writes data to the file. The number of bytes that are read or written and the file attributes are returned upon completion of the operation.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRW): | 31-bit |
| AMODE (BPX4VRW): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRW,(Vnode_token,
              OSS,
              UIO,
              Attr_length,
              Attr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRW with the same parameters. The FUIO may contain a 64-bit address.

## Parameters

**Vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the file that is to be read from or written into.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contain operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see "BPXYOSS — Map operating system specific information" on page 469).

**UIO**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fuio#Len (from the BPXYFUIO macro) |

The name of an area that contains the user input and output block. This area is mapped by the BPXYFUIO macro (see "BPXYFUIO — Map file system user I/O block" on page 452).

**Attr_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the area that is passed in the Attr parameter. To determine the value of Attr_length, use the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**Attr**

Returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area, of length Attr_length, in which the v_rdwr service returns the file attribute structure for the file that is specified by the vnode token. This area is mapped by the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

The file attributes information is returned only if the read or write operation is successful.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rdwr service returns the number of bytes read or written if the request is successful, or −1 if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rdwr service stores the return code. The v_rdwr service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_rdwr service can return one of the following values in the Return_code parameter:

| **Return_code** | **Explanation** |
|---|---|
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallAttr, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS, JRRwNotRegFile, JRInvalidFuio, JRBytes2RWZero. |
| EFBIG | Writing to the specified file would exceed either the file size limit for the process or the maximum file size that is supported by the physical file system. |
| EACCES | The caller does not have the requested (read or write) access to the file. |

| Return_code | Explanation |
|---|---|
| EIO | An I/O error occurred while reading or writing the file. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |
| EMVSPFSPERM | An internal error occurred in the PFS. Consult Reason_code to determine the exact reason the error occurred. |

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the v_rdwr service stores the reason code. The v_rdwr service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The following UIO fields are provided to specify the details of the read or write request:

   **FuioSync** Requests that all data that is associated with the file is to be transferred to the storage device before completion of this write request.

   **FuioChkAcc** Requests the PFS to perform required access checking before performing the requested read or write operation.

   **FuioBufferAddr** Contains the address of a buffer that contains the data that is to be read or written.

   **FuioBuff64Vaddr** Contains the 64-bit virtual address of a buffer that contains the data that is to be read or written.

   **FuioIBytesRW** Specifies the number of bytes to be read or written.

   **FuioRWInd** Specifies the operation requested; read or write.

   **FuioCursor** Specifies the byte offset in the file where the read or write operation is to begin.

   **FuioRealPage** Specifies that the buffer is a real-storage page and the DATOFF services of MVS must be used to move the data.

   **FuioInternal** Used internally by the LFS during a call; this field must be zeroed out before each call.

2. The FuioAddr64 setting determines whether the pointer to the user buffer is a 64-bit pointer in FuioBuff64Vaddr or a 31-bit pointer in FuioBufferAddr.
3. An open token from a prior v_open may be passed in the OSS to indicate that this read or write operation is being done within the open context of that token. Consequently, the operation does not have to be verified against the share reservations that may currently be in effect for this file. If an open token is unavailable to pass on a call, there are three levels of share reservation checking that can be requested:

| | |
|---|---|
| **Oss#NoTokAdvChk** | Advisory checking. The operation will only be validated against non-NLM share reservations. This corresponds to a read or write from a version 2 or 3 NFS client. These clients do not issue an open request and the NLM share reservations that they make are only advisory with respect to the reads and writes of other version 2 or 3 clients. |
| **Oss#NoTokMandChk** | Mandatory checking. The operation will be validated against all share reservations. This corresponds to a version 4 NFS client read or write with a stateid of 0 or a write with a stateid of –1. |
| **Oss#NoTokOverride** | No checking. The operation will be permitted without any share reservation checking. This is only allowed for read operations and corresponds to a version 4 NFS client read with a stateid of –1. |

In general, version 4 share reservations are enforced against all clients; read and write operations from version 4 clients cannot violate any share reservations. Read and write operations from version 2 and 3 clients are allowed to violate version 2 and 3 share reservations.

## Related services

- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333

## Characteristics and restrictions

A process must be registered as a server before the v_rdwr service is permitted; see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

## Examples

For an example using this callable service, see “BPX1VRW, BPX4VRW (v_rdwr) example” on page 488.

# v_readdir (BPX1VRD, BPX4VRD) — Read entries from a directory

## Function

The v_readdir service accepts a vnode token that represents a directory and returns as many directory entries from this directory as will fit in the caller's buffer.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRD): | 31-bit |
| AMODE (BPX4VRD): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRD,(Vnode_token,
              OSS,
              UIO,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRD with the same parameters. The FUIO may contain a 64-bit address.

## Parameters

**Vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory to read directory entries from.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see "BPXYOSS — Map operating system specific information" on page 469).

**UIO**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fuio#Len (from the BPXYFUIO macro) |

The name of an area that contains the user input and output block. This area is mapped by the BPXYFUIO macro (see “BPXYFUIO — Map file system user I/O block” on page 452).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_readdir service returns the number of directory entries that were returned if the request is successful, or −1 if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_readdir service stores the return code. The v_readdir service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_readdir service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | The calling process does not have permission to read a specified directory. |
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRInvalidFuio, JrBytes2RWZero, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS |
| ENOTDIR | The supplied token did not represent a directory. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_readdir service stores the reason code. The v_readdir service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. For an overview of the process of reading from directories, see “Reading directories” on page 257.
2. Two protocols are supported for reading through large directories with successive calls:
   - **Cursor protocol.** The cursor, or offset, that is returned in the UIO by the v_readdir service contains file-system-specific information that locates the next directory entry. The cursor and buffer must be preserved by the caller from one v_readdir call to the next, and reading proceeds based on the cursor.

- **Index protocol.** The index that is set in the UIO by the caller determines which entry to start reading from. To read through the directory, the caller starts at one and increments the index by the number of entries that were returned on the previous call.

3. The following UIO fields are provided to specify the details of the read directory request:

   **FuioID** Contains Fuio#ID (from the BPXYFUIO macro).

   **FuioLen** Contains the length of the UIO structure.

   **FuioChkAcc** Requests the PFS to perform required access checking before performing the requested readdir operation.

   **FuioBufferAddr** Contains the address of a buffer where the directory entries are to be returned.

   **FuioBuff64Vaddr** Contains the 64-bit virtual address of a buffer where the directory entries are to be returned.

   **FuioIBytesRW** Specifies the maximum number of bytes that can be written to the output buffer.

   **FuioRDIndex** Specifies the first directory entry that is to be returned when the index protocol is used.

   **FuioCursor** When the cursor protocol is used, this specifies a value that was returned on the previous v_readdir call and that indicates the next entry to be read, or 0 on the first call.

   **FuioRddPlus** Indicates that the request is for the ReaddirPlus function. The attributes for each entry should be included in the output.

4. The following UIO fields are returned by the v_readdir service:

   **FuioPSWKey** Is set to the caller's key.

   **FuioCursor** Is set to the cursor value representing the directory position. This value is used if the next call uses the cursor protocol.

   **FuioCVerRet** Indicates that the Cookie Verifier (FuioCVer) is being returned.

   **FuioCVer** When FuioCVerRet is on, this field is set to the Cookie Verifier for the directory that is being read. When a directory is being read with multiple reads, you can use the FuioCVer that is returned to compare each Cookie Verifier with the last one. If the directory has been modified between reads, you can reject the request because the results will not be valid.

5. The buffer contents that are returned by the v_readdir service are mapped by BPXYDIRE macro (see "BPXYDIRE — Map directory entries for readdir" on page 449).

6. The FuioAddr64 setting determines whether the pointer to the user buffer is a 64-bit pointer in FuioBuff64Vaddr or a 31-bit pointer in FuioBufferAddr.

7. The OssXmtpt flag allows a v_readdir operation to cross mount points when the FuioRddPlus flag is set. Normally, the attributes that are returned with each

| name are for objects in the same file system as the directory being read. However, some of the objects may be mount point directories. To have the attributes of the mounted root directory returned (instead of the attributes of the mount point), set the OssXmtpt flag in the input OSS structure. When the directory being read is the root of a mounted file system (but not the system root), the attributes for the “..” entry will be replaced with the attributes of the parent of the underlying mount point. In such cases, the device number in the Attrdev field in that entry’s attributes will differ from the device number of the directory being read and the VFS_Token of the other file system will be returned in the AttrCharSetID field.

## Related services

- “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333

## Characteristics and restrictions

A process must be registered as a server before the v_readdir service is permitted; see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

## Examples

For an example using this callable service, see “BPX1VRD, BPX4VRD (v_readdir) example” on page 489.

# v_readlink (BPX1VRA, BPX4VRA) — Read a symbolic link

## Function

The v_readlink service reads the symbolic link file that is represented by Vnode_token, and returns the contents in the buffer that is described by UIO. The symbolic link file contains the pathname or external name that was specified when the symbolic link was defined (see “v_symlink (BPX1VSY, BPX4VSY) — Create a symbolic link” on page 361).

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRA): | 31-bit |
| AMODE (BPX4VRA): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRA,(Vnode_token,
              OSS,
              UIO,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRA with the same parameters. The FUIO may contain a 64-bit address.

## Parameters

**Vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the symbolic link file to read.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**UIO**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Fuio#Len (from the BPXYFUIO macro) |

The name of an area that contains the user input and output block. This area is mapped by the BPXYFUIO macro (see "BPXYFUIO — Map file system user I/O block" on page 452).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_readlink service returns the number of bytes read into the buffer if the request is successful, or −1 if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_readlink service stores the return code. The v_readlink service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_readlink service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRInvalidFuio, JrFileNotSymLink, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_readlink service stores the reason code. The v_readlink service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The following UIO fields are provided by the caller:

| | |
|---|---|
| **FuioID** | Contains Fuio#ID (from the BPXYFUIO macro). |
| **FuioLen** | Contains the length of the UIO structure. |
| **FuioBufferAddr** | Contains the address of a buffer where the link contents are to be returned. |
| **FuioBuff64Vaddr** | Contains the 64-bit virtual address of a buffer where the link contents are to be returned. |

| | |
|---|---|
| **FuioIBytesRW** | Specifies the maximum number of bytes that can be written to the output buffer. |

2. The following UIO field is returned by the v_readlink service.:

| | |
|---|---|
| **FuioPSWKey** | Is set to the caller's key. |

3. If the buffer that is supplied to v_readlink is too small to contain the contents of the symbolic link, the value is truncated to the length of the buffer (FuioBytesRW). The length of the symbolic link can be determined from an ATTR structure that is returned on a call to the VFS callable services API (that is, to "v_getattr (BPX1VGA, BPX4VGA) — Get the attributes of a file" on page 285). The maximum length is 1023 bytes.
4. The FuioAddr64 setting determines whether the pointer to the user buffer is a 64-bit pointer in FuioBuff64Vaddr or a 31-bit pointer in FuioBufferAddr.

## Related services

- "v_getattr (BPX1VGA, BPX4VGA) — Get the attributes of a file" on page 285
- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_symlink (BPX1VSY, BPX4VSY) — Create a symbolic link" on page 361

## Characteristics and restrictions

A process must be registered as a server before the v_readlink service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VRA, BPX4VRA (v_readlink) example" on page 490.

# v_reg (BPX1VRG, BPX4VRG) — Register a process as a server

## Function

The v_reg service registers a process as a server. A process must be registered using this service before it may use any other VFS callable services API.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRG): | 31-bit |
| AMODE (BPX4VRG): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRG,(Nreg_length,
              Nreg,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRG with the same parameters.

## Parameters

**Nreg_length**

Supplied parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword that contains the length of the Nreg parameter list area.

**Nreg**

Supplied and returned parameter

**Type:** Structure

**Length:** Specified by the Nreg_length parameter

The name of an area that contains the registration parameters. The entries in this area are mapped by BPXYNREG (see "BPXYNREG — Map interface block to vnode registration" on page 467). The following registration parameters must be supplied:

| Parameter | Description |
|---|---|
| ID | Set to Nreg#ID. |
| Len | Set to Nreg#Len. |
| Ver | Set to Nreg#Version. |

**Type**
: Set to server type:
  *NRegSType#FILE* — for a file server
  *NRegSType#LOCK* — for a lock server
  *NRegSType#FEXP* — for a file exporter

**NameLen**
: Set to the length of the supplied server name.

**Name**
: Up to 32 bytes of character string that is used as the name of this server. This name appears in DISPLAY OMVS output.

If the process is to be registered as a server-type file exporter, the following parameters must also be supplied:

**ExitName**
The name of the program that is to control local access to exported file systems.

**InitParm**
A parameter that is to be passed to the ExitName program when it is initialized.

**Hotc Flag**
An indication that the ExitName program should be invoked with a pre-initialized C environment (HOTC).

The following registration parameters may be supplied:

**No Wait Flag**
An indication that server threads should not be suspended during a request that is made to a file system that is quiesced, such as for an HSM backup. The request will fail instead of waiting.

**MaxVnTok**
| An upper bound on the number of vnode tokens and, separately, the number of open tokens that the server is to be allowed to have active at one time.

| **AllocDevno Flag**
| Requests that a file system device number, as in AttrDev, be allocated for exclusive use by the server. This number will not be used by the LFS for any mounted file system so the server can use this number as the device number for a non-UNIX file system that it is exporting. On a successful v_reg call, the device number is returned in the Devno field of the Nreg structure.

If the process is responsible for posting threads that are waiting within a specific PFS, the process can establish special recovery by specifying the PFS with:

**PfsType**
The name of the Physical File System that is dependent on this process for osi_post. This is the name that was specified when the PFS was defined in the BPXPRMxx parmlib member with either FILESYSTYPE TYPE() or SUBFILESYSTYPE NAME().

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_reg service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_reg service stores the return code. The v_reg service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_reg service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, the server name length that was supplied in the registration parameter list was too long; or the server type that was supplied is not a recognized value. The following reason codes can accompany this return code: JRNameTooLong, JRInvalidNReg, and JRInvalidRegType. |
| EPERM | The operation is not permitted. The caller of the service is not privileged; or the caller is already registered. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_reg service stores the reason code. The v_reg service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. Registration as a server is not inherited across a fork.
2. The MaxVNTokens field in the registration parameter list is an input and output parameter. If supplied by the caller, it indicates the value that should be used for this server. If a value of 0 is supplied, or if the value that is supplied exceeds the maximum allowed value, the maximum allowed value is used and returned.
3. The main difference between the file server and file exporter types is that file exporters control all access, both local and remote, to the file systems that they export.

   Refer to “DFS-style file exporters” on page 255 for more information on file exporters and the exit program.
4. If the exit program cannot be loaded, the Nreg abend code and abend reason code fields are filled in with the corresponding values returned by the system load service.

   If the exit program fails, v_reg also fails, and the exit’s return and reason codes are returned as the corresponding values from v_reg.
5. If the server’s address space is started before the z/OS UNIX address space, a v_reg that is issued during initialization fails. To account for this, an Event Notification Facility (ENF) signal is issued whenever z/OS UNIX is started. During initialization, a server can set up an ENF Listen for this event and call v_reg. If the v_reg call fails with EMVSNOTUP, the ENF signal is eventually issued, and v_reg can be called again after the server’s ENF Listen exit is

invoked. The ENF Qualifier Constant is defined in macro BPXYENFO. The MVS ENF service is documented in *z/OS MVS Programming: Assembler Services Guide*.

6. When a PFS is dependent on a separate address space calling osi_post to wake up threads that are in osi_wait within that PFS, recovery can be established to protect these threads from waiting forever if the separate address space terminates abnormally.

   To do this, the separate address space registers and specifies a PfsType name. This creates a process, if one did not already exist. When the registered process terminates, the system scans for and wakes up any users that are in osi_wait from within the specified PFS. The PFS's osi_wait call returns with a return code of OSI_POSTERTRM if it is posted for this reason.

   This recovery support is process-related. A process is usually the same as the address space, but if the registering task is the only task to use z/OS UNIX services, or if set_dub_default (BPX1SDD/BPX4SDD) has been called to make each task a separate process, this recovery is invoked when the registering task terminates.

   If this recovery support is the only reason the server is registering, use the server type for a file server.
7. There is no specific way to unregister. If necessary, the task can call mvsprocclp (BPX1MPC/BPX4MPC) to terminate the process, which also unregisters the server.
8. If z/OS UNIX terminates and restarts while the server address space is active, mvsprocclp (BPX1MPC) must be called on each task that has used z/OS UNIX services to remove its binding to the old instance of z/OS UNIX before V_reg can be recalled to reregister as a server.

## Characteristics and restrictions

In order to register, the caller must have appropriate privileges.

## Examples

For an example using this callable service, see "BPX1VRG, BPX4VRG (v_reg) example" on page 491.

# v_rel (BPX1VRL, BPX4VRL) — Release a vnode token

## Function

The v_rel service accepts a Vnode_token value that represents a file or a directory and releases that token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRL): | 31-bit |
| AMODE (BPX4VRL): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRL,(Vnode_token,
              OSS,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRL with the same parameters.

## Parameters

**Vnode_token**

Supplied parameter

**Type:** Token

**Length:** 8 bytes

The name of an 8-byte area that contains a vnode token that is to be released.

**OSS**

Supplied and returned parameter

**Type:** Structure

**Length:** OSS#LENGTH (from the BPXYOSS macro)

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Return_value**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_rel service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rel service stores the return code. The v_rel service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_rel service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, Vnode_token has already been released. The following reason codes can accompany the return code: JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rel service stores the reason code. The v_rel service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The vnode token is no longer valid and cannot be used for subsequent requests after the v_rel service has successfully processed it.
2. All vnode tokens that are obtained from other operations must be released by calling this service.

## Related services

- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333

## Characteristics and restrictions

A process must be registered as a server before the v_rel service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VRL, BPX4VRL (v_rel) example" on page 492.

# v_remove (BPX1VRM, BPX4VRM) — Remove a link to a file

## Function

The v_remove service removes a link to a file.

The name of the link is specified as input, along with a Directory_vnode_token value that identifies the directory that contains the name that is to be removed. The name can identify a file, a link name to a file, or a symbolic link.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRM): | 31-bit |
| AMODE (BPX4VRM): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRM,(Directory_vnode_token,
              OSS,
              Name_length,
              Name,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRM with the same parameters.

## Parameters

**Directory_vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory from which the v_remove service is to remove the entry that is supplied in the Name parameter.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro, see “BPXYOSS — Map operating system specific information” on page 469.

**Name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of Name. The name can be up to 255 bytes long.

**Name**
Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Name_length parameter |

The name of an area, of length Name_length, that contains the name that is to be removed. It must not contain null characters (X'00').

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_remove service returns 0 if the request completes successfully, or −1 if the request is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_remove service stores the return code. The v_remove service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_remove service can return one of the following values in the Return_code parameter:

| **Return_code** | **Explanation** |
|---|---|
| EACCES | The process did not have write permission for the directory that contains the name that is to be removed. |
| EAGAIN | The name cannot be removed, because it is temporarily unavailable. The following reason code can accompany the return code: JRInvalidVnode. |
| EBUSY | The file is open by a remote NFS client with a share reservation that conflicts with the requested operation. |
| EINVAL | Parameter error; for example, the vnode token parameter is stale. The following reason codes can accompany the return code: JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS, JRNoName, JRNullInPath. |
| ENAMETOOLONG | Name_length exceeds 255 characters. |
| ENOENT | Name was not found. |
| ENOTDIR | The file that was specified by Directory_vnode_token is not a directory. The following reason code can accompany the return code: JRTokNotDir. |

| Return_code | Explanation |
|---|---|
| EPERM | The operation is not permitted. The caller of the service is not registered as a server; or Name specifies a directory. The following reason codes can accompany the return code: JRNotRegisteredServer, JRNotForDir. |
| EROFS | The name that is to be removed is on a read-only file system. The following reason code can accompany the return code: JRReadOnlyFS. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_remove service stores the reason code. The v_remove service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. If the sticky bit is on in the parent directory, the file cannot be deleted.
2. If the name that is specified refers to a symbolic link, the symbolic link file that is named by Name is deleted.
3. If the v_remove service request is successful and the link count becomes zero, the file is deleted. The contents of the file are discarded, and the space it occupied is freed for reuse. However, if another process (or more than one) has the file open, or has a valid vnode token, when the last link is removed, the file contents are not discarded until the last process closes the file or releases the vnode token.
4. When the v_remove service is successful in removing a directory entry and decrementing the link count, whether or not the link count becomes zero, it returns control to the caller with Return_value set to 0. It updates the change and modification times for the parent directory, and the change time for the file itself (unless the file is deleted).
5. Directories cannot be removed using v_remove. To remove a directory, refer to "v_rmdir (BPX1VRE, BPX4VRE) — Remove a directory" on page 347.
6. A file may not be removed if it is currently open by a remote NFS client with a share reservation that would prevent the file from being opened for write access.

## Related services


- "v_link (BPX1VLN, BPX4VLN) — Create a link to a file" on page 288
- "v_lookup (BPX1VLK, BPX4VLK) — Look up a file or directory" on page 303
- "v_mkdir (BPX1VMK, BPX4VMK) — Create a directory" on page 307
- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_rel (BPX1VRL, BPX4VRL) — Release a vnode token" on page 337
- "v_remove (BPX1VRM, BPX4VRM) — Remove a link to a file" on page 339
- "v_rename (BPX1VRN, BPX4VRN) — Rename a file or directory" on page 343
- "v_rmdir (BPX1VRE, BPX4VRE) — Remove a directory" on page 347

## Characteristics and restrictions

A process must be registered as a server before the v_remove service is permitted. See “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333.

## Examples

For an example using this callable service, see “BPX1VRM, BPX4VRM (v_remove) example” on page 493.

# v_rename (BPX1VRN, BPX4VRN) — Rename a file or directory

## Function

The v_rename service renames a file or a directory that is specified by the Old_name parameter in the directory that is represented by Old_directory_vnode_token to the name that is specified by the New_name parameter in the directory that is represented by New_directory_vnode_token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRN): | 31-bit |
| AMODE (BPX4VRN): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRN,(Old_directory_vnode_token,
              OSS,
              Old_name_length,
              Old_name,
              New_directory_vnode_token,
              New_name_length,
              New_name,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRN with the same parameters.

## Parameters

**Old_directory_vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory in which the file or directory that is to be renamed exists.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating system specific parameters. This area is mapped by the BPXYOSS macro, see “BPXYOSS — Map operating system specific information” on page 469.

**Old_name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the file or directory name that is to be renamed. The name can be up to 255 bytes long.

**Old_name**
Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Old_name_length parameter |

The name of an area, of length Old_name_length, that contains the file or directory name that is to be renamed. It must not contain null characters (X'00').

**New_directory_vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory in which the renamed file or directory is to exist.

**New_name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of the file or directory name to which the file or directory is to be renamed. The name can be up to 255 bytes long.

**New_name**
Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by New_name_length parameter |

The name of an area, of length New_name_length, that contains the file or directory name to which the file or directory is to be renamed. It must not contain null characters (X'00').

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rename service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rename service stores the return code. The v_rename service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_rename service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | The calling process does not have permission to write in a specified directory. |
| EAGAIN | One of the files or directories was temporarily unavailable. The following reason code can accompany the return code: JRInvalidVnode. |
| EBUSY | The name that was specified is in use as a mount point or the file is open by a remote NFS client with a share reservation that conflicts with the requested operation. The following reason code can accompany the return code: JRIsFSRoot. |
| EINVAL | Parameter error—for example, attempting to rename a file named "..". The following reason codes can accompany the return code: JRDotorDotDot, JrOldPartOfNew, JrNoName, JrNullInPath, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EISDIR | An attempt was made to rename something other than a directory to a directory. |
| ENAMETOOLONG | A name is longer than 255 characters. |
| ENOSPC | The directory that is intended to contain New_name cannot be extended. |
| ENOTDIR | The supplied token did not represent a directory; or an attempt was made to rename a directory to something other than a directory. |
| ENOTEMPTY | New_name specified an existing directory that was not empty. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |
| EROFS | The specified file system is read-only. The following reason code can accompany the return code: JRReadOnlyFS. |
| EXDEV | An attempt was made to rename across file systems. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_rename service stores the reason code. The v_rename service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. If the sticky bit is on in the parent directory, special ownership is required to rename the file.
2. The v_rename service changes the name of a file or a directory from Old_name to New_name. When renaming completes successfully, the change and modification times for the parent directories of Old_name and New_name are updated.
3. For renaming to succeed, the calling process needs write permission for the directory that contains Old_name and the directory that contains New_name. If Old_name and New_name are the names of directories, the caller does not need write permission for the directories themselves.

4. **Renaming Files:**
   - If Old_name and New_name are links that refer to the same file, v_rename returns successfully and does not perform any other action.
   - If Old_name is the name of a file, New_name must also name a file, not a directory. If New_name is an existing file, it is unlinked. Then the file that is specified as Old_name is given New_name. The pathname New_name always stays in existence; at the beginning of the operation, New_name refers to its original file, and at the end, it refers to the file that used to be Old_name.
   - If Old_name is the name of a file that is currently open by a remote NFS client with a share reservation that would prevent the file from being opened for writing, the file cannot be renamed.
5. **Renaming Directories:**

   If Old_name is the name of a directory, New_name must also name a directory, not a file. If New_name is an existing directory, it must be empty, containing no files or subdirectories. If empty, it is removed, as described in "v_remove (BPX1VRM, BPX4VRM) — Remove a link to a file" on page 339.

   New_name cannot be a directory under Old_name; that is, the old directory cannot be part of the pathname prefix of the new one.

## Related services

- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333

## Characteristics and restrictions

A process must be registered as a server before the v_rename service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VRN, BPX4VRN (v_rename) example" on page 494.

# v_rmdir (BPX1VRE, BPX4VRE) — Remove a directory

## Function

The v_rmdir service removes a directory. The directory must be empty.

Directory_name is specified as input, along with a Directory_vnode_token value that identifies the directory that contains the directory that is to be removed.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRE): | 31-bit |
| AMODE (BPX4VRE): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRE,(Directory_vnode_token,
              OSS,
              Directory_name_length,
              Directory_name,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRE with the same parameters.

## Parameters

**Directory_vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8 byte area that contains a vnode token that represents the directory from which the v_rmdir service is to remove the directory that is supplied in the Directory_name parameter.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Directory_name_length**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of Directory_name. The name can be up to 255 bytes long.

**Directory_name**
Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Directory_name_length parameter |

The name of an area, of length Directory_name_length, that contains the name of the directory that is to be removed. It must not contain null characters (X'00').

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rmdir service returns `0` if the request completes successfully, or `−1` if the request is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rmdir service stores the return code. The v_rmdir service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_rmdir service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | The process did not have write permission for the directory that contains the directory that is to be removed. |
| EBUSY | The directory cannot be removed, because it is being used as a mount point. The following reason code can accompany the return code: JRIsFSRoot. |
| EAGAIN | The directory cannot be removed, because it is temporarily unavailable. The following reason code can accompany the return code: JRInvalidVnode. |
| EINVAL | Parameter error; for example, the Vnode_token parameter is obsolete. The following reason codes can accompany the return code: JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS, JRDotOrDotDot, JRNoName, JRNullInPath. |
| ENAMETOOLONG | Directory_name_length exceeds 255 characters. |
| ENOENT | The directory that was specified by Directory_name was not found. |
| ENOTDIR | The file that was specified by Directory_vnode_token is not a directory; or the name that was specified by Directory_name is not a directory. The following reason codes can accompany the return code: JRTokNotDir, JRNotDir. |
| ENOTEMPTY | The directory contains files or subdirectories. |

| Return_code | Explanation |
| --- | --- |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |
| EROFS | The directory that is to be removed is on a read-only file system. The following reason code can accompany the return code: JRReadOnlyFS. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_rmdir service stores the reason code. The v_rmdir service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. If the sticky bit is on in the parent directory, the target directory cannot be removed.
2. The directory that is specified by Directory_name must be empty.
3. If the directory is successfully removed, the change and modification times for the parent directory are updated.
4. If any process has the directory open when it is removed, the directory itself is not removed until the last process has closed the directory. New files cannot be created under a directory that is removed, even if the directory is still open.

## Related services

- "v_mkdir (BPX1VMK, BPX4VMK) — Create a directory" on page 307
- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_rel (BPX1VRL, BPX4VRL) — Release a vnode token" on page 337
- "v_remove (BPX1VRM, BPX4VRM) — Remove a link to a file" on page 339

## Characteristics and restrictions

A process must be registered as a server before the v_rmdir service is permitted. See "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VRE, BPX4VRE (v_rmdir) example" on page 495.

# v_rpn (BPX1VRP, BPX4VRP)) — Resolve a pathname

## Function

The v_rpn service accepts an absolute pathname of a file or a directory and returns a vnode token that represents this file or directory, and the VFS token that represents the mounted file system that contains the file or directory. These tokens must be supplied by the server on any subsequent VFS callable services API that is related to these files, directories, or file systems. The v_rpn service also returns file attribute information for the file or directory, and mount information for the file system.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VRP): | 31-bit |
| AMODE (BPX4VRP): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VRP,(OSS,
              Pathname_length,
              Pathname,
              VFS_token,
              Vnode_token,
              Mnte_length,
              Mnte,
              Attr_length,
              Attr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VRP with the same parameters.

## Parameters

**OSS**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Pathname_length**

Supplied parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword that contains the length of the full pathname of the file or directory that is to be resolved to a token. The name can be up to 1023 bytes long; each component of the name (between delimiters) can be up to 255 bytes long.

**Pathname**
Supplied parameter
**Type:** Character string
**Length:** Specified by Pathname_length parameter

The name of an area, of length Pathname_length, that contains the full name of the file or directory that is to be resolved.

**VFS_token**
Returned parameter
**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area in which the v_rpn service returns the VFS token of the file system that contains the file or directory that is supplied in the Pathname parameter.

**Vnode_token**
Returned parameter
**Type:** Token
**Length:** 8 bytes

The name of an 8-byte area in which the v_rpn service returns a vnode token of the file or directory that is supplied in the Pathname parameter.

**Mnte_length**
Supplied parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword that contains the length of the area that is passed in the Mnte parameter.

The length of this area must be large enough to contain a mount entry header (MnteH) and one mount entry (Mnte). These fields are mapped by the BPXYMNTE macro (see “BPXYMNTE — Map response and element structure of w_getmnte” on page 463).

**Mnte**
Returned parameter
**Type:** Structure
**Length:** Specified by the Mnte_length parameter

The name of an area, of length Mnte_length, in which the v_rpn service returns information about the file system that contains the file or directory that is supplied in the Pathname parameter. This area is mapped by the BPXYMNTE macro (see “BPXYMNTE — Map response and element structure of w_getmnte” on page 463).

**Attr_length**
Supplied parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword that contains the length of the area that is passed in the Attr parameter. To determine the value of Attr_length, use the ATTR structure (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**Attr**

Returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area, of length Attr_length, in which the v_rpn service returns the file attribute structure for the file or directory that is supplied in the Pathname parameter. This area is mapped by the ATTR structure (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rpn service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rpn service stores the return code. The v_rpn service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_rpn service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, the Pathname parameter did not contain an absolute pathname; or one of the supplied areas was too small. The following reason codes can accompany the return code: JRNoLeadingSlash, JRSmallAttr, JRSmallMnte, JRInvalidOSS, JRNullInPath. |
| ELOOP | Too many symbolic links were encountered in the pathname. |
| EMFILE | The maximum number of vnode tokens have been created. |
| ENAMETOOLONG | The pathname or a component in the pathname is too long. |
| ENFILE | An error occurred while storage was being obtained for a vnode token. |
| ENOENT | A directory or file that was supplied in the Pathname parameter does not exist; or the Pathname_length parameter is not greater than 0. |
| ENOTDIR | A node in the pathname is not a directory. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_rpn service stores the reason code. The v_rpn service returns a Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. Vnode tokens that are returned by the v_rpn service are not inherited across a fork callable service.
2. VFS tokens that are returned by the v_rpn service are inherited across a fork callable service.
3. The mount point pathname is not returned in the Mnte structure that is returned by v_rpn.
4. The caller is responsible for freeing the vnode token that is returned by the v_rpn service, by calling to the v_rel service when it is no longer needed.

## Related services

- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_rel (BPX1VRL, BPX4VRL) — Release a vnode token" on page 337

## Characteristics and restrictions

A process must be registered as a server before the v_rpn service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VRP, BPX4VRP (v_rpn) example" on page 496.

# v_setattr (BPX1VSA, BPX4VSA) — Set the attributes of a file

## Function

The v_setattr service sets the attributes that are associated with the file that is represented by Vnode_token. It can be used to change the mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, and file size. It can also be used to set the initial security label for a file or directory.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VSA): | 31-bit |
| AMODE (BPX4VSA): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VSA,(Vnode_token,
              OSS,
              Attr_length,
              Attr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VSA with the same parameters.

## Parameters

**Vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the file.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Attr_length**

Supplied parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword that contains the length of Attr. To determine the value of Attr_length, use the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**Attr**

Supplied and returned parameter

**Type:** Structure

**Length:** Specified by the Attr_length parameter

The name of an area, of length Attr_length, that contains the file attributes to be set for the file that is specified by the vnode token. The attributes of the file are also returned in this area, overlaying the input values. This area is mapped by the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**Return_value**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_setattr service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_setattr service stores the return code. The v_setattr service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_setattr service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallAttr, JRInvalidAttr, JRNegativeValueInvalid, JRTrNotRegFile, JRTrNegOffset, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| EACCES | The calling process did not have appropriate permissions. Possible reasons include:<br>• In an attempt to set access time or modification time to current time, the effective UID of the calling process does not match the owner of the file, the process does not have write permission for the file, and the process does not have appropriate privileges.<br>• In an attempt to truncate the file, the calling process does not have write permission for the file. |

| Return_code | Explanation |
|---|---|
| EFBIG | A process attempted to change the size of a file, but the new length that was specified is greater than the maximum file size limit for the process. The following reason code can accompany the return code: JRWriteBeyondLimit. |
| EPERM | The operation is not permitted for one of the following reasons:<br>• The caller of the service is not registered as a server.<br>• In an attempt to change the mode or the file format, the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.<br>• In an attempt to change the owner, the calling process does not have appropriate privileges.<br>• In an attempt to change the general attribute bits, the calling process does not have write permission for the file.<br>• In an attempt to set a time value (not current time), the effective user ID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.<br>• In an attempt to set the change time or reference time to current time, the calling process does not have write permission for the file.<br>• In an attempt to change auditing flags, the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.<br>• In an attempt to change the security auditor's auditing flags, the user does not have auditor authority.<br>• In an attempt to set the security label, one or more of the following conditions applies:<br>– The calling process does not have RACF SPECIAL authorization and appropriate privileges.<br>– The security label that is currently associated with the file is already set. |
| EROFS | The file is on a read-only file system. The following reason code can accompany the return code: JRReadOnlyFS. |
| ESTALE | On input, the AttrGuardTimeChk bit was on, and the input AttrGuardTime value did not match the Ctime of the file. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the v_setattr service stores the reason code. The v_setattr service returns Reason_code only if Return_value is −1.

Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

*Table 7. Attributes fields*

| Set Flags | Attribute Fields Input | Description |
|---|---|---|
| ATTRMODECHG | ATTRMODE | Set the mode according to the value in ATTRMODE. |
| ATTRUID<br>ATTRGID | Set the owner user ID (UID) and group ID (GID) to the values specified in ATTRUID and ATTRGID. | |
| ATTRSETGEN | ATTRGENVALUE<br>ATTRGENMASK | Only the bits corresponding to the bits set ON in the ATTRGENMASK are set to the value (ON or OFF) in ATTRGENVALUE. Other bits will be unchanged. |
| ATTRTRUNC | ATTRSIZE | Truncate the file size to ATTRSIZE bytes. |
| ATTRATIMECHG | ATTRATIME | Set the access time of the file to the value specified in ATTRATIME. |
| ATTRATIMECHG and ATTRATIMETOD | None | Set the access time of the file to the current time. |
| ATTRMTIMECHG | ATTRMTIME | Set the modification time of the file to the value specified in ATTRMTIME. |
| ATTRMTIMECHG and ATTRMTIMETOD | None | Set the modification time of the file to the current time |
| ATTRMAAUDIT | ATTRAUDITORAUDIT | Set the security auditor's auditing flags to the value specified in ATTRAUDITORAUDIT. |
| ATTRMUAUDIT | ATTRUSERAUDIT | Set the user's auditing flags to the value specified in ATTRUSERAUDIT. |
| ATTRCTIMECHG | ATTRCTIME | Set the change time of the file to the value specified in ATTRCTIME. |
| ATTRCTIMECHG and ATTRCTIMETOD | None | Set the change time of the file to the current time. |
| ATTRREFTIMECHG | ATTRREFTIME | Set the reference time of the file to the value specified in ATTRREFTIME. |
| ATTRREFTIMECHG and ATTRREFTIMETOD | None | Set the reference time of the file to the current time. |
| ATTRFILEFMTCHG | ATTRFILEFMT | Set the file format of the file to the value specified in ATTRFILEFMT. |
| ATTRSECLABELCHG | ATTRSECLABEL | Set the initial security label for a file or directory. |

1. Flags in the attributes parameter are set to indicate which attributes should be updated. To set an attribute, turn the corresponding Set Flag on, and set the corresponding attributes field according to Table 7 on page 357. Multiple attributes may be changed at the same time.

   The **Set Flag** field should be cleared before any bits are turned on. It is considered an error if any of the reserved bits in the flag field are turned on.
2. In addition to the attribute fields that are specified according to Table 7 on page 357, the following ATTR header fields must be provided by the caller:

   **ATTRID** Contains “ATTR”.

   **ATTRLEN** Specifies the length of the ATTR structure.

   **AttrGuardTimeChk** Indicates whether the AttrGuardTime should be checked.

   **AttrGuardTime** If this bit is on, the PFS checks the Ctime of the file against the value that is specified in AttrGuardTime. If they do not match, the request fails with ESTALE.

   Other fields in the ATTR should be set to 0s.
3. Changing mode (ATTRMODECHG = ON):
   - The file mode field in the ATTR area is mapped by the BPXYMODE macro (see “BPXYMODE — Map the mode constants of the file services” on page 466). For information on the values for file type, see “BPXYFTYP — File type definitions” on page 451.
   - Files that are open when the v_setattr service is called retain the access permission they had when the file was opened.
   - The effective UID of the calling process must match the file’s owner UID, or the caller must have appropriate privileges.
   - Setting the set-group-ID-on-execution permission (in mode) means that when this file is run, through the exec service, the effective GID of the caller is set to the file’s owner GID, so that the caller seems to be running under the GID of the file, rather than that of the actual invoker.

     The set-group-ID-on-execution permission is set to zero if both of the following are true:
     – The caller does not have appropriate privileges.
     – The GID of the file’s owner does not match the effective GID or one of the supplementary GIDs of the caller.
   - Setting the set-user-ID-on-execution permission (in mode) means that when this file is run, the process’s effective UID is set to the file’s owner UID, so that the process seems to be running under the UID of the file’s owner, rather than that of the actual invoker.
4. Changing owner (ATTROWNERCHG = ON):
   - For changing the owner UID of a file, the caller must have appropriate privileges.
   - For changing the owner GID of a file, the caller must have appropriate privileges, or meet all of these conditions:
     – The effective UID of the caller matches the file’s owner UID.
     – The Owner_UID value that is specified in the change request matches the file’s owner UID.
     – The Group_ID value that is specified in the change request is the effective GID, or one of the supplementary GIDs, of the caller.

   - When owner is changed, the set-user-ID-on-execution and set-group-ID-on-execution permissions of the file mode are automatically turned off.
   - When owner is changed, both UID and GID must be specified as they are to be set. If only one of these values is to be changed, the other must be set to its present value or to -1 in order to remain unchanged.
5. Changing general attribute bits (ATTRSETGEN = ON):
   - For general attribute bits to be changed, the calling process must have write permission for the file.
6. Truncating a file (ATTRTRUNC = ON):
   - The truncation of a file to ATTRSIZE bytes changes the file size to ATTRSIZE, beginning from the first byte of the file. If the file was originally larger than ATTRSIZE bytes, the data from ATTRSIZE to the original end of file is removed. If the file was originally shorter than ATTRSIZE, bytes between the old and new lengths are read as zeros.
   - Full blocks are returned to the file system so that they can be used again. The file offset is not changed.
   - When a file is truncated successfully, it clears the set-user-ID, the set-group-ID, and the save-text (sticky bit) attributes of the file unless the caller has authority to access the root.
   - Changing a file's size is considered to be a write operation and an open token from a prior v_open may be passed in the OSS to indicate that this change is being done within the open context of that token. Consequently, the operation does not have to be verified against the share reservations that may currently be in effect for the file. If no open token is available to pass on the call, there are three levels of share reservation checking that can be requested (see "v_rdwr (BPX1VRW, BPX4VRW) — Read from and write to a file" on page 322 for details).
7. Changing times:
   - All time fields in the Attr area are in POSIX format.
   - For the access time or the modification time to be set explicitly (ATTRATIMECHG = ON or ATTRMTIMECHG = ON), the effective ID must match the file's owner, or the process must have appropriate privileges.
   - For the access time or modification time to be set to the current time (ATTRATIMETOD = ON or ATTRMTIMETOD = ON), the effective ID must match the file's owner, the calling process must have write permission for the file, or the process must have appropriate privileges.
   - For the change time or the reference time to be set explicitly (ATTRCTIMECHG = ON or ATTRREFTIMECHG = ON), the effective ID must match the file's owner or the process must have appropriate privileges.
   - For the change time or reference time to be set to the current time (ATTRCTIMETOD = ON or ATTRREFTIMETOD = ON), the calling process must have write permission for the file.
   - When any attribute field is changed successfully, the file's change time is updated as well.
   - The setting of the AttrLP64times bit in the BPXYATT structure, and not the AMODE of the caller, determines whether 4-byte or 8-byte time fields are used.
8. Changing auditor audit flags (ATTRMAAUDIT = ON):

- For auditor audit flags to be changed, the user must have auditor authority. Users with auditor authority can set the auditor options for any file, even those they do not have path access to or authority to use for other purposes.

  You can establish auditor authority by running the TSO/E command ALTUSER Auditor.

9. Changing user audit flags (ATTRMUAUDIT = ON):
   - For the user audit flags to be changed, the user must have appropriate privileges (See Authorization in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*) or be the owner of the file.
10. Changing file format (ATTRFILEFMTCHG = ON):
    - The effective UID of the calling process must match the file's owner UID or the caller must have appropriate privileges.
11. Changing the security label (ATTSECLABELCHG=ON):
    - For the security label to be changed, the user must have RACF SPECIAL authorization and appropriate privileges, and no security label must currently exist on the file. Only an initial security label can be set. An existing security label cannot be changed. The function will successfully set the security label if the SECLABEL class is active. If the SECLABEL class is not active, the request will return successfully, but the security label will not be set.

## Related services

- "v_getattr (BPX1VGA, BPX4VGA) — Get the attributes of a file" on page 285
- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333

## Characteristics and restrictions

1. A process must be registered as a server before the v_setattr service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.
2. The ATTREXTLINK flag in the ATTRGENVALUE field of the ATTR cannot be modified with BPX1VSA.
3. The general attribute fields (set by ATTRSETGEN, ATTRGENMASK, and ATTRGENVALUE fields) are not intended as a general-use programming interface on v_setattr.
4. The security label (ATTRSECLABELCHG) flag requires RACF SPECIAL authorization and appropriate privileges. See Authorization in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for information about appropriate privileges.
5. The security label (ATTRSECLABELCHG) flag cannot be used to change an existing security label; it can only be used to set an initial security label on a file.

## Examples

For an example using this callable service, see "BPX1VSA, BPX4VSA (v_setattr) example" on page 497.

# v_symlink (BPX1VSY, BPX4VSY) — Create a symbolic link

## Function

The v_symlink service creates a symbolic link to a pathname or external name. A file whose name is specified in the Link_name parameter, of type “symbolic link”, is created within the directory that is represented by Directory_vnode_token. The contents of the symbolic link file is the pathname or external name that is specified in Pathname.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | PASN = HASN |
| AMODE (BPX1VSY): | 31-bit |
| AMODE (BPX4VSY): | 64-bit |
| ASC mode: | Primary mode |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL BPX1VSY,(Directory_vnode_token,
              OSS,
              Link_name_length,
              Link_name,
              Pathname_length,
              Pathname,
              Attr_length,
              Attr,
              Return_value,
              Return_code,
              Reason_code)
```

AMODE 64 callers use BPX4VSY with the same parameters.

## Parameters

**Directory_vnode_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of an 8-byte area that contains a vnode token that represents the directory in which the v_symlink service creates the new symbolic link file that is named in the Link_name parameter.

**OSS**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | OSS#LENGTH (from the BPXYOSS macro) |

The name of an area that contains operating-system-specific parameters. This area is mapped by the BPXYOSS macro (see “BPXYOSS — Map operating system specific information” on page 469).

**Link_name_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of Link_name. The Link_name can be up to 255 bytes long.

**Link_name**

Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by Link_name_length parameter |

The name of a field that contains the symbolic link that is being created. It must not contain null characters (X'00').

**Pathname_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of Pathname. The Pathname can be up to 1023 bytes long. If the Pathname is not an external name (AttrExtLink = 0), each component of the name (between delimiters) can be up to 255 bytes long.

**Pathname**

Supplied parameter

| | |
|---|---|
| **Type:** | Character string |
| **Length:** | Specified by the Pathname_length parameter |

The name of a field that contains the pathname or external name for which you are creating a symbolic link.

A pathname can begin with or without a slash.

- If the pathname begins with a slash, it is an *absolute* pathname, the slash refers to the root directory, and the search for the file starts at the root directory.
- If the pathname does not begin with a slash, it is a *relative* pathname, and the search for the file starts at the parent directory of the symbolic link file.

A pathname must not contain null characters (X'00').

An external name is the name of an object that is outside the hierarchical file system. There are no restrictions on the characters that may be used in an external name.

**Attr_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the length of Attr. To determine the value of Attr_length, use the ATTR structure (see “BPXYATTR — Map file attributes for v_ system calls” on page 445).

**Attr**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Attr_length parameter |

The name of an area, of length Attr_length, that is to be used by the v_symlink service to set the attributes of the file that is to be created. This area is mapped by the ATTR structure (see "BPXYATTR — Map file attributes for v_ system calls" on page 445).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_symlink service returns `0` if the request is successful, or `−1` if it is not successful.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_symlink service stores the return code. The v_symlink service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The v_symlink service can return one of the following values in the Return_code parameter:

| **Return_code** | **Explanation** |
|---|---|
| EACCES | The calling process does not have permission to write in the directory specified. |
| EEXIST | Link_name already exists. |
| EFBIG | The file size limit for the process is set to zero, prohibiting the creation of symbolic links. |
| EINVAL | Parameter error; for example, a supplied area was too small. The following reason codes can accompany the return code: JRSmallAttr, JRInvalidAttr, JRNoName, JRInvalidSymLinkLen, JRNULLInPath, JRInvalidSymLinkComp, JRVTokenFreed, JRWrongPID, JRStaleVnodeTok, JRInvalidVnodeTok, JRInvalidOSS. |
| ENAMETOOLONG | Link_name is longer than 255 characters. |
| ENOTDIR | The supplied token did not represent a directory. |
| EPERM | The operation is not permitted. The caller of the service is not registered as a server. |
| EROFS | Directory_vnode_token specifies a directory on a read-only file system. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the v_symlink service stores the reason code. The v_symlink service returns a Reason_code only if Return_value is `−1`. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. The following Attr fields are provided by the caller:

   | | |
   |---|---|
   | **AttrID** | Contains Attr#ID (from the ATTR structure) |
   | **AttrLen** | Specifies the length of the ATTR structure. |
   | **AttrExtLink** | Specifies whether the Pathname is an external name (1) or a pathname in a hierarchical file system (0). |
   | **AttrMode** | Specifies directory mode permission bits. See “BPXYMODE — Map the mode constants of the file services” on page 466 for the mapping of this field. |

   Other fields in the ATTR should be set to zeros.
2. Like a hard link (described in “v_link (BPX1VLN, BPX4VLN) — Create a link to a file” on page 288), a symbolic link allows a file to have more than one name. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link, however, provides no such assurance; in fact, the file identified by Pathname need not exist when the symbolic link is created. In addition, a symbolic link can cross file system boundaries, and can refer to objects that are outside a hierarchical file system.
3. When a component of a pathname refers to a symbolic link (but not an external symbolic link) rather than to a directory, the pathname that is contained in the symbolic link is resolved. When the VFS callable services API, v_rpn, or other z/OS UNIX callable services are being used, a symbolic link in a pathname parameter is resolved as follows:
   - If the pathname in the symbolic link begins with / (slash), the symbolic link pathname is resolved relative to the process root directory.
   - If the pathname in the symbolic link does not begin with /, the symbolic link pathname is resolved relative to the directory that contains the symbolic link.
   - If the symbolic link is not the last component of the original pathname, remaining components of the original pathname are resolved from there.
   - When a symbolic link is the last component of a pathname, it may or may not be resolved. Resolution depends on the function that is using the pathname. For example, a rename request does not have a symbolic link resolved when it appears as the final component of either the new or old pathname. However, an open request does have a symbolic link resolved when it appears as the last component.
   - When a slash is the last component of a pathname, and it is preceded by a symbolic link, the symbolic link is always resolved.
   - The mode of a symbolic link is ignored during the lookup process. Any files and directories to which a symbolic link refers are checked for access permission.
4. The external name that is contained in an external symbolic link is not resolved. Link_name cannot be used as a directory component of a pathname.
5. If the file size limit for the process is set to zero, symbolic link creation is prohibited and fails with EFBIG.
6. The valuethat is set by **umask()** for the process does not affect the setting of the mode permission bits.

## Related services


- "v_getattr (BPX1VGA, BPX4VGA) — Get the attributes of a file" on page 285
- "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333
- "v_readlink (BPX1VRA, BPX4VRA) — Read a symbolic link" on page 330
- "v_remove (BPX1VRM, BPX4VRM) — Remove a link to a file" on page 339
- "v_link (BPX1VLN, BPX4VLN) — Create a link to a file" on page 288


## Characteristics and restrictions

A process must be registered as a server before the v_symlink service is permitted; see "v_reg (BPX1VRG, BPX4VRG) — Register a process as a server" on page 333.

## Examples

For an example using this callable service, see "BPX1VSY, BPX4VSY (v_symlink) example" on page 498.

# Chapter 6. OSI services

The LFS provides several Operating System Interface (OSI) callable services specifically for PFSs.

The addresses of these routines are passed to a PFS during initialization in the OSI operations vector table (OSIT structure). For information about how the OSIT is passed to the PFS during initialization, see "Activating and deactivating the PFS" on page 4. See also Appendix D for more information on C language structures that are referred to in this document.

Table 8 shows the OSI services.

*Table 8. OSI services*

| Service | Description |
|---|---|
| osi_copyin | Copy data to a PFS buffer |
| osi_copyout | Copy data to a user buffer |
| osi_copy64 | Move data between user and PFS buffers with 64-bit addresses |
| osi_ctl | Pass control information to the kernel |
| osi_getcred | Obtain SAF UIDs, GIDs, and supplementary IDs |
| osi_getvnode | Get a vnode |
| osi_kipcget | Query interprocess communications |
| osi_kmsgctl | Control in-kernel messages |
| osi_kmsgget | Get a message queue ID |
| osi_kmsgrcv | Receive an in-kernel message from the queue |
| osi_kmsgsnd | Send an in-kernel message to the queue |
| osi_mountstatus | Report file system status |
| osi_post | General post |
| osi_sched | Schedule Part 2 of Async I/O |
| osi_selpost | Select post |
| osi_signal | Send a signal |
| osi_sleep | Wait for a resource |
| osi_thread | Fetch a module from a thread |
| osi_uiomove | Move data between buffers |
| osi_upda | Update Async I/O request |
| osi_wait | General wait |
| osi_wakeup | Wake a task waiting for a resource |

This chapter describes each of the OSI services, which are arranged in alphabetic order. The OSI services are callable services that are generally called only from within a PFS. Some of these services must be called from the same thread that is making a VFS or vnode call. The information about callable services from Chapter 5 applies here, with a few exceptions:

- The service name that is listed above is a C-language macro that invokes the particular service based on its address in the OSIT structure.

© Copyright IBM Corp. 1996, 2006

- The three ways of invoking a module that are listed in Chapter 5 do not apply to these services. They must be called with the saved OSIT structure address, by using the macros listed in Table 8 on page 367.

  Assembler language programs must use the OSIT structure offsets for each service. These offsets can be found in the OSIT typedef in Appendix D.

**Note:** Any of the output parameters of a call can be modified by the system, whether the call is successful or not.

## Using OSI services from a non-kernel address space

The osi_post, osi_selpost, and osi_wakeup services can be called from a non-kernel address space to wake up a thread that is waiting for some event to occur. Osi_ctl can be used from a non-kernel address space to communicate with a file exporter exit program. The osi_sched service can be called to initiate Part 2 of an asynchronous I/O.

For example, if a PFS establishes its own communication mechanism to another separate address space, there may be times when it needs to wait for a reply from that address space. In these cases the PFS can call osi_wait, while running on the user's thread in either the kernel or the other address space, and a program in that other address space can call osi_post to wake it up. A recovery option is available through the v_reg() function that will ensure that these waiting processes are posted if the separate address space should terminate abnormally.

Similarly, if the PFS participates in select() processing and the selected event occurs in another address space, osi_selpost can be called from that other address space.

This section does not apply to calls that are made by the PFS while in the kernel or in a colony address space. For these calls, the OSIT table address that is passed during initialization should be used.

To use the OSI services from an independent (non-cross-memory) thread in another address space, or from an end user thread that has PCed from the PFS to another address space, you must perform the following steps from an authorized program that is running in non-cross-memory mode in that other address space:

1. Issue an MVS LOAD for the module BPXVOSIT.
2. Branch to the address that is returned by LOAD, passing the standard return_value, return_code, and reason_code parameters with OS linkage.

   The program must be authorized at the time of this branch, so that a PC (Program Call instruction) can be set up between this address space and the kernel.

   If return_value is not -1, it will be the address of an OSIT in this address space.
3. Save the OSIT address returned from a successful LOAD and branch.
4. Do not DELETE the BPXVOSIT load module. All the addresses of the OSI services are within this load module.

The constants and prototype related to doing this are included in Appendix D.

From this point on, you can call the OSI services from this address space (via the saved OSIT address) from C or assembler programs the same way a PFS does. The calling program does not have to be authorized at the time of an OSI service call, unless the service specifically requires it.

The following restrictions on using the OSI services from an independent task apply:

- A task in the server process can use the standard IPC message interface to communicate with a PFS that is using the osi_kmsg interface, so osi_getipc and the osi_kmsg services are not intended to be used from an independent task.
- A task in the server process can use the standard kill() function to send a signal; osi_signal should not be used.
- Osi_copyin, osi_copyout, and osi_uiomove should not be used to copy from or to the user address space buffers that were passed on a PFS operation.
- Osi_getvnode, osi_sleep, and osi_thread may not be used. Osi_wait may be used after some special setup. Refer to the Usage Notes for osi_wait for details.

The effect of loading and calling BPXVOSIT is tied to the address space. BPXVOSIT cannot be called twice unless z/OS UNIX has terminated and restarted.

If z/OS UNIX terminates, new OSI service requests fail with an EMVSNOTUP return code. Calls that are in progress when z/OS UNIX terminates may receive a cross memory abend. After z/OS UNIX is restarted, BPXVOSIT must be recalled to reestablish the PC to the new kernel.

If the separate address space is started before the kernel address space, a call to BPXVOSIT that is issued during initialization fails. Generally, a PFS contacts its partner address space during z/OS UNIX initialization, and the load and call can be performed at this time. As an alternative, you can listen for the Event Notification Facility (ENF) Signal, which is issued whenever z/OS UNIX is started. During initialization, an address space can set up an ENF Listen for this event and call BPXVOSIT. If BPXVOSIT fails with EMVSNOTUP, the ENF Signal is eventually issued and BPXVOSIT can be called again after the server’s ENF Listen exit is invoked. The ENF Qualifier Constant is defined in macro BPXYENFO.

# osi_copyin — Move data from a user buffer to a PFS buffer

## Function

The osi_copyin service moves a block of data from a user buffer to a PFS buffer.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state; any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_copyin(Destination_buffer,
           Destination_ALET,
           Source_buffer,
           Source_ALET,
           Source_key,
           Move_length,
           Return_value,
           Return_code,
           Reason_code);
```

## Parameters

**Destination_buffer**
Supplied parameter

| | |
|---|---|
| **Type:** | Char |
| **Length:** | Value specified by Move_length. |

The name of the buffer in the PFS to which data is copied.

**Destination_ALET**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The ALET for the specified Destination_buffer in the PFS.

**Source_buffer**
Supplied parameter

| | |
|---|---|
| **Type:** | Char |
| **Length:** | Value specified by Move_length. |

The name of the buffer in the user address space from which data is copied.

**Source_ALET**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The ALET for the specified Source_buffer in the user address space.

**Source_key**
Supplied parameter

**Type:** Integer
**Length:** Fullword

The storage key for the Source_buffer in the user address space. The specified key should be in the last 4 bits of the word. The key is typically the same value as the key stored in the UIO field UIO.u_key.

**Move_length**
Supplied parameter

**Type:** Integer
**Length:** Fullword

The number of bytes to move.

**Return_value**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the osi_copyin service returns the results of the request, as one of the following:

| Return_value | Meaning |
|---|---|
| **-1** | The operation was not successful. The Return_code and Reason_code parameters contain the values returned by the service. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the osi_copyin service stores the return code. The osi_copyin service returns Return_code only if Return_value is –1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

The osi_copyin operation supports the following error value:

| Return_code | Explanation |
|---|---|
| EFAULT | A specified buffer address is not in addressable storage. |

**Reason_code**
Returned parameter

**Type:** Integer
**Length:** Fullword

A fullword in which the osi_copyin service stores the reason code. The osi_copyin service returns Return_code only if Return_value is –1. Reason_code further qualifies the Return_code value. The reason codes are described in *z/OS UNIX System Services Messages and Codes*.

## Usage notes

1. The address of the osi_copyin routine is passed to the PFS in the OSIT structure when the PFS is initialized.
2. The storage key for the destination buffer can be any storage key.

## Related services

- “osi_uiomove — Move data between PFS buffers and buffers defined by a UIO structure” on page 426
- “osi_copyout — Move data from a PFS buffer to a user buffer” on page 373
- “osi_copy64 — Move data between user and PFS buffers with 64-bit addresses” on page 376

## Characteristics and restrictions

This routine must be used only on the dispatchable unit (task or SRB) that made the vnode or VFS call because the service requires the use of the cross-memory environment of the calling dispatchable unit.

# osi_copyout — Move data from a PFS buffer to a user buffer

## Function

The osi_copyout service moves a block of data from a PFS buffer to a user buffer.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state; any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_copyout(Destination_buffer,
            Destination_ALET,
            Source_buffer,
            Source_ALET,
            Destination_key,
            Move_length,
            Return_value,
            Return_code,
            Reason_code);
```

## Parameters

**Destination_buffer**
Supplied parameter

| | |
|---|---|
| **Type:** | Char |
| **Length:** | Value specified by Move_length. |

The name of the buffer in the user address space to which data is copied.

**Destination_ALET**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The ALET for the specified Destination_buffer in the user address space.

**Source_buffer**
Supplied parameter

| | |
|---|---|
| **Type:** | Char |
| **Length:** | Value specified by Move_length. |

The name of the buffer in the PFS from which data is copied.

**Source_ALET**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The ALET for the specified Source_buffer in the PFS.

**Destination_key**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The storage key for the Destination_buffer in the user address space. The specified key should be in the last 4 bits of the word. The key is typically the same value as the key stored in the UIO field UIO.u_key

**Move_length**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The number of bytes to move.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_copyout service returns the results of the request, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **–1** | The operation was not successful. The Return_code and Reason_code parameters contain the values returned by the service. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_copyout service stores the return code. The osi_copyout service returns Return_code only if Return_value is –1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

The osi_copyout operation supports the following error value:

| **Return_code** | **Explanation** |
|---|---|
| EFAULT | A buffer address that was specified is not in addressable storage. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_copyout service stores the reason code. The osi_copyout service returns Return_code only if Return_value is –1. Reason_code further qualifies the Return_code value. The reason codes are described in *z/OS UNIX System Services Messages and Codes*.

## Usage notes

1. The address of the osi_copyout routine is passed to the PFS in the OSIT structure when the PFS is initialized.
2. The storage key for the source buffer can be any storage key.

## Related services

- “osi_uiomove — Move data between PFS buffers and buffers defined by a UIO structure” on page 426
- “osi_copyin — Move data from a user buffer to a PFS buffer” on page 370
- “osi_copy64 — Move data between user and PFS buffers with 64-bit addresses” on page 376

## Characteristics and restrictions

This routine must be used only on the dispatchable unit (task or SRB) that made the vnode or VFS call because the service requires the use of the cross-memory environment of the calling dispatchable unit.

# osi_copy64 — Move data between user and PFS buffers with 64-bit addresses

## Function

The osi_copy64 service moves a block of data in either direction between 64-bit addressed user and PFS buffers.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state; any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31- or 64-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be 31-bit addressable by the caller and in the primary address space. |

## Format

```
osi_copy64(copy64_struct,
          Workarea);
```

## Parameters

**copy64_struct**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the 64_length field |

The parameters of this service are contained in the copy64_struct. See “Usage notes” for a description of the fields in this structure.

**Workarea**
Supplied parameter

| | |
|---|---|
| **Type:** | Character |
| **Length:** | 512 bytes |

**Workarea** is a 512-byte buffer that resides below the 2GB line and is aligned on a doubleword boundary. It can be used by the service for dynamic storage.

## Usage notes

1. The osi_copy64 service can be called in AMODE 31 or AMODE 64, and the buffers may be above or below the 2GB line. In all cases the full 64-bit addresses must be valid. In releases prior to z/OS V1R5, the osi_copy64 service may be called only in AMODE 31.
2. The size of the R1 address and of the parameter list addresses that it points to are assumed to correspond to the AMODE of the caller at the time of the call.
3. **copy64_struct** contains the following fields:

| | |
|---|---|
| **c64_sourcebuff** | The source address for the copy. The source is always copied to the destination (**c64_destbuff**). |
| **c64_destbuff** | The destination address for the copy. |
| **c64_direction** | Specifies whether MVCSK (In) or MVCDK (Out) should be used. |
| **c64_keybits** | Contains the 4-bit key of the user's data. |
| **c64_copylen** | Specifies the length of the data to be copied. This is a 32-bit field. |
| **c64_dontincrsrc** | The source address will be incremented by the c64_copylen to facilitate looping calls, unless this flag is set. |
| **64_dontincrdest** | The destination address will be incremented by the c64_copylen to facilitate looping calls, unless this flag is set. |
| **c64_gotrecovery** | If the PFS has its own EFAULT recovery, you can avoid the overhead involved in the setting up and taking down of an FRR on each call to this service by setting this flag. |
| **c64_rc** | Indicates the success or failure of the operation, as described below. |
| **c64_rsn** | Indicates the success or failure of the operation, as described below. |
| **c64_sourcealet** | Contains the ALET of the source buffer. |
| **c64_destalet** | Contains the ALET of the destination buffer. |
| **c64_length** | Contains the length of the copy64_struct itself. |

4. The results of the operation are returned in c64_rc as either:

   **0** The operation was successful, and c64_copylen bytes were moved.

   **Nonzero** The operation failed. This is the failing return code, and c64_rsn contains the failing reason code. The osi_copy64 service may return the following return code:

   | Return_code | Explanation |
   |---|---|
   | EFAULT | A specified buffer address is not in addressable storage. |

5. The osi_copy64 routine is a high-performance routine. It does not issue program calls (PC) into the kernel.
6. If the PFS has no storage below the 2GB line for the Workarea, the OSI WorkArea can be used.

## Related services

- "osi_uiomove — Move data between PFS buffers and buffers defined by a UIO structure" on page 426
- "osi_copyout — Move data from a PFS buffer to a user buffer" on page 373
- "osi_copyin — Move data from a user buffer to a PFS buffer" on page 370

## Characteristics and restrictions

Whenever it is used to copy user address space areas, this routine must be used only on the dispatchable unit (task or SRB) that made the original vnode or VFS call because the service requires the use of the cross-memory environment of the calling dispatchable unit.

# osi_ctl — Pass control information to the kernel

## Function

The osi_ctl service passes control information to the kernel or to a file exporter exit in the kernel.

## Requirements

| | |
|---|---|
| Authorization: | Problem or Supervisor state, any key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_ctl(Command,
        Argument_length,
        Argument,
        Return_value,
        Return_code,
        Reason_code);
```

## Parameters

**Command**
Supplied parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword that contains the command code for this operation. The allowable value is:
*1* — for file exporter exit control

**Argument_length**
Supplied parameter.

**Type:** Integer
**Length:** Fullword

The name of a fullword that contains the length of the argument. The length of the argument is specified as an integer value in the range 0-256.

**Argument**
Parameter supplied and returned

**Type:** Defined by the receiver.
**Length:** Specified by the Argument_length parameter

Specifies the name of a buffer, of length Argument_Length, that contains the argument of the operation.

The buffer may be modified to return information to the caller.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_ctl service returns 0 if the request is successful, and −1 if the request is not successful.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_ctl service stores the return code. The osi_ctl service returns Return_code only if Return_value is −1. For a complete list of possible return code values, see *z/OS UNIX System Services Messages and Codes*. The return code may come from the exporter exit.

The osi_ctl service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EINVAL | A supplied parameter is incorrect.<br><br>One of the following Reason_codes may accompany this Return_code:<br>• JRNotRegisteredServer - The caller is not registered or is not a file exporter type.<br>• JRInvIoctlCmd - The Command was not a supported value. |
| ENOMEM | A C environment cannot be obtained to invoke the exit. |
| ESRCH with JrPfsNotDubbed | The caller is not on a POSIX thread. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_ctl service stores the reason code. The osi_ctl service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. For the reason codes, see *z/OS UNIX System Services Messages and Codes*.

The reason code may come from the exporter exit, in which case it would be documented with that product.

## Usage notes

1. This service is provided for general communication between a file exporter and the exporter exit that was established during v_reg().

   The argument buffer may be modified to convey information from the exit to the caller. The exit must not write in the argument buffer beyond the amount that was passed in by the caller. The caller and the exit should agree on the size of the argument, or should use an imbedded length field to limit the amount of data that is moved to the argument buffer for output.

   If the amount of data to be transferred is more than the amount that is allowed by this service, the caller should use the argument to pass the address of a buffer that contains the data. The exit can use osi_copyin and osi_copyout to move data between the caller's address space and the kernel.

Refer to "DFS-style file exporters" on page 255 for more information on file exporters.

2. The address of the osi_ctl routine is passed to the PFS in the OSIT structure when the PFS is initialized.

## Characteristics and restrictions

None.

# osi_getcred — Obtain SAF UIDs, GIDs and supplementary GIDs

## Function

The osi_getcred obtains the real, effective, and saved user IDs; group IDs; and supplementary group IDs from SAF.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and, except for the Getcred_Parms and area for the supplemental GIDs, must be in the primary address space. |

## Format

```
osi_getcred(OSI_structure,
        Workarea,
        Alet,
        Getcred_Parms,
        Return_value,
        Return_code,
        Reason_code);
```

## Parameters

**OSI_structure**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Osilen field |

OSI_structure contains information that is used by the OSI operations. The PFS receives this structure on each PFS interface operation.

Refer to Appendix D for a full description of this structure.

**Workarea**

Supplied parameter

| | |
|---|---|
| **Type:** | Char |
| **Length:** | 3072 bytes |

Workarea is a buffer of 3072 bytes (3K), aligned on a doubleword boundary, that is to be used by this OSI operation.

**Alet**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The Alet for the Getcred_Parms structure and the supplementary GID list that is pointed to by Getcred_Parms.

**Getcred_parms**

Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by sizeof(OGCDPRM) |

An area that contains the osi_getcred parameters. The entries in this area are mapped by the OGCDPRM typedef, which is defined in the BPXYPFSI header.

Refer to Appendix D for a full description of this structure. Following is a description of the parameters in this structure:

| | |
|---|---|
| **oc_hdr** | A header that contains an eyecatcher and length. It can be initialized using OGCDPRM_HDR. |
| **oc_real_uid** | The real UID, returned by the security product. |
| **oc_effective_uid** | The effective UID, returned by the security product. |
| **oc_saved_uid** | The saved UID, returned by the security product. |
| **oc_real_gid** | The real GID, returned by the security product. |
| **oc_effective_gid** | The effective GID, returned by the security product. |
| **oc_saved_gid** | The saved GID, returned by the security product. |
| **oc_maxsgids** | Set by the invoker to the maximum number of supplementary GIDs that will fit in the area that is pointed to by oc_gid_list. If there is not enough room for all available GIDs, this maximum is returned. In this case, this field is updated, on return to the caller, to indicate the total number of GIDs which could have been returned had there been room for all. |
| **oc_numsgids** | The number of supplementary GIDs returned by the security product. |
| **oc_gid_list** | A pointer to an area to contain the array of supplementary GIDs. |

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_getcred service returns the results of the service, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **–1** | The operation was not successful. The Return_code and Reason_code parameters contain the values returned by the service. |

| | |
|---|---|
| **0** | The operation was successful, and there was room for all supplementary GIDs in the caller-provided area. |
| **+1** | The operation was successful, but there were more supplementary GIDs than could fit in the caller-provided area. A partial list of GIDs has been returned. The oc_maxsgids field has been updated with the actual number of supplementary GIDs that are available. The oc_maxsgids field should be reset to the proper value, if necessary, before the Getcred_Parms structure is used again on a subsequent call. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_getcred service stores the return code. The osi_getcred service returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_getcred service stores the reason code. The osi_getcred service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. The reason codes are described in *z/OS UNIX System Services Messages and Codes*.

Note that if the Return_code that is returned by osi_getcred is EMVSSAF2ERR, the low-order two bytes of the Reason_code will be the RACF return and reason codes.

## Usage notes

1. The osi_getcred calls SAF to obtain the UID and GID information.
2. If there is not room in the supplementary GID area, SAF returns as many as will fit. A return value of 1 indicates that this has occurred. In this case, the oc_maxsgids field is updated with the number that would have been returned had there been room for all supplementary GIDs. The caller should not depend upon those GIDs that are returned when there is not enough room for all supplementary GIDs. The subset of the available GIDs that is returned may differ among various security products, or even from call to call for some security products.
3. The OSI_structure contains an area that is pointed to by osi_workarea, which may be passed to this service as the Workarea parameter.

## Related services

None.

## Characteristics and restrictions

None.

# osi_getvnode — Get or return a vnode

## Function

The osi_getvnode service is used by a PFS to create a vnode or return a vnode that it created but never used.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state, PSW key 0 |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_getvnode(Entry_code,
        Token_structure,
        attribute_structure,
        PFS_token,
        Vnode_token,
        Return_value,
        Return_code,
        Reason_code);
```

## Parameters

**Entry_code**

Supplied parameter

**Type:** Integer

**Length:** Fullword

Entry_code specifies the function that is being requested for the osi_getvnode service.

| Entry_code | Explanation |
|---|---|
| OSI_BUILDVNOD | Get a vnode |
| OSI_BUILDVNODNL | Get a vnode that is never locked by the LFS. |
| OSI_RTNVNOD | Return an unused vnode. |
| OSI_BUILDVNODXL | Get a vnode that is always under an exclusive lock. |
| OSI_PURGELLA | Purge LLA entries for a vnode. |
| OSI_INACTASAP | Inactivate a vnode as soon as possible. |
| OSI_MEMCRITICAL | PFS requests memory relief for its file systems. |

**Token_structure**

Supplied parameter

**Type:** TOKSTR

**Length:** Specified by TOKSTR.ts_hdr_cblen.

This token_structure is the one that was passed to the vnode or VFS operation from which this call is being made. It represents the parent file or file system of the file for which a vnode is being created. This parameter is 0 for OSI_PURGELLA and OSI_INACTASAP.

**attribute_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the structure's attr.cbhdr.cblen field. |

The file attributes of the file for which this vnode is being created. This structure is mapped by typedef ATTR in the BPXYVFSI header file (see Appendix D).

**PFS_token**
Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 |

The PFS token for the file for which this vnode is being created.

**Vnode_token**
Returned parameter for entry code OSI_BUILDVNOD, OSI_BUILDVNODXL, and OSI_BUILDVNODNL; supplied parameter for entry code OSI_RTNVNOD, OSI_PURGELLA, and OSI_INACTASAP.

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 |

The vnode token for the file.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_getvnode service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_getvnode service stores the return code. The osi_getvnode service can return one of the following values in the Return_code parameter only if Return_value is −1. Reason_code further qualifies the Return_code value.

| Return_code | Explanation |
|---|---|
| 0 | Successful completion |
| Osi_BadParm | Invalid OSI_structure |
| Osi_Abend | Abend in osi_getvnode |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |

**Length:** Fullword

A fullword in which the osi_getvnode service stores the reason code. The osi_getvnode service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value.

## Usage notes

1. For additional information, see “Creating, referring to, and inactivating file vnodes” on page 31.
2. The PFS should use the OSI_RTNVNOD function to return an unused vnode only when it gets a vnode, but decides it does not need it, before returning the vnode token to the LFS.

   The Token_structure, attribute_structure, and PFS_token parameters are not referenced for OSI_RTNVNOD, and the PFS may pass a zero for each parameter.
3. The address of the osi_getvnode routine is passed to the PFS in the OSIT when the PFS is initialized.
4. OSI_BUILDVNODNL is used when the PFS does not need the vnode serialization provided by the LFS. A vnode that is obtained in this way is locked only for vn_open and vn_close.
5. The PFS may pass a minimum File_Attribute_Structure, for performance reasons. This structure must include:

   **at_hdr.cbid** Set to ATT2 to distinguish this subset ATTR
   **at_hdr.cblen** Set to the correct length
   **at_mode** The file type field, at least
   **at_ino**
   **at_major**
   **at_minor**
   **at_genvalue** The LFS bits, at least
   **at_fid**
6. No Token_structure is required on an OSI_PURGELLA or OSI_INACTASAP request. This parameter may be 0.

## Characteristics and restrictions

1. This routine can be called only for a vnode or VFS operation that returns a vnode token on the interface—for example, vn_lookup.
2. This routine must be used only on the task that made the vnode or VFS call, with the exception of the OSI_INACTASAP requests. OSI_INACTASAP requests can be invoked on a physical file system worker task; no serialization is necessary for these operations.
3. OSI_MEMCRITICAL is not a vnode-related function. Only the Token_Structure is used as input. The PFS should first check the PFS initialization block (PFSI) to see if the OSI_MEMCRITICAL function is supported. If it is, the PFS may use it to request memory relief by requesting that LFS clean up the vnode cache quickly. The PFS will also be called to harden its cached data to disk for each of its mounted file systems, using vfs_sync. To indicate the completion of this LFS memory-critical function, LFS will set the ts_sysd1 field to OSI_MEMCRITICAL for the last vfs_sync operation.

# osi_kipcget — Query interprocess communications

## Function

The osi_kipcget service queries shared memory, messages and semaphors for the ″next or specified member″.

| This is a secondary interface to the w_getipc service. It is provided for use by a PFS that is running in a colony address space. For information on the w_getipc service, see w_getipc (BPX1GET, BPX4GET) — Query interprocess communications in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state; any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL osi_kipcget(Ipc_Token | Ipc_Member_ID,
                 Buffer_Address,
                 Buffer_Length,
                 Command,
                 Return_value,
                 Return_code,
                 Reason_code);
```

## Parameters

**Ipc_Token**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Word |

Specifies a token that corresponds to a message queue, shared memory segment, or semaphore member ID. Zero represents the first member ID. The token that is to be used in the next invocation is passed back in Return_value. Ipc_Token is ignored when Ipc_OVER is specified.

**Ipc_Member_ID**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Word |

Specifies a message queue ID, semaphore ID, or shared member ID.

**Buffer_address**

Supplied parameter

| | |
|---|---|
| **Type:** | Address |
| **Length:** | Fullword |

Address of the buffer structure that is defined by IPCQ. For the structure that describes this buffer, see "BPXYIPCQ — Map w_getipc structure" on page 460.

**Buffer_Length**

Supplied parameter

| | |
|---|---|
| **Type:** | Address |
| **Length:** | Fullword |

Length of the structure that is defined by IPCQ. This parameter is set to IPCQ#LENGTH. Field IPCQLENGTH differs from IPCQ#LENGTH when the system call is at a different level than the included IPCQ. An error is returned if this length is less than 4. The buffer is filled to the lesser of IPCQ#LENGTH or the value specified here.

**Command**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

| Command | Description |
|---|---|
| **Ipcq#ALL** | Retrieve next shared memory, message and semaphore member. |
| **Ipcq#MSG** | Retrieve next message member. |
| **Ipcq#SEM** | Retrieve next semaphore member. |
| **Ipcq#SHM** | Retrieve next shared memory member. |
| **Ipcq#OVER** | Overview of system variables. Ignores the value of the first operand (Ipc_Token). |

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_kipcget service returns the next Ipc_Token (a negative number), `0`, or `−1` (error). If Ipc_Token is specified, `0` indicates end of file. If Ipc_Member_ID is specified, `0` indicates success.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_kipcget service stores the return code. The osi_kipcget service returns Return_code only if Return_value is `−1`. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The osi_kipcget service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
| --- | --- |
| EINVAL | One of the following errors occurred:<br>• The Ipc_Member_ID is not valid for the command that was specified.<br>• The Command parameter is not a valid command.<br>• The buffer pointer was zero; or the buffer length was less than 4.<br><br>The following reason codes can accompany the return code: JRBuffTooSmall, JRIpcBadID, or JRBadEntryCode. |
| EFAULT | An input parameter specified an address that caused the callable service to program check. The following reason code can accompany the return code: JRBadAddress. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the osi_kipcget service stores the reason code. The osi_kipcget service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. With Ipc_Token, return_values should be tested for 0 (end of file) or -1 (error). Other values are negative and will be the next Ipc_Token.
2. With Ipc_Member_ID, return_values should be tested for -1 (error).
3. A member's accessibility can change if the permissions are changed.
4. A given Ipc_Token may not always retrieve the same member.
5. If a specific member is desired and has been found using Ipc_Token, subsequent requests may place it at that token or later (never earlier).
6. The address of the osi_kipcget routine is passed to the PFS in the OSIT structure when the PFS is initialized.

## Related services

None.

## Characteristics and restrictions

This service may be invoked only from a colony address space.

# osi_kmsgctl — Perform message queue control operations

## Function

The osi_kmsgctl service provides a variety of message control operations as specified by command. These functions include reading and changing message variables within the MSQID_DS data structure and removing a message queue from the system.

This is a secondary interface to the msgctl service. It is provided for use by a PFS running in a colony address space. For information on the msgctl service, see msgctl (BPX1QCT, BPX4QCT) — Perform message queue control operations in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state; any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL osi_kmsgctl,(Message_Queue_ID,
              Command,
              Buffer,
              Return_value,
              Return_code,
              Reason_code)
```

## Parameters

**Message_Queue_ID**
Supplied parameter
**Type:** Integer
**Length:** Fullword

Specifies the message queue identifier.

**Command**
Supplied parameter
**Type:** Integer
**Character set:**
N/A
**Length:**
Fullword

The name of a fullword field that indicates the message command that is to be executed. For the structure that contains these constants, see “BPXYIPCP — Map interprocess communication permissions” on page 459. The values for Command are:

| | |
|---|---|
| **Ipc_STAT** | This command obtains status information about the message queue that is identified by the Message_Queue_ID parameter, if the current process has read permission. This information is stored in the area that is pointed to by argument Buffer and mapped by area MSQID_DS data structure. For the data structure, see “BPXYMSG — Map interprocess communication message queues” on page 462, MSQID_DS DSECT. |
| **Ipc_SET** | Set the value of the IPC_UID, IPC_GID, IPC_MODE and MSG_QBYTES for associated Message_queue_ID. The values that are to be set are taken from the MSQID_DS data structure that is pointed to by argument Buffer. Any value for IPC_UID and IPC_GID may be specified. Only mode bits that are defined by BPX1QGT under the Message_Flag argument may be specified in the IPC_MODE field. This Command can only be executed by a task that has an effective user ID equal to either that of a task with appropriate privileges, or the value of IPC_CUID or IPC_UID in the MSQID_DS data structure that is associated with Message_Queue_ID. This information is taken from the buffer that is pointed to by the Buffer parameter. For the data structure, see “BPXYMSG — Map interprocess communication message queues” on page 462, MSQID_DS DSECT. |
| **Ipc_RMID** | Remove the message identifier that is specified by Message_Queue_ID from the system, and destroy the message queue and MSQID_DS data structure that are associated with it. This Command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges, or the value of IPC_CUID or IPC_UID in the MSQID_DS data structure that is associated with Message_Queue_ID. |

**Buffer**

Parameter supplied and returned

| | |
|---|---|
| **Type:** | Address |
| **Length:** | Fullword |

The name of the fullword that contains the address of the buffer into which or from which the message queue information is to be copied. This buffer is mapped by MSQID_DS.

**Return_value**

Returned parameter

**Type:** Integer
**Length:** Fullword

The name of a fullword in which the osi_kmsgctl service returns −1 or 0.

**Return_code**

Returned parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword in which the osi_kmsgctl service stores the return code. The osi_kmsgctl service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The osi_kmsgctl service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | The command specified was Ipc_STAT, and the calling process does not have read permission. The following reason code can accompany the return code: JRIpcDenied. |
| EINVAL | One of the following errors occurred:<br>• Message_Queue_ID is not a valid Message queue identifier<br>• The Command parameter is not a valid command.<br>• The mode bits were not valid (SET).<br><br>The following reason codes can accompany the return code: JRIpcBadFlags, JRMsqQBytes, or JRIpcBadID. |
| EPERM | The command specified was Ipc_RMID or Ipc_SET, and the effective user ID of the caller is not that of a process with appropriate privileges, and is not the value of IPC_CUID or IPC_UID in the MSQID_DS data structure that is associated with Message_Queue_ID.<br><br>The command specified was Ipc_SET, and an attempt is being made to increase MSG_QBYTES. The effective user ID of the caller does not have superuser privileges. The following reason codes can accompany the return code: JRIpcDenied or JRMsqQBytes. |
| EFAULT | The Buffer parameter specified an address that caused the syscall to program check. The following reason code can accompany the return code: JRBadAddress. |

**Reason_code**

Returned parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword in whic the osi_kmsgctl service stores the reason code. The osi_kmsgctl service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. Changing the access permissions only affects message queue syscall invocations that occur after msgctl has returned. msgsnd and msgrcv, which are waiting while the permission bits are changed by msgctl, are not affected.
2. Ipc_SET can change permissions, and may affect the ability of a thread to use the next message queue syscall.
3. Quiescing a message queue stops additional messages from being added, while allowing existing messages to be received. A message queue can be quiesced by clearing (Ipc_SET) write permission bits.
4. A message queue can also be quiesced by reducing MSG_QBYTES (Ipc_SET) to zero. (Note: it would take superuser authority to re-raise the limit.) Requesters are told EAGAIN or wait.
5. When a message queue ID is removed (Ipc_RMID) from the system, all waiting threads regain control with RV=-1, RC=EIDRM, and RC=JRIpcRemoved.
6. If you do not wish to change all the fields, first initialize (Ipc_STAT) the buffer, change the desired fields, and then make the change (Ipc_SET).
7. For Command Ipc_RMID, the remove is complete by the time control is returned to the caller.

## Related services

- “osi_kmsgget — Create or find a message queue” on page 395

## Characteristics and restrictions

This service may be invoked only from a colony address space.

The caller is restricted by ownership, read, and read-write permissions that are defined by OSI_kmsgget and OSI_kmsgctl Ipc_SET.

# osi_kmsgget — Create or find a message queue

## Function

The osi_kmsgget service returns a message queue ID.

This is a secondary interface to the msgget service. It is provided for use by a PFS running in a colony address space. For information on the msgget service, see msgget (BPX1QGT, BPX4QGT) — Create or find a message queue in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state; any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL osi_kmsgget,(Key,
              Message_Flag,
              Return_value,
              Return_code,
              Reason_code)
```

## Parameters

**Key**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

Identification for this message queue. This is a user-defined value that serves as a lookup value to determine if this message queue already exists, or the reserved value Ipc_PRIVATE.

**Message_Flag**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

Valid values for this field include any combination of the following (additional bits cause an EINVAL):

| | |
|---|---|
| **Ipc_CREAT** | Creates a message queue if the key that is specified does not already have an associated ID. Ipc_CREATE is ignored when Ipc_PRIVATE is specified. |

| | |
|---|---|
| **Ipc_EXCL** | Causes the msgget function to fail if the key that is specified has an associated ID. Ipc_EXCL is ignored when Ipc_CREAT is not specified, or when Ipc_PRIVATE is specified. |
| **S_IRUSR** | Permits the process that owns the message queue to read it. |
| **S_IWUSR** | Permits the process that owns the message queue to alter it. |
| **S_IRGRP** | Permits the group that is associated with the message queue to read it. |
| **S_IWGRP** | Permits the group that is associated with the message queue to alter it. |
| **S_IROTH** | Permits others to read the message queue. |
| **S_IWOTH** | Permits others to alter the message queue. |

The values that begin with an ″Ipc_″ prefix are defined in BPXYIPCP, and are mapped onto S_TYPE, which is in BPXYMODE.

The values that begin with an ″S_I″ prefix are defined in BPXYMODE, and are a subset of the access permissions that apply to files.

**Return_value**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the osi_kmsgget service returns −1 or the message queue identifier.

**Return_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the osi_kmsgget service stores the return code. The osi_kmsgget service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The osi_kmsgget service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | A message queue identifier exists for the Key parameter, but operation permission, as specified by the low-order 9-bits of the Message_Flag parameter, is not granted (the ″S_″ items). The following reason code can accompany the return code: JRIpcDenied. |
| EEXIST | A message queue identifier exists for the Key parameter, and both Ipc_CREAT and Ipc_EXCL are specified. The following reason code can accompany the return code: JRIpcExists. |
| EINVAL | The Message_Flag operand included bits that are not supported by this function. The following reason code can accompany the return code: JRIpcBadFlags. |

| Return_code | Explanation |
|---|---|
| ENOENT | A message queue identifier does not exist for the Key parameter, and Ipc_CREAT was not set. The following reason code can accompany the return code: JRIpcNoExist. |
| ENOSPC | The system limit of the number of message queue IDs has been reached. The following reason code can accompany the return code: JRIpcMaxIDs. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the osi_kmsgget service stores the reason code. The osi_kmsgget service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. As long as a thread knows the message queue ID, it may issue a msgctl, msgsnd, or msgrcv (msgget is not needed).
2. This function returns the message queue identifier that is associated with the Key parameter.
3. This function creates a data structure, which is defined by MSQID_DS, if one of the following is true:
   - The Key parameter is equal to Ipc_PRIVATE.
   - The Key parameter does not already have a message queue identifier associated with it, and Ipc_CREAT is set.
4. Upon creation, the data structure that is associated with the new message queue identifier is initialized as follows:
   - Ipc_CUID and Ipc_UID are set to the effective user ID of the calling task.
   - Ipc_CGID and Ipc_GID are set to the effective group ID of the calling task.
   - The low-order 9-bits of Ipc_MODE are equal to the low-order 9-bits of the Message_Flag parameter.
   - MSG_QBYTES is set to the system limit that is defined by parmlib.
5. The message queue is removed from the system when BPX1QCT (msgctl) is called with command Ipc_RMID.
6. Users of message queues are responsible for removing them when they are no longer needed. Failure to do so ties up system resources.

## Related services

- “osi_kmsgctl — Perform message queue control operations” on page 391

## Characteristics and restrictions

1. This service may only be invoked from a colony address space.
2. There is a maximum number of message queues that are allowed in the system.
3. The caller is restricted by ownership, read, and read-write permissions that are defined by OSI_kmsgget and OSI_kmsgctl Ipc_SET.

# osi_kmsgrcv — Receive from a message queue

## Function

The osi_kmsgrcv service receives a message from a message queue.

| This is a secondary interface to the msgrcv service. It is provided for use by a PFS
| that is running in a colony address space. For information on the msgrcv service,
| see msgrcv (BPX1QRC, BPX4QRC) — Receive from a message queue in *z/OS*
| *UNIX System Services Programming: Assembler Callable Services Reference*.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state; any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL osi_kmsgrcv,(Message_Queue_ID,
             Message_Address,
             Message_Alet,
             Message_Length,
             Message_Type,
             Message_Flag,
             Return_value,
             Return_code,
             Reason_code)
```

## Parameters

**Message_Queue_ID**
Supplied parameter
**Type:** Integer
**Length:** Fullword

Specifies the message queue identifier.

**Message_Address**
Supplied parameter
**Type:** Address
**Length:** Fullword

The name of a field that contains the address of a a buffer that is mapped by MSGBUF or MSGXBUF (see “BPXYMSG — Map interprocess communication message queues” on page 462).

**Message_Alet**
Supplied parameter
**Type:** Address

**Length:** Fullword

The name of the fullword that contains the ALET for Message_Address, which identifies the address space or data space in which the buffer resides.

You should specify a Message_Alet of 0 if the buffer resides in the user's address space (current primary address space).

You should specify a Message_Alet of 2 if the buffer resides in the home address space.

If a value other than 0 or 2 is specified for the Message_ALET, the value must represent a valid entry in the dispatchable unit access list (DUAL).

**Message_Length**

Supplied parameter

**Type:** Integer

**Length:** Fullword

Specifies the length of the message text that is to be placed in the buffer that is pointed to by Message_Address parameter. If Msg_Info is specified, this buffer is 20 bytes longer than Message_Length; otherwise this buffer is 4 bytes longer than Message_Length. The message that is received may be truncated (see MSG_NOERROR of Message_Flag). A value of zero with MSG_NOERROR is useful for receiving the message type without the message text.

**Message_Type**

Supplied parameter

**Type:** Integer

**Length:** Fullword

Specifies the type of message that is requested, as follows:

- If Message_Type is equal to zero, the first message on the queue is received.
- If Message_Type is greater than zero, the first message of Message_Type is received.
- If Message_Type is less than zero, the first message of the lowest type that is less than or equal to the absolute value of Message_Type is received.

**Message_Flag**

Supplied parameter

**Type:** Integer

**Length:** Fullword

MSG_NOERROR specifies that the received message is to be truncated to Message_Length (mapped in BPXYMSG). The truncated part of the message is lost, and no indication of the truncation is given to the caller.

MSG_INFO specifies that the received message is to be of the MSGXBUF and not the MSGBUF format mapped in BPXYMSG. MSG_INFO specifies that extended information is to be received, which is similar to the msgxrcv() C language function.

Ipc_NOWAIT specifies the action that is to be taken if a message of the desired type is not on the queue, as follows:

- If Ipc_NOWAIT is specified, the caller is to return immediately with an error (ENOMSG).
- If Ipc_NOWAIT is not specified, the calling thread is to suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.

– The message queue is removed from the system (EIDRM).
– The caller receives a signal (EINTR).

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_kmsgrcv service returns −1, or the number of MSG_MTEXT bytes returned.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_kmsgrcv service stores the return code. The osi_kmsgrcv service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The osi_kmsgrcv service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| E2BIG | MSG_MTEXT is greater than Message_Length, and MSG_NOERROR is not set. The following reason code can accompany the return code: JRMsq2Big. |
| EACCES | Operation permission is denied to the calling task. The following reason code can accompany the return code: JRIpcDenied. |
| EIDRM | The Message_Queue_ID was removed from the system while the invoker was waiting. The following reason code can accompany the return code: JRIpcRemoved. |
| EINTR | The function was interrupted by a signal. The following reason code can accompany the return code: JRIpcSignaled. |
| EINVAL | Message_Queue_ID is not a valid message queue identifier; or the Message_Length parameter is less than 0. The following reason codes can accompany the return code: JRIpcBadID or JRMsqBadSize. |
| EFAULT | The Message_Address parameter specified an address that caused the syscall to program check. The following reason code can accompany the return code: JRBadAddress. |
| ENOMSG | The queue does not contain a message of the desired type, and Ipc_NOWAIT is set. The following reason code can accompany the return code: JRMsqNoMsg. |

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_kmsgrcv service stores the reason code. The osi_kmsgrcv service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

1. Within the type specifications, the longest waiting thread is reactivated first (FIFO). For example, if there are two threads waiting on message type 3 and one thread waiting on message type 2, when a message send for type 3 occurs, the oldest waiter for message type 3 receive is posted first.
2. Read access to the specified message queue is required.

## Related services

- “osi_kmsgctl — Perform message queue control operations” on page 391
- “osi_kmsgget — Create or find a message queue” on page 395
- “osi_kmsgsnd — Send a message to a message queue” on page 402

## Characteristics and restrictions

- This service may only be invoked from a colony address space.
- There is a maximum number of message queues that are allowed in the system.
- The caller is restricted by ownership, read, and read-write permissions that are defined by OSI_kmsgrcv and OSI_kmsgctl Ipc_SET.

# osi_kmsgsnd — Send a message to a message queue

## Function

The osi_kmsgsnd service sends a message to a message queue.

This is a secondary interface to the msgsnd service. It is provided for use by a PFS that is running in a colony address space. For information on the msgsnd service, see msgsnd (BPX1QSN, BPX4QSN) — Send to a message queue in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state or problem state; any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
CALL osi_kmsgsnd,(Message_Queue_ID,
              Message_address,
              Message_Alet,
              Message_Size,
              Message_Flag,
              Return_value,
              Return_code,
              Reason_code)
```

## Parameters

**Message_Queue_ID**
Supplied parameter
**Type:** Integer
**Character set:**
N/A
**Length:**
Fullword

Specifies the message queue identifier.

**Message_address**
Supplied parameter
**Type:** Address
**Length:** Fullword

The name of a field that contains the address of the message that is to be sent. This area is mapped by MSGBUF. The message type is the first word of the message, and must be greater than zero.

**Message_Alet**

Supplied parameter

| | |
|---|---|
| **Type:** | Address |
| **Length:** | Fullword |

The name of the fullword that contains the ALET for Message_address that identifies the address space or data space in which the buffer resides.

You should specify a Message_address of 0 if the buffer resides in the user's address space (current primary address space).

You should specify a Message_address of 2 if the buffer resides in the home address space.

If a value other than 0 or 2 is specified for the Message_ALET, the value must represent a valid entry in the dispatchable unit access list (DUAL).

**Message_Size**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

Specifies the length of the message text that is pointed to by the Message_address parameter. The length does not include the 4-byte type that precedes the message text. For example, a message with a MSG_TYPE and no MSG_MTEXT has a Message_Size of zero.

**Message_Flag**

Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

Specifies the action that is to be taken if one or more of the following are true:

- Placing the message on the message queue would cause the current number of bytes on the message queue (msg_cbytes) to be greater than the maximum number of bytes that are allowed on the message queue (msg_qbytes).
- The total number of messages on the message queue (msg_qnum) is equal to the system-imposed limit.

The actions that are taken are as follows:

- If Ipc_NOWAIT is specified, the caller returns immediately with an error (EAGAIN).
- If Ipc_NOWAIT is not specified, the calling thread suspends execution until one of the following occurs:
  – The message is sent.
  – The message queue is removed from the system (EIDRM).
  – The caller receives a signal (EINTR).

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_kmsgsnd service returns `−1` or `0`. The message was sent unless a `−1` is received.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |

**Length:** Fullword

The name of a fullword in which the osi_kmsgsnd service stores the return code. The osi_kmsgsnd service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of possible return code values. The osi_kmsgsnd service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| EACCES | Operation permission is denied to the calling task. The following reason code can accompany the return code: JRIpcDenied. |
| EAGAIN | The message cannot be sent, and Message_Flag is set to Ipc_NOWAIT. The following reason codes can accompany the return code: JRMsqQueueFullMessages, JRMsqQueueFullBytes. |
| EIDRM | The Message_Queue_ID was removed from the system while the invoker was waiting. The following reason code can accompany the return code: JRIpcRemoved. |
| EINTR | The function was interrupted by a signal, and the message was not sent. The following reason code can accompany the return code: JRIpcSignaled. |
| EINVAL | Message_Queue_ID is not a valid message queue identifier; the value of MSG _TYPE is less than 1; or the value of Message_Size is less than zero or greater than the system-imposed limit. The following reason codes can accompany the return code: JRIpcBadID, JRMsqBadSize or JRMsqBadType. |
| EFAULT | The Message_address parameter specified an address that caused the syscall to program check. The following reason code can accompany the return code: JRBadAddress. |
| ENOMEM | There are not enough system storage exits to send the message; the message was not sent. The following reason code can accompany the return code: JrSmNoStorage. |

**Reason_code**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the osi_kmsgsnd service stores the reason code. The osi_kmsgsnd service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value. See *z/OS UNIX System Services Messages and Codes* for the reason codes.

## Usage notes

| | |
|---|---|
| **Authorization:** | Supervisor state or problem state, any PSW key |
| **Dispatchable unit mode:** | Task |
| **Cross memory mode:** | PASN = HASN |
| **AMODE:** | 31-bit |
| **ASC mode:** | Primary mode |
| **Interrupt status:** | Enabled for interrupts |
| **Locks:** | Unlocked |
| **Control parameters:** | All parameters must be addressable by the caller and in the primary address space |

## Related services

- “osi_kmsgget — Create or find a message queue” on page 395
- “osi_kmsgctl — Perform message queue control operations” on page 391
- “osi_kmsgrcv — Receive from a message queue” on page 398

## Characteristics and restrictions

1. This service may only be invoked from a colony address space.
2. The caller is restricted by ownership, read, and read-write permissions that are defined by OSI_kmsgsnd and OSI_kmsgctl Ipc_SET.

# osi_mountstatus — Report file system status to LFS

## Function

The osi_mountstatus service is used by a PFS to indicate to the LFS a change in the status of a file system, such as completion of an asynchronous mount operation.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state; any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_mountstatus(Entry_code,
                StDev,
                Return_value,
                Return_code,
                Reason_code);
```

## Parameters

**Entry_code**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

Entry_code specifies the function that is being requested for the osi_mountstatus service.

| Entry_code | Explanation |
|---|---|
| OSI_MOUNTCOMPLETE | Mount complete |

**StDev**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

This is a copy of MTAB.mt_stdev that is passed by the LFS on the original vfs_mount. It identifies the file system for which status is being reported.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_mountstatus service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| −1 | The operation was not successful. |
| 0 | The operation was successful. |

**Return_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword in which the osi_mountstatus service stores the return code. The osi_mountstatus service can return the following value in the Return_code parameter only if Return_value is −1. Reason_code further qualifies the Return_code value.

| Return_code | Explanation |
|---|---|
| EINVAL | Parameter error. Consult Reason_code to determine the exact reason the error occurred. The following reason codes can accompany the return code: JRIsMounted, JRBadStDev. |

**Reason_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

A fullword in which the osi_mountstatus service stores the reason code. The osi_mountstatus service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value.

## Usage notes

1. For the OSI_MOUNTCOMPLETE Entry_code:
   - The PFS uses this entry code to inform the LFS of the completion of a mount operation that was previously identified as asynchronous. The LFS calls vfs_mount again to complete the mount. See “Asynchronous mounting” on page 28.
   - If the PFS has a Return_value, Return_code, and Reason_code to present, indicating the status of the mount, they must be returned to the LFS at the time vfs_mount is called again.
2. The address of the osi_mountstatus routine is passed to the PFS in the OSIT when the PFS is initialized.

# osi_post — Post an OSI waiter

## Function

The osi_post service posts a process that is waiting in osi_wait.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state, any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_post(OSI_post_token,
         Return_code);
```

## Parameters

**OSI_post_token**

Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 24 |

OSI_post_token is the post token that is saved from the OSI_structure of the task that is to be posted.

Refer to Appendix D for a full description of this structure.

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_post service stores the return code. The osi_post service can return one of the following values in the Return_code parameter:

| **Return_code** | **Explanation** |
|---|---|
| 0 | Successful completion |
| Osi_BadParm | Invalid OSI_structure |
| Osi_Abend | Abend in osi_post |

## Usage notes

1. For additional information, see “Waiting and posting” on page 21.
2. The task that is posted is the task that is represented by OSI_post_token. Before a PFS uses OSI_wait, it should copy the OSI_post_token from the OSI

structure to a place that is addressable by the task that performs the OSI_post. The storage for the OSI for the waiting task is freed if the task terminates.

3. The PFS must never call OSI_post for a waiting process more than once, and it should have sufficient logic and recovery to avoid calling OSI_post for a task that is no longer waiting.
4. The address of the osi_post routine is passed to the PFS in the OSIT structure when the PFS is initialized.

## Related services

- "osi_wait — Wait for an event to occur" on page 431

# osi_sched — Schedule async I/O completion

## Function

The osi_sched service schedules the completion of an asynchronous request.

## Requirements

| | |
|---|---|
| Authorization: | Problem or Supervisor state, any key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_sched(Saved_Osi_AsyTok,
          Return_value,
          Return_code,
          Reason_code);
```

## Parameters

**Saved_Osi_AsyTok**

Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 8 bytes |

The name of the field that contains the osi_asytok value that was saved by the PFS during Part 1 of the asynchronous vnode operation that is now completing.

**Return_value**

Supplied and returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the PFS passes the results of Part 1 of the asynchronous operation, and the LFS returns the results of the scheduling.

**On input to osi_sched:**

| Return_value | Meaning |
|---|---|
| `0 or greater` | The request is successful.<br>The LFS invokes the PFS for Part 2 of the asynchronous operation.<br>If the user has requested a preprocessing exit call, this value is passed to the exit before Part 2 is invoked. |

On receive-type operations, the PFS should pass the actual length of the data that is to be received, if it can do so at this point. This allows a preprocessing exit to allocate smaller buffers than the size that was originally specified at the beginning of the operation. If this value cannot be passed to osi_sched, a Return_value of 0 should be used, and the exit will allocate buffers to accommodate the amount that was originally requested. See asyncio (BPX1AIO, BPX4AIO) — Asynchronous I/O for sockets in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for more details about these operations.

**-1**
The request has failed.

The LFS does not invoke the PFS for Part 2 of the asynchronous operation.

The Return_value, Return_code, and Reason_code are passed back to the user as the results of the operation.

If the PFS has resources to free that cannot be freed by the caller of osi_sched, or if for any other reason Part 2 needs to be called, it should set Return_value to 0 and report the failure of the user's operation as output from Part 2.

**On output from osi_sched:**

| Return_value | Meaning |
|---|---|
| **0** | The scheduling was successful. The LFS invokes the PFS for Part 2 of the asynchronous operation based on the input Return_value, as explained above. If Part 2 cannot be run because of process termination, the PFS gets a vn_cancel instead. |
| **-1** | The Saved_Osi_AsyTok value was not recognized. The LFS always accepts valid calls to osi_sched. Even when the user's process is terminating and Part 2 cannot be run, cleanup for the request is deferred to vn_cancel. If the saved LFS token is bad, it is not clear what the PFS should do about it. This could be a logic error in the PFS. If the value that was passed was once an LFS token, and this is not a late or redundant call, then it is unlikely that this can happen, because the LFS does not clean up its request while there is still any valid chance that osi_sched will be called. |

**Return_code**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the PFS passes the return code from Part 1, and the LFS returns the return code from the scheduling.

The Return_code parameter is meaningful only if Return_value is −1.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the PFS passes the reason code from Part 1 and the LFS returns the reason code from the scheduling. Reason_code further qualifies the Return_code value.

The Reason_code parameter is meaningful only if Return_value is −1.

For the reason codes, see *z/OS UNIX System Services Messages and Codes*..

## Usage notes

1. Refer to "Asynchronous I/O processing" on page 55 for details on asynchronous operations.
2. osi_sched is called by the PFS when an asynchronous vnode operation is ready to complete. For instance, data has arrived for receive-type operations or buffers are available for write-type operations.
3. Osi_asytok on entry to Part 1 of an asynchronous vnode operation contains the LFS's request token. This value must be saved by the PFS, and is used here to identify the operation that is completing.
4. As a result of calling osi_sched, the LFS re-calls the PFS for Part 2 of the original operation. Part 2 is run from an SRB in the user's address space.

## Characteristics and restrictions

None.

# osi_selpost — Post a process waiting for select

## Function

The osi_selpost service posts a process that is waiting because of a select request.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state, any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_selpost(Select_token,
            Return_value,
            Return_code,
            Reason_code);
```

## Parameters

**Select_token**

Supplied parameter

| | |
|---|---|
| **Type:** | Token |
| **Length:** | 16 bytes |

Select_token is the token that was saved by the PFS when it was called for the select query request. The PFS does not need to be aware of the contents of this field; it just needs to save it on the select request and pass it to this module when it is time to post.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_selpost operation returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. |
| **0** | The operation was successful. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_selpost operation stores the return code. The osi_selpost operation returns Return_code only if Return_value is −1. For a complete list of supported return code values, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_selpost operation stores the reason code. The osi_selpost operation returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value.

## Usage notes

1. For additional information, see "Select/poll processing" on page 45.
2. The task that is posted is the task that is represented by Select_token. Before a PFS uses osi_selpost, it should copy the Select_token to a place that is addressable by the task that will perform the osi_selpost.
3. The PFS must never call osi_selpost for a waiting process more than once, and it should have sufficient logic and recovery to avoid calling osi_selpost for a task that is no longer waiting.
4. The address of the osi_selpost routine is passed to the PFS in the OSIT structure when the PFS is initialized.

## Related services

- "vn_select — Select or poll on a vnode" on page 204

## Characteristics and restrictions

The caller of this service must be on a process thread.

# osi_signal — Generate the requested signal event

## Function

The osi_signal service generates the requested signal to the target process.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_signal(OSI_structure,
         Target_Osipid,
         Signal_value,
         Signal_options,
         Return_value,
         Return_code,
         Reason_code);
```

## Parameters

**OSI_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Osilen field. |

OSI_structure contains information that is used by the OSI operations. The PFS receives this structure on each PFS interface operation.

Refer to Appendix D for a full description of this structure.

**Target_Osipid**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A copy of the Osipid field from the target process.

**Signal_value**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the signal value. See *z/OS XL C/C++ Run-Time Library Reference* for a description of the **signal.h** header and the signal values.

**Signal_options**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword that contains the signal option flags. See kill (BPX1KIL, BPX4KIL) — Send a signal to a process in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for a description of the declaration of signal option flags.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_signal service returns the results of the signal request, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. |
| **0** | The operation was successful. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_signal service stores the return code. The osi_signal service returns Return_code only if Return_value is −1. See *z/OS UNIX System Services Messages and Codes* for a complete list of supported return code values.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_signal service stores the reason code. The osi_signal service returns Return_code only if Return_value is −1. Reason_code further qualifies the Return_code value. The reason codes are described in *z/OS UNIX System Services Messages and Codes*.

## Usage notes

1. The PFS must have the process ID of the task that is to receive the signal. This information must be retrieved from the target OSI_structure and placed in a variable that is visible to the task that will eventually invoke the osi_signal service.
2. The address of the osi_signal routine is passed to the PFS in the OSIT structure when the PFS is initialized.

# osi_sleep — Sleep until a resource is available

## Function

The osi_sleep service waits for an osi_wakeup to be called with a matching Resource_id and Pfs_id.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_sleep(OSI_structure,
          Resource_id,
          Time_interval,
          Return_value,
          Return_code,
          Reason_code);
```

## Parameters

**OSI_structure**
Supplied parameter

**Type:** OSI
**Length:** Specified by OSI.osi_hdr.cblen.

OSI_structure contains information that is used by the OSI operations. The PFS receives this structure on each PFS interface operation.

Refer to Appendix D for a full description of this structure.

**Resource_id**
Supplied parameter

**Type:** Token
**Length:** Fullword

The Resource_id identifies the resource for which the thread is waiting.

**Time_interval**
Supplied parameter

**Type:** Integer
**Length:** Doubleword

The Time_interval is the maximum time for which osi_sleep will sleep. The value is specified in timer units and is rounded up to approximate seconds (the value of the high-order word). See *z/Architecture Principles of Operation* for more information about timer units. The rounded-up value is added to the current time; therefore a very large time interval added to the current time could wrap to a very small number and result in an immediate timeout of osi_sleep. A value of 0 indicates that there is no time limit.

**Return_value**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_sleep service returns the results of the operation as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. |
| **0** | The operation was successful, and the task was awakened by osi_wakeup. |

**Return_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_sleep service stores the return code. The osi_sleep service can return one of the following values in the Return_code parameter only if Return_value is −1. Reason_code further qualifies the Return_code value.

| Return_code | Explanation |
|---|---|
| EDEADLK | An FRR was active when the service was requested. |
| EINTR | The service was interrupted. Consult Reason_code to determine the exact reason the error occurred. The following reason codes can accompany the return code: JRSIGDURINGWAIT, JRTIMEOUT. |
| EINVAL | Incorrect parameter. Consult Reason_code to determine the exact reason the error occurred. The following reason codes can accompany the return code: JRBADOSI, JRBADPFSID. |
| EIO | The file system was unmounted while LFS serialization was dropped. |
| EMVSNOTUP | The system is being stopped. |

**Reason_code**

Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_sleep service stores the reason code. The osi_sleep service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value.

## Usage notes

1. For additional information, see “Waiting and posting” on page 21.
2. All LFS serialization is dropped during an osi_sleep and reestablished after the osi_wakeup.
3. Before calling osi_sleep, the PFS must copy the osi_pfsid value to a location that is addressable by the task that will call osi_wakeup. It must be passed as

the Pfs_id on osi_wakeup. The osi_pfsid value that is passed to the PFS is the same for all operations of this PFS. It is also passed as pfsi_pfsid during PFS initialization. This initialization value may be used on osi_wakeup instead of saving the OSI value.

4. The osi_wakeup service does not wake up a task that is not currently sleeping. If osi_wakeup is issued before osi_sleep for the same resource, the task sleeps until the next osi_wakeup for that resource. Therefore, the PFS must have sufficient logic and recovery to ensure that sleeping tasks are eventually awakened.
5. The address of the osi_sleep routine is passed to the PFS in the OSIT structure when the PFS is initialized.

## Related services

- “osi_wakeup — Wake up OSI sleepers” on page 435

## Characteristics and restrictions

1. This routine must be used only on the task that made the vnode or VFS call.
2. An osi_sleep is not permitted if an FRR is established.

# osi_thread — Fetch and call a module from a colony thread

## Function

The osi_thread service is used by a PFS to call a module on an asynchronous colony thread that is in the same colony address space that the PFS is running on. For a synchronous request, the caller's task is put into a wait while the module is running.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state, any PSW key |
| Dispatchable unit mode: | Task |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_thread(OSI_structure,
           OsitThread_Parm_structure,
           Return_value,
           Return_code,
           Reason_code);
```

## Parameters

**OSI_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Osilen field |

OSI_structure contains information that is used by the OSI operations. The PFS receives this structure on each PFS interface operation.

Refer to Appendix D for a full description of this structure.

**OsitThread_Parm_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Othdlen field |

An area that contains the OsitThread parameters. The entries in this area are mapped by the OTHDPRM typedef.

Refer to Appendix D for a full description of this structure. The following OsitThread parameters must be supplied:

| | |
|---|---|
| **ot_modname** | The name of the module that is to be fetched and called on the colony thread. The name must be a null-terminated string that is acceptable to the C fetch function. |

**ot_parms**
The address of the parameters that are to be passed to the module specified by ot_modname. This parameter is also passed to the named exit if it is called. If any parameters are passed, the first parameter is used by the LFS to pass a state token to the named module or exit routine. The area whose address is passed in ot_parms must reserve the first word for this purpose.

The address that is specified in this parameter points to a structure, or control block, in whose first word the LFS inserts the address of the 8-byte state token. A pointer containing ot_parms is the first parameter to the module and to the exit routine.

**ot_exitname**
The name of the exit routine that may be called after the module completes. This routine is called for a request that specifies NOWAIT, or when the caller's wait is terminated before the module completes. The name must be a null-terminated string that is acceptable to the C fetch function.

**ot_option_flags**
A field in which the caller can specify:

- OSI_SIGWAIT—the caller's task is put into a signal-enabled wait until the module that is named in ot_modname completes.
- OSI_NOWAIT—the caller's task is not put into a wait; the module is run asynchronously.

  If neither OSI_SIGWAIT nor OSI_NOWAIT is specified, the caller's task is placed in a wait that is not signal-enabled.
- OSI_RELEASEMODS—the fetched module and exit routine, if called, are released when the request is complete.

  When a module is released, any state token that is associated with this module on the current osi worker thread is freed.

  IF OSI_RELEASEMODS is not specified, the named module and the exit routine, if called, remain in storage. The next request that specifies these routines does not fetch them before calling them.

**Return_value**
Returned parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword in which the osi_thread service returns the results of the operation, as one of the following:

| **Return_value** | **Meaning** |
|---|---|

| | |
|---|---|
| **−1** | The operation was not successful. The resources that are associated with this request can be safely freed. |
| **0** | The operation was successful. The resources that are associated with this request can be safely freed. |
| **+1** | The named module was scheduled to be called, but may not have completed. Resources that are associated with this request should not be freed. This value is returned if the request specified OSI_NOWAIT, or if the caller's wait is terminated before the request completes. |

**Note:** The return value indicates the results of the osi_thread service. It does not indicate the results of the named module. Some other mechanism must be used by the caller to determine these results.

**Return_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword in which the osi_thread service stores the return code. The osi_thread service can return one of the following values in the Return_code parameter only if Return_value is +1 or −1. Reason_code further qualifies the Return_code value.

| Return_code | Explanation |
|---|---|
| EINTR | The service was interrupted by a signal. |
| EINVAL | Parameter error. Consult Reason_code to determine the exact reason the error occurred. The following reason codes can accompany the return code: JROWaitSetupErr, JRNoClnyThreadSuppt. |

**Reason_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

A fullword in which the osi_thread service stores the reason code. The osi_thread service returns Reason_code only if Return_value is +1 or −1. Reason_code further qualifies the Return_code value.

## Usage notes

1. The osi_thread service may be invoked only from a PFS that is running in a colony address space.
2. For more information, see "Using daemon tasks within a PFS" on page 41.
3. The osi_thread service is not available for use until kernel initialization is complete. The PFS can determine when kernel initialization is complete by interrogating the ot_available flag whose address is passed in the pfsi_otstatptr field.
4. The caller must not free any resources that may be used by the module that is running on the colony thread unless a return value of 0 or −1 is returned. If a return value of +1 is returned, the resources must be freed by the exit routine.

5. The osi_thread service undoes any Osi_Wait Setup that was done before this service was called.
6. The named module and the named exit routine are fetched on the colony thread using the C/370 **fetch()** function. The named module must comply with any requirements of this function. See *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821 for more information.
7. The named module, and the exit routine, if it is called, remain in storage after the request completes, unless OSI_RELEASEMODS was specified.
8. The named module and the exit routine may use C/RTL or POSIX services. The writer of the PFS should remember that this thread could be used to fetch and call the specified module on another osi_thread call. Therefore, the named module should not request any services that would affect the process that is associated with this thread, such as exit or exec. Pthread services should not be requested either.
9. The named module and the exit routine must be reentrant.
10. The named module and the exit routine are invoked using OS linkage conventions.
11. The named module and the exit routine receive control in the following environment:

| | |
|---|---|
| **Authorization:** | Problem state, PSW key 8 |
| **Dispatchable unit mode:** | Task |
| **Cross memory mode:** | PASN=HASN=SASN |
| **AMODE:** | 31-bit |
| **ASC mode:** | Primary |
| **Interrupt status:** | Enabled for interrupts |
| **Locks:** | Unlocked |
| **Signals:** | All signals are blocked except SIGALRM |

12. If any parameters are passed to the named module or exit routine, the first parameter should be the address of an 8-byte state token. The first time a named module or exit routine is invoked on a particular osi worker thread, this token is zeros. The named module or exit routine can modify this token to preserve some state information from one invocation to the next. For all subsequent invocations of this module on this particular worker thread, the token is provided unmodified by the LFS.

    When the PFS uses a parameter structure, the first word is used by the LFS to point to the state token. The input to the module and exit looks like this:

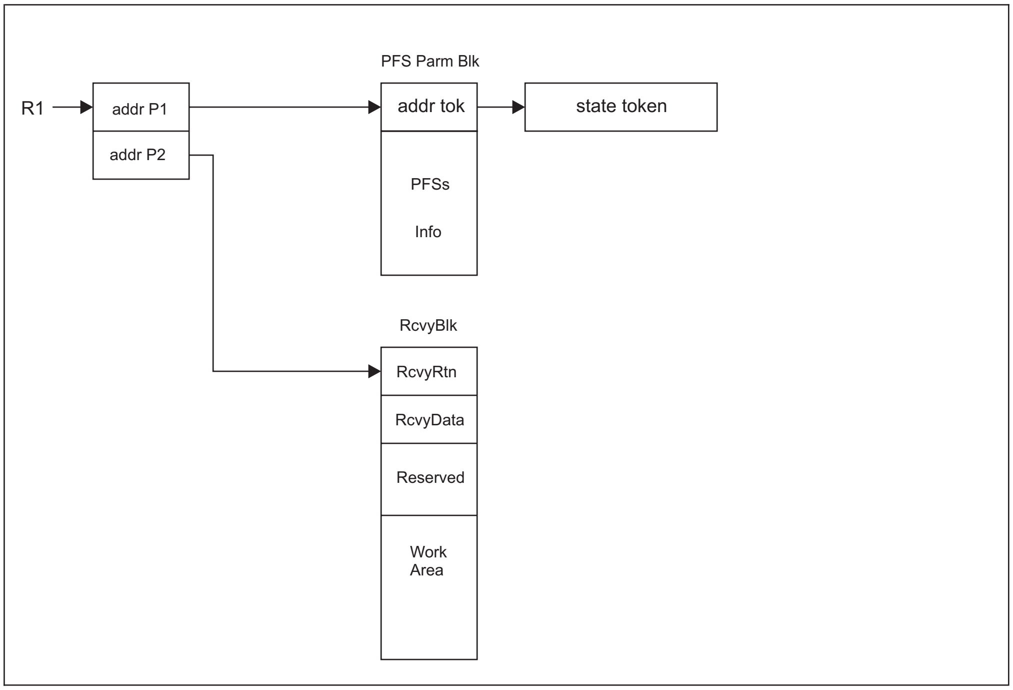


*Figure 7. Input to module and exit using a parameter structure*

When the PFS does not use a parameter structure, the input to the module and exit looks like this:

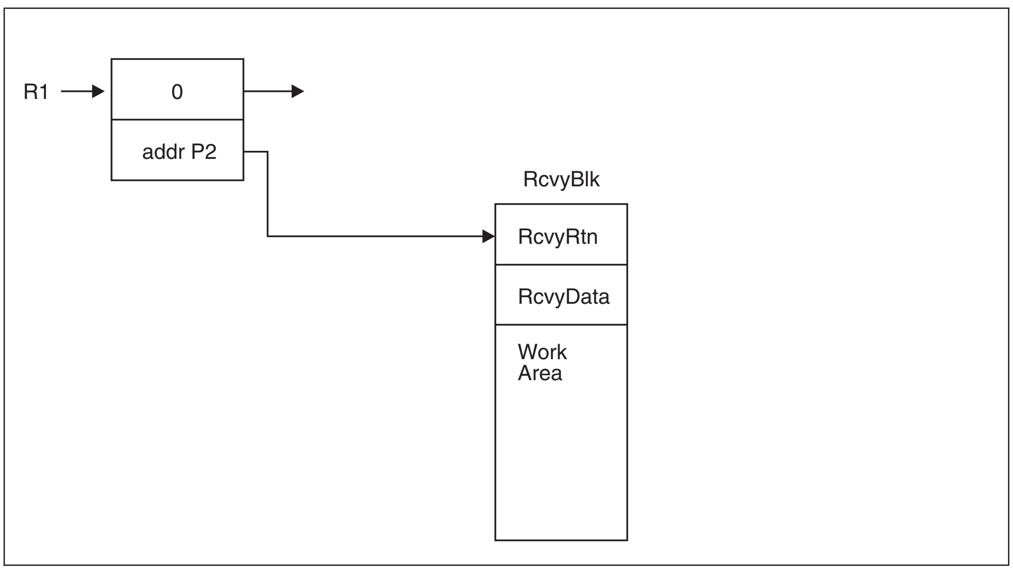


*Figure 8. Input to module and exit without using a parameter structure*

13. ESTAE-like recovery is available to the module and exit routines. This saves the overhead involved in having these routines set up and take down their own ESTAEs on each entry.

    A pointer to a Recovery Block (RcvyBlk above) is passed as the second parameter to these routines. The pointer is used as follows:

    a. On entry, or when recovery protection is needed, the module or exit sets the RcvyData pointer to the address of its own recovery information.

Pertinent data can also be placed in the work area. This data will be available to the recovery routine.

The RcvyRtn pointer is set to the address of a recovery routine.

b. If the module or exit ends abnormally, and RcvyRtn is non-zero, the RcvyRtn routine is called from the LFS's ESTAE and passed all the parameters provided by RTM, including the pointer to RcvyBlk. An exception is that register 15 contains the address of the recovery routine (RcvyRtn), rather than the address of the LFS's permanent ESTAE exit. When the RcvyRtn routine returns, it returns directly to RTM, rather than to the LFS's permanent ESTAE exit.

c. Under normal circumstances, before returning, or when recovery protection is no longer needed, the module or exit zeros out the RcvyRtn field.

The recovery routine is invoked in the same way as an MVS ESTAE routine, not a C subroutine. The registers on entry are:

| Register | Contents |
|---|---|
| **R0** | 12, if an SDWA is not provided; otherwise, an SDWA address is provided in R1. |
| **R1** | SDWA address, if an SDWA is provided; otherwise, completion code. |
| **R2** | Pointer to RcvyBlk, with or without an SDWA. This value is also contained in SDWAPARM when an SDWA is provided. |
| **R13** | Address of the save area provided by RTM. |
| **R14** | Return address, as provided by RTM. |
| **R15** | Address of the recovery routine that is being called (RcvyRtn). |

14. The recovery block (RcvyBlk) is mapped by OTHDCRCV in bpxypfsi.h.
15. The work area in the recovery block can be used to pass information to the recovery routine. It can also be used as a work area for the recovery routine to build dump titles or list forms of assembler macros.
16. The recovery routine is entered in problem program state, key 8.
17. The address of the osi_thread routine is passed to the PFS in the OSIT when the PFS is initialized.

## Characteristics and restrictions

This routine must be used only on the task that made the vnode or VFS call.

# osi_uiomove — Move data between PFS buffers and buffers defined by a UIO structure

## Function

The osi_uiomove service moves blocks of data between PFS buffers and buffers that are defined by a UIO structure.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state, PSW key 0 |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_uiomove(OSI_structure,
         Workarea,
         PFS_Buffer,
         PFS_Buffer_Alet,
         Number_of_bytes,
         User_IO_structure,
         Return_value,
         Return_code,
         Reason_code);
```

## Parameters

**OSI_structure**
Supplied parameter

| | |
|---|---|
| **Type:** | Structure |
| **Length:** | Specified by the Osilen field |

OSI_structure contains information that is used by the OSI operations. The PFS receives this structure on each PFS interface operation.

Refer to Appendix D for a full description of this structure.

**Workarea**
Supplied parameter

| | |
|---|---|
| **Type:** | Char |
| **Length:** | 2048 bytes |

Workarea is a buffer of 2048 bytes, aligned on a word boundary, that is to be used by this OSI operation.

**PFS_Buffer**
Supplied parameter

| | |
|---|---|
| **Type:** | Char |
| **Length:** | N/A |

The name of the buffer to or from which data is to be moved.

**PFS_Buffer_Alet**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The Alet for the specified PFS_Buffer.

**Number_of_bytes**
Supplied parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The number of bytes to move.

**User_IO_structure**
Supplied and returned parameter

| | |
|---|---|
| **Type:** | UIO |
| **Length:** | Specified by UIO.u_hdr.cblen. |

An area that contains the parameters for the I/O that is to be performed. This area is mapped by the UIO typedef in the BPXYVFSI header file (see Appendix D). See "Specific processing notes" on page 428 for details on how the fields in this structure are processed.

**Return_value**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_uiomove service returns the results of the service, as one of the following:

| **Return_value** | **Meaning** |
|---|---|
| **−1** | The operation was not successful. The Return_code and Reason_code parameters contain the values that are returned by the service. |
| **0 or greater** | The operation was successful. The value represents the number of bytes that were transferred. |

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_uiomove service stores the return code. The osi_uiomove service returns Return_code only if Return_value is −1. For a complete list of return codes, see *z/OS UNIX System Services Messages and Codes*.

**Reason_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

A fullword in which the osi_uiomove service stores the reason code. The osi_uiomove service returns Return_code only if Return_value is −1.

Reason_code further qualifies the Return_code value. The reason codes are described in *z/OS UNIX System Services Messages and Codes*.

## Usage notes

1. The osi_uiomove service moves the number of bytes of data that is specified by the Number_of_bytes parameter or the UIO.u_count field, whichever is less. If either of these parameters is zero, no data is moved, and Return_value field is set to 0.
2. The u_iovresidualcnt and u_totalbytesrw fields, described below, are not set until after the first call to osi_uiomove.
3. This service requires the calling program to be in key 0 storage, because it must update the UIO, and this structure is usually in key 0 storage. Osi_copyin and osi_copyout do not require the calling program to be in key 0 storage.
4. The address of the osi_uiomove routine is passed to the PFS in the OSIT structure when the PFS is initialized.
5. The OSI_structure contains an area, pointed to by osi_workarea, that may be passed to this service as the Workarea parameter.

### Specific processing notes

The following UIO fields are provided by the LFS:

**UIO.u_count**
Specifies the number of bytes in the buffer, or the number of elements in the IOV array.

**UIO.u_rw**
Specifies whether the request is a read (0) or a write (1). On a read, the contents of PFS_buffer are moved to Uiouserbuffer. On a write, the contents of Uiouserbuffer are moved to PFS_buffer.

**UIO.u_iovinuio**
Specifies whether the user_IO_structure points to an iov structure.

**UIO.u_realpage**
Specifies whether the user_IO_structure contains addresses of real pages. This flag must be OFF (0), or the osi_uiomove service fails the request.

**UIO.u_key**
Specifies the storage key of the caller's buffer.

**UIO.u_iovresidualcnt**
Specifies the number of bytes remaining in the buffer or iov structure that is pointed to by the user_IO_structure.

**u_totalbytesrw**
Specifies the total number of bytes that are to be moved.

## Related services

- "osi_copyin — Move data from a user buffer to a PFS buffer" on page 370
- "osi_copyout — Move data from a PFS buffer to a user buffer" on page 373
- "osi_copy64 — Move data between user and PFS buffers with 64-bit addresses" on page 376

## Characteristics and restrictions

1. This routine must be used only on the dispatchable unit (task or SRB) that made the vnode or VFS call because the service requires the use of the cross-memory environment of the calling dispatchable unit.
2. The osi_uiomove service does not support DATOFF moves; that is, it fails requests if the UIO.u_realpage flag is ON.

# osi_upda — Update async I/O request

## Function

The osi_upda service updates an asynchronous request with the PFS's request token.

## Requirements

| | |
|---|---|
| Authorization: | Supervisor state, key 0 |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_upda(Osi_AsyTok,
         PFS_AsyTok);
```

## Parameters

**Osi_AsyTok**

Supplied parameter

**Type:** Token

**Length:** 8 bytes

The name of the field that contains the osi_asytok value that was passed to the PFS on this vnode operation.

The field from the input osi itself may be used on this call.

**PFS_AsyTok**

Supplied parameter.

**Type:** Token

**Length:** 8 bytes

The name of the field containing the PFS's token for this asynchronous request. This value is saved by the LFS and passed back to the PFS on the second part of the asynchronous operation, or on vn_cancel.

## Usage notes

1. Refer to "Asynchronous I/O processing" on page 55 for details on asynchronous operations.
2. osi_upda is called by the PFS early in Part 1 of an asynchronous vnode operation. It must be called some time before there is any possibility that osi_sched will be called for an asynchronous completion of this I/O.

When an operation can be completed immediately, Osi_upda does not have to be called if osi_ok2compimd=ON, or if the PFS does not need to participate in Part 2.

3. On entry to Part 1, Osi_asytok contains the LFS's request token, and osi_upda is called so that the LFS can save the PFS's request token.

   Osi_asytok is also saved by the PFS during Part 1, and is used later for osi_sched.

4. Osi_asytok on entry to Part 2 contains this PFS_AsyTok value.

   It is important that osi_upda be called before osi_sched is called, when the PFS is participating in Part 2, because Part 2 could run anytime after osi_sched is called, and the LFS might not have the PFS's request token to pass.

5. This PFS_AsyTok value is also passed on vn_cancel to identify the request that is being canceled.

   Canceled requests do not generate a call to vn_cancel if osi_upda has not been called.

6. If the Osi_Asytok value is not valid, osi_upda issues an 0xEC6 abnormal end with a reason code of 0x11450727.

## Characteristics and restrictions

None.

# osi_wait — Wait for an event to occur

## Function

The osi_wait service waits for a signal to occur or for osi_post to be called.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state, any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_wait(Entry_code,
         OSI_structure,
         Return_code,
        [Wait_Flags,
         Time_interval]);
```

## Parameters

**Entry_code**

Supplied parameter

**Type:** Integer

**Length:** Fullword

The Entry_code specifies the function that is being requested for the osi_wait service.

| Entry_code | Explanation |
|---|---|
| OSI_SETUP | Set up for a subsequent wait request. |
| OSI_SETUPSIG | Set up for a subsequent wait request with signals enabled. |
| OSI_SUSPEND | Wait to be posted from osi_post. |
| OSI_WAITX | Wait to be posted from osi_post or for a timer to expire. |
| OSI_INIT2 | Initialize for use by an independent task. See Usage Notes. |

**OSI_structure**

Supplied and returned parameter

**Type:** OSI

**Length:** Specified by OSI.osi_hdr.cblen.

OSI_structure contains information that is used by the OSI operations. The PFS receives this structure on each PFS interface operation.

Refer to Appendix D for a full description of this structure.

**Return_code**
Returned parameter

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

The name of a fullword in which the osi_wait service stores the return code. The osi_wait service can return one of the following values in the Return_code parameter:

| Return_code | Explanation |
|---|---|
| OSI_POSTED | Successful completion. |
| OSI_SIGNALRCV | A signal arrived. |
| OSI_SHUTDOWN | The system is being stopped. |
| OSI_TIMEOUT | A specified time interval expired before a post or signal. |
| OSI_UNMOUNTED | The file system was unmounted while LFS serialization was dropped. |
| OSI_POSTERTRM | The address space that is responsible for doing the osi_post has terminated. |
| OSI_BADPARM | Incorrect OSI_structure. |
| OSI_ESTAEF | Unable to establish a recovery environment. |
| OSI_ABEND | Abnormal end in osi_wait. |
| OSI_SYSTEMERR | Unable to release latches before a signal wait. |

**Wait_Flags**
Supplied parameter (only when Entry_code is OSI_WAITX)

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Fullword |

Wait_flags contains flags that specify options on the wait request.

| Flag | Explanation |
|---|---|
| osi_wtdroplocks | Drop LFS serialization during the wait, and reestablish it after the wait. |

Refer to Appendix D for a full description of this structure.

**Time_interval**
Supplied parameter (only when Entry_code is OSI_WAITX)

| | |
|---|---|
| **Type:** | Integer |
| **Length:** | Doubleword |

The Time_interval is the time for which osi_wait will wait. The value is specified in timer units. If the high-order word is non-zero, the 8-byte value is rounded to approximate seconds. See *z/Architecture Principles of Operation* for more detail on timer units. The value is added to the current time; therefore a very large time interval added to the current time could wrap to a very small number and result in an immediate timeout of osi_wait. A value of 0 indicates there is no time limit.

## Usage notes

1. For additional information, see "Waiting and posting" on page 21.
2. The PFS must call osi_wait for setup before making the call to do the wait and before OSI_post is called to wake up the task. On the setup call, Entry_code specifies whether the PFS wants the wait to be terminated if the process receives a signal.

The order of the calls to wait and to OSI_post is not important after the setup call has been made.

3. If Entry_code is OSI_SUSPEND and a signal-enabled wait was set up, all LFS serialization is dropped during the wait and reestablished after the wait.

   If Entry_code is OSI_WAITX, the Wait_flags specify whether LFS serialization is dropped during the wait and reestablished after the wait.

   For writes on stream sockets, the default socket option of exclusive write will prevent the dropping of LFS serialization during signal-enabled waits.

4. Between the calls to setup and suspend, the PFS should make sure that the OSI token that is returned by setup is addressable to the program that will eventually call OSI_post. The PFS can copy the OSI token. If only the address is used, be careful using this OSI, because the storage for a task will be freed if the task terminates.

5. The PFS must never call OSI_post for a waiting task more than once, and should have sufficient logic and recovery to avoid calling OSI_post for a task that is no longer waiting.

6. The osi_thread service undoes any osi_wait setup that was done before osi_thread was called.

7. Osi_wait issues an MVS WAIT or SUSPEND, respectively, as appropriate for TCB or SRB mode callers. Osi_post invokes the corresponding MVS service to wake up osi_waiters.

8. When osi_wait is called from an SRB, OSI_SETUPSIG may be requested, but signals are not really enabled. This is because signals are not delivered to SRBs, therefore the wait is not interrupted by a signal.

   Using OSI_SETUPSIG allows z/OS UNIX to interrupt an SRB's wait if the associated user process goes into termination. It is awakened as if a signal had been delivered.

9. The OSI_INIT2 Entry_code is used to initialize an OSI_Structure for use by an independent task (TCB) in an address space that is associated with the PFS. This allows the task to wait with osi_wait and be posted with osi_post. An independent task is one that was attached in that address space; it is not running from within a vnode operation.

   **Note:** Generally an independent task would use MVS WAIT, and be posted by MVS POST. Osi_wait and osi_post take several hundred more instructions to execute than MVS WAIT/POST.

   There are several restrictions on this service:

   a. Only tasks (TCBs) are supported, not SRBs.

   b. The task must already be dubbed a z/OS UNIX thread. If this is not the case, the task can get dubbed by calling a z/OS UNIX service such as getpid() before calling osi_wait for OSI_INIT2.

   c. The only osi service that is expected to be used by this task with the OSI_structure returned is osi_wait.

   Osi_wait(OSI_INIT2) needs to be called only once for the life of a TCB.

   The storage for the OSI_structure is provided by the caller as input, and osi_wait(OSI_INIT2) initializes this area for use on subsequent calls for setup and suspension. This storage belongs to the caller, and is freed by the caller, usually at task termination. Calls for setup and suspension may be made with a copy of the structure that is built from this call.

   Only the task that made the OSI_INIT2 call can use this OSI_structure.

The OSI_structure must be initialized with the length of the area that is being passed before osi_wait(OSI_INIT2) is called. For example, `osi.osi_hdr.cblen=sizeof(OSI)`.

10. The address of the osi_wait routine is passed to the PFS in the OSIT structure when the PFS is initialized. Calls that are made from independent address spaces require their own loaded OSIT structure. Refer to “Using OSI services from a non-kernel address space” on page 368 for details.

## Related services

- “osi_post — Post an OSI waiter” on page 408

## Characteristics and restrictions

Calls that are made with the OSI_structure that was passed to the PFS on a vnode or VFS operation must be made only on the task that made the vnode or VFS call.

# osi_wakeup — Wake up OSI sleepers

## Function

The osi_wakeup service wakes up all threads that are sleeping in osi_sleep with a matching Resource_id and Pfs_id.

## Requirements

| | |
|---|---|
| Authorization: | Problem or supervisor state, any PSW key |
| Dispatchable unit mode: | Task or SRB |
| Cross memory mode: | Any |
| AMODE: | 31-bit |
| ASC mode: | Any |
| Interrupt status: | Enabled for interrupts |
| Locks: | Unlocked |
| Control parameters: | All parameters must be addressable by the caller and in the primary address space. |

## Format

```
osi_wakeup(Resource_id,
           Pfs_id,
           Return_value,
           Return_code,
           Reason_code);
```

## Parameters

**Resource_id**

Supplied parameter

**Type:** Token

**Length:** Fullword

The Resource_id identifies the resource that is available. All osi_sleep services that are waiting for this Resource_id are to return to their callers.

**Pfs_id**

Supplied parameter

**Type:** Token

**Length:** Fullword

The Pfs_id identifies the calling PFS. The PFS receives its unique identifier from the LFS in the osi_pfsid field of the OSI structure on each VFS and vnode operation. This identifier is also passed as pfsi_pfsid during PFS initialization, and the initialization value may be used instead of the OSI value that is saved from osi_sleep.

**Return_value**

Returned parameter

**Type:** Integer

**Length:** Fullword

The name of a fullword in which the osi_wakeup service returns the results of the operation, as one of the following:

| Return_value | Meaning |
|---|---|
| **−1** | The operation was not successful. |
| **0 or greater** | The operation was successful; the value represents the number of sleeping tasks that were awakened. |

**Return_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

The name of a fullword in which the osi_wakeup service stores the return code. The osi_wakeup service returns Return_code only if Return_value is −1. Reason_code further qualifies Return_code.

**Reason_code**
Returned parameter
**Type:** Integer
**Length:** Fullword

A fullword in which the osi_wakeup service stores the reason code. The osi_wakeup service returns Reason_code only if Return_value is −1. Reason_code further qualifies the Return_code value.

## Usage notes

1. For additional information, see “Waiting and posting” on page 21.
2. Before calling osi_sleep, the PFS must copy the osi_pfsid value to a location that is addressable by the task that is to call osi_wakeup. It must be passed as the Pfs_id on osi_wakeup. The osi_pfsid value that is passed to the PFS is the same for all operations of this PFS. It is also passed as pfsi_pfsid during PFS initialization. This initialization value may be used on osi_wakeup instead of the OSI value that is saved from osi_sleep.
3. The osi_wakeup service does not wake up a task that is not currently sleeping. If osi_wakeup is issued before osi_sleep for the same resource, the task sleeps until the next osi_wakeup for that resource. Therefore, the PFS must have sufficient logic and recovery to ensure that sleeping tasks will eventually be awakened.
4. The address of the osi_wakeup routine is passed to the PFS in the OSIT structure when the PFS is initialized.

## Related services

- “osi_sleep — Sleep until a resource is available” on page 417

## Characteristics and restrictions

The caller of this service must be on a process thread.

# Appendix A. System control offsets to callable services

An alternative to loading or link-editing the service stub is to include in the code the system control offset to the callable service. For example, use decimal 52 for the offset of access (BPX1ACC).

When using the offsets, set the registers up as follows:

**Register 1**
: To contain the address of your parameter list. Set bit `0` of the last address in the list on.

**Register 14**
: To contain the return address in the invoking module.

**Register 15**
: To contain the address of the callable service code.

## Example

The following is an example of code that specifies the offset. The example assumes that register 1 is set up with the address of the parameter list. Replace *offset* with the appropriate value from the following offset table.

```
L     15,16           CVT - common vector table
L     15,544(15)      CSRTABLE
L     15,24(15)       CSR slot
L     15,offset(15)   Address of the service
BALR  14,15           Branch and link
```

## List of offsets

*Table 9. System control offsets to callable services*

| Service | Offset | Function |
|---|---|---|
| BPX1ACC | 52 | access |
| BPX1ACK | 972 | auth_check_rsrc_np |
| BPX1ACP | 508 | accept |
| BPX1AIO | 988 | asyncio |
| BPX1ALR | 224 | alarm |
| BPX1ANR | 1060 | accept_and_recv |
| BPX1ASP | 1088 | aio_suspend |
| BPX1ATM | 668 | attach_execmvs |
| BPX1ATX | 664 | attach_exec |
| BPX1BND | 512 | bind |
| BPX1CCA | 480 | cond_cancel |
| BPX1CCS | 1012 | console_np |
| BPX1CHA | 84 | chaudit |
| BPX1CHD | 56 | chdir |
| BPX1CHM | 60 | chmod |
| BPX1CHO | 64 | chown |
| BPX1CHP | 764 | chpriority |
| BPX1CHR | 500 | chattr |
| BPX1CID | 968 | convert_id_np |
| BPX1CLD | 68 | closedir |
| BPX1CLO | 72 | close |
| BPX1CON | 516 | connect |
| BPX1CPL | 1132 | __cpl |
| BPX1CPO | 484 | cond_post |
| BPX1CRT | 872 | chroot |
| BPX1CSE | 488 | cond_setup |

© Copyright IBM Corp. 1996, 2006

*Table 9. System control offsets to callable services (continued)*

| Service | Offset | Function |
|---|---|---|
| BPX1CTW | 492 | cond_timed_wait |
| BPX1CWA | 496 | cond_wait |
| BPX1DEL | 888 | deletehfs |
| BPX1DSD | 1124 | sw_signaldelv |
| BPX1ENV | 960 | oe_env_np |
| BPX1EXC | 228 | exec |
| BPX1EXI | 232 | _exit |
| BPX1EXM | 236 | execmvs |
| BPX1EXT | 200 | extlink_np |
| BPX1FAI | 1168 | FreeAddrInfo |
| BPX1FCA | 140 | fchaudit |
| BPX1FCD | 852 | fchdir |
| BPX1FCM | 88 | fchmod |
| BPX1FCO | 92 | fchown |
| BPX1FCR | 504 | fchattr |
| BPX1FCT | 96 | fcntl |
| BPX1FPC | 100 | fpathconf |
| BPX1FRK | 240 | fork |
| BPX1FST | 104 | fstat |
| BPX1FSY | 108 | fsync |
| BPX1FTR | 112 | ftruncate |
| BPX1FTV | 848 | FstatVfs |
| BPX1GAI | 1164 | GetAddrInfo |
| BPX1GCL | 1024 | getclientid |
| BPX1GCW | 116 | getcwd |
| BPX1GEG | 244 | getegid |
| BPX1GEP | 860 | getpgid |
| BPX1GES | 864 | getsid |
| BPX1GET | 736 | w_getipc |
| BPX1GEU | 248 | geteuid |
| BPX1GGE | 772 | getgrent |
| BPX1GGI | 252 | getgrgid |
| BPX1GGN | 256 | getgrnam |
| BPX1GGR | 260 | getgroups |
| BPX1GHA | 1160 | gethostbyaddr |
| BPX1GHN | 1156 | gethostbyname |
| BPX1GID | 264 | getgid |
| BPX1GIV | 1028 | givesocket |
| BPX1GLG | 268 | getlogin |
| BPX1GMN | 76 | w_getmntent |
| BPX1GNI | 1172 | GetNameInfo |
| BPX1GNM | 524 | getpeername |
| BPX1GPE | 776 | getpwent |
| BPX1GPG | 272 | getpgrp |
| BPX1GPI | 276 | getpid |
| BPX1GPN | 280 | getpwnam |
| BPX1GPP | 284 | getppid |
| BPX1GPS | 428 | w_getpsent |
| BPX1GPT | 916 | grantpt |
| BPX1GPU | 288 | getpwuid |
| BPX1GPY | 744 | getpriority |
| BPX1GRL | 820 | getrlimit |
| BPX1GRU | 824 | getrusage |
| BPX1GTH | 1056 | __getthent |

*Table 9. System control offsets to callable services (continued)*

| Service | Offset | Function |
|---|---|---|
| BPX1GTR | 752 | getitimer |
| BPX1GUG | 292 | getugrps |
| BPX1GUI | 296 | getuid |
| BPX1GWD | 936 | getwd |
| BPX1HST | 520 | gethostid |
| BPX1IOC | 120 | w_ioctl |
| BPX1IPT | 396 | MvsIptAffinity |
| BPX1ITY | 12 | isatty |
| BPX1KIL | 308 | kill |
| BPX1LCO | 832 | lchown |
| BPX1LCR | 1180 | lchattr |
| BPX1LNK | 124 | link |
| BPX1LOD | 880 | loadhfs |
| BPX1LSK | 128 | lseek |
| BPX1LSN | 532 | listen |
| BPX1LST | 132 | lstat |
| BPX1MAT | 720 | shmat |
| BPX1MCT | 724 | shmctl |
| BPX1MDT | 728 | shmdt |
| BPX1MGT | 732 | shmget |
| BPX1MKD | 136 | mkdir |
| BPX1MKN | 144 | mknod |
| BPX1MMI | 1136 | __map_init |
| BPX1MMP | 796 | mmap |
| BPX1MMS | 1140 | __map_service |
| BPX1MNT | 148 | mount |
| BPX1MP | 688 | MVSpause |
| BPX1MPC | 408 | mvsprocclp |
| BPX1MPI | 680 | MVSpauseInit |
| BPX1MPR | 800 | mprotect |
| BPX1MSD | 336 | mvsunsigsetup |
| BPX1MSS | 312 | mvssigsetup |
| BPX1MSY | 804 | msync |
| BPX1MUN | 808 | munmap |
| BPX1NIC | 748 | nice |
| BPX1OPD | 152 | opendir |
| BPX1OPN | 156 | open |
| BPX1OPT | 528 | getsockopt |
| BPX1OSE | 1100 | __osenv |
| BPX1PAF | 1072 | __pid_affinity |
| BPX1PAS | 316 | pause |
| BPX1PCF | 160 | pathconf |
| BPX1PCT | 768 | pfsctl |
| BPX1PIO | 984 | w_pioctl |
| BPX1PIP | 164 | pipe |
| BPX1POE | 1176 | __poe |
| BPX1POL | 932 | poll |
| BPX1PQG | 1152 | Pthread_quiesce_and_get_np |
| BPX1PSI | 460 | pthread_setintr |
| BPX1PST | 472 | Pthread_setintrtype |
| BPX1PTB | 448 | pthread_cancel |
| BPX1PTC | 432 | pthread_create |
| BPX1PTD | 444 | pthread_detach |
| BPX1PTI | 476 | Pthread_testintr |

*Table 9. System control offsets to callable services (continued)*

| Service | Offset | Function |
|---|---|---|
| BPX1PTJ | 440 | pthread_join |
| BPX1PTK | 464 | pthread_kill |
| BPX1PTQ | 412 | pthread_quiesc |
| BPX1PTR | 320 | ptrace |
| BPX1PTS | 452 | pthread_self |
| BPX1PTT | 1016 | pthread_tag_np |
| BPX1PTX | 436 | pthread_xandg |
| BPX1PWD | 788 | password |
| BPX1QCT | 692 | msgctl |
| BPX1QDB | 948 | querydub |
| BPX1QGT | 696 | msgget |
| BPX1QRC | 700 | msgrcv |
| BPX1QSE | 388 | quiesce |
| BPX1QSN | 704 | msgsnd |
| BPX1RCV | 540 | recv |
| BPX1RDD | 168 | readdir |
| BPX1RDL | 172 | readlink |
| BPX1RDV | 536 | readv |
| BPX1RDX | 940 | read_extlink |
| BPX1RD2 | 856 | readdir2 |
| BPX1RED | 176 | read |
| BPX1REN | 180 | rename |
| BPX1RFM | 544 | recvfrom |
| BPX1RMD | 188 | rmdir |
| BPX1RMG | 8 | resource |
| BPX1RMS | 548 | recvmsg |
| BPX1RPH | 884 | realpath |
| BPX1RW | 1108 | Pread |
| BPX1RWD | 184 | rewinddir |
| BPX1SA2 | 1084 | __Sigactionset |
| BPX1SCT | 708 | semctl |
| BPX1SDD | 300 | setdubdefault |
| BPX1SEC | 1044 | __security |
| BPX1SEG | 424 | setegid |
| BPX1SEL | 552 | select |
| BPX1SEU | 420 | seteuid |
| BPX1SF | 1064 | send_file |
| BPX1SGE | 780 | setgrent |
| BPX1SGI | 328 | setgid |
| BPX1SGQ | 1104 | sigqueue |
| BPX1SGR | 792 | setgroups |
| BPX1SGT | 712 | semget |
| BPX1SHT | 572 | shutdown |
| BPX1SIA | 324 | sigaction |
| BPX1SIN | 1004 | server_init |
| BPX1SIP | 340 | sigpending |
| BPX1SLK | 1068 | __shm_lock |
| BPX1SLP | 344 | sleep |
| BPX1SMC | 1112 | __smc |
| BPX1SMF | 1036 | __smf_record |
| BPX1SMS | 560 | sendmsg |
| BPX1SND | 556 | send |
| BPX1SOC | 576 | socket_pair |
| BPX1SOP | 716 | semop |

*Table 9. System control offsets to callable services (continued)*

| Service | Offset | Function |
|---|---|---|
| BPX1SPB | 416 | sigputback |
| BPX1SPE | 784 | setpwent |
| BPX1SPG | 348 | setpgid |
| BPX1SPM | 352 | sigprocmask |
| BPX1SPN | 760 | spawn |
| BPX1SPR | 568 | setpeer |
| BPX1SPW | 1008 | server_pwu |
| BPX1SPY | 740 | setpriority |
| BPX1SRG | 896 | setregid |
| BPX1SRL | 816 | setrlimit |
| BPX1SRU | 892 | setreuid |
| BPX1SRX | 1080 | srx_np |
| BPX1SSI | 356 | setsid |
| BPX1SSU | 360 | sigsuspend |
| BPX1STA | 192 | stat |
| BPX1STE | 1076 | Set_Timer_Event |
| BPX1STF | 80 | w_statfs |
| BPX1STL | 684 | Set_limits |
| BPX1STO | 564 | sendto |
| BPX1STQ | 1144 | server_thread_query |
| BPX1STR | 756 | setitimer |
| BPX1STV | 844 | StatVfs |
| BPX1STW | 1096 | sigtimedwait |
| BPX1SUI | 364 | setuid |
| BPX1SWT | 468 | sigwait |
| BPX1SYC | 368 | sysconf |
| BPX1SYM | 196 | symlink |
| BPX1SYN | 868 | sync |
| BPX1TAF | 1148 | MvsThreadAffinity |
| BPX1TAK | 1032 | takesocket |
| BPX1TDR | 24 | tcdrain |
| BPX1TFH | 20 | tcflush |
| BPX1TFW | 28 | tcflow |
| BPX1TGA | 32 | tcgetattr |
| BPX1TGC | 900 | tcgetcp |
| BPX1TGP | 36 | tcgetpgrp |
| BPX1TGS | 912 | tcgetsid |
| BPX1TIM | 372 | times |
| BPX1TLS | 964 | pthread_security_np |
| BPX1TRU | 828 | truncate |
| BPX1TSA | 40 | tcsetattr |
| BPX1TSB | 44 | tcsendbreak |
| BPX1TSC | 904 | tcsetcp |
| BPX1TSP | 48 | tcsetpgrp |
| BPX1TST | 908 | tcsettables |
| BPX1TYN | 16 | ttyname |
| BPX1UMK | 204 | umask |
| BPX1UMT | 208 | umount |
| BPX1UNA | 376 | uname |
| BPX1UNL | 212 | unlink |
| BPX1UPT | 920 | unlockpt |
| BPX1UQS | 392 | unquiesce |
| BPX1UTI | 216 | utime |
| BPX1VAC | 944 | v_access |

*Table 9. System control offsets to callable services (continued)*

| Service | Offset | Function |
| --- | --- | --- |
| BPX1VCL | 1188 | v_close |
| BPX1VCR | 620 | v_create |
| BPX1VEX | 876 | v_export |
| BPX1VGA | 632 | v_getattr |
| BPX1VGT | 596 | v_get |
| BPX1VLK | 604 | v_lookup |
| BPX1VLN | 640 | v_link |
| BPX1VLO | 660 | v_lockctl |
| BPX1VMD | 624 | v_mkdir |
| BPX1VOP | 1184 | v_open |
| BPX1VPC | 1040 | v_pathconf |
| BPX1VRA | 616 | v_readlink |
| BPX1VRD | 612 | v_readdir |
| BPX1VRE | 644 | v_rmdir |
| BPX1VRG | 584 | v_reg |
| BPX1VRL | 600 | v_rel |
| BPX1VRM | 648 | v_remove |
| BPX1VRN | 652 | v_rename |
| BPX1VRP | 588 | v_rpn |
| BPX1VRW | 608 | v_rdwr |
| BPX1VSA | 636 | v_settatr |
| BPX1VSF | 656 | v_fstatfs |
| BPX1VSY | 628 | v_symlink |
| BPX1WAT | 380 | wait |
| BPX1WLM | 1048 | __wlm |
| BPX1WRT | 220 | write |
| BPX1WRV | 580 | writev |
| BPX1WTE | 840 | waitid/wait3 |
| BPX2ITY | 928 | isatty2 |
| BPX2MNT | 1128 | __mount |
| BPX2OPN | 1052 | openstat |
| BPX2RMS | 976 | recvmsg2 |
| BPX2SMS | 980 | sendmsg2 |
| BPX2TYN | 924 | ttyname2 |

# Appendix B. Mapping macros

Mapping macros map the parameter options in many callable services. The fields with the comment “Reserved for IBM use” are not programming interfaces. A complete list of the options for each macro is listed in the macro in “Macros mapping parameter options” on page 444.

Most of the mapping macros can be expanded with or without a *DSECT* statement. The invocation operand *DSECT=YES* (default) can be used with either reentrant or nonreentrant programs with the appropriate rules governing the storage backed by the *USING*.

Many of the mapping macros exploit the fact that *DC* expands as a *DS* in a *DSECT* and as a *DC* with its initialized value in a *CSECT*. When these fields are expanded as or within DSECTs, the program is responsible for initializing the necessary fields.

© Copyright IBM Corp. 1996, 2006

## Macros mapping parameter options

Specifying DSECT=YES (the default for all macros) creates a DSECT. Addressability requires a USING and a register pointing to storage.

Specifying DSECT=NO (exceptions are listed when this is not allowed) allocates space in the current DSECT or CSECT. In reentrant programs, programmers can place these macros in the DSECT with DSECT=NO, and addressability is accomplished without the individual USING required by DSECT=YES. Nonreentrant programs can place their macros in the program's CSECT, and addressability is obtained through the program base register(s).

Specifying LIST=YES (the default for most macros) causes the expansion of the macro to appear in the listing. You can override this by using PRINT OFF.

Specifying LIST=NO removes the macro expansion from the listing.

Additional keywords VARLEN and PREFIX are described in the individual sections where they apply.

## BPXYATTR — Map file attributes for v_ system calls

```
 **  BPXYATTR:  File attributes for callable services                01626000
**  Used By:  VRP VLK VRW VCR VMD VSY VGA VSA                        01627000
         AIF   ('&DSECT;' EQ 'NO').B411                               01628000
ATTR                 DSECT ,                                         01629000
         AGO   .C411                                                 01630000
.B411    ANOP  ,                                                     01640000
                    DS    0D                 Clear storage           01650000
ATTR                DC    XL(ATTR#LENGTH)'00'                        01660000
                    ORG   ATTR                                       01670000
.C411    ANOP  ,                                                     01680000
ATTRBEGIN           DS    0D                                         01690000
*                                                                    01700000
ATTRHDR             DS    0D     ATTR Header                         01710000
ATTRID              DC    C'ATTR'                                   *01720000
                                 Eye Catcher                         01730000
ATTRSP              DC    AL1(ATTR#SP)                              *01740000
                                 Subpool number of this ATTR         01750000
ATTRLEN             DC    AL3(ATTR#LENGTH)                          *01760000
                                 Length of this Attr                 01770000
ATTRSTAT            DS    0D     stat() structure                    01780000
ATTRMODE            DS    0F     File Mode mapped by BPXYMODE        01790000
ATTRTYPE            DS    AL1    First byte of mode is file type,   *01800000
                                 mapped by BPXYFTYP                  01810000
ATTRREMMODE         DS    AL3    Name to know the last 3 byte        01820000
ATTRINO             DS    F      File Serial Number                  01830000
ATTRDEV             DS    F      Device ID of the file               01840000
ATTRLINK            DS    F      Number of links                     01850000
ATTRUID             DS    F      User ID of owner of the file        01860000
ATTRGID             DS    F      Group ID of Group of file           01870000
ATTRSIZE            DS    0D     File Size in bytes, for            *01880000
                                 regular file.  This is             *01890000
                                 unspecified for others.             01900000
ATTRSIZE_H          DS    F      First word of size                  01910000
ATTRSIZE_L          DS    F      Second word of size                 01920000
ATTRATIME           DS    F      Time of last access                 01930000
ATTRMTIME           DS    F      Time of last data mod               01940000
ATTRCTIME           DS    F      Time of last file stat chng         01950000
ATTRMAJORNUMBER     DS    F      Major number for this file,        *01960000
                                 if it is a character               *01970000
                                 special file.                       01980000
ATTRMINORNUMBER     DS    F      Minor number for this file,        *01990000
                                 if it is a character               *02000000
                                 special file.                       02010000
ATTRSTAT2           DS    0F     second part of the stat             02020000
ATTRAUDITORAUDIT    DS    0F     Area for auditor audit info         02030000
ATTRAUDITORAUDIT1   DS    XL1     Auditor audit byte 1               02040000
ATTRAUDITORAUDIT2   DS    XL1     Auditor audit byte 2               02050000
ATTRAUDITORAUDIT3   DS    XL1     Auditor audit byte 3               02060000
ATTRAUDITORAUDIT4   DS    XL1     Auditor audit byte 4               02070000
ATTRAAUDIT          EQU   X'01'    ON = auditor audit info          *02080000
                                        change request              *02090000
                                        (ON when AttrMAAudit = ON)   02100000
ATTRUSERAUDIT       DS    0F     Area for user audit info            02110000
ATTRUSERAUDIT1      DS    XL1     User audit byte 1                  02120000
ATTRUSERAUDIT2      DS    XL1     User audit byte 2                  02130000
ATTRUSERAUDIT3      DS    XL1     User audit byte 3                  02140000
ATTRUSERAUDIT4      DS    XL1     User audit byte 4                  02150000
ATTRNOTAAUDIT       EQU   X'01'    Always OFF to indicate           *02160000
                                          this is NOT auditor audit *02170000
                                          info                       02180000
ATTRBLKSIZE         DS    F      File Block Size                     02190000
ATTRCREATETIME      DS    F      File Creation Time                  02200000
ATTRAUDITID         DS    CL16   RACF File ID for auditing           02210000
*                                                                    02220000
ATTRGUARDTIME       ORG   ATTRAUDITID   Guard Time              @D7A 02230000
```

```
ATTRGUARDTIMESEC     DS    F               Seconds              @D7A 02240000
ATTRGUARDTIMEMSEC    DS    F               Micro_Seconds        @D7A 02250000
                     ORG   ATTRAUDITID                          @D7A 02260000
ATTRCVER             DS    CL8           Creation Verifier      @D7A 02270000
                     DS    CL8           Spacer                 @D7A 02280000
*                                                                    02290000
ATTRRES01            DS    F      Reserved                           02300000
ATTRGENMASK          DS    0F     Mask to indicate which            *02310000
                                  General attributes bit to         *02320000
                                  modify                            *02330000
                                  --Masks AttrGenValue               02340000
ATTROPAQUEMASK       DS    XL3    Opaque attribute flags -          *02350000
                                  Reserved for ADSTAR use            02360000
ATTRVISIBLEMASK      DS    XL1    Visible attribute flags            02370000
ATTRNODELFILESMASK   EQU   X'20'  Files should not be deleted   @P1A 02380000
ATTRSHARELIBMASK     EQU   X'10'  Shared Library                @D6A 02390000
ATTRNOSHAREASMASK    EQU   X'08'  No shareas flag               @D8A 02400000
ATTRAPFAUTHMASK      EQU   X'04'  APF authorized flag           @D6A 02410000
ATTRPROGCTLMASK      EQU   X'02'  Program controlled flag       @D6A 02420000
ATTREXTLINKMASK      EQU   X'01'  External Symlink flag             *02430000
                                  Mask bit not used on              *02440000
                                  vn_setattr                         02450000
ATTRSETFLAGS         DS    0XL4   Flags - which fields to set        02460000
ATTRSETFLAGS1        DS    XL1    Flag byte 1                        02470000
ATTRMODECHG          EQU   X'80'  Change to the mode indicated       02480000
ATTROWNERCHG         EQU   X'40'  Change to Owner indicated          02490000
ATTRSETGEN           EQU   X'20'  Set General attributes             02500000
ATTRTRUNC            EQU   X'10'  Truncate size                      02510000
ATTRATIMECHG         EQU   X'08'  Change the Atime                   02520000
ATTRATIMETOD         EQU   X'04'  Change to the Current Time         02530000
ATTRMTIMECHG         EQU   X'02'  Change the Mtime                   02540000
ATTRMTIMETOD         EQU   X'01'  Change to the Current Time         02550000
ATTRSETFLAGS2        DS    XL1    Flag byte 2                        02560000
ATTRMAAUDIT          EQU   X'80'  Modify auditor audit info          02570000
ATTRMUAUDIT          EQU   X'40'  Modify user audit info             02580000
ATTRCTIMECHG         EQU   X'20'  Change the Ctime                   02590000
ATTRCTIMETOD         EQU   X'10'  Change Ctime to the Current       *02600000
                                  Time                               02610000
ATTRREFTIMECHG       EQU   X'08'  Change the RefTime                 02620000
ATTRREFTIMETOD       EQU   X'04'  Change RefTime to Current Time     02630000
ATTRFILEFMTCHG       EQU   X'02'  Change File Format            @D5A 02640000
ATTRGUARDTIMECHK     EQU   X'01'  Guard Time Check Requested    @D7A 02650000
ATTRSETFLAGS3        DS    XL1    Flag byte 3 - reserved             02660000
ATTRCVERSET          EQU   X'80'  Creation Verifier Set         @D7A 02670000
ATTRCHARSETIDCHG     EQU   X'40'  CharSetId Change              @D9A 02680000
ATTRSETFLAGS4        DS    XL1    Flag byte 4 - reserved            *02690000
                                                                     02700000
ATTRSTAT3            DS    0F     Third part of the stat             02710000
ATTRCHARSETID        DS    CL12   Coded Character set id             02720000
                     ORG   ATTRCHARSETID                        @D9A 02730000
ATTRFILETAG          DS    CL4       File Tag                   @D9A 02740000
                     DS    CL8       Reserved                   @D9A 02750000
ATTRBLOCKS_D         DS    0F     Double word num blocks             02760000
ATTRBLOCKS_H         DS    F      First word of blocks               02770000
ATTRBLOCKS           DS    F      Number of blocks allocated         02780000
ATTRGENVALUE         DS    0F     General attribute values          *02790000
                                  --Masked by  AttrGenMask           02800000
ATTROPAQUE           DS    XL3    Opaque attribute flags -          *02810000
                                  Reserved for ADSTAR use            02820000
ATTRVISIBLE          DS    XL1    Visible attribute flags            02830000
ATTRNODELFILES       EQU   X'20'  Files should not be deleted   @P1A 02840000
ATTRSHARELIB         EQU   X'10'  Shared Library flag           @D8A 02850000
ATTRNOSHAREAS        EQU   X'08'  No shareas flag               @D6A 02860000
ATTRAPFAUTH          EQU   X'04'  APF authorized flag           @D6A 02870000
ATTRPROGCTL          EQU   X'02'  Program controlled flag       @D6A 02880000
ATTREXTLINK          EQU   X'01'  External Symlink                   02890000
ATTRREFTIME          DS    F      Reference Time -                  *02900000
```

```
                                  Reserved for ADSTAR use          02910000
                   DS    0F       Align ATTRFID                    02920000
ATTRFID            DS    CL8      File Identifier                  02930000
ATTRFILEFMT        DS    XL1      File Format                 @D5A 02940000
ATTRFSPFLAG2       DS    XL1      IFSP_FLAG2                  @DBA 02950000
ATTRACCESSACL      EQU   X'80'    Access ACL exists           @DBA 02960000
ATTRFMODELACL      EQU   X'40'    File Model ACL exists       @DBA 02970000
ATTRDMODELACL      EQU   X'20'    Directory Model ACL exists  @DBA 02980000
ATTRRES02          DS    CL2      Reserved for future         @DBC 02990000
*                                                                  03000000
ATTRCTIMEMSEC      DS    F        Ctime Micro_Seconds         @D7A 03010000
ATTRSECLABEL       DS    CL8      Security Label              @DBA 03020000
ATTRRES03          DS    CL4      Reserved for future         @DBC 03030000
ATTRENDVER1        EQU   *        End of Version 1 ATTR            03040000
*                                                                  03050000
ATTRATIME64        DS    D        Access Time                 @DAA 03060000
ATTRMTIME64        DS    D        Data Mod Time               @DAA 03070000
ATTRCTIME64        DS    D        Medadata Change Time        @DAA 03080000
ATTRCREATETIME64   DS    D        File Creation Time          @DAA 03090000
ATTRREFTIME64      DS    D        Reference Time              @DAA 03100000
                   DS    D                                    @DAA 03110000
                   DS    CL16     Reserved                    @DAA 03120000
ATTRENDVER2        EQU   *        End of Version 2 ATTR       @DAA 03130000
*                                                                  03140000
*   Constants                                                      03150000
*                                                                  03160000
ATTR#LEN           EQU   *-ATTRBEGIN                              *03170000
                                    Length of ATTR                 03180000
ATTR#LENGTH        EQU   ATTR#LEN Length of ATTR                   03190000
ATTR#MINLEN        EQU   ATTRENDVER1-ATTRBEGIN                    *03200000
                                    Minimum length of valid ATTR   03210000
ATTR#SP            EQU   2          Subpool for the ATTR           03220000
** BPXYATTR End                                                    03230000
```

# BPXYBRLK — Map the byte range lock request for fcntl

```
          BPXYBRLK   ,
** BPXYBRLK: External Byte Range Locking interface control block
**  Used By: FCT
BRLK                 DSECT ,
L_TYPE               DS    H      Requested lock type:
F_RDLCK              EQU   1       Shared or read lock
F_WRLCK              EQU   2       Exclusive or write lock
F_UNLCK              EQU   3       Unlock
L_WHENCE             DS    H      Flag for starting offset
L_START              DS    0CL8   Relative offset in bytes
L_START_H            DS    F      High word of relative offset
L_START_L            DS    F      Low word of relative offset
L_LEN                DS    0CL8   Size of lock in bytes
L_LEN_H              DS    F      High word of size of lock in bytes
L_LEN_L              DS    F      Low word of size of lock in bytes
L_PID                DS    F      Process ID of process holding lock
BRLK#LENGTH          EQU   *-BRLK Length of this area
** BPXYBRLK End
```

## BPXYDIRE — Map directory entries for readdir

DSECT=NO is not allowed, the basing for the PFSOTHER data is not known as it depends on the length of the name.

```
          BPXYDIRE   ,
** BPXYDIRE: Mapping of directory entry
**  Used By: RDD
* LA     RegOne,buffer            RegOne->BPX1RDD buffer and 1st DIRE
* USING  DIRE,RegOne              Addressability to DIRE
DIRE                 DSECT ,
DIRENTINFO           DS    0X     Fixed length information
DIRENTLEN            DS    H      Entry length
DIRENTNAML           DS    H      Name length
DIRENTNAME           DS    0C     Name
* LR     RegTwo,RegOne            RegTwo->DIRE
* LA     RegTwo,4(RegTwo)         RegTwo->start of name
* SLR    RegThree,RegThree        Clear register
* ICM    RegThree,3,DIRENTNAML    Load name length
* ALR    RegTwo,RegThree          RegTwo->end of name+1
* USING  DIRENTPFSDATA,RegTwo     Addressability to DIRENTPFSDATA
DIRENTPFSDATA        DSECT ,      Physical file system-specific data
DIRENTPFSINO         DS    CL4    File Serial Number = st_ino
DIRENTPFSOTHER       DS    0C     Other PFS specific data
* ICM    RegThree,3,DIRENTLEN     Load entry length
* ALR    RegOne,RegThree          RegOne->Next DIRE in buffer
* BCT    Return_Value,Back_to_process_next_DIRE
** BPXYDIRE End
```

## BPXYFDUM — Logical file system dump parameter list

```
          BPXYFDUM   ,
** BPXYFDUM: FDUM - LFS dump list passed to PFS initialization
FDUM                 DSECT ,
FDUMBEGIN            DS    0D
*
FDUMPHDRINFO         DS    0F
FDUMPENTS            DS    F            NUMBER OF ENTRIES
FDUMPID              DC    C'FDUM'      EYE CATCHER
FDUMPHRES1           DS    CL8          SPACE RESERVED FOR EXPANSION
*
FDUM#LENH            EQU   *-FDUMBEGIN
*
FDUMPDATA            DSECT ,
                     DS    0F           ONE SET FOR EACH AREA TO DUMP
FDUMPSTOKEN          DS    CL8          STOKEN FOR DUMP
FDUMPRES1            DS    CL8          RESERVED
FDUMPSTART           DS    F            FIRST BYTE TO DUMP
FDUMPEND             DS    F            LAST BYTE TO DUMP
*
FDUM#LENENT          EQU   *-FDUMPDATA
*
* To access the FDUM header (dumpptr must be a copy of pfsi_dumpptr):
* L      RegOne,dumpptr             RegOne->pfsi_dumpents from BPXYPFSI
* USING  FDUM,RegOne                Addressability to FDUM
*
* To access the first FDUMPDATA:
* LR     RegTwo,RegOne              RegTwo->FDUM
* LA     RegTwo,FDUM#LENH(RegTwo)   RegTwo->FDUMPDATA
* USING  FDUMPDATA,RegTwo           Addressability to FDUMPDATA fields
*
* To access the next FDUMPDATA:
* LA     RegTwo,FDUM#LENENT(RegTwo) RegTwo-> next FDUMPDATA
*
** BPXYFDUM End
```

## BPXYFTYP — File type definitions

BPXYFTYP is composed only of EQUates. DSECT= is allowed but ignored.

```
          BPXYFTYP   ,
** BPXYFTYP: File type definitions
**  Used By: FST MKD MKN OPN
FT_DIR               EQU   1      Directory File
FT_CHARSPEC          EQU   2      Character Special File
FT_REGFILE           EQU   3      Regular File
FT_FIFO              EQU   4      Named Pipe (FIFO) File
FT_SYMLINK           EQU   5      Symbolic link
*                    EQU   6      Reserved for Block Special
FT_SOCKET            EQU   7      Socket File
*
** File format definitions (for chattr)
FTFFNA               EQU   0      Not specified
FTFFBINARY           EQU   1      Binary data
*                                 Text data delimiters:
FTFFNL               EQU   2        New Line
FTFFCR               EQU   3        Carrage Return
FTFFLF               EQU   4        Line Feed
FTFFCRLF             EQU   5        CR & LF
FTFFLFCR             EQU   6        LF & CR
** BPXYFTYP End
```

# BPXYFUIO — Map file system user I/O block

```
         BPXYFUIO   ,
**  BPXYFUIO: User I/O block
**  Used By:  VRW VRD VRA
FUIO                 DSECT ,
FUIOBEGIN           DS    0D
FUIOHDR             DS    0D
*
FUIOID              DC    C'FUIO'                                   X
                                 EBCDIC ID - FUIO
FUIOLEN             DC    AL4(FUIO#LENGTH)                          X
                                 Length of this FUIO
FUIOINFO            DS    0D     Note: The following fields must    X
                                       map to BPXZDDPL
FUIOBUFFERADDR      DS    F      Buffer address for READ or         X
                                        WRITE, etc.  Address of iov X
                                        for READV and WRITEV
FUIOBUFFALET        DS    F      Alet associated with Buffer
FUIOCURSOR          DS    0F     Current position in the file
FUIOCUR1            DS    F      Word 1 of cursor
FUIOCUR2            DS    F      Word 2 of cursor
FUIOIBYTESRW        DS    F      Num of bytes to read or write      X
                                 (or iovcnt for READV and WRITEV)
FUIOASID            DS    H      Address Space ID
*
*
FUIOFLAGS           DS    XL1    Flags
*
FUIORWIND           EQU   FUIOFLAGS                                 X
                                 Indicates if READ or WRITE         X
                                 0 - Read, 1 - Write
FUIO#RD             EQU   X'7F'  Read: AND with FUIORWIND
FUIO#WRT            EQU   X'80'  Write: OR with FUIORWIND
*
FUIOPSWKEY          EQU   FUIOFLAGS                                 X
                                 Describes bits 1 through 4 of      X
                                 byte FUIOFLAGS
FUIOPSWKEYMASK      EQU   X'78'  AND with FUIOPSWKEY to clear       X
                                 non-PSWKEY bits in FUIOFLAGS
*
FUIOSYNC            EQU   X'04'  Sync on write requested
FUIOSYNCDONE        EQU   X'02'  Sync on write was done
FUIOCHKACC          EQU   X'01'  Perform access checking
FUIOFLAG2           DS    XL1    More flags
FUIOREALPAGE        EQU   X'80'  Real page address provided
FUIOLIMITEX         EQU   X'40'  File size limit exceeded
FUIOIOVINUIO        EQU   X'20'  uio contains an iov struc
FUIOSOCKINVALID     EQU   X'10'  Invalid Sockaddr address
FUIOVSPECIFIC       DS    CL8    Vnop Specific Fields
FUIOFSSIZELIMIT     DS    0CL8   Rlimit support
FUIOFSSIZELIMITHW   DS    F      hiword - filesize limit
FUIONONEWFILES      EQU   X'80'  can't create new files
FUIOFSSIZELIMITLW   DS    F      loword - filesize limit
FUIOREL2SIZE        DS    0F     Fuio before Rel 3 expansion
FUIOCURRBUFFPTR     DS    F      Buffer currently being processed
FUIOCURRBUFFLEN     DS    F      Length of current buffer
FUIOCURRBUFFOFFSET  DS    F      Offset into current buffer
FUIOCURRIOVENTRY    DS    F      Iov entry being processed
FUIOIOVRESIDUALCNT  DS    F      Num bytes remaining in iov str
FUIOTOTALBYTESRW    DS    F      Total number of bytes to be moved  X
                                 If FuioIovinUio=on,                X
                                 this is the sum of all bytes       X
                                 in the iov. Otherwise, this is     X
                                 the same as FuioIBytesRW
FUIORES01           DS    CL8    Reserved area
FUIOEND             DS    0F     End of FUIO
```

```
*---------------------------------------------------------------
*   ReadDir Specific Information
*---------------------------------------------------------------
FUIOREADDIR          ORG   FUIOVSPECIFIC
FUIORDINDEX          DS    F      Readdir Index
FUIORDRES01          DS    F      Reserved
*---------------------------------------------------------------
*   VN_ReadWriteV and VN_SRMsg Specific Information
*---------------------------------------------------------------
FUIOSOCKETALETS      ORG   FUIOVSPECIFIC
FUIOIOVALET          DS    F      SRMsg IOV Alet
FUIOIOVBUFALET       DS    F      All IOV's Buff's Alet
                     ORG
*
*   Constants
*
FUIO#LEN             EQU   FUIOEND-FUIOBEGIN                   X
                                    Length of FUIO
FUIO#LENGTH          EQU   FUIO#LEN Length of FUIO
FUIO#REL2LEN         EQU   FUIOREL2SIZE-FUIOBEGIN              X
                                    Length of Release 2 FUIO
FUIO#SP              EQU   3        Subpool for the FUIO
** BPXYFUIO End
```

## BPXYIOC6 — ioctl network mapping information for IPV6

```
*       %GOTO  IOC6PLX ;      /* Bilingual header
***********************************************************************
* NetConfHdr Structure                                                *
***********************************************************************
*
.A411    ANOP  ,
         AIF   ('&DSECT;' EQ 'NO').B411
NETCONFHDR          DSECT ,
         AGO   .C411
.B411    ANOP  ,
NETCONFHDR          DS    0F
.C411    ANOP  ,
NCHEYECATCHER       DS    CL4    Eye catcher
NCHIOCTL            DS    F      Ioctl being processed (RAS)
NCHBUFFERLENGTH     DS    F      Buffer Length
NCHBUFFERPTR        DS    F      Buffer Pointer
NCHNUMENTRYRET      DS    F      Number of HomeIF returned via       *
                                      SIOCGHomeIf6 or the number of   *
                                      GRT6RtEntry's returned via      *
                                      SIOCGRT6TABLE.
NETCONFHDR#LENGTH   EQU   *-NETCONFHDR   Length of NETCONFHDR
*
***********************************************************************
* HomeIf Structure                                                    *
***********************************************************************
*
         AIF   ('&DSECT;' EQ 'NO').D411
HOMEIF              DSECT ,      HomeIf structure
         AGO   .E411
.D411    ANOP  ,
HOMEIF              DS    0F     HomeIf Structure
.E411    ANOP  ,
HomeIfAddress       DS    CL16   Home Interface Address
*
HomeIf#LENGTH       EQU   *-HOMEIF   Length of HOMEIF
*
***********************************************************************
* GRT6RtEntry Structure                                               *
***********************************************************************
*
         AIF   ('&DSECT;' EQ 'NO').F411
GRT6RTENTRY         DSECT ,      GRT6RtEntry Structure
         AGO   .G411
.F411    ANOP  ,
GRT6RTENTRY         DS    0F     GRT6RtEntry Structure
.G411    ANOP  ,
*
GRT6DESTINATION     DS    CL16 Destination IP Address
GRT6GATEWAY         DS    CL16 First HOP on the trip if going through *
                                a gateway
GRT6DESTPREFIXLEN   DS    F    Destination's Prefix Length which is a *
                                decimal value that specifies how many *
                                of the leftmost contiguous bits of the*
                                address comprise the prefix
GRT6RTMETRIC        DS    F    Metric - hop count. Currently Tcp/Ip   *
                                returns 1 for indirect routes and 0   *
                                for direct routes. If route is from   *
                                routing daemon, metric is whatever    *
                                routing daemon set it to.
GRT6RTFLAGS         DS    F    IPV6 Route Flags.
*
GRT6RTENTRY#LENGTH  EQU   *-GRT6RTENTRY   Length of GRT6RTENTRY
*
***********************************************************************
* RT6Entry Structure                                                  *
```

```
***********************************************************************
*
         AIF   ('&DSECT;' EQ 'NO').H411
RT6ENTRY            DSECT ,          Rt6Entry Structure
         AGO   .I411
.H411    ANOP  ,
RT6ENTRY               DS    0F      Rt6Entry Structure
.I411    ANOP  ,
*
RT6DESTINATION         DS    CL28    Destination IP address (in an IPV6   *
                                       sockaddr structure)
RT6GATEWAY             DS    CL28    First HOP on the trip if going       *
                                       through a gateway (in an IPV6      *
                                       sockaddr structure)
RT6DESTPREFIXLEN       DS    F       Destination's Prefix Length,         *
                                         which is a decimal value         *
                                         that specifies how many of       *
                                         the leftmost contiguous          *
                                         bits of the address              *
                                         comprise the prefix.
RT6METRIC              DS    F       Metric - hop count                   *
                                        Currently Tcp/IP returns          *
                                         1 for indirect route and         *
                                         0 for direct route.              *
                                        If route is from routing          *
                                        daemon, metric is whatever        *
                                        routing daemon set it to.
RT6FLAGS               DS    F       IPV6 Route Flags.
*
RT6ENTRY#LENGTH        EQU   *-RT6ENTRY   Length of RT6ENTRY
*
***********************************************************************
* IPV6RtFlags Structure                                               *
***********************************************************************
*
         AIF   ('&DSECT;' EQ 'NO').J411
IPV6RTFLAGS         DSECT ,          IPV6RtFlags Structure
         AGO   .K411
.J411    ANOP  ,
IPV6RTFLAGS            DS    0F      IPV6RtFlags Structure
.K411    ANOP  ,
*
IPV6FLGBYTE1           DS    XL1     Reserved
IPV6FLGBYTE2           DS    XL1     Reserved
IPV6FLGBYTE3           DS    XL1     Reserved
IPV6FLGBYTE4           DS    XL1     FLAGS:
*                      EQU   X'80'   Reserved
*                      EQU   X'40'   Reserved
*                      EQU   X'20'   Reserved
*                      EQU   X'10'   Reserved
IPV6BITHOME            EQU   X'08'   1 = Home interface
IPV6BITHOST            EQU   X'04'   1 = Host Route. 0 = Network Route
IPV6BITGATE            EQU   X'02'   1 = Gateway
IPV6BITRTUP            EQU   X'01'   1 = Route is active
*
** BPXYIOC6 End
         SPACE 3
         AIF   ('&LIST;' EQ 'YES').Z411
         POP   PRINT
.Z411    ANOP  ,
         MEND
  Terminating PL/X comment                                            */
 %IOC6PLX : ;


 /*********************************************************************/
 /* The following code was requested by Raleigh                       */
```

```
/********************************************************************/
%dcl bpxyioc6_inc fixed external;
%If bpxyioc6_inc = 0
  %Then
   %Do;
    %bpxyioc6_inc=1;


/********************************************************************/
/*                                                                  */
/*  Internet Protocol address type                                  */
/*                                                                  */
/********************************************************************/
Declare
 IPV6Addr Type Char(16);            /* IPV6 IP Address Type         */

/********************************************************************/
/*                                                                  */
/* The network configuration header is used with the SIOCGHomeIf6   */
/* and SIOCGRT6TABLE IOCTLs.                                        */
/*                                                                  */
/* When the USS Pre-Router issues the SIOCGHomeIf6 IOCTL, it passes */
/* the NetConfHdr filled in with the EyeCatcher, IOCTL, buffer      */
/* length and buffer ptr. The stack returns HomeIf records in       */
/* the supplied buffer and sets NchNumEntryRet with the number      */
/* HomeIf records being returned.                                   */
/*                                                                  */
/* When the USS Pre-Router issues the SIOCGRT6TABLE IOCTL, it passes*/
/* the NetConfHdr filled in with the EyeCacther, IOCTL, buffer      */
/* length and buffer ptr. The stack returns GRT6RtEntry records     */
/* in the supplied buffer and sets NchNumEntryRet with the          */
/* number of GRT6RtEntry records being returned.                    */
/*                                                                  */
/*                                                                  */
/********************************************************************/
Declare
 1 NetConfHdr Bdy(Word) Based, /* Network Configuration Header      */
  3 NchEyeCatcher Char(4),     /* Eye Catcher '6NCH'                */
  3 NchIOCTL Fixed(32),        /* The IOCTL being processed
                                       with this instance of the
                                       NetConfHdr. (RAS item)       */
  3 NchBufferLength Fixed(31), /* Buffer Length                     */
  3 NchBufferPtr    Ptr(31),   /* Buffer Pointer                    */
  3 NchNumEntryRet Fixed(31);  /* Number of HomeIF returned via
                                       SIOCGHomeIf6 or the number of
                                       GRT6RtEntry's returned via
                                       SIOCGRT6TABLE.               */
/********************************************************************/
/*                                                                  */
/*The mapping for the home interface record that is returned via the*/
/*SIOCGHomeIf6 IOCTL.                                               */
/*                                                                  */
/********************************************************************/
Declare
 1 HomeIf Bdy(Word) Based,              /* Home Interface Structure */
  3 HomeIfAddress(*) Isa(IPV6Addr);     /* Home Interface Address   */

/********************************************************************/
/*                                                                  */
/* The mapping for the Pre-Router Route Entry that is returned by   */
/* the stack via SIOCGRT6TABLE.                                     */
/*                                                                  */
/* NOTE: The only difference between this route entry and the      */
/*       route entry used with the SIOCMSADDRT6 and SIOCMSDELRT6    */
/*       IOCTLs is that "destination" and "gateway" are IP addresses*/
/*       not SockAddr Structures. The reason for that is because    */
/*       the Pre-Router only needs the IP address so additional     */
```

```
/*       buffer storage should not be obtained to hold SockAddr     */
/*       fields that are not used.                                  */
/*                                                                  */
/*       The reasons why a SockAddr structure was used with the     */
/*       SIOCMSADDRT6 and SIOCMSDELRT6 route entry is:              */
/*        (1) This interface may be externalized via SIOCADDRT6 and */
/*            SIOCDELRT6 which will use a Sockaddr structure between*/
/*            routing daemons and the stack.                        */
/*        (2) If the Pre-Router "intercepts" these IOCTLs, it could */
/*            process them thus eliminating the need for the stack  */
/*            to send a SIOCMSADDRT6/DELRT6 IOCTL to the Pre-Router */
/*            for route updates received from a routing daemon.     */
/*                                                                  */
/********************************************************************/
Declare
 1 GRT6RtEntry    Bdy(Word) Based, /* Route entry used with the
                                        SIOCGRT6TABLE IOCTL.        */
  3 GRT6Destination Isa(IPV6Addr), /* Destination IP address.       */
  3 GRT6Gateway     Isa(IPV6Addr), /* First HOP on the trip if
                                        going through a gateway.    */
  3 GRT6DestPrefixLen Fixed(31),   /* Destination's Prefix Length
                                        which is a decimal value
                                        that specifies how many of
                                        the leftmost contiguous
                                        bits of the address
                                        comprise the prefix.        */
  3 GRT6RtMetric     Fixed(31),    /* Metric - hop count
                                        Currently Tcp/IP returns
                                         1 for indirect route and
                                         0 for direct route.
                                        If route is from routing
                                        daemon, metric is whatever
                                        routing daemon set it to.*/
  3 GRT6RtFlags Isa(Ipv6RtFlags); /* IPV6 Route Flags.             */

/********************************************************************/
/*                                                                  */
/* The mapping for the Route Entry that is specified on the         */
/* SIOCMSADDRT6 and SIOCMSDELRT6 IOCTLs.                            */
/*                                                                  */
/* NOTE: This structure uses SockAddr structures for the            */
/*       "destination" and "gateway" addresses, the Grt6RtEntry     */
/*       uses Ip addresses. See the Grt6RtEntry structure for       */
/*       the reasons.                                               */
/*                                                                  */
/********************************************************************/
Declare
 1 Rt6Entry    Bdy(Word) Based,     /* Route entry used with the
                                       SIOCMSADDRT6/DELRT6 IOCTLs   */

  3 Rt6Destination                  /* Destination IP address (in a */
           Isa(Sock_Inet6_SockAddr),/*  sockaddr structure)         */

  3 Rt6Gateway                      /* First HOP on the trip if     */
           Isa(Sock_Inet6_SockAddr),/*   going through a gateway.   */

  3 Rt6DestPrefixLen Fixed(31),/* Destination's Prefix Length,
                                       which is a decimal value
                                       that specifies how many of
                                       the leftmost contiguous
                                       bits of the address
                                       comprise the prefix.         */
  3 Rt6Metric     Fixed(31),        /* Metric - hop count
                                      Currently Tcp/IP returns
                                       1 for indirect route and
                                       0 for direct route.
                                      If route is from routing
```

```
                                          daemon, metric is whatever
                                          routing daemon set it to.  */
  3 Rt6Flags Isa(Ipv6RtFlags);       /* IPV6 Route Flags.            */

/*******************************************************************/
/*                                                                 */
/* The mapping for the route flags                                 */
/*                                                                 */
/*******************************************************************/
Declare
 1 IPV6RtFlags Type,              /* Route Flags                   */
  3 IPV6FlgByte1 Bit(8),          /* Reserved                      */
  3 IPV6FlgByte2 Bit(8),          /* Reserved                      */
  3 IPV6FlgByte3 Bit(8),          /* Reserved                      */
  3 IPV6FlgByte4 Bit(8),          /* Reserved                      */
   5 *              Bit(4),       /* Reserved                      */
   5 IPV6BitHome   Bit(1),        /* 1 = Home Interface            */
   5 IPV6BitHost   Bit(1),        /* 1 = Host Route, 0=Network route*/
   5 IPV6BitGate   Bit(1),        /* 1 = Gateway                   */
   5 IPV6BitRtUp   Bit(1);        /* 1 = Route is active           */

/*******************************************************************/
/*                                                                 */
/*   Constants                                                     */
/*                                                                 */
/*******************************************************************/
Declare
  IOC6_#HomeIfPrefixLen Constant(128);   /* The prefix length for a
                                            home interface address
                                            returned on the
                                            SIOCGHomeIf6 IOCTL.    */
Declare
 IOC6_Nch#Eye Char(4)  Constant('6NCH'); /* IPV6 Network Configuration
                                            Header EyeCatcher.      */
/*******************************************************************/
/*                                                                 */
/* Maximum hop count for the Metric fields:                        */
/*    GRT6RtMetric                                                 */
/*    Rt6Metric                                                    */
/*                                                                 */
/*******************************************************************/
Declare
  IOC6_#MaxHopMetric  Fixed(31) constant(16);

/*******************************************************************/
/*                                                                 */
/*   Constants used for size of control areas                      */
/*                                                                 */
/*******************************************************************/
Declare
  IOC6_#MaxRoutes     Fixed(31) Constant(600),
  IOC6_#Grt6RouteLen  Fixed(31) Constant(Length(GRT6RtEntry)),
  /*****************************************************************/
  /* Initial buffer size for SIOCGHomeIf6 and SIOCGRT6TABLE.       */
  /*****************************************************************/
  IOC6_#MAXGRT6Len    Fixed(31) Constant(IOC6_#MaxRoutes *
                                         Ioc6_#Grt6RouteLen),
  IOC6_#NetConfHdrLen Fixed(31) Constant(Length(NetConfHdr));


   %End; /* End if bpxyioc6_inc = 1      */
```

## BPXYIPCP — Map interprocess communication permissions

```
          BPXYIPCP   ,
** BPXYIPCP: Interprocess Communications Permission
**  Used By: MCT, MGT, SCT, SGT, QCT, QGT
IPC_PERM            DSECT ,      Interprocess Communications
IPC_UID             DS    F      Owner's effective user ID
IPC_GID             DS    F      Owner's effective group ID
IPC_CUID            DS    F      Creator's effective user ID
IPC_CGID            DS    F      Creator's effective group ID
IPC_MODE            DS    XL4    Mode, mapped by BPXYMODE
IPC#LENGTH   EQU   *-IPC_PERM  Length of Interprocess Control block
* Key:
IPC_PRIVATE          EQU   0      Private key.
* Mode bits:                      Map over S_TYPE in BPXYMODE
IPC_CREAT            EQU   1      Create entry if key does not exist.
IPC_EXCL             EQU   2      Fail if key exists.
* Flag bits - semop, msgrcv, msgsnd:
IPC_NOWAIT           EQU   1      Error if request must wait.
* Control Command:
IPC_RMID             EQU   1      Remove identifier.
IPC_SET              EQU   2      Set options.
IPC_STAT             EQU   3      Access status.
* CONSTANTS WHICH MAP OVER BYTE S_TYPE, SEE BPXYMODE
** BPXYIPCP End
```

## BPXYIPCQ — Map w_getipc structure

```
          BPXYIPCQ   ,
** BPXYIPCQ: w_getipc interface mapping
**  Used By: GET
IPCQ          DSECT ,      Interprocess Communications - Query
IPCQLENGTH    DS    F      IPCQ#LENGTH used by system call.  If not
*                          equal, check BPXYIPCQ and system levels.
IPCQTYPE      DS    CL4    "IMSG", "ISEM", "ISHM", "OVER"
IPCQOVER      DS    0F     OVERVIEW MAPPING STARTS HERE
IPCQMID       DS    FL4    MEMBER ID
IPCQKEY       DS    XL4    KEY
IPCQIPCP      DS    CL20   MAPPED BY BPXYIPCP
IPCQGTIME     DS    XL4    TIME_T OF LAST ...GET()
IPCQCTIME     DS    XL4    TIME_T OF LAST ...CTL()
IPCQTTIME     DS    XL4    TIME_T CHANGED BY TERMINATION
IPCQREST      DS    0C     IPCQMSG, IPCQSHM, IPCQSEM
              ORG   IPCQREST Message Queue unique data
              DS    0F
IPCQBYTES     DS    F      # BYTES OF MESSAGES ON QUEUE
IPCQQBYTES    DS    F      MAX # BYTES OF MESSAGES ALLOWED ON QUEUE
IPCQLSPID     DS    F      PID OF LAST MSGSND()
IPCQLRPID     DS    F      PID OF LAST MSGRCV()
IPCQSTIME     DS    F      TIME_T OF LAST MSGSND()
IPCQRTIME     DS    F      TIME_T OF LAST MSGRCV()
IPCQNUM       DS    F      # OF MESSAGES ON QUEUE
IPCQRCNT      DS    F      COUNT OF WAITING MSGRCV
IPCQSCNT      DS    F      COUNT OF WAITING MSGSND
              DS    0CL16  MSGRCV AND MSGSND WAITERS
              DS    0CL8    MSGRCV - WAIT FOR TYPE
IPCQQRPID     DS    F         PROCESS ID
IPCQQRMSGTYPE DS    F         MESSAGE TYPE
              DS    0CL8    MSGSND - WAIT FOR ROOM TO SEND
IPCQQSPID     DS    F         PROCESS ID
IPCQQSMSGLEN  DS    F         MESSAGE LENGTH
              DS    9CL16  MSGSND AND MSGRCV WAITERS
              ORG   IPCQREST Semaphore Unique data
              DS    0F
IPCQLOPID     DS    XL4    PID OF LAST SEMOP
IPCQOTIME     DS    F      TIME_T LAST SEMOP
IPCQADJBADCNT DS    F      TERMINATION BUMPS SEM_VAL LIMITS
IPCQNSEMS     DS    FL2    NUMBER OF SEMAPHORES IN THIS SET
IPCQADJCNT    DS    FL2    NUMBER OF UNDO STRUCTURES
IPCQNCNT      DS    FL2    COUNT OF WAITERS FOR >0
IPCQZCNT      DS    FL2    COUNT OF WAITERS FOR =0
              DS    0CL16  WAITERS AND ADJUSTERS
              DS    0CL8    WAITER
IPCQSWPID     DS    F         PROCESS ID
IPCQSWNUM     DS    H         SEMAPHORE NUMBER
IPCQSWOP      DS    H         SEMAPHORE OPERATION
              DS    0CL8    ADJUSTER
IPCQSAPID     DS    F         PROCESS ID
IPCQSANUM     DS    H         SEMAPHORE NUMBER
IPCQSAADJ     DS    H         SEMAPHORE OPERATION
              DS    9CL16  WAITERS AND ADJUSTERS
              ORG   IPCQREST Shared Memory unique data
              DS    0F
IPCQACNT      DS    F      USE COUNT (#SHMAT - #SHMDT)
IPCQSEGSZ     DS    F      MEMORY SEGMENT SIZE
IPCQDTIME     DS    F      TIME_T OF LAST SHMDT()
IPCQATIME     DS    F      TIME_T OF LAST SHMAT()
IPCQLPID      DS    F      PID OF LAST SHMAT() OR SHMDT()
IPCQCPID      DS    XL4    PID OF CREATOR
IPCQATPID     DS    F      ATTACHED PROCESS ID
IPCQATADDRESS DS    F      SEGMENT ADDRESS FOR PROCESS
              DS    18F    MORE ATTACHED PROCESS IDS AND
*                          SEGMENT ADDRESS
```

```
              ORG   IPCQOVER Overview
                 DS    0F  MESSAGE QUEUES
IPCQOMSGNIDS     DS    F        Maximum number MSQs allowed
IPCQOMSGHIGHH2O  DS    F        Most MSQs at one time
IPCQOMSGFREE     DS    F        Number MSQs available
IPCQOMSGPRIVATE  DS    F        Number MSQs with Ipc_PRIVATE
IPCQOMSGKEYED    DS    F        Number MSQs with KEYs
IPCQOMSGREJECTS  DS    F        TIMES MSGGET DENIED
IPCQOMSGQBYTES   DS    F        MAX BYTES PER QUEUE
IPCQOMSGQMNUM    DS    F        MAX NUMBER MESSAGES PER QUEUE
IPCQOMSGNOALC    DS    F        # MSGSNDS THAT RETURNED ENOMEM
                 DS    F
                 DS    0F  SEMAPHORE
IPCQOSEMNIDS     DS    F        Maximum number SEMs allowed
IPCQOSEMHIGHH2O  DS    F        Most SEMs at one time
IPCQOSEMFREE     DS    F        Number SEMs available
IPCQOSEMPRIVATE  DS    F        Number SEMs with Ipc_PRIVATE
IPCQOSEMKEYED    DS    F        Number SEMs with KEYs
IPCQOSEMREJECTS  DS    F        TIMES SEMGET DENIED
IPCQOSEMSNSEMS   DS    F        MAX NUMBER OF SEMAPHORES PER SET
IPCQOSEMSNOPS    DS    F        MAX NUMBER OPERATION IN SEMOP
IPCQOSEMSBYTES   DS    F        STORAGE LIMIT
IPCQOSEMCBYTES   DS    F        STORAGE COUNT
                 DS    F
                 DS    0F  SHARED MEMORY
IPCQOSHMNIDS     DS    F        Maximum number SHMs allowed
IPCQOSHMHIGHH2O  DS    F        Most SHMs at one time
IPCQOSHMFREE     DS    F        Number SHMs available
IPCQOSHMPRIVATE  DS    F        Number SHMs with Ipc_PRIVATE
IPCQOSHMKEYED    DS    F        Number SHMs with KEYs
IPCQOSHMREJECTS  DS    F        TIMES SHMGET DENIED
IPCQOSHMSPAGES   DS    F        MAX # PAGES PER SYSTEM LIMIT
IPCQOSHMMPAGES   DS    F        MAX # PAGES PER SEGMENT LIMIT
IPCQOSHMNSEGS    DS    F        MAX # SEGMENTS PER PROCESS LIMIT
IPCQOSHMCPAGES   DS    F        CURRENT # BYTES SYSTEM WIDE
IPCQOSHMBIGGEST  DS    F        LARGEST SEGMENT ALLOCATED
              ORG   ,
IPCQ#LENGTH   EQU   *-IPCQ Storage needed for w_getipc function
* w-getipc Command:
IPCQ#MSG      EQU   1      Retrieve next message queue
IPCQ#SHM      EQU   2      Retrieve next shared memory segment
IPCQ#SEM      EQU   3      Retrieve next semaphore set
IPCQ#ALL      EQU   4      Retrieve next member, all mechanisms
IPCQ#OVER     EQU   5      Retrieve overview
** BPXYIPCQ End
```

## BPXYMSG — Map interprocess communication message queues

DSECT (MSGBUF) will be generated with either DSECT=NO or DESECT=YES. If DSECT=NO is specified, you may need an additional DSECT / CSECT statement to return to the current DSECT or CSECT.

```
          BPXYMSG   ,
** BPXYMSG: Interprocess Communication Message Queue Structure
**  Used By: msgctl
MSQID_DS             DSECT ,      message queue structure
MSG_PERM             DS    CL(IPC#LENGTH) Mapped by BPXYIPCP
MSG_QNUM             DS    F      # of messages on queue
MSG_QBYTES           DS    F      max bytes allowed on queue
MSG_LSPID            DS    F      process ID of last msgsnd()
MSG_LRPID            DS    F      process ID of last msgrcv()
MSG_STIME            DS    F      time of last msgsnd()
MSG_RTIME            DS    F      time of last msgrcv()
MSG_CTIME            DS    F      time of last change  get/ctl
MSQ#LENGTH           EQU *-MSQID_DS Length of this DSECT
MSGBUF               DSECT ,      Message buffer - msgsnd, msgrcv
MSG_TYPE             DS    F      Message type
MSG_MTEXT            DS    CL100  Message text
MSGB#LENGTH          EQU *-MSGBUF Length of this DSECT
MSGXBUF              DSECT ,      Message buffer - msgxrcv
MSGX_MTIME           DS    F      time message sent
MSGX_UID             DS    F      sender's effective UID
MSGX_GID             DS    F      sender's effective GID
MSGX_PID             DS    F      sender's PID
MSGX_TYPE            DS    F      Message type
MSGX_MTEXT           DS    CL100  Message text
MSGX#LENGTH          EQU *-MSGXBUF Length of this DSECT
* Flag bits - msgrcv (also IPC_NOWAIT
MSG_NOERROR          EQU   4      No error if big message.
MSG_INFO             EQU   8      Use MSGXBUF not MSGBUF format
** BPXYMSG End
```

## BPXYMNTE — Map response and element structure of w_getmnte

DSECT (MNTENTPARMDATA) will be generated with either DSECT=NO or DESECT=YES. If DSECT=NO is specified, you may need an additional DSECT / CSECT statement to return to the current DSECT or CSECT.

```
          BPXYMNTE   ,
** BPXYMNTE: OpenMVS w_getmntent response structure and element
**  Used By: GMN
MNTEH               DSECT ,
MNTEH               DS    0F
MNTEHID             DC    C'MNTE'   Eye catcher
MNTEHID             DC    C'MNT2'   Eye catcher
MNTEHSIZE           DC    A(MNTE#LENGTH)   Size of area (MNTEH+MNTE)
MNTEHCUR            DC    XL8'0000000000000000'
*                                Index of next element to return
*                                   - must be zero (i.e.
*                                     X'0000000000000000'),
*                                     on initial call
*                                   - must be left undisturbed
*                                     for subsequent calls
MNTEHDEVNO          DS     F'0'  Device number - this value is
*                                specified if information about only
*                                one file system is requested
MNTEHRES1           DS     A(0)  Reserved for future - must be zero
*                                 on entry

MNTEHBLEN           DS     F     Length of mnte body used
MNTEHRES1           DS     BL8   Reserved for future - must be zero
*                                on entry
MNTEH#LENGTH        EQU   *-MNTEH Length of header structure*
MNTE                DSECT ,
*
MNTE                DS    0F
MNTENTFSTYPE        DS    F      File system type
MNTENTFSTYPEMVS     EQU   1      MVS Local File System
MNTENTFSTYPEREMOTE  EQU   2      Remote File System
MNTENTFSTYPEPIPE    EQU   3      Pipe file system
MNTENTFSTYPESOCKET  EQU   4      Socket file system
MNTENTFSTYPEXPFS    EQU   5      Cross System PFS (XPFS)
MNTENTFSTYPECSPS    EQU   6      Char special streams
MNTENTFSTYPENFS     EQU   MNTENTFSTYPEREMOTE
MNTENTFSMODE        DS    0F     File system mount flags
MNTENTFSMODE1       DS    B      File system mount method - byte 1
MNTENTFSMODE2       DS    B      File system mount method - byte 2
MNTENTFSMODE3       DS    B      File system mount method - byte 3
MNTENTFSMODE4       DS    B      File system mount method - byte 4
MNTENTFSCLIENT      EQU   X'20'  File system is a client
MNTENTFSNOAUTOMOVE  EQU   X'10'  Automove allowed
MNTENTFSMODENOSEC   EQU   X'08'  No Security checks enforced
MNTENTFSMODEEXPORT  EQU   X'04'  File system exported by DFS
MNTENTFSMODENOSUID  EQU   X'02'  SetUID not permitted for
*                                files in this file system
MNTENTFSMODERDONLY  EQU   X'01'  File system mounted read only
MNTENTFSMODERDWR    EQU   X'00'  File system mounted read/write
MNTENTFSDEV         DS    F      st_dev value to be returned by
*                                the stat system call for all files
*                                in this file system
MNTENTPARENTDEV     DS    F      st_dev of the parent file system
MNTENTROOTINO       DS    F      ino of the mount point
MNTENTSTATUS        DS    B      Status of the file system
MNTENTFILEACTIVE    EQU   B'00000000' File system is active
MNTENTFILEDEAD      EQU   B'00000001' File system is dead
MNTENTFILERESET     EQU   B'00000010' File system being reset
MNTENTFILEDRAIN     EQU   B'00000100' File system being unmounted with
*                                        drain option
MNTENTFILEFORCE     EQU   B'00001000' File system being unmounted with
```

```
*                                        force option
MNTENTFILEIMMED      EQU   B'00010000' File system being unmounted with
*                                        immed option
MNTENTFILENORM       EQU   B'00100000' File system being unmounted with
*                                        normal option
MNTENTIMMEDTRIED     EQU   B'01000000' File system Umount immed failed
MNTENTQUIESCED       EQU   B'10000000' File system is quiesced
MNTENTMNTINPROGRESS  EQU   B'10000001' Mount in progress for0
*                                        this file system
MNTENTASYNCHMOUNT    EQU   B'10000010' Asynchronous mount in progress
*                                        for this file system
MNTENTFSDDNAME       DS    CL9    DDNAME specified on mount - null
*                                 terminated
MNTENTFSTNAME        DS    CL9    File system type name -
*                                 from the FILESYSTYPE parmlib
*                                 statement - null terminated
MNTENTFSNAM44        DS    CL44   File system name - as a 44 byte field
                     ORG   MNTENTFSNAM44
MNTENTFSNAME         DS    CL45   File system name - for PDSE/X, this
*                                 is the name of the PDSE/X containing
*                                 file system, null terminated
MNTENTPATHLEN        DS    F      length of mount point path name
MNTENTMOUNTPOINT     DS    CL1024 Name of directory where the file
*                                 system is mounted - (mount point
*                                 path name - null terminated
MNTENTJOBNAME        DS    CL8    Job name of quiesce requestor
MNTENTPID            DS    F      PID of quiesce requestor
MNTENTPARMOFFSET     DS    F      Offset of MntEntParm from MNTE
*                                 (Zero if none)
MNTENTPARMLEN        DS    H      Length of mount parameter
*                                 (Zero if none)
MNTENTRES01          DS    H      Reserved for future expansion
MNTENTRES02          DS    13F    Reserved for future expansion
MNTENTSYSNAME        DS    CL8    Name of system to mount on
MNTENTQSYSNAME       DS    CL8    Name of queise system name
MNTENTFROMSYS        DS    CL8    Filesystems to be moved from here
MNTENTRES00          DS    2B     Alignment
MNTENTRFLAGS         DS    0F     Request flags
MNTENTRFLAGS1        DS    B      Request flags - byte 1
MNTENTRFLAGS2        DS    B      Request flags - byte 2
MNTENTRFLAGS3        DS    B      Request flags - byte 3
MNTENTRFLAGS4        DS    B      Request flags - byte 4
MNTENTCHANGE         EQU   X'01'  Change f.s. server request
MNTENTNEWAUTO        EQU   X'02'  Change automove setting
MNTENTSTATUS2        DS    0F     Status of filesystem
MNTENTSTATUS2B1      DS    B      Status of filesystem - byte 1
MNTENTSTATUS2B2      DS    B      Status of filesystem - byte 2
MNTENTSTATUS2B3      DS    B      Status of filesystem - byte 3
MNTENTSTATUS2B4      DS    B      Status of filesystem - byte 4
MNTENTUNOWNED        EQU   B'00000001' File system unowned
MNTENTINRECOVERY     EQU   B'00000010' File system in recovery
MNTENTSUPERQUIESCED  EQU   B'00000100' File system super quiesced
MNTENTSUCCESS        DS    F      Successful moves
MNTENTREADCT         DS    F      Number of reads from filesys
MNTENTWRITECT        DS    F      Number of writes done
MNTENTDIRIBC         DS    F      Number of directory I/O blocks
MNTENTREADIBC        DS    F      Number of read I/O blocks
MNTENTWRITEIBC       DS    F      Number of write I/O blocks
MNTENTBYTESREAD      DS    BL8    Number of bytes read
MNTENTBYTESWRITTEN   DS    BL8    Number of bytes written
MNTENTRES01          DS    6F     Reserved for future expansion
MNTE#LENGTH          EQU   *-MNTE Length of this structure
*
MNTENTPARMDATA       DSECT ,      Mount() parameter data dsect
MNTENTPARM           DS    0C     Parameter specified with mount()
*
* To access MNTEH, MNTE and MNTENTPARM:
```

```
* LA     RegOne,buffer            RegOne->BPX1GMN buffer and MNTEH
* USING  MNTEH,RegOne             Addressability to MNTEH
*
* LR     RegTwo,RegOne            RegTwo->MNTEH
* LA     RegTwo,MNTEH#LENGTH(RegTwo) RegTwo->MNTE
* USING  MNTE,RegTwo              Addressability to MNTENTPARMLEN
*                                 and MNTENTPARMOFFSET
*
* ICM    RegThree,15,MNTENTPARMOFFSET Load offset from start of
*                                 entry  (i.e. start of MNTE)
* BZ     SkipParm                 If zero, skip processing parm
* ALR    RegThree,RegTwo          RegTwo->MNTE,
*                                 RegThree=MNTENTPARMOFFSET
*                                 RegThree->MNTENTPARMDATA (after)
* USING  MNTENTPARMDATA,RegThree  Addressability to MNTENTPARMDATA
*
** BPXYMNTE End
```

## BPXYMODE — Map the mode constants of the file services

```
          BPXYMODE   ,
** BPXYMODE: Mode constants specified on system calls
**  Used By: CHM FCM MKD MKN OPN UMK
S_MODE              DSECT ,
                    DS    0F
*
S_TYPE              DS    B      File types, mapped by BPXYFTYP
*                                Flag bytes
S_MODE3B            DS    0XL3   All flag bytes
S_RES01             DS    0BL.8  Reserved
S_MODE1             DS    B      Flag byte 1 - reserved
*
S_RES02             DS    0BL.4  Reserved
S_MODE2             DS    B      Flag byte 2
*                                Set ID flags
S_ISUID             EQU   X'08'  Set user ID on execution
S_ISGID             EQU   X'04'  Set group ID on execution
S_ISVTX             EQU   X'02'  Sticky Bit: For executables, look
*                                 first in normal MVS search order
*                                 For directories, deletion rstd
*                                 to owner or superuser.
*                                Owner flags
S_IRWXU1            EQU   X'01'  All permissions for user - part I
S_IRUSR             EQU   X'01'  Read permission
*
S_MODE3             DS    B      Flag byte 3
*                                Owner flags - continued
S_IRWXU2            EQU   X'C0'  All permissions for user - Part II
S_IWUSR             EQU   X'80'  Write permission
S_IXUSR             EQU   X'40'  Search (if a directory) or
*                                execute (otherwise) permission
*                                Group flags
S_IRWXG             EQU   X'38'  All permissions for group
S_IRGRP             EQU   X'20'  Read permission
S_IWGRP             EQU   X'10'  Write permission
S_IXGRP             EQU   X'08'  Search (if a directory) or
*                                execute (otherwise) permission
*                                Other flags
S_IRWXO             EQU   X'07'  All permissions for other
S_IROTH             EQU   X'04'  Read permission
S_IWOTH             EQU   X'02'  Write permission
S_IXOTH             EQU   X'01'  Search (if a directory) or
*                                execute (otherwise) permission
S_MODE#LENGTH       EQU   *-S_MODE Length this structure
** BPXYMODE End
```

## BPXYNREG — Map interface block to vnode registration

```
          BPXYNREG   ,
** BPXYNREG: NREG - LFS Registration routine parameter list
**  Used By:  VRG
NREG                 DSECT ,
NREGBEGIN            DS    0D
*
NREGID               DC    C'NREG'            Eye catcher
NREGLEN              DC    AL2(NREG#LENGTH)   Length of the structure
NREGVER              DC    AL2(NREG#VERSION)  NReg version number
NREGSTYPE            DS    F                  Server Type
NREGSNAMELEN         DS    F                  Length of Server name
NREGSNAME            DS    CL32               Server Name
NREGMAXVNTOKENS      DS    F                  Max # of VNTokens
NREGFLAGS            DS    CL1                Flags
NREGFXHOTC           EQU   X'80'               Exit uses HOTC
NREGNOWAIT           EQU   X'40'               for Quiesced FS
NREGRES01            DS    CL3                Reserved field
NREGENDOFVER1        DS    0F                 End of Version 1
NREGFXEXITNAME       DS    CL8                Exit program name
NREGFXINITPARM       DS    CL8                Init parm for Exit
NREGABENDCODE        DS    F                  Abend Code received
NREGABENDRSN         DS    F                  Abend Reason Code
NREGPFSTYPE          DS    CL8                Dependant PFS
*
*   Constants
*
NREG#LENGTH          EQU   *-NREGBEGIN        Length of NREG
NREG#LENGTHVER1      EQU   NREGENDOFVER1-NREGBEGIN  Length of V1 NREG
NREG#VERSION1        EQU   1                  NReg Version 1
NREG#VERSION2        EQU   2                  NReg Version 2
NREG#VERSION         EQU   NREG#VERSION2      NReg Current Version
*      NRegSType constants
NREGSTYPE#FILE       EQU   1                  File Server type
NREGSTYPE#LOCK       EQU   2                  Lock Server type
NREGSTYPE#FEXP       EQU   3                  File Exporter type
NREGSTYPE#MAX        EQU   3                  Max allowed srvr type
** BPXYNREG End
```

# BPXYOPNF — Map flag values for open

```
          BPXYOPNF   ,
** BPXYOPNF: File status flags
**  Used By: FCT OPN
O_FLAGS             DSECT ,
O_FLAGS1            DS    B      Open flags - byte 1
OPNFHIGH            EQU   X'80'  DO NOT USE THIS BIT!
*                                O_FLAGS must never be < 0
O_FLAGS2            DS    B      Open flags - byte 2
OPNFEXEC            EQU   X'80'  Execute access requested -
*                                authorization required for use
O_FLAGS3            DS    B      Open flags - byte 3
O_ASYNCSIG          EQU   X'02'  An asynchronous signal may occur
O_SYNC              EQU   X'01'  Force synchronous updates
O_FLAGS4            DS    B      Open flags - byte 4
O_CREXCL            EQU   X'C0'  Create file only if non-existent
O_CREAT             EQU   X'80'  Create file
O_EXCL              EQU   X'40'  Exclusive flag
O_NOCTTY            EQU   X'20'  Not a controlling terminal
O_TRUNC             EQU   X'10'  Truncate flag
O_APPEND            EQU   X'08'  Set offset to EOF on write
O_NONBLOCK          EQU   X'04'  Don't block this file
FNDELAY             EQU   X'04'  Don't block this file
O_RDWR              EQU   X'03'  Open for Read and Write
O_RDONLY            EQU   X'02'  Open for Read Only
O_WRONLY            EQU   X'01'  Open for Write Only
O_ACCMODE           EQU   X'03'  Mask for file access modes
O_GETFL             EQU   X'0F'  Mask for file access modes and
*                                file status flags together
OPNF#LENGTH         EQU   *-O_FLAGS Length of this structure
** BPXYOPNF End
```

## BPXYOSS — Map operating system specific information

The numbers of file blocks read and written, along with the number of directory blocks processed, are returned in the OssReadIBC, OssWriteIBC and OssDirIBC, fields of the OSS. On return from the VFS Callable Service API, the block counts present initially in the OSS have been incremented to reflect the counts for this call to the service. Thus, to obtain the numbers of blocks processed on a particular call to a VFS Callable Service API, set the block count fields to zero before calling the service. To accumulate the block counts across a series of calls, pass the same OSS to each, without modifying the count fields

The following OSS fields must be provided by the caller:

**OssId** Contains 'OSS '

**OssLen**
Specifies the length of the OSS structure, OSS#LENGTH.

**OSSReadIBC**
Contains number of blocks read.

**OSSWriteIBC**
Contains number of blocks written.

**OSSDirIBC**
Contains number of directory blocks processed.

```
          BPXYOSS    ,
** BPXYOSS: OSS - Operating System Specific Information
**  Used By: v_ callable services
OSS                 DSECT ,
OSSBEGIN            DS    0D
*
OSSID               DC    C'OSS '            Eye catcher
OSSLEN              DC    AL4(OSS#LENGTH)    Length of the structure
OSSDIRIBC           DS    F                  Directory I/O block cnt
OSSREADIBC          DS    F                  Read I/O block cnt
OSSWRITEIBC         DS    F                  Write I/O block cnt
OSSRSVD             DS    3F                 Reserved
*
*   Constants
*
OSS#LENGTH          EQU   *-OSSBEGIN         Length of OSS
** BPXYOSS End
```

## BPXYPCF — Map pathconf values

BPXYPCF is composed only of EQUates. DSECT= is allowed but ignored.

```
          BPXYPCF    ,
** BPXYPCF:  Command values
**  Used By: FPC PCF
PC_CHOWN_RESTRICTED  EQU 1        _POSIX_CHOWN_RESTRICTED option
PC_LINK_MAX          EQU 2        LINK_MAX option
PC_MAX_CANON         EQU 3        _POSIX_MAX_CANON option
PC_MAX_INPUT         EQU 4        _POSIX_MAX_INPUT option
PC_NAME_MAX          EQU 5        NAME_MAX option
PC_NO_TRUNC          EQU 6        _POSIX_NO_TRUNC option
PC_PATH_MAX          EQU 7        PATH_MAX option
PC_PIPE_BUF          EQU 8        PIPE_BUF option
PC_VDISABLE          EQU 9        _POSIX_VDISABLE option
** BPXYPCF End
```

## BPXYSSTF — Map the response structure for file system status

```
          BPXYSSTF   ,
** BPXYSSTF: file system status response structure
**  Used By: STF STV FTV VSF
SSTF                DSECT ,
SSTFID              DC    C'SSTF'  EBCDIC ID - SSTF (f_OEcbid)
SSTFLEN             DC    A(SSTF#LENGTH)  Length of SSTF (f_OEcblen)
SSTFBLOCKSIZE       DS    F      Block size (f_bsize)
                    DS    F      Reserved
SSTFDBLTOTSPACE     DS    0D     Name of dblword field - total
                    DS    F      Reserved
SSTFTOTALSPACE      DS    F      Total space. The total number of    X
                                 blocks on file system in units of   X
                                 f_frsize (f_blocks)
SSTFDBLUSEDSPACE    DS    0D     Name of dblword field - used
                    DS    F      Reserved
SSTFUSEDSPACE       DS    F      Allocated space in block size units X
                                 (f_OEusedspace)
SSTFDBLFREESPACE    DS    0D     Name of dblword field - free
                    DS    F      Reserved
SSTFFREESPACE       DS    F      Space available to unprivileged     X
                                 users in block size units           X
                                 (f_bavail)
SSTFENDVER1         EQU   *      End of Version 1 SSTF
SSTFFSID            DS    F      File system ID (f_fsid)             X
                                 Set by LFS
SSTFFLAG            DS    0BL.32 Bit mask of f_flag vals
SSTFFLAGB1          DS    XL1    byte 1
SSTFEXPORTED        EQU   X'40'  Filesys is exported                 X
                                 (ST_OEEXPORTED)                     X
                                 Set by LFS
SSTFFLAGB2          DS    XL1    byte 2
SSTFFLAGB3          DS    XL1    byte 3
SSTFFLAGB4          DS    XL1    byte 4
SSTFNOSUID          EQU   X'02'  SetUID/SetGID not supported         X
                                 (ST_NOSUID)                         X
                                 Set by LFS
SSTFRDONLY          EQU   X'01'  Filesys is read only                X
                                 (ST_RDONLY)                         X
                                 Set by LFS
SSTFMAXFILESIZE     DS    0D     Name of dblword field - maximum     X
                                 file size                           X
                                 May be set by LFS
SSTFMAXFILESIZEHW   DS    F      High word of max file size          X
                                 (f_OEmaxfilesizehw)
SSTFMAXFILESIZELW   DS    F      Low word of max file size           X
                                 (f_OEmaxfilesizelw)
                    DS    CL16   Reserved
SSTFENDLFSINFO      EQU   *      End of LFS information
SSTFFRSIZE          DS    F      Fundamental filesystem block size   X
                                 (f_frsize)
                    DS    F      Reserved
SSTFDBLBFREE        DS    0D     Name of dblword field -             X
                                 total number of free blocks
                    DS    F      Reserved
SSTFBFREE           DS    F      Total number of free blocks         X
                                 (f_bfree)
SSTFFILENODES       DS    0CL12  File nodes
SSTFFILES           DS    F      Total number of file nodes          X
                                 in the file system (f_files)
SSTFFFREE           DS    F      Total number of free file nodes     X
                                 (f_ffree)
SSTFFAVAIL          DS    F      Number of free file nodes available X
                                 to unprivileged users (f_favail)
SSTFNAMEMAX         DS    F      Maximum file name len (f_namemax)
SSTFINVARSEC        DS    F      Number of seconds file system       X
```

```
                                     will remain unchanged                X
                                     (f_OEinvarsec)
                      DS    CL20   Reserved
SSTF#LENGTH           EQU   *-SSTF Length of this structure
SSTF#MINLEN           EQU   SSTFENDVER1-SSTF
SSTF#LFSLEN           EQU   SSTFENDLFSINFO-SSTF
** BPXYSSTF End
```

# BPXYSTAT — Map the response structure for stat

```
          BPXYSTAT   ,
** BPXYSTAT: stat system call structure
**  Used By: FST LST STA
STAT                 DSECT ,
ST_BEGIN             DS    0D
*
ST_EYE               DC    C'STAT'  Eye catcher
ST_LENGTH            DC    AL2(STAT#LENGTH)                            X
                                  Length of this structure
ST_VERSION           DC    AL2(ST#VER)                                 X
                                  Version of this structure
ST_MODE              DS    F      File Mode, mapped by BPXYMODE
ST_INO               DS    F      File Serial Number
ST_DEV               DS    F      Device ID of the file
ST_NLINK             DS    F      Number of links
ST_UID               DS    F      User ID of the owner of the file
ST_GID               DS    F      Group ID of the Group of the file
ST_SIZE              DS    0D     File Size in bytes, for regular
*                                 files.  Unspecified, for others
ST_SIZE_H            DS    F      First word of size
ST_SIZE_L            DS    F      Second word of size
ST_ATIME             DS    F      Time of last access
ST_MTIME             DS    F      Time of last data modification
ST_CTIME             DS    F      Time of last file status change
*                                 Time is in seconds since
*                                 00:00:00 GMT, Jan. 1, 1970
ST_RDEV              DS    0F     Device Information
ST_MAJORNUMBER       DS    H      Major number for this file, if it
*                                    is a character special file.
ST_MINORNUMBER       DS    H      Minor number for this file, if it
*                                    is a character special file.
ST_AUDITORAUDIT      DS    F      Area for auditor audit info
ST_USERAUDIT         DS    F      Area for user audit info
ST_BLKSIZE           DS    F      File Block size
ST_CREATETIME        DS    F      File Creation Time
ST_AUDITID           DS    4F     RACF File ID for auditing
ST_RES01             DS    F
ST_CHARSETID         DS    3F     Coded Character Set ID
ST_BLOCKS_D          DS    0D     Double word number - blocks allocated
ST_RES02             DS    F
ST_BLOCKS            DS    F      Number of blocks allocated
ST_GENVALUE          DS    F      General attribute values
ST_REFTIME           DS    F      Reference time
ST_FID               DS    2F     File identifier
ST_FILEFMT           DS    XL1    File Format
ST_RES03             DS    CL19   Reserved for future
*
*   Constants
*
ST#VER               EQU   ST#VER01 Current version
ST#VER01             EQU   1        Version 1 of this structure
STAT#LENGTH          EQU   *-STAT   Length of STAT
ST#LEN               EQU   STAT#LENGTH Length of STAT
** BPXYSTAT End
```

## BPXYVLOK — Map the interface block for v_lockctl

The BPXYVLOK macro maps the interface block to pass locking information via the v_lockctl service.

```
         BPXYVLOK    ,
** BPXYVLOK: VLOK - Vnode Service Byte Range Locking structure
VLOK                DSECT ,
VLOKBEGIN           DS    0D
*
VLOKID              DC    C'VLOK'            Eye catcher
VLOKLEN             DC    AL4(VLOK#LENGTH)   Length of the structure
VLOKLOCKER          DS    0F                 Locker
VLOKSERVERPID       DS    F                    Server's Process ID
VLOKCLIENTPID       DS    F                    Server's Client's PID
VLOKLOCKERTOK       DS    CL8                Locker Token
VLOKCLIENTTID       DS    CL8                Client's Thread ID
VLOKOBJECT          DS    0F                 Object - a locked file
VLOKOBJCLASS        DS    F                    Object Class
VLOKOBJID           DS    0CL12                Object ID         @D1C
VLOKOBJDEV          DS    CL4                    Object Device ID
VLOKOBJFID          DS    CL8                    Object File ID
VLOKOBJTOK          DS    CL8                Object token
VLOKDOS             DS    0F                 DOS file sharing fields
VLOKDOSMODE         DS    CL1                  DOS client mode
VLOKDOSACCESS       DS    CL1                  DOS client access
VLOKBLKLOCKLEN      DS    CL1                VlokBlockingLock length
VLOKSUBFUNCTION     DS    CL1                Internal SubFunction
VLOKRSVD            DS    CL4                Reserved
VLOKVNTOKEN         DS    CL8                Vnode Token
VLOKBRLK            DS    CL24               Lock Information mapped
*                                            by BPXYBRLK
VLOKENDVER1         DS    0F              --- END OF VERSION 1 -------
                    DS    F
VLOKBLOCKINGLOCK    DS    A                  Ptr to Ret Blocking Lock
VLOKUNION           DS    0CL12
VLOKAIOEXT          ORG VLOKUNION    Async Extension
                    DS    F            Reserved
VLOKAIOCB           DS    A            Async Locking Aiocb
VLOKAIOCBLEN        DS    F            Async Aiocb Length
*
VLOKUNLOADLOCKSEXT  ORG VLOKUNION    Unload Locks Extension
                    DS    F            Reserved
VLOKULLOUTLISTPTR   DS    A            Output List Ptr
VLOKULLSUBPOOL      DS    CL1          Storage Subpool
                    DS    CL1
VLOKULLRETWAITERS   DS    CL1          Return Waiters too
*
VLOKPURGEEXT        ORG VLOKUNION    Purge Locks Mask Ext
                    DS    F            Reserved
VLOKPGMASKS         DS    A            VlokObjOwnMasks
VLOKPGMASKSLEN      DS    F            Length of the two masks
*
                    DS    CL12
VLOKENDVER2         DS    0F         --- End of Version 2 -------
*
*   Constants
*
VLOK#LENGTH         EQU   *-VLOKBEGIN        Length of VLOK
VLOK#HFS            EQU   0                  HFS Object Class
VLOK#MVS            EQU   1                  MVS Object Class
```

```
VLOK#LFSESA          EQU   2                  LFS/ESA Object Class
*
*   Constants for V_lockctl commands
*
VLOK#REGLOCKER       EQU   1               Register Locker
VLOK#UNREGLOCKER     EQU   2               Unregister Locker
VLOK#LOCK            EQU   3               Lock object's byte range
VLOK#LOCKWAIT        EQU   4               Lock object's byte range    +
                                           - wait if blocked
VLOK#UNLOCK          EQU   5               UnLock object's byte range
VLOK#QUERY           EQU   6               Query byte range for locks
VLOK#PURGE           EQU   7               Purge all locks for a locker
VLOK#LOCKASY         EQU   8               Lock Asynchronously
VLOK#LOCKCANCEL      EQU   9               Cancel Async Lock
VLOK#UNLOADLOCKS     EQU  10               Unload BRLM Locks
*
*   Constants for UnLoadLocks
*
VLOK#RETWAITERS      EQU   1               Ret Held & Waiters
VLOK#RETALLOBJ       EQU   3               Total UnLoad
*
*   Mask structure for Purge Locks
*
VLOKOBJOWNMASKS      DSECT ,
VLOKOBJECTMASK       DS  0CL16             Object Id Mask
VLOKOBJCLASSMASK     DS  CL4                Object Class
VLOKOBJDEVMASK       DS  CL4                Object Devno (HFS)
VLOKOBJFIDMASK       DS  CL8                Object Fid   (HFS)
VLOKOWNERMASK        DS  0CL16              Owner Id Mask
VLOKLOCKERMASK       DS  0CL8              Locker Mask
VLOKSPIDMASK         DS  CL4                Server PID Mask
VLOKCPIDMASK         DS  CL4                Client PID Mask
VLOKTIDMASK          DS  CL8               Thread Id Mask
*
** BPXYVLOK END
```

## BPXYVOPN — Map the open parameters structure for v_open

The BPXYVOPN macro maps the structure of the Open_Parms parameter of the v_open service.

```
         BPXYVOPN    ,
** BPXYVOPN: V_open Parameters
VOPN                 DSECT ,
VOPNOPENTYPE         DS    F      Type of v_open
VOPNOPENOWNER        DS    CL16   Owner identification
VOPNSHRACCESS        DS    F      Read, Write, or Both
VOPNSHRDENY          DS    F      None, Read, Write, Both
VOPNOPENTOKEN        DS    CL8    Output/Input Open Token
VOPNVNTOKEN          DS    CL8    Output Vnode Token
VOPNFLAGS            DS    F      Open Flags @D1A
                     DS    CL12  @D1C
*
VOPN#LENGTH          EQU   *-VOPN Length of this structure
*
** VopnOpenType Values:
*
OPEN_CREATE_UNCHECKED  EQU  1
OPEN_CREATE_GUARDED    EQU  2
OPEN_CREATE_EXCLUSIVE  EQU  3
OPEN_FILE              EQU  4
OPEN_NLM_SHR           EQU  5
OPEN_UPGRADE           EQU  6
OPEN_DOWNGRADE         EQU  7
*
** VopnShrAccess Values:
*
SHRACC_WRITE       EQU  1
SHRACC_READ        EQU  2
SHRACC_BOTH        EQU  3
*
** VopnShrDeny Values:  @D1C
*
SHRDENY_NONE       EQU  0
SHRDENY_WRITE      EQU  1
SHRDENY_READ       EQU  2
SHRDENY_BOTH       EQU  3
*
** VopnFlags Values:
*
SHRMOD_NONE        EQU  0
SHRMOD_DENY        EQU  1
SHRMOD_ACC         EQU  2
SHRMOD_BOTH        EQU  3
*
** BPXYVOPN End
```

# Appendix C. Callable services examples

These examples follow the rules of reentrancy. They use DSECT=NO and place the variables in the program's dynamic storage DSECT, which is allocated upon entry.

The examples are arranged alphabetically and have references to the mapping macros they use. The declaration for all local variables used in the examples follows the examples.

© Copyright IBM Corp. 1996, 2006 

## Reentrant entry linkage

This entry linkage is reentrant and saves the caller's registers, allocates a save area and dynamic storage, and establishes program and dynamic storage base registers. This entry linkage is paired with the return linkage that is located at the end of the executable program; see "Reentrant return linkage" on page 499.

```
         TITLE 'Alphabetical Invocation of OpenMVS Callable Services'
BPXB5SM6 CSECT ,                     Reentrant entry linkage
BPXB5SM6 AMODE 31
BPXB5SM6 RMODE ANY
         USING *,R15                 Program addressability
@ENTRY0  B     @ENTRY1               Branch around program header
         DROP  R15                   R15 not needed for addressability
         DC    C'BPXB5SM6 - Reentrant callable service examples'
         DS    0H                    Ensure half word boundary
@ENTRY1  STM   R14,R12,12(R13)       Save caller's registers
         LR    R2,R13                Hold address of caller's area
         LR    R3,R1                 Hold parameter register
         LR    R12,R15               R12 program base register
         LA    R11,2048(,R12)        Second program base register
         LA    R11,2048(,R11)        Second program base register
         LA    R9,2048(,R11)         Third program base register
         LA    R9,2048(,R9)          Third program base register
         USING @ENTRY0,R12,R11,R9    Program addressability
         L     R0,@SIZEDAT           Size this program's getmain area
         GETMAIN RU,LV=(0)           Getmain storage
         LR    R13,R1                R13 -> this program's save area
         LA    R10,2048(,R13)        Second getmain base register
         LA    R10,2048(,R10)        Second getmain base register
         USING @STORE,R13,R10        Getmain addressability
         ST    R2,@BACK              Save caller's save area pointer
         ST    R13,8(,R2)            Give caller our save area
         LR    R1,R3                 Restore parameter register
@ENTRY2  EQU   *  *  *  *  *  *  *   End of the entry linkage code
         SPACE ,
PSEUDO   EQU   *                     Dummy label used throughout
```

## BPX1VCR, BPX4VCR (v_create) example

The following code creates a new and empty regular file named **fnewprots** in a previously looked-up directory whose vnode token is in DIRVNODETOK with user read-execute, group write, other read-execute permissions. For the callable service, see "v_create (BPX1VCR, BPX4VCR) — Create a file" on page 270. For the data structures, see "BPXYATTR — Map file attributes for v_ system calls" on page 445, "BPXYFTYP — File type definitions" on page 451, "BPXYMODE — Map the mode constants of the file services" on page 466 and "BPXYOSS — Map operating system specific information" on page 469.

```
MVC   BUFFERA(9),=CL9'fnewprots'
MVC   BUFLENA,=F'9'
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
MVC   ATTRSTOR,ATTR         Initialize BPXYATTR area
XC    S_MODE,S_MODE         Clear mode
MVI   S_TYPE,FT_REGFILE     Set regular file type
MVI   S_MODE2,S_IRUSR       Read-execute/write/read-execute
MVI   S_MODE3,S_IXUSR+S_IWGRP+S_IROTH+S_IXOTH
LA    R5,ATTRSTOR           Address and
USING ATTR,R5               map BPXYATTR area
MVC   ATTRMODE,S_MODE       Move mode data to attribute      +
                            structure
DROP  R5
SPACE ,
CALL  BPX1VCR,              Create a file                    +
      (DIRVNODETOK,         Input: Directory vnode token     +
      OSSSTOR,              Input/output: BPXYOSS            +
      BUFLENA,              Input: New file name length      +
      BUFFERA,              Input: New file name             +
      =A(ATTR#LENGTH),      Input: BPXYATTR length           +
      ATTRSTOR,             Input/output: BPXYATTR           +
      VNODETOK,             Output: New file Vnode token     +
      RETVAL,               Return value: 0 or -1            +
      RETCODE,              Return code                      +
      RSNCODE),             Reason code                      +
      VL,MF=(E,PLIST)       ---------------------------------
```

## BPX1VSF, BPX4VSF (v_fstatfs) example

The following code obtains the status of the file system containing the previously looked-up file whose vnode token is in VNODETOK. For the callable service, see "v_fstatfs (BPX1VSF, BPX4VSF) — Return file system status" on page 279. For the data structures, see "BPXYSSTF — Map the response structure for file system status" on page 471, and "BPXYOSS — Map operating system specific information" on page 469.

```
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
SPACE ,
CALL  BPX1VSF,              Obtain file system status        +
      (VNODETOK,            Input: Vnode token               +
      OSSSTOR,              Input/output: BPXYOSS            +
      =A(SSTF#LENGTH),      Input: BPXYSSTF length           +
      SSTF,                 Output: BPXYSSTF                 +
      RETVAL,               Return value: 0 or -1            +
      RETCODE,              Return code                      +
      RSNCODE),             Reason code                      +
      VL,MF=(E,PLIST)       ----------------------------------
```

## BPX1VGT, BPX4VGT (v_get) example

The following code obtains a vnode token for the file or directory specified via the input FID, residing within the mounted file system represented by the input VFS token. Previously, the FID might have been obtained from an attribute structure returned by v_lookup, and the VFS token via v_rpn. For the callable service, see “v_get (BPX1VGT, BPX4VGT) — Convert an FID to a vnode Token” on page 282. For the data structure, see “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   OSSSTOR,OSS          Initialize BPXYOSS area
SPACE ,
CALL  BPX1VGT,             Obtain a Vnode token            +
      (VFSTOK,             Input: VFS token                +
      OSSSTOR,             Input/output: BPXYOSS           +
      FID,                 Input: File identifier          +
      VNODETOK,            Output: Vnode token for file    +
      RETVAL,              Return value: 0 or -1           +
      RETCODE,             Return code                     +
      RSNCODE),            Reason code                     +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VGA, BPX4VGA (v_getattr) example

The following code obtains the status of a file whose previously looked-up vnode token is in VNODETOK. For the callable service, see “v_getattr (BPX1VGA, BPX4VGA) — Get the attributes of a file” on page 285. For the data structures, see “BPXYATTR — Map file attributes for v_ system calls” on page 445 and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   OSSSTOR,OSS            Initialize BPXYOSS area
SPACE ,
CALL  BPX1VGA,               Obtain file status                  +
      (VNODETOK,             Input: Vnode token                  +
      OSSSTOR,               Input/output: BPXYOSS               +
      =A(ATTR#LENGTH),       Input: BPXYATTR length              +
      ATTRSTOR,              Output: BPXYATTR                    +
      RETVAL,                Return value: 0 or -1               +
      RETCODE,               Return code                         +
      RSNCODE),              Reason code                         +
      VL,MF=(E,PLIST)        ----------------------------------
```

## BPX1VLN, BPX4VLN (v_link) example

The following code creates a new name, **dataproc.next**, for a previously looked-up file whose vnode token is in VNODETOK in a previously looked-up directory whose vnode token is in DIRVNODETOK. For the callable service, see "v_link (BPX1VLN, BPX4VLN) — Create a link to a file" on page 288. For the data structure, see "BPXYOSS — Map operating system specific information" on page 469.

```
MVC   OSSSTOR,OSS          Initialize BPXYOSS area
MVC   BUFLENA,=F'13'
MVC   BUFFERA(13),=CL13'dataproc.next'
SPACE ,
CALL  BPX1VLN,             Create a link to a file          +
      (VNODETOK,           Input: File vnode token          +
      OSSSTOR,             Input/output: BPXYOSS            +
      BUFLENA,             Input: Name length: new name     +
      BUFFERA,             Input: New file name             +
      DIRVNODETOK,         Input: Vnode for directory       +
      RETVAL,              Return value: 0 or -1            +
      RETCODE,             Return code                      +
      RSNCODE),            Reason code                      +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VLO, BPX4VLO (v_lockctl) example

The following code requests a read lock on the file with the input DEVNO and FID. The locker has been previously registered as LOCKERTOK, and the request is for client thread CTID. The byte-range to lock is from the start of the file to byte 10. For the callable service, see "v_lockctl (BPX1VLO, BPX4VLO) — Lock a file" on page 292. For the data structures, see "BPXYOSS — Map operating system specific information" on page 469, "BPXYVLOK — Map the interface block for v_lockctl" on page 474, and "BPXYBRLK — Map the byte range lock request for fcntl" on page 448.

```
MVC   OSSSTOR,OSS                  Initialize BPXYOSS area
MVC   VLOKSTOR,VLOK                Initialize BPXYVLOK area
XC    BRLK(BRLK#LENGTH),BRLK       Initialize BPXYBRLK
MVI   L_TYPE,F_RDLCK               Lock type = read
MVI   L_WHENCE,SEEK_SET            Whence = start of file
MVC   L_LEN,=F'10'                 Len = 10 bytes
LA    R5,VLOKSTOR                  Address and
USING VLOK,R5                      map BPXYVLOK area
MVC   VLOKLOCKERTOK,LOCKERTOK      Move Locker Token to VLOK
MVC   VLOKCLIENTTID,CTID           Move Thread ID to VLOK
MVI   VLOKOBJCLASS,VLOK#HFS        Object Class = HFS
MVC   VLOKOBJDEV,DEVNO             Move Device ID
MVC   VLOKOBJFID,FID               Move File ID
MVC   VLOKBRLK,BRLK                Move Lock info to VLOK
DROP  R5
SPACE ,
CALL  BPX1VLO,             Create a link to a file          +
      (OSSSTOR,            Input/output: BPXYOSS            +
      =A(VLOK#LOCK),       Input: Command = Lock            +
      =A(VLOK#LENGTH),     Input: BPXYVLOK length           +
      VLOKSTOR,            Input/output: BPXYVLOK           +
      RETVAL,              Return value: 0 or -1            +
      RETCODE,             Return code                      +
      RSNCODE),            Reason code                      +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VLK, BPX4VLK (v_lookup) example

The following code looks up a file named **fnewprots** in a previously looked-up directory whose vnode token is in DIRVNODETOK. In the returned attribute structure, ATTRFID contains the file identifier (FID) which can be used to obtain a vnode token for the file, subsequent to freeing the vnode token returned by v_lookup via v_rel. For the callable service, see “v_lookup (BPX1VLK, BPX4VLK) — Look up a file or directory” on page 303. For the data structures, see “BPXYATTR — Map file attributes for v_ system calls” on page 445 and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   BUFFERA(9),=CL9'fnewprots'
MVC   BUFLENA,=F'9'
MVC   OSSSTOR,OSS          Initialize BPXYOSS area
SPACE ,
CALL  BPX1VLK,             Lookup a file                    +
      (DIRVNODETOK,        Input: Directory Vnode token     +
      OSSSTOR,             Input/output: BPXYOSS            +
      BUFLENA,             Input: File name length          +
      BUFFERA,             Input: File name                 +
      =A(ATTR#LENGTH),     Input: BPXYATTR length           +
      ATTRSTOR,            Output: BPXYATTR                 +
      VNODETOK,            Output: File Vnode token         +
      RETVAL,              Return value: 0 or -1            +
      RETCODE,             Return code                      +
      RSNCODE),            Reason code                      +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VMK, BPX4VMK (v_mkdir) example

The following code creates a new and empty directory named **newprots** in a previously looked-up directory whose vnode token is in DIRVNODETOK with user read-execute, group write, other read-execute permissions. For the callable service, see “v_mkdir (BPX1VMK, BPX4VMK) — Create a directory” on page 307. For the data structures, see “BPXYATTR — Map file attributes for v_ system calls” on page 445, “BPXYFTYP — File type definitions” on page 451, “BPXYMODE — Map the mode constants of the file services” on page 466 and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   BUFFERA(8),=CL8'newprots'
MVC   BUFLENA,=F'8'
MVC   OSSSTOR,OSS          Initialize BPXYOSS area
MVC   ATTRSTOR,ATTR        Initialize BPXYATTR area
XC    S_MODE,S_MODE        Clear mode
MVI   S_TYPE,FT_DIR        Set directory file type
MVI   S_MODE2,S_IRUSR      Read-execute/write/read-execute
MVI   S_MODE3,S_IXUSR+S_IWGRP+S_IROTH+S_IXOTH
LA    R5,ATTRSTOR          Address and
USING ATTR,R5              map BPXYATTR area
MVC   ATTRMODE,S_MODE      Move mode data to attribute      +
                           structure
DROP  R5
SPACE ,
CALL  BPX1VMK,             Make a directory                 +
      (DIRVNODETOK,        Input: Directory vnode token     +
      OSSSTOR,             Input/output: BPXYOSS            +
      BUFLENA,             Input: New directory name length +
      BUFFERA,             Input: New directory name        +
      =A(ATTR#LENGTH),     Input: BPXYATTR length           +
      ATTRSTOR,            Input/output: BPXYATTR           +
      DIRVNODETOK2,        Output: New directory Vnode token +
      RETVAL,              Return value: 0 or -1            +
      RETCODE,             Return code                      +
      RSNCODE),            Reason code                      +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VPC, BPX4VPC (v_pathconf) example

The following code obtains current values of configurable options of a file or directory whose vnode token is in VNODETOK. For the callable service, see “v_pathconf (BPX1VPC, BPX4VPC) — Get pathconf information for a directory or file” on page 319. For the data structures, see “BPXYATTR — Map file attributes for v_ system calls” on page 445, “BPXYPCF — Map pathconf values” on page 470, and “BPXYOSS — Map operating system specific information” on page 469.

```
         MVC   OSSSTOR,OSS          Initialize BPXYOSS area
         MVC   ATTRSTOR,ATTR        Initialize BPXYATTR area
         CALL  BPX1VPC,                                            +
               (VNODETOK,           Input: File Vnode token        +
               OSSSTOR,             Input/output: BPXYOSS          +
               =A(PCFG#LEN),        Input: PCFG length             +
               BUFFERA,             Output: PCFG buffer area       +
               =A(ATTR#LENGTH),     Input: BPXYATTR length         +
               ATTRSTOR,            Output: BPXYATTR               +
               RETVAL,              Return value: PCFG len or -1   +
               RETCODE,             Return code                    +
               RSNCODE),            Reason code                    +
               VL,MF=(E,PLIST)      ----------------------------------
Note: PCFG#LEN is defined as follows. It is not constant in the BPXYPCF macro.

               BPXYPCF              pathconf
      PCFG#LEN EQU                  *-PCFG
```

## BPX1VRW, BPX4VRW (v_rdwr) example

The following code writes data to a previously looked-up file whose vnode token is in VNODETOK, from the buffer provided. Control is not to be returned to the calling program until the data have been written, and authorization to write to the file is to be verified. For the callable service, see “v_rdwr (BPX1VRW, BPX4VRW) — Read from and write to a file” on page 322. For the data structures, see “BPXYFUIO — Map file system user I/O block” on page 452, and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   OSSSTOR,OSS          Initialize BPXYOSS area
MVC   FUIOSTOR,FUIO        Initialize BPXYFUIO area
LA    R5,FUIOSTOR          Address and
USING FUIO,R5              map BPXYFUIO area
LA    R15,BUFFERA          Set address of buffer
ST    R15,FUIOBUFFERADDR   to be written in FUIO
OI    FUIOFLAGS,FUIO#WRT+FUIOSYNC+FUIOCHKACC                  +
                           Indicate write action, write       +
                           to medium before return,           +
                           and check authorization
MVC   FUIOCURSOR,=F'100'   Set offset to begin writing
MVC   FUIOIBYTESRW,=F'80'  Max number of bytes to write
DROP  R5
SPACE ,
CALL  BPX1VRW,             Read or write data to or from file+
      (VNODETOK,           Input: Vnode token for file       +
      OSSSTOR,             Input/output: BPXYOSS             +
      FUIOSTOR,            Input/output: BPXYFUIO            +
      =A(ATTR#LENGTH),     Input: BPXYATTR length            +
      ATTRSTOR,            Output: BPXYATTR                  +
      RETVAL,              Return value: 0, -1 or char count +
      RETCODE,             Return code                       +
      RSNCODE),            Reason code                       +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VRD, BPX4VRD (v_readdir) example

The following code reads the multiple entries from a directory, whose previously looked-up vnode token is in DIRVNODETOK, into the buffer provided. FUIOCURSOR, set to zero by the BPXYFUIO macro, indicates that the system is to begin reading with the first entry in the directory. Presuming that this is the first time the directory is read, FUIOCHKACC is set, in order to verify access authority. For the callable service, see “v_readdir (BPX1VRD, BPX4VRD) — Read entries from a directory” on page 326. For the data structures, see “BPXYDIRE — Map directory entries for readdir” on page 449, “BPXYFUIO — Map file system user I/O block” on page 452, and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
MVC   FUIOSTOR,FUIO         Initialize BPXYFUIO area
LA    R5,FUIOSTOR           Address and
USING FUIO,R5               map BPXYFUIO area
LA    R15,BUFFERA           Set address of buffer
ST    R15,FUIOBUFFERADDR    for directory data in FUIO
MVC   FUIOIBYTESRW,=F'1023' Max number of bytes to read
OI    FUIOFLAGS,FUIOCHKACC  Check authorization
DROP  R5
SPACE ,
CALL  BPX1VRD,              Read directory entries           +
      (DIRVNODETOK,         Input: Vnode token for directory  +
      OSSSTOR,              Input/output: BPXYOSS            +
      FUIOSTOR,             Input/output: BPXYFUIO           +
      RETVAL,               Return value: 0, -1 or char count +
      RETCODE,              Return code                      +
      RSNCODE),             Reason code                      +
      VL,MF=(E,PLIST)       ----------------------------------
```

## BPX1VRA, BPX4VRA (v_readlink) example

The following code reads the contents of a previously looked up symbolic link file whose vnode token is in VNODETOK, into the buffer provided. This will be the pathname that was specified when the symbolic link was defined. For the callable service, see “v_readlink (BPX1VRA, BPX4VRA) — Read a symbolic link” on page 330. For the data structures, see “BPXYFUIO — Map file system user I/O block” on page 452, and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
MVC   FUIOSTOR,FUIO         Initialize BPXYFUIO area
LA    R5,FUIOSTOR           Address and
USING FUIO,R5               map BPXYFUIO area
LA    R15,BUFFERA           Set address of buffer
ST    R15,FUIOBUFFERADDR    for symlink in FUIO
MVC   FUIOIBYTESRW,=F'1023' Max number of bytes to read
DROP  R5
SPACE ,
CALL  BPX1VRA,              Read the value of a symbolic link +
      (VNODETOK,            Input: Vnode token for file       +
      OSSSTOR,              Input/output: BPXYOSS             +
      FUIOSTOR,             Input/output: BPXYFUIO            +
      RETVAL,               Return value: 0, -1 or char count +
      RETCODE,              Return code                       +
      RSNCODE),             Reason code                       +
      VL,MF=(E,PLIST)       ----------------------------------
```

## BPX1VRG, BPX4VRG (v_reg) example

The following code registers a file server named **File server**, and accepts the default maximum number of vnode tokens by allowing NREGMAXVNTOKENS to remain zero. For the callable service, see “v_reg (BPX1VRG, BPX4VRG) — Register a process as a server” on page 333. For the data structure, see “BPXYNREG — Map interface block to vnode registration” on page 467.

```
MVC   NREGSTOR,NREG         Initialize BPXYNREG area
 LA    R5,NREGSTOR           Address and
 USING NREG,R5               map BPXYNREG area
 MVC   NREGSTYPE,=A(NREGSTYPE#FILE)     Set server type
 MVC   NREGSNAME(11),=CL11'File server' Set server name
 MVC   NREGSNAMELEN,=F'11'
 DROP  R5
 SPACE ,
 CALL  BPX1VRG,              Register server                  +
       (=A(NREG#LENGTH),     Input: BPXYNREG length           +
       NREGSTOR,             Input/output: BPXYNREG           +
       RETVAL,               Return value: 0 or -1            +
       RETCODE,              Return code                      +
       RSNCODE),             Reason code                      +
       VL,MF=(E,PLIST)       ----------------------------------
```

## BPX1VRL, BPX4VRL (v_rel) example

The following code releases a vnode token, specified in VNODETOK. For the callable service, see “v_rel (BPX1VRL, BPX4VRL) — Release a vnode token” on page 337. For the data structure, see “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   OSSSTOR,OSS          Initialize BPXYOSS area
SPACE ,
CALL  BPX1VRL,             Release Vnode token              +
      (VNODETOK,           Input: Vnode token               +
      OSSSTOR,             Input/output: BPXYOSS            +
      RETVAL,              Return value: 0 or -1            +
      RETCODE,             Return code                      +
      RSNCODE),            Reason code                      +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VRM, BPX4VRM (v_remove) example

The following code deletes the file named **newprots** located in a previously looked-up directory whose vnode token is in DIRVNODETOK. For the callable service, see “v_remove (BPX1VRM, BPX4VRM) — Remove a link to a file” on page 339. For the data structure, see “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   BUFFERA(8),=CL8'newprots'
MVC   BUFLENA,=F'8'
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
SPACE ,
CALL  BPX1VRM,              Remove a file                    +
      (DIRVNODETOK,         Input: Directory vnode token     +
      OSSSTOR,              Input/output: BPXYOSS            +
      BUFLENA,              Input: File name length          +
      BUFFERA,              Input: File name                 +
      RETVAL,               Return value: 0 or -1            +
      RETCODE,              Return code                      +
      RSNCODE),             Reason code                      +
      VL,MF=(E,PLIST)       ----------------------------------
```

## BPX1VRN, BPX4VRN (v_rename) example

The following code changes the name of a file from **samantha** in a previously looked-up directory whose vnode token is in DIRVNODETOK to **sam** in a previously looked-up directory whose vnode token is in DIRVNODETOK2. For the callable service, see "v_rename (BPX1VRN, BPX4VRN) — Rename a file or directory" on page 343. For the data structure, see and "BPXYOSS — Map operating system specific information" on page 469.

```
MVC   BUFFERA(08),=CL08'samantha' Old name
MVC   BUFLENA,=F'08'
MVC   BUFFERB(03),=CL03'sam'      New name
MVC   BUFLENB,=F'03'
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
SPACE ,
CALL  BPX1VRN,              Rename a file                     +
      (DIRVNODETOK,         Input: Old directory vnode token  +
      OSSSTOR,              Input/output: BPXYOSS             +
      BUFLENA,              Input: Old name length            +
      BUFFERA,              Input: Old name                   +
      DIRVNODETOK2,         Input: New directory Vnode token  +
      BUFLENB,              Input: New name length            +
      BUFFERB,              Input: New name                   +
      RETVAL,               Return value: 0 or -1             +
      RETCODE,              Return code                       +
      RSNCODE),             Reason code                       +
      VL,MF=(E,PLIST)       ----------------------------------
```

## BPX1VRE, BPX4VRE (v_rmdir) example

The following code deletes the directory named **newprots** located in a previously looked-up directory whose vnode token is in DIRVNODETOK. For the callable service, see "v_rmdir (BPX1VRE, BPX4VRE) — Remove a directory" on page 347. For the data structure, see "BPXYOSS — Map operating system specific information" on page 469.

```
MVC   BUFFERA(8),=CL8'newprots'
MVC   BUFLENA,=F'8'
MVC   OSSSTOR,OSS            Initialize BPXYOSS area
SPACE ,
CALL  BPX1VRE,               Remove a directory              +
      (DIRVNODETOK,          Input: Directory vnode token    +
      OSSSTOR,               Input/output: BPXYOSS           +
      BUFLENA,               Input: Directory name length    +
      BUFFERA,               Input: Directory name           +
      RETVAL,                Return value: 0 or -1           +
      RETCODE,               Return code                     +
      RSNCODE),              Reason code                     +
      VL,MF=(E,PLIST)        ----------------------------------
```

## BPX1VRP, BPX4VRP (v_rpn) example

The following code resolves (i.e. looks up) the fully qualified path named **/usr/fnewprots**. For the callable service, see “v_rpn (BPX1VRP, BPX4VRP)) — Resolve a pathname” on page 350. For the data structures, see “BPXYATTR — Map file attributes for v_ system calls” on page 445, “BPXYMNTE — Map response and element structure of w_getmnte” on page 463, and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   BUFFERA(14),=CL14'/usr/fnewprots'
MVC   BUFLENA,=F'14'
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
SPACE ,
CALL  BPX1VRP,              Resolve a pathname              +
      (OSSSTOR,             Input/output: BPXYOSS           +
      BUFLENA,              Input:  Path name length        +
      BUFFERA,              Input:  Path name               +
      VFSTOK,               Output: VFS token               +
      VNODETOK,             Output: Vnode token             +
      =A(MNTEH#LENGTH+MNTE#LENGTH), Input: MNTE length      +
      MNTE,                 Output: BPXYMNTE                +
      =A(ATTR#LENGTH),      Input:  BPXYATTR length         +
      ATTRSTOR,             Output: BPXYATTR                +
      RETVAL,               Return value: 0 or -1           +
      RETCODE,              Return code                     +
      RSNCODE),             Reason code                     +
      VL,MF=(E,PLIST)       ----------------------------------
```

## BPX1VSA, BPX4VSA (v_setattr) example

The following code sets attributes for a previously looked-up file whose vnode token is in VNODETOK. The owning user and group ids are changed, the file change time is set to the current time and the user read-execute, group write, other read-execute permissions are set. For the callable service, see "v_setattr (BPX1VSA, BPX4VSA) — Set the attributes of a file" on page 354. For the data structures, see "BPXYATTR — Map file attributes for v_ system calls" on page 445, "BPXYMODE — Map the mode constants of the file services" on page 466 and "BPXYOSS — Map operating system specific information" on page 469.

```
MVC   OSSSTOR,OSS          Initialize BPXYOSS area
MVC   ATTRSTOR,ATTR        Initialize BPXYATTR area
XC    S_MODE,S_MODE        Clear mode
MVI   S_MODE2,S_IRUSR      Read-execute/write/read-execute
MVI   S_MODE3,S_IXUSR+S_IWGRP+S_IROTH+S_IXOTH
LA    R5,ATTRSTOR          Address and
USING ATTR,R5              map BPXYATTR area
MVC   ATTRMODE,S_MODE      Move mode data to attribute      +
                           structure
MVC   ATTRUID,=F'7'        Specify new UID
MVC   ATTRGID,=F'77'       Specify new GID
OI    ATTRSETFLAGS1,ATTRMODECHG+ATTROWNERCHG                +
                           Flag UID and GID changes
OI    ATTRSETFLAGS2,ATTRCTIMECHG+ATTRCTIMETOD               +
                           Set change time to current time
DROP  R5
SPACE ,
CALL  BPX1VSA,             Set file attributes              +
      (VNODETOK,           Input: File vnode token          +
      OSSSTOR,             Input/output: BPXYOSS            +
      =A(ATTR#LENGTH),     Input: BPXYATTR length           +
      ATTRSTOR,            Input/output: BPXYATTR           +
      RETVAL,              Return value: 0 or -1            +
      RETCODE,             Return code                      +
      RSNCODE),            Reason code                      +
      VL,MF=(E,PLIST)      ----------------------------------
```

## BPX1VSY, BPX4VSY (v_symlink) example

The following code creates an external symbolic link to data set **MY.DATASET**, the ″pathname″, for link name **mydataset**, the ″link name″, which is contained in a previously looked-up directory whose vnode token is in DIRVNODETOK. For the callable service, see “v_symlink (BPX1VSY, BPX4VSY) — Create a symbolic link” on page 361. For the data structures, see “BPXYATTR — Map file attributes for v_ system calls” on page 445, “BPXYFTYP — File type definitions” on page 451, “BPXYMODE — Map the mode constants of the file services” on page 466 and “BPXYOSS — Map operating system specific information” on page 469.

```
MVC   BUFFERA(09),=CL09'mydataset'  Name of link
MVC   BUFLENA,=F'09'
MVC   BUFFERB(10),=CL10'MY.DATASET' Contents of link
MVC   BUFLENB,=F'10'
MVC   OSSSTOR,OSS           Initialize BPXYOSS area
MVC   ATTRSTOR,ATTR         Initialize BPXYATTR area
LA    R5,ATTRSTOR           Address and
USING ATTR,R5               map BPXYATTR area
OI    ATTRVISIBLE,ATTREXTLINK                               +
                            Flag as external link
DROP  R5
SPACE ,
CALL  BPX1VSY,              Create a symbolic link          +
      (DIRVNODETOK,         Input: Directory vnode token    +
      OSSSTOR,              Input/output: BPXYOSS           +
      BUFLENA,              Input: Link name length         +
      BUFFERA,              Input: Link name                +
      BUFLENB,              Input: Pathname length          +
      BUFFERB,              Input: Path name                +
      =A(ATTR#LENGTH),      Input: BPXYATTR length          +
      ATTRSTOR,             Input/output: BPXYATTR          +
      RETVAL,               Return value: 0 or -1           +
      RETCODE,              Return code                     +
      RSNCODE),             Reason code                     +
      VL,MF=(E,PLIST)       ----------------------------------
```

## Reentrant return linkage

```
         XR    R15,R15               Zero return code
         L     R0,@SIZEDAT           Size this program's getmain area
         LR    R1,R13                R1 -> this program's getmain area
         L     R13,@BACK             R2 -> caller's save area
         DROP  R13
         FREEMAIN RU,LV=(0),A=(1)
         L     R14,12(,R13)          Restore caller's R14
         LM    R0,R12,20(R13)        Restore caller's R0-R12
         BSM   0,R14                 Branch back to caller
         SPACE ,  * * * * * * * * * *.* Program constants * * * * * * *
@SIZEDAT DC    A(@ENDSTOR-@STORE)    Size of this getmain storage
MNTEL    DC    A(MNTE#LENGTH+MNTEH#LENGTH)
*                                    Length of MNTEH and 1 MNTE area
         SPACE ,
PRIMARYALET  DC   A(0)               Primary ALET
```

```
* * * * * * * * * * * * * * * * * * *.* OpenMvs EQUates * * * * * * * *
* With EQUate only macros, DSECT= is allowed but is ignored
         BPXYCONS LIST=NO            OpenMVS constants
         BPXYERNO LIST=NO            Errno, Errnojr constants
         BPXYFTYP ,                  File type constants
         BPXYSEEK ,                  lseek constants
* * * * * * * * * * * * * * * * * * *.* Structures requiring a USING  *
* * * * * * * * * * * * * * * * * * *.* Standard linkage save area  * *
@STORE   DSECT ,
@SAVE00  DS    0D                    Standard 72-byte save area
         DS    A
@BACK    DS    A                     Back to caller's save area
@FORWARD DS    A                     Forwards to callee's save area
         DS    15A                   Regs 14,15,0-12
* * * * * * * * * * * * * * * * * * *.* Structures initializing storage
         BPXYATTR DSECT=NO           Attributes for Vnode services
         BPXYFUIO DSECT=NO           User I/O block
         BPXYNREG DSECT=NO           Registration structure for       +
                                     for Vnode services
         BPXYOSS  DSECT=NO           Operating system info for Vnode   +
                                     services
         BPXYVLOK DSECT=NO           Lock request info for v_lockctl
         SPACE 2  * * * * * * * * * *.* Getmain for mappings * * * * *
ATTRSTOR DS    CL(ATTR#LENGTH)       BPXYATTR storage
BRLKA    DS    A                     ->BPXYBRLK
         BPXYBRLK DSECT=NO           Byte range locking for v_lockctl
FUIOSTOR DS    CL(FUIO#LENGTH)       BPXYFUIO storage
         BPXYMODE DSECT=NO           Mode constants
         BPXYMNTE DSECT=NO           Get mount entries for v_rpn
@STORE   DSECT ,                     because MNTE has DSECT in it
NREGSTOR DS    CL(NREG#LENGTH)       BPXYNREG storage
OSSSTOR  DS    CL(OSS#LENGTH)        BPXYOSS storage
VLOKSTOR DS    CL(VLOK#LENGTH)       BPXYVLOK storage
         BPXYSSTF DSECT=NO           Response data for v_fstatfs
* * * * * * * * * * * * * * * * * * *.* Program getmain variables * * *
              DS    0D
ARGCNT        DS    F                Argument count
ARGLLST       DS    3A               Argument lengths list
ARGSLST       DS    3A               Arguments list
BUFA          DS    F                ->buffer
BUFFERA       DS    CL1024           Utility buffer A, length 1024
BUFLENA       DS    F                Number of bytes used in buffer A
BUFFERB       DS    CL1024           Utility buffer B, length 1024
BUFLENB       DS    F                Number of bytes used in buffer B
BUFW          DS    F                Number of words used in BUF
COMMAND       DS    F                User defined command
CTID          DS    CL8              Client Thread ID
DEVNO         DS    F                Device ID
DIRECTDES     DS    F                Directory descriptor
DIRVNODETOK   DS    2F               Directory Vnode token
DIRVNODETOK2  DS    2F               Directory Vnode token
ENVCNT        DS    F                Number of environment variables
ENVLENS       DS    F                Length of environment variables
ENVPARMS      DS    F                Environment variables
EVENTLIST     DS    A                Event list for thread posting
EXITRTNA      DS    A                Exit routine address
EXITPLA       DS    A                Exit Parm list address
FID           DS    2F               File identifier (FID)
FILEDESC      DS    F                File descriptor
FILEDES2      DS    F                File descriptor
FSNAME        DS    CL44             File system name
FSTYPE        DS    CL8              File system type
GRNAMELN      DS    F                Group name length
GROUP         DS    F                Group
GROUPCNT      DS    F                Group count
GROUPID       DS    F                Group ID (PID of group leader)
```

```
GRPGMNAME     DS    CL8              Group program name
INTMASK       DS    XL8              Signal mask
INITRTNA      DS    A                ->Initialization routine
INTRSTATE     DS    A                Interrupt state
INTRTYPE      DS    A                Interrupt type
LOCKERTOK     DS    CL8              Locker Token
NANOSECONDS   DS    F                Count of nanoseconds
NCATCHER      DS    A                New catcher
NEWFLAGS      DS    F                New flags
NEWHANDL      DS    F                New Handler
NEWLEN        DS    XL8              Length file
NEWMASK       DS    XL8              New mask for signals
NEWMASKA      DS    A                ->New mask
NEWTIMES      DS    D                New access/modification time
OCATCHER      DS    A                Old catcher
OFFSET        DS    CL8              File offset
OLDHANDL      DS    F                Old handler
OLDFLAGS      DS    F                Old flags
OLDMASK       DS    CL8              Old signal mask
OLDMASKA      DS    A                ->Old mask
OPTIONS       DS    F                Options
PGMNAME       DS    CL8              Program name
PGMNAMEL      DS    F                Length PGMNAME
PLIST         DS    13A              Max number of parms
PROCID        DS    F                Process ID
PROCTOK       DS    F                Relative process number
READFD        DS    F                File descriptor - input file
REFPT         DS    F                File reference point
RETCODE       DS    F                Return code (ERRNO)
RETVAL        DS    F                Return value (0, -1 or other)
RSNCODE       DS    F                Reason code (ERRNOJR)
SECONDS       DS    F                Time in seconds
SIGNAL        DS    A                Signal
SIGNALREG     DS    A                Signal registration, user data
SIGNALOPTIONS DS    A                Signal options
SIGRET        DS    CL8              Signal return mask
SIRTNA        DS    A                Signal interrupt routine
STATFLD       DS    A                Status field
STATUS        DS    F                Status
STATUSA       DS    A                ->STATUS
TERMMASK      DS    XL8              Signal termination mask
THID          DS    XL8              Thread ID
USERID        DS    F                User ID
USERNAME      DS    CL8              User name
USERNLEN      DS    F                Length USERNAME
USERWORD      DS    F                User data
WAITMASK      DS    F                Mast for signal waits
WRITEFD       DS    F                File descriptor - output file
VFSTOK        DS    2F               VFS token
VNODETOK      DS    2F               Vnode token
              SPACE ,
@ENDSTOR      EQU   *                End of getmain storage
         SPACE 3  * * * * * * * * * *.* Register equates  * * * * * * *
              SPACE ,
R0            EQU   0
R1            EQU   1                Parameter list pointer
R2            EQU   2
R3            EQU   3
R4            EQU   4
R5            EQU   5
R6            EQU   6
R7            EQU   7
R8            EQU   8
R9            EQU   9                Third program base register
R10           EQU   10               Second getmain storage register
R11           EQU   11               Second program base register
R12           EQU   12               Program base register
```

```
R13          EQU   13               Savearea & getmain storage base
R14          EQU   14               Return address
R15          EQU   15               Branch location
             END
```

# Appendix D. Interface structures for C language servers and clients

Two header files are described in this appendix. The first header file is for the VFS callable services API (v_); the second is for the PFS interface (vfs_ and vn_).

These headers are placed in sys1.SFOMHDRS when z/OS UNIX is installed.

## BPXYVFSI—VFS interface definitions

```
 /*****START OF SPECIFICATIONS****************************************
*
*    $MAC (BPXYVFSI) COMP(SCPX4) PROD(FOM):
*
*01* MACRO NAME: BPXYVFSI
*
*01* DSECT NAME: N/A
*
*01* DESCRIPTIVE NAME: Virtual File System Interface Definition for C
*
*02*   ACRONYM: N/A
*                                                                    */
 /*01* PROPRIETARY STATEMENT=                                        */
 /***PROPRIETARY_STATEMENT********************************************/
 /*                                                                  */
 /*                                                                  */
 /* LICENSED MATERIALS - PROPERTY OF IBM                             */
 /* THIS MACRO IS "RESTRICTED MATERIALS OF IBM"                      */
 /* 5694-A01 (C) COPYRIGHT IBM CORP. 1993, 2003                      */
 /*                                                                  */
 /* STATUS= HOT7709                                                  */
 /*                                                                  */
 /***END_OF_PROPRIETARY_STATEMENT*************************************/
 /*
*01* EXTERNAL CLASSIFICATION:  GUPI
*01* END OF EXTERNAL CLASSIFICATION:
*
*01* FUNCTION: Provide a C language header file for the VFS Callable
*              Services Interface.
*
*     Defines C structures for the control blocks and tokens that
*     are used with the v_ (BPX1V) Callable Services.
*
*     Defines C prototypes and macros for the Callable Services.
*     The macros make use of the callable services vector tables
*     so that the caller does not have to be statically bound
*     to the services or to their stubs.
*     The callable services may be invoked by either their official
*     names or by C-friendly names, i.e. as bpx1vgt() or v_get().
*
*  The following structures are defined here:
*
*     Common structures used on both the VFS and PFS interfaces.
*     ----------------------------------------------------------
*       GTOK   - General Eight Byte Token
*       FID    - File Identifier
*       CBHDR  - General Control Block Header
*       ATTR   - File Attribute Structure
*       UIO    - User I/O Structure
*       DIRENT - Directory Entries for v_readdir/vn_readdir.
*       FSATTR - File System Attributes of v_fstatfs/vfs_statfs
*
*     Structures specific to the VFS interface.
```

```
*     ----------------------------------------
*       VFSTOK & VNTOK - Opaque Tokens for file systems and files.
*       OSS     - Operating System Specific Information Structure
*       RPNMNTE - Mount Entry Structures returned by v_rpn.
*       NREG    - Registration Parameter Block used with v_reg
*       VLOCK   - Byte Range Locking Structure for v_lockctl.
*
*
*  Conditional Processing is controlled by the following symbols:
*
*     _NOFCNTL - suppresses the inclusion of fcntl.h
*
*         To suppress the inclusion of fcntl.h #define _NOFCNTL
*         and do one of the following before you include this header:
*
*          (1) If you are not going to call v_lockctl:
*
*              #define FLOCK char - to provide a dummy type for vl_flock
*          or
*
*          (2) If you will call v_lockctl
*
*              #define or typedef FLOCK to your program's flock struct
*
*
*     _BPX_MNTE2 - Produces Version 2 of the Mount Entry         @P5A
*
*
*     _BPXLL - converts the following fields from (Highword,LowWord)
*              pairs into a single 8-byte long long data type:   @P5A
*
*                  at_size
*                  at_blocks
*                  u_offset
*                  u_fssizelimit
*                  fs_maxfilesize
*                  me_bytesread
*                  me_byteswritten
*
*
*     _BPXRTL_VFSI - Makes adjustments necessary for the RTL.    @P7A
*
*     _LP64 - Makes the UIO and ATTR compatable with LP64        @P7A
*
*     __XPLINK__ - Makes the V_XXXX Linkages OS_UPSTACK          @P7A
*
*     __BPXYVFSI_INT - Changes existing fields to unsigned       @01A
*
*
*  Structures that are input to the services must be initialized
*  prior to being passed on the calls.
*  This means that the id and length fields are set correctly
*  and that unused fields are zero.
*
*     Macros are provided for initializing these structures
*     in two ways:
*
*     (1) For each potential input structure, XXX, there is an XXX_HDR
*         macro defined that can be used to initialize the header and
*         zero out the rest of the structure when the local copy
*         is declared.  For example:
*
*         ATTR attr2 = { ATTR_HDR };
*
*
*     (2) The CBINIT macro can be used to initialize an area after
*         it has been declared.  For example:
```

```
*
*         struct { int  abc;
*                  UIO  uio2;
*                  ATTR attr2;
*                  int  def;
*             } area2;
*         ...
*
*         CBINIT(area2.uio2,UIO);
*
*
*******************************************************************
*
*01* METHOD OF ACCESS:
*
*02*   C/370:
*
*        #include <bpxyvfsi.h>
*
*01* NOTES:
*
*     This header file is consistent with the following mappings:
*
*           BPXYATTR
*           BPXYDIRE
*           BPXYFUIO
*           BPXYMNTE
*           BPXYNREG
*           BPXYOSS
*           BPXYSSTF
*           BPXYVLOK
*
*01* COMPONENT: OpenMVS (SCPX4)
*
*01* DISTRIBUTION LIBRARY:  AFOMHDR1
*
****END OF SPECIFICATIONS*********************************************/

 #ifndef __BPXYVFSI_Common
   #define __BPXYVFSI_Common

 #ifdef __BPXYVFSI_INT                                        /* 7@01A*/
   typedef unsigned int bpx_int;
   typedef unsigned long long bpx_longlong;
 #else
   typedef int bpx_int;
   typedef long long bpx_longlong;
 #endif

 /*******************************************************************/
 /*******************************************************************/
 /**                                                               **/
 /**  Common structures used on both the VFS and PFS interfaces.   **/
 /**                                                               **/
 /*******************************************************************/
 /*******************************************************************/

 /*-----------------------------------------------------------------*/
 /* Opaque Tokens                                                   */
 /*-----------------------------------------------------------------*/
  typedef struct s_gtok {             /* General Eight Byte Token   */
                  void *gtok[2];
        }  GTOK ;

  typedef struct s_fid {              /* File Identifier            */
                  int fid[2];         /*   PFS Specific values      */
        }  FID  ;
```

```
 /*--------------------------------------------------------------------*/
 /* General Control Block Header and Typedef for BIT                   */
 /*--------------------------------------------------------------------*/
  typedef struct s_cbhdr {
                  char cbid[4];       /* Eye catcher                   */
                  int  cblen;         /* Length                        */
       } CBHDR;

  typedef unsigned int BIT;


 /*--------------------------------------------------------------------*/
 /* ATTR - File Attribute Structure                        (BPXYATTR)*/
 /*                                                                    */
 /*    File types and permissions of at_mode are defined in modes.h. */
 /*    Audit bits of at_aaudit & at_uaudit are defined in stat.h.    */
 /*--------------------------------------------------------------------*/
#ifdef _BPXLL
  typedef _Packed struct s_attr {
#else
  typedef struct s_attr {
#endif
           CBHDR   at_hdr;
                                                     /* POSIX fields */
           int     at_mode;           /* Type & Permissions  st_mode */
           int     at_ino;            /* inode number        st_ino  */
           int     at_dev;            /* device number       st_dev  */
           int     at_nlink;          /* link count          st_nlink*/
           int     at_uid;            /* uid of owner        st_uid  */
           int     at_gid;            /* group id of owner   st_gid  */
#ifdef _BPXLL
           long long at_size;
#else
           int     at_sizeh;          /* file size (high word)       */
           bpx_int at_size;           /* file size      st_size  @01C*/
#endif
           int     at_atime;          /* last access time    st_atime*/
           int     at_mtime;          /* last modified time  st_mtime*/
           int     at_ctime;          /* status change time  st_ctime*/
                                                    /* OE Extensions */
           int     at_major;          /* Major number for char spec  */
           int     at_minor;          /* Minor number for char spec  */
           int     at_aaudit;         /* auditor audit info          */
           int     at_uaudit;         /* user audit info             */
           int     at_blksize;        /* File block size             */
           int     at_createtime;     /* File Creation time          */
           union {                                          /*@DFA*/
              char   AT_auditid[16];    /* SAF Audit ID              */

              struct {                  /* Guard Time Value:     @DFA*/
                       int sec;            /* Seconds            @DFA*/
                       int msec;           /* Micro-seconds      @DFA*/
                   } AT_guardtime;                          /*@DFA*/
              char   AT_cver[8];        /* Creation Verifier     @DFA*/
                            /* See below for non-union names @DFA*/
           } at_u1;                                         /*@DFA*/

           char       rsvd1[4];
           int     at_genmask;        /* Setgen Mask                 */
                                      /* SetAttr Change Flags:       */
           BIT     at_modechg   :1;   /*     to mode indicated       */
           BIT     at_ownchg    :1;   /*     to UID indicated        */
           BIT     at_setgen    :1;   /*     to General Attr flags   */
           BIT     at_trunc     :1;   /*     truncate size           */
           BIT     at_atimechg   :1;  /*     the Atime               */
           BIT     at_atimeTOD   :1;  /*     Atime to TOD            */
```

```
          BIT     at_mtimechg   :1;  /*     the Mtime              */
          BIT     at_mtimeTOD   :1;  /*     Mtime to TOD           */
          BIT     at_aauditchg   :1; /*     auditor audit info     */
          BIT     at_uauditchg   :1; /*     user audit info        */
          BIT     at_ctimechg    :1; /*     the Ctime              */
          BIT     at_ctimeTOD    :1; /*     Ctime to TOD           */
          BIT     at_reftimechg   :1; /*    Reference time change   */
          BIT     at_refTOD       :1; /*    Reference time to TOD   */
          BIT     at_filefmtchg   :1; /*    File format change  @DAA*/
          BIT     at_guardtimechk :1; /*    Guard Time Check    @DFA*/
          BIT     at_cverset       :1; /*   Creation Ver Set    @DFA*/
          BIT     at_charsetidchg  :1; /*   Change File Info    @DNA*/
          BIT     at_lp64times     :1; /*   64-bit fields used  @P7A*/
          BIT     at_seclabelchg   :1; /*   change seclabel     @DQA*/
          BIT                  :12;

#ifdef _BPXRTL_VFSI                                         /*@P7A*/
          struct file_tag at_filetag;  /* Ccsid and TxtFlag     @P7A*/
          char    at_charsetid[8];     /* (Not used)            @P7C*/
#else                                                       /*@P7A*/
          char    at_charsetid[12];  /* CharSetId                   */
                  /* First 4 bytes of CharSetId is the FileTag      */
#endif                                                      /*@P7A*/
#ifdef _BPXLL
          long long at_blocks;
#else
          int     at_blocksh;        /* blocks allocated (high word)*/
          bpx_int at_blocks;         /* blocks allocated       @01C*/ /
#endif
          int     at_genvalue;       /* General Attribute Flags     */
          int     at_reftime;        /* Reference Time              */
          FID     at_fid;            /* File FID                    */
          char    at_filefmt;        /* File format            @DAA*/
          char    at_fspflag2;       /* Fsp flag2 w/acl flags   @DOA*/
          char       rsvd2[02];                              /* @DOC*/
          int     at_ctimemsec;      /* Micro-seconds of Ctime  @DFA*/
          char    at_seclabel[8];    /* security label          @DOA*/
          char       rsvd3[4];                               /* @DOC*/
                                     /* +A0 --- End Ver 1 ---   @P5A*/
          char    at_atime64[8];     /* Large Time Fields       @P5A*/
          char    at_mtime64[8];                            /*@P5A*/
          char    at_ctime64[8];                            /*@P5A*/
          char    at_createtime64[8];                       /*@P5A*/
          char    at_reftime64[8];                          /*@P5A*/
          char    at_rsvd4[8];                              /*@P5A*/
          char    at_rsvd5[16];                             /*@P5A*/
                                     /* +E0 --- End Ver 2 ---   @P5A*/
    } ATTR ;

          #define  ATTR_ID   "ATTR"
          #define  ATTR_HDR  {{ATTR_ID}, sizeof(ATTR)}

             /* Field names without the union qualifiers       @DFA*/
          #define  at_auditid   at_u1.AT_auditid             /*@DFA*/
          #define  at_guardtime at_u1.AT_guardtime           /*@DFA*/
          #define  at_cver      at_u1.AT_cver                /*@DFA*/

              /*-------------------------------------------------*/
              /*  FSP Flag2 (ACL) constants                       */
              /*---------------------------------------------@DOA*/
          #define  ATTR_ACCESS_ACL 128 /* access acl exists     @DOA*/
          #define  ATTR_FMODEL_ACL  64 /* file model acl exists @DOA*/
          #define  ATTR_DMODEL_ACL  32 /* dir model acl exists  @DOA*/

              /*-------------------------------------------------*/
              /*  File Format Type Constants                      */
              /*---------------------------------------------@DAA*/
```

```
          #define  ATTR_FFNA      0  /* Not specified              */
          #define  ATTR_FFBinary  1  /* Binary data                */
                                     /* Text data delimiters:      */
          #define  ATTR_FFNL      2  /*   New Line                 */
          #define  ATTR_FFCR      3  /*   Carrage Return           */
          #define  ATTR_FFLF      4  /*   Line Feed                */
          #define  ATTR_FFCRLF    5  /*   CR & LF                  */
          #define  ATTR_FFLFCR    6  /*   LF & CR                  */
          #define  ATTR_FFCRNL    7  /*   CR & NL                  */

              /*--------------------------------------------- 7@DGA*/
              /*  genvalue Constants -- use for ATTR or any other   */
              /*                        structures with genvalue    */
              /*----------------------------------------------------*/
    #define  GENVAL_PROGCTL 2  /* file can be program controlled    */
    #define  GENVAL_APF     4  /* file can be APF authorized        */
    #define  GENVAL_NOSHRAS 8  /* file cannot run in a shared AS    */
    #define  GENVAL_NODELFILES 32 /* files are not to be deleted from
                                     this directory            @P4A*/


    /*--------------------------------------------------------------*/
    /* The macro below tests the at_mode and at_genvalue fields     */
    /*   to see if the file is an External Symbolic Link.           */
    /*--------------------------------------------------------------*/
    #ifndef S_IFEXTL
      #define S_IFEXTL 0x00000001  /* External Link in at_genvalue  */

      #define S_ISEXTL(m,gv) ( S_ISLNK(m) && ((gv) & S_IFEXTL) )
    #endif

/*------------------------------------------------------------------*/
/* UIO - User I/O Structure                               (BPXYFUIO)*/
/*                                                                  */
/*   For 31-Bit addresses: u_buffaddr points to the buffer          */
/*                                                                  */
/*   For 64-Bit addresses: u_addr64 is on and                       */
/*                         u_buff64vaddr points to the buffer       */
/*                                                                  */
/*------------------------------------------------------------------*/
 typedef struct s_uio {
          CBHDR   u_hdr;
                                 /* u_buffaddr64 (Real)     @DMA*/
#ifndef _LP64                                                /*@P7A*/
          char   *u_buffaddr;       /*   Buffer 31-bit address     */
 #else                                                      /*@P7A*/
            char    u_buffaddr[4];                          /*@P7A*/
 #endif                                                     /*@P7A*/
          int     u_buffalet;       /*   Alet for Buffer Address   */

#ifdef _BPXLL
          bpx_longlong u_offset;    /*                        @01C*/
#else
 #ifndef _LP64                                              /*@P7A*/
          bpx_int u_offseth;        /* Cursor (high word)     @01C*/
          bpx_int u_offset;         /* Cursor                 @01C*/
  #else                                                     /*@P7A*/
            off_t   u_offset;                               /*@P7A*/
  #endif                                                    /*@P7A*/
#endif
          int     u_count;          /* Number of bytes              */
          short   u_asid;           /* Addr Space ID:  set by LFS  */
          BIT     u_rw     :1;      /* 0=Read, 1=Write              */
          BIT     u_key    :4;      /* Storage Key:    set by LFS  */
          BIT     u_sync   :1;      /* Sync data on write           */
          BIT     u_syncd  :1;      /* Sync was done:  LFS/PFS only*/
          BIT     u_chkacc :1;      /* Perform Access check         */
```

```
           BIT     u_realpage :1;    /* u_buffaddr -> real page @D4A*/
           BIT     u_limitex  :1;    /* File size limit exceeded
                                                                @D6A*/
           BIT     u_iovinuio :1;    /* uio buff is an iov      @D9A*/
           BIT     u_shutd    :1;    /* do shutdown after send  @DMA*/
           BIT     u_addr64   :1;    /* 64-Bit Addressing       @DMA*/
           BIT     u_seekcur  :1;    /* For pread/pwrite        @P5A*/
           BIT                :2;    /* Reserved                @D6C*/

           union {                   /* Vnop Specific Fields:       */
             int     u_rdindex;      /*  Readdir Index              */
             int     u_iovalet;      /*  Sendmsg/Recvmsg IOV's ALET */
                 } u_vs;             /*                             */
           union {                   /* Vnop Specific Fields2:  @DFA*/
             int     U_rddflags;     /*  Readdir Flags          @DFA*/
             int     U_iovbufalet;   /*  IOV's Buffer's ALET    @DFC*/
                           /* See below for non-union names @DFA*/
                 } u_vs2;                                     /*@DFA*/
#ifndef _BPXLL
           int     u_fssizelimithw;  /* filesize limit hiword   @D6A*/
           bpx_int u_fssizelimitlw;  /* filesize limit loword   @01C*/
#else
           long long u_fssizelimit;
#endif
           union {                   /* Vnop Specific Fields3:  @DFA*/
             char    U_cver[8];      /*  Readdir Cookie Verifier@DFA*/
             char    u_internal[16]; /*  Internal fields        @D9A*/
                           /* See below for non-union names @DFA*/
                 } u_vs3;                                     /*@DFA*/
           int     u_iovresidualcnt; /* remaining bytes to be moved
                                                                @D9A*/
           bpx_int u_totalbytesrw;   /* total number of bytes to be
                                        moved                   @01C*/
#ifndef _LP64                                                 /*@P7A*/
           char    u_buff64vaddr[8]; /* 64-Bit Virtual Addr     @P5A*/
 #else                                                        /*@P7A*/
             char   *u_buff64vaddr;                           /*@P7A*/
 #endif                                                       /*@P7A*/
    } UIO ;

             /* Field names w/o the union qualifier             @DFA*/
          #define  u_iovbufalet  u_vs2.U_iovbufalet           /*@DFA*/
          #define  u_rddflags    u_vs2.U_rddflags             /*@DFA*/
          #define  u_cver        u_vs3.U_cver                 /*@DFA*/

             /* u_rddflags - Readdir/ReaddirPlus Flags          @DFA*/
          #define  FUIOCVERRET 2  /* Cookie Ver being Returned  @DFA*/
          #define  FUIORDDPLUS 1  /* ReaddirPlus requested      @DFA*/


          #define  UIO_ID   "FUIO"
          #define  UIO_HDR  {{UIO_ID}, sizeof(UIO)}

#ifdef _BPXLL
          #define  UIO_NONEWFILES 0x8000000000000000LL
#else
          #define  UIO_NONEWFILES 0x80000000 /* No new files
                                        can be created          @D6A*/
#endif


 /*------------------------------------------------------------------*/
 /* DIRENT - Directory Entries for v_readdir/vn_readdir.   (BPXYDIRE)*/
 /*                                                                  */
 /*       Entry                      Extension                       */
 /*       ---- ---- ------//-------- -------- -------                */
 /*      | TL | NL | name           |  Ino   |       |               */
```

```
/*       ---- ---- ------//-------- -------- -------              */
/*       0    2    4                4+NL     8+NL    TL            */
/*                                                                 */
/*      The Extension may not be returned by all PFSes.            */
/*                                                                 */
/*   When (u_rddflags & FUIORDDPLUS) == FUIORDDPLUS            @DFA*/
/*     the directory entries look like:                        @DFA*/
/*                                                             @DFA*/
/*       ---- ---- ------//-------- -------------------------  @DFA*/
/*      | TL | NL | name           | Attributes              | @DFA*/
/*       ---- ---- ------//-------- -------------------------  @DFA*/
/*       0    2    4                4+NL                   TL @DFA*/
/*                                                                 */
/*----------------------------------------------------------------*/

 typedef struct s_dirent {           /* Directory Entry            */
           short  dir_len;           /*   Total entry length       */
           short  dir_namelen;       /*   Name length              */
           char   dir_name[1];       /*   File name, 1-255 bytes   */
   } DIRENT ;

 typedef struct s_dirext {           /* Directory Extension        */
           int    dir_ino;           /*   File Ino number          */
           char   dir_other[1];      /*   PFS specific data        */
   } DIREXT;


     /* The dir_name field is of variable length.
      |
      | Given the following two pointers:
      |      DIRENT  *dp;
      |      DIREXT  *dx;
      |
      | To move from one entry to the next:
      |
      |    dp = (DIRENT *) ((int)dp + dp->dir_len);
      |
      | To copy the name field to a standard C string buffer:
      |
      |    memcpy(dp->dir_name, name, dp->dir_namelen);
      |    name[dp->dir_namelen] = '\0';
      |
      | To address the optional extension structure:
      |
      |   if ((dp->dir_len) > (4 + dp->dir_namelen)) {
      |
      |      dx = (DIREXT *) ((int)dp + 4 + dp->dir_namelen);
      |
      |      ino = dx->dir_ino;
      |     }
      |   else
      |     ino = 0;
      |
      | To address the readdirplus attributes:                 @DFA
      |                                                        @DFA
      |   ATTR *da;                                            @DFA
      |                                                        @DFA
      |   da = (ATTR *) ((int)dp + 4 + dp->dir_namelen);       @DFA
      |   ino = da->at_ino;                                    @DFA
      |
      */

/*----------------------------------------------------------------*/
/* FSATTR - File System Attributes of v_fstatfs/vfs_statfs(BPXYSSTF)*/
/*----------------------------------------------------------------*/
         struct fsf_prop {           /*  NFS V3 Properties     @DFA*/
             BIT fs_fsf_v3ret       :1; /* V3 Prop Returned    @DFA*/
```

```
            BIT                    :2;
            BIT fs_fsf_CanSetTime  :1; /* time_delta accuracy  @DFA*/
            BIT fs_fsf_homogeneous :1; /* pathconf same for all@DFA*/
            BIT                    :1;
            BIT fs_fsf_symlink     :1; /* Supports Symlinks    @DFA*/
            BIT fs_fsf_link        :1; /* Supports Hard Links  @DFA*/
         };                                                /*@DFA*/

  typedef struct s_fsattr {
            CBHDR  fs_hdr;           /* (f_OEcbid and f_OEcblen)   */
    unsigned long    fs_blocksize;     /* Block size (f_bsize)       */
            int       rsvd1;         /* Reserved                   */
            int       rsvd2;         /* Reserved                   */
    unsigned long    fs_totalspace;    /* Total space. The total
                                          number of blocks on file
                                          system in units of f_frsize
                                          (f_blocks)            @D6C*/
            int       rsvd3;         /* Reserved                   */
    unsigned long    fs_usedspace;     /* Used space in blocks
                                          (f_OEusedspace)          */
            int       rsvd4;         /* Reserved                   */
    unsigned long    fs_freespace;     /* Free space in blocks
                                          (f_bavail)               */
    unsigned long    fs_fsid;          /* File system ID (f_fsid)@DBM*/
                                     /* Flags:                     */
            BIT                :1;  /*  Reserved              @DCA*/
            BIT    fs_exported  :1;  /*  Filesys is exported
                                           (ST_OEEXPORTED)       @DCA*/
            BIT                :6;  /*  Reserved              @DFC*/
    struct fsf_prop  fs_nfsprop;       /*  NFS V3 Properties     @DFA*/
            BIT                :8;  /*  Reserved              @DFC*/
            BIT                :5;  /*  Reserved              @DJC*/
            BIT    fs_nosec     :1;  /*  No Security Checks    @DJA*/
            BIT    fs_nosuid    :1;  /*  SetUID/SetGID not supported
                                           (ST_NOSUID)           @DBM*/
            BIT    fs_rdonly    :1;  /*  Filesys is read only
                                           (ST_RDONLY)           @DBM*/
#ifdef _BPXLL
            long long fs_maxfilesize;
#else
            int    fs_maxfilesizehw; /* High word of max file size
                                          (f_OEmaxfilesizehw)    @DBM*/
    unsigned long    fs_maxfilesizelw; /* Low word of max file size
                                          (f_OEmaxfilesizelw)    @DBM*/
#endif
            char      rsvd5[16];     /* Reserved                @DBA*/
    unsigned long    fs_frsize;        /* Fundamental filesystem
                                          block size (f_frsize)  @D6C*/
            int       rsvd6;         /* Reserved                @DBC*/
            int       rsvd7;         /* Reserved                @DBC*/
    unsigned long    fs_bfree;         /* Total number of free blocks
                                          (f_bfree)              @D6A*/
    unsigned long    fs_files;         /* Total number of file nodes
                                          in the file system (f_files)
                                                                 @D6A*/
    unsigned long    fs_ffree;         /* Total number of free file
                                          nodes (f_ffree)        @D6A*/
    unsigned long    fs_favail;        /* Number of free file nodes
                                          available to unprivileged
                                          users (f_favail)       @D6A*/
    unsigned long    fs_namemax;       /* Maximum file name length
                                          (f_namemax)            @D6A*/
    unsigned long    fs_invarsec;      /* Number of seconds fs will
                                          remain unchanged
                                          (f_OEinvarsec)         @D6A*/
    unsigned long    fs_time_delta_sec;/* Granularity of setting @DFA*/
    unsigned long    fs_time_delta_ns; /*   file times           @DFA*/
```

```
            char      rsvd8[12];     /* Reserved              @DBC*/
   } FSATTR ;


         #define  FSATTR_ID  "SSTF"
         #define  FSATTR_HDR  {{FSATTR_ID}, sizeof(FSATTR)}

/*------------------------------------------------------------------*/
/* PFS Type values for me_fstype and pfsi_pfstype.                  */
/*------------------------------------------------------------------*/
         #define MNT_FSTYPE_LOCAL  1    /* Files are local         */
         #define MNT_FSTYPE_REMOTE 2    /* Files are remote        */
         #define MNT_FSTYPE_PIPE   3    /* Files are pipes/fifos   */
         #define MNT_FSTYPE_SOCKET 4    /* Files are sockets       */
         #define MNT_FSTYPE_CSPS   6    /* STREAMS char spec   @DHA*/


/*------------------------------------------------------------------*/
/* Alternative Macro for Initializing Input Structures.             */
/*------------------------------------------------------------------*/
   #define CBINIT(cb,typ) {                                  \
        memset(&(cb),'\0',sizeof(typ));                       \
        memcpy(((CBHDR *)(&(cb))) -> cbid, typ ## _ID, 4);     \
        ((CBHDR *)(&(cb))) -> cblen = sizeof(typ);              \
      }

/*------------------------------------------------------------------*/
/* I/O Control Command Codes used by w_pioctl & vn_ioctl  (BPXYIOCC)*/
/*------------------------------------------------------------------*/
   #ifndef IOC_EDITACL
      #define IOC_EDITACL 0x2000C100    /* Edit ACL: _IO('A',0) @P2A*/
   #endif


/*------------------------------------------------------------------*/
/* File Group Pathconf structure used by v_pathconf       (BPXYPCF) */
/*------------------------------------------------------------------*/
                                                             /*@DFA*/
  struct PC_filegrp {                  /* PathConf File Group  @DFA*/
         int   pcfglinkmax;            /*   Link Max           @DFA*/
         int   pcfgnamemax;            /*   Name Max           @DFA*/
                                       /*   Flags:             @DFA*/
         BIT   pcfgnotrunc          :1;   /*  No Trunc         @DFA*/
         BIT   pcfgchownRstd        :1;   /*  Chown Rstd       @DFA*/
         BIT   pcfgcaseinsensitive   :1;   /*  Case Insensitive @DFA*/
         BIT   pcfgcasenonpreserving :1;   /*  Case non-presrv  @DFA*/
         BIT                        :4;                      /*@DFA*/
         char  pcfgRsvd[3];                                  /*@DFA*/
     } ;
#endif                                 /* End of Common Structures */


#if !defined(__BPXYVFSI) && !defined(__BPXYVFSI_Common_Only)
  #define __BPXYVFSI
/*******************************************************************/
/*******************************************************************/
/**                                                               **/
/**  Structures specific to the VFS interface.                    **/
/**                                                               **/
/*******************************************************************/
/*******************************************************************/

/*------------------------------------------------------------------*/
/* VFSTOK & VNTOK - Opaque Tokens for file systems and files.       */
/*------------------------------------------------------------------*/
 typedef struct s_vfstok {            /* VFS Token                 */
```

```
               char vfstok[8];
      }  VFSTOK ;

 typedef struct s_vntok {             /* Vnode Token               */
               char vntok[8];
      }  VNTOK  ;

/*-----------------------------------------------------------------*/
/* OSS - Operating System Specific Information Structure  (BPXYOSS )*/
/*-----------------------------------------------------------------*/
 typedef struct s_oss {
            CBHDR  os_hdr;
              bpx_int os_diribc;    /* Directory I/O blk cnt  @01C*/
              bpx_int os_readibc;   /* Read I/O block cnt     @01C*/
              bpx_int os_writeibc;  /* Write I/O block cnt    @01C*/
              BIT  os_xmtpt :1;     /* v_lookup cross mt pts  @P5A*/
              BIT           :31;
             void    *rsvd[2];      /* Reserved                  */
   } OSS ;

         #define  OSS_ID   "OSS "
         #define  OSS_HDR  {{OSS_ID}, sizeof(OSS)}


/*-----------------------------------------------------------------*/
/* RPNMNTE - Mount Entry Structures returned by v_rpn.    (BPXYMNTE)*/
/*     NOTE: me_mountpoint is not filled in by v_rpn.              */
/*-----------------------------------------------------------------*/

 #ifdef _BPX_MNTE2

  #ifndef __syslistdef                                         /*@P8A*/
   #define __syslistdef 1                                      /*@P8A*/

   typedef struct s_syslistdef {
           short int mt_syslistnum;   /* Number of systems in list
                                                                @DRA*/
           short int mt_syslistflags; /* Flags                  @DRA*/
           char      mt_syslist[32] [8];/* System names         @DRA*/
           } SYSLISTDEF;                                       /*@DRA*/

  #else                   /* Use the definition from mntent.h  @P8A*/
   #define mt_syslistnum   mnt2_syslistnum                     /*@P8A*/
   #define mt_syslistflags mnt2_syslistflags                   /*@P8A*/
   #define mt_syslist      mnt2_syslist                        /*@P8A*/
  #endif                                                       /*@P8A*/

   /*
    * Values for mt_syslistflags
    */
   #define MNT_SYSLIST_INCLUDE 0x0000                          /*@DRA*/
   #define MNT_SYSLIST_EXCLUDE 0x0001                          /*@DRA*/

 #endif

 typedef struct s_mnteh {             /* w_getmntent header        */
       CBHDR       mh_hdr;            /* Header with total length   */
       char        mh_cursor[8];      /* Internal cursor            */
       int         mh_devno;          /* File System devno to find  */
  #ifndef _BPX_MNTE2                  /* mnte header definition @DLA*/
       int             rsvd;          /* Reserved - must be
                                         zero on entry          @D5C*/
  #else                               /* mnte2 header definition@DLA*/
       int         mh_bodylen;        /* Length of the mnte body@DLA*/
       char            rsvd[8];       /* Reserved - must be
                                         zero on entry          @DLC*/
  #endif                                                       /*@DLA*/
```

```
    } MNTEH;

  typedef struct s_mnte {              /* w_getmntent returned entry */
         int         me_fstype;        /* File system type           */
         int         me_mode;          /* File system mount mode     */
         int         me_dev;           /* st_dev of this file system */
         int         me_parentdev;     /* st_dev of parent file sys  */
         int         me_rootino;       /* st_ino of the mount point  */
         char        me_status;        /* status of the file system. */
         char        me_ddname[9];     /* ddname specified on mount  */
         char        me_fstname[9];    /* FILESYSTYPE Name           */

         char        me_fsname[45];    /* File System Name (HFS DSN) */

         int         me_pathlen;       /* Length of mount point path */
         char        me_mountpoint[1024];  /* Mount point pathname  */
         char        me_jobname[8];    /* Job Name issuing Quiesce   */
         int         me_pid;           /* PID that issued Quiesce    */
         int         me_parmoffset;    /* Offset of mount parameter
                                          from me_fstype        @D8C*/
         short       me_parmlen;       /* Length of mount parameter
                                                                @D5A*/
   #ifndef  _BPX_MNTE2                 /* mnte base   definition @DLA*/
         char            rsvd[54];     /* Reserved for expansion @D5C*/
   #else                               /* mnte2 base   definition@DLA*/
         char        me_sysname[8];    /* system mounted on      @DKA*/
         char        me_qsysname[8];   /* quiesce system name    @DKA*/
         char        me_fromsys[8];    /* filesystem moved from this
                                          system                 @DKA*/
         short       rsvd1;            /* reserved for alignment @DKA*/
         int         me_rflags;        /* request flags          @DKA*/
         int         me_status2;       /* more status fields     @DKA*/
         int         me_success;       /* filesystems moved ok   @DKA*/
         bpx_int     me_readct;        /* Number of reads        @01C*/
         bpx_int     me_writect;       /* Number of writes done  @01C*/
         bpx_int     me_diribc;        /* Number dir I/O blks    @01C*/
         bpx_int     me_readibc;       /* Number read I/O blocks @01C*/
         bpx_int     me_writeibc;      /* Number write I/O blks  @01C*/
#ifdef _BPXLL
         bpx_longlong me_bytesread;    /*                        @01C*/
         bpx_longlong me_byteswritten; /*                        @01C*/
#else
         bpx_int     me_bytesreadhw;   /* Total number bytes read
                                          high word              @01C*/
         bpx_int     me_bytesreadlw;   /* Total number bytes read
                                          low word               @01C*/
         bpx_int     me_byteswrittenhw;/*Total number bytes-wrote
                                          high word              @01C*/
         bpx_int     me_byteswrittenlw;/*Total number bytes-wrote
                                          low word               @01C*/
#endif
         char        me_filetag[4];    /* File tag               @DNA*/
         int         me_syslistoff;    /* offset to syslist      @DRA*/
         short       me_syslistlen;    /* length of syslist      @DRA*/
         short       me_aggnamelen;    /* length of aggregate name
                                                                 @DSA*/
         int         me_aggnameoff;    /* offset to aggregate name
                                                                 @DSA*/
         char        me_roseclabel[8];/* read only seclabel      @DTA*/
    #endif                                                     /* @DLA*/
    } MNTE;

  typedef struct s_rpnmnte {           /* v_rpn returned entry:      */
         MNTEH rpn_mnteh;              /*   w_getmntent header       */
         MNTE  rpn_mnte;               /*   one w_getmntent entry    */
    } RPNMNTE;
```

```
  #define  MNTEH_ID   "MNTE"
/*#define  MNTE2H_ID  "MNT2"    */                  /*@DLA*/
  #define  MNTE2H_ID  "\xD4\xD5\xE3\xF2" /* MNT2 In Hex @DSC*/
/*#define MNTE2H MNTEH          delete conflict    @P8D @DLA*/
  #define  MNTEH_HDR  {{MNTEH_ID}, sizeof(MNTEH)+sizeof(MNTE)}


        /* Values for me_fstype are in the common area.     */

        /* Values for me_mode                               */
  #define MNT_MODE_RDWR    0x00000000            /*@P8C*/
  #define MNT_MODE_RDONLY  0x00000001            /*@P8C*/
  #define MNT_MODE_NOSUID  0x00000002            /*@P8C @DCA*/
  #define MNT_MODE_EXPORT  0x00000004            /*@P8C @DCA*/
  #define MNT_MODE_NOSEC   0x00000008            /*@P8C*/
  #define MNT_MODE_NOAUTO  16                         /*@DKA*/
  #define MNT_MODE_CLIENT  32                         /*@DKA*/
  #define MNT_MODE_AUNMOUNT 64                        /*@DPA*/
  #define MNT_MODE_SECACL  128                        /*@DOA*/
  #define MNT_MODE_RSVD1   256                        /*@P9A*/

        /* Values for me_status                             */
  #define MNT_FILE_ACTIVE           0x00              /*@D5A*/
  #define MNT_FILE_DEAD             0x01
  #define MNT_FILE_RESET            0x02
  #define MNT_FILE_DRAIN            0x04
  #define MNT_FILE_FORCE            0x08
  #define MNT_FILE_IMMED            0x10
  #define MNT_FILE_NORM             0x20
  #define MNT_FILE_IMMED_TRIED      0x40
  #define MNT_FILE_QUIESCED         0x80
  #define MNT_FILE_MOUNT_IN_PROGRESS 0x81             /*@D7A*/
  #define MNT_FILE_ASYNCH_MOUNT     0x82              /*@D5A*/

        /* Values for me_status2                            */
  #define MNT_FILE_UNOWNED          0x01
  #define MNT_FILE_INRECOVERY       0x02
  #define MNT_FILE_SUPERQUIESCED    0x04

        /* Values for me_rflags                             */
  #define MNT_REQUEST_CHANGE        0x01
  #define MNT_REQUEST_NEWAUTO       0x02

/*-----------------------------------------------------------------*/
/* NREG - Registration Parameter Block used with v_reg    (BPXYNREG)*/
/*                                                                 */
/*   NOTE: The CBINIT macro cannot be used with this structure.    */
/*-----------------------------------------------------------------*/
 typedef struct s_nreg {
            char   nr_id[4];        /* Identifier                 */
     signed short  nr_len;          /* Length                     */
            short  nr_ver;          /* Version                    */

            int    nr_type;         /* Server Type              in*/
            int    nr_namelen;      /* Server Name Length       in*/
            char   nr_name[32];     /* Server Name              in*/
            int    nr_maxvntok;     /* Maximum VNTOK limit   inout*/
            BIT    nr_hotc   :1;    /* Exit needs Hotc Env      in*/
            BIT    nr_nowait :1;    /* Don't wait for Quiesce   in*/
            BIT    nr_secsfd :1;    /* secondary sfd server     in*/
            BIT              :5;    /* Reserved                   */
            char      rsvd1[3];     /* Reserved                   */
            char   nr_exitname[8];  /* Exit name                in*/
            char   nr_initparm[8];  /* Init parm for Exit       in*/
            int    nr_abendcode;    /* Abend Code              out*/
            int    nr_abendrsn;     /* Abend Reason            out*/
```

```
          char   nr_pfstype[8];    /* Dependant PFS            in*/
  } NREG;

        #define  NREG_ID      "NREG"
        #define  NREG_VERSION  2
        #define  NREG_HDR  {NREG_ID}, sizeof(NREG), NREG_VERSION

        #define  NREG_FILE  1       /* File Server e.g.NFSS      */
        #define  NREG_LOCK  2       /* Lock Server e.g.LOCKD     */
        #define  NREG_FEXP  3       /* File Exporter e.g. DFS    */
        #define  NREG_SFDS  4       /* Shared FD server: NW  @DIA*/


/*---------------------------------------------------------------*/
/* VLOCK - Byte Range Locking Structure for v_lockctl.   (BPXYVLOK)*/
/*                                                               */
/*    The POSIX flock structure and locking constants are defined   */
/*    in the fcntl.h header, which is included here.             */
/*---------------------------------------------------------------*/
 #ifndef _NOFCNTL                   /* If pgm does not request that*/
   #include <fcntl.h>               /*   fcntl.h be suppressed,    */
   #define FLOCK struct flock       /*   see prolog for details.   */
 #endif


 typedef struct s_vlock {
          CBHDR   vl_hdr;
                  /* LOCKER: fields that are used with VL_REGLOCKER*/
          int     vl_serverpid;     /* Server's PID                */
          int     vl_clientpid;     /* Server's Client's PID       */

          GTOK    vl_lockertok;     /* Token for Locker(Spid+Cpid) */

                  /* TID: individual lock owner within a locker.   */
          char    vl_clienttid[8];  /* Client's Thread's TID       */

                  /* OBJECT: represents a single locked file       */
          int     vl_objclass;      /* Object Class: HFS, MVS, etc */
          char    vl_objid[12];     /* Obj's Unique Id within class*/

          GTOK    vl_objtok;        /* Token for Object(Class+Id)  */

                  /* DOS file sharing fields                       */
          char    vl_dosmode;
          char    vl_dosaccess;

          char    vl_rsvd1[14];

                  /* Lock information: range and type, etc         */
          FLOCK   vl_flock;         /* POSIX flock structure       */

   } VLOCK;

 /* The vl_objid used by fcntl() for POSIX locking of HFS files is: */
   struct hfsobjid {
        int   hfsobj_devno;          /*   device number (at_dev)   */
        FID   hfsobj_fid;            /*   file ID      (at_fid)    */
   };

        #define  VLOCK_ID   "VLOK"
        #define  VLOCK_HDR  {{VLOCK_ID}, sizeof(VLOCK)}

                  /* Values for Object Class: vl_objclass          */
        #define  VL_HFS    0        /* z/OS UNIX file              */
        #define  VL_MVS    1        /* MVS data set                */
        #define  VL_LFSESA 2        /* Lan File Server             */
```

```
            /* Values for v_lockctl cmd                         */
         #define  VL_REGLOCKER     1
         #define  VL_UNREGLOCKER   2
         #define  VL_LOCK          3
         #define  VL_LOCKWAIT      4
         #define  VL_UNLOCK        5
         #define  VL_QUERY         6
         #define  VL_PURGE         7

/*********************************************************************/
/*********************************************************************/
/**                                                                 **/
/**  Calling Interface Definitions                                  **/
/**                                                                 **/
/*********************************************************************/
/*********************************************************************/

/*-------------------------------------------------------------------*/
/* Macros to translate the vnode calls to their callable services    */
/*-------------------------------------------------------------------*/
#define v_reg      _VCALL(V_REG     ,145)
#define v_rpn      _VCALL(V_RPN     ,146)
#define v_get      _VCALL(V_GET     ,148)
#define v_rel      _VCALL(V_REL     ,149)
#define v_lookup   _VCALL(V_LOOKUP  ,150)
#define v_rdwr     _VCALL(V_RDWR    ,151)
#define v_readdir  _VCALL(V_READDIR ,152)
#define v_readlink _VCALL(V_READLINK,153)
#define v_create   _VCALL(V_CREATE  ,154)
#define v_mkdir    _VCALL(V_MKDIR   ,155)
#define v_symlink  _VCALL(V_SYMLINK ,156)
#define v_getattr  _VCALL(V_GETATTR ,157)
#define v_setattr  _VCALL(V_SETATTR ,158)
#define v_link     _VCALL(V_LINK    ,159)
#define v_rmdir    _VCALL(V_RMDIR   ,160)
#define v_remove   _VCALL(V_REMOVE  ,161)
#define v_rename   _VCALL(V_RENAME  ,162)
#define v_fstatfs  _VCALL(V_FSTATFS ,163)
#define v_lockctl  _VCALL(V_LOCKCTL ,164)
#define v_export   _VCALL(V_EXPORT  ,218)                     /*@DCA*/
#define v_access   _VCALL(V_ACCESS  ,235)                     /*@DDA*/
#define w_pioctl   _VCALL(W_PIOCTL  ,245)                     /*@P2A*/
#define v_pathconf _VCALL(V_PATHCONF,259)                     /*@DFA*/

/*-------------------------------------------------------------------*/
/* Callable Services Typedefs and Prototypes                         */
/*                                                                   */
/*   NOTE: Each "len" parameter contains the length of the           */
/*         parameter that follows.                                   */
/*-------------------------------------------------------------------*/
typedef void V_REG     (int len, NREG *,
                          int *rv, int *rc, int *rsn);
typedef void V_RPN     (OSS *,
                          int pathlen, char *path,
                          VFSTOK *, VNTOK *,
                          int mlen, RPNMNTE *,
                          int alen, ATTR *,
                          int *rv, int *rc, int *rsn);
typedef void V_EXPORT  (OSS *,                                /*@DCA*/
                          int   function,
                          char *filesysname,
                          VFSTOK *, VNTOK *,
                          int mlen, RPNMNTE *,
                          int alen, ATTR *,
                          char *volhdl,
                          int *rv, int *rc, int *rsn);
typedef void V_GET     (VFSTOK, OSS *,
```

```
                          FID, VNTOK *,
                          int *rv, int *rc, int *rsn);
typedef void V_REL     (VNTOK, OSS *,
                          int *rv, int *rc, int *rsn);
typedef void V_LOOKUP  (VNTOK, OSS *,
                          int namelen, char *name,
                          int alen, ATTR *,
                          VNTOK *,
                          int *rv, int *rc, int *rsn);
typedef void V_RDWR    (VNTOK, OSS *,
                          UIO *,
                          int alen, ATTR *,
                          int *rv, int *rc, int *rsn);
typedef void V_READDIR (VNTOK, OSS *,
                          UIO *,
                          int *rv, int *rc, int *rsn);
typedef void V_READLINK(VNTOK, OSS *,
                          UIO *,
                          int *rv, int *rc, int *rsn);
typedef void V_CREATE  (VNTOK, OSS *,
                          int namelen, char *name,
                          int alen, ATTR *,
                          VNTOK *,
                          int *rv, int *rc, int *rsn);
typedef void V_MKDIR   (VNTOK, OSS *,
                          int namelen, char *name,
                          int alen, ATTR *,
                          VNTOK *,
                          int *rv, int *rc, int *rsn);
typedef void V_SYMLINK (VNTOK, OSS *,
                          int namelen, char *name,
                          int pathlen, char *pathname,
                          int alen, ATTR *,
                          int *rv, int *rc, int *rsn);
typedef void V_GETATTR (VNTOK, OSS *,
                          int alen, ATTR *,
                          int *rv, int *rc, int *rsn);
typedef void V_SETATTR (VNTOK, OSS *,
                          int alen, ATTR *,
                          int *rv, int *rc, int *rsn);
typedef void V_LINK    (VNTOK, OSS *,
                          int namelen, char *name,
                          VNTOK todir,
                          int *rv, int *rc, int *rsn);
typedef void V_RMDIR   (VNTOK, OSS *,
                          int namelen, char *name,
                          int *rv, int *rc, int *rsn);
typedef void V_REMOVE  (VNTOK, OSS *,
                          int namelen, char *name,
                          int *rv, int *rc, int *rsn);
typedef void V_RENAME  (VNTOK, OSS *,
                          int oldlen, char *oldname,
                          VNTOK newdir,
                          int newlen, char *newname,
                          int *rv, int *rc, int *rsn);
typedef void V_FSTATFS (VNTOK, OSS *,
                          int falen, FSATTR *,
                          int *rv, int *rc, int *rsn);
typedef void V_ACCESS  (VNTOK, OSS *,
                          int mode,
                          int *rv, int *rc, int *rsn);
typedef void V_LOCKCTL (OSS *,
                          int cmd,
                          int vlen, VLOCK *,
                          int *rv, int *rc, int *rsn);

typedef void W_PIOCTL  (int pathlen, char *path,              /*@P2A*/
```

```
                          int cmd,
                          int arglen, char *arg,
                          int *rv, int *rc, int *rsn);

 typedef void V_PATHCONF (VNTOK, OSS *,                        /*@DFA*/
                          int pc_len, struct PC_filegrp *,
                          int alen, ATTR *,
                          int *rv, int *rc, int *rsn);

 /*------------------------------------------------------------------*/
 /* Macros & structures used to address the OE Callable Services      */
 /*------------------------------------------------------------------*/

 /* _VCALL Macro to invoke the i'th Vnode Callable Service            */
 #define _VCALL(op,i) ((op *) ( (*(struct _v_cvt **)0x10) ->         \
                       _v_cvtcsrt -> _v_csrtvopt -> _v_vopptr[i] ))

 /* Stub System Control Blocks for addressing the Callable Services   */
  struct _v_vopt {             /* z/OS UNIX Callable Services         */
     void *filler;
     void *_v_vopptr[200];
  };
  struct _v_csrt {             /* MVS Callable Services Table     */
     char          filler[0x18];
     struct _v_vopt *_v_csrtvopt;
  };
  struct _v_cvt  {             /* The CVT                         */
     char          filler[0x220];
     struct _v_csrt *_v_cvtcsrt;
  };

 /*------------------------------------------------------------------*/
 /* Interface Linkages                                                */
 /*------------------------------------------------------------------*/
#if !defined(__XPLINK__) && !defined(_LP64)                    /*@P7A*/
 #pragma linkage(V_REG     , OS)
 #pragma linkage(V_RPN     , OS)
 #pragma linkage(V_GET     , OS)
 #pragma linkage(V_REL     , OS)
 #pragma linkage(V_LOOKUP  , OS)
 #pragma linkage(V_RDWR    , OS)
 #pragma linkage(V_READDIR , OS)
 #pragma linkage(V_READLINK, OS)
 #pragma linkage(V_CREATE  , OS)
 #pragma linkage(V_MKDIR   , OS)
 #pragma linkage(V_SYMLINK , OS)
 #pragma linkage(V_GETATTR , OS)
 #pragma linkage(V_SETATTR , OS)
 #pragma linkage(V_LINK    , OS)
 #pragma linkage(V_RMDIR   , OS)
 #pragma linkage(V_REMOVE  , OS)
 #pragma linkage(V_RENAME  , OS)
 #pragma linkage(V_FSTATFS , OS)
 #pragma linkage(V_LOCKCTL , OS)
 #pragma linkage(V_EXPORT  , OS)                                /*@DCA*/
 #pragma linkage(V_ACCESS  , OS)                                /*@DDA*/
 #pragma linkage(W_PIOCTL  , OS)                                /*@P2A*/
 #pragma linkage(V_PATHCONF, OS)                                /*@DFA*/

#else                  /* Linkage Versions for LP54 & XPLINK     @P7A*/
#ifdef _LP64                                                    /*@P7A*/
 #pragma linkage(V_REG     , OS64_NOSTACK)
 #pragma linkage(V_RPN     , OS64_NOSTACK)
 #pragma linkage(V_GET     , OS64_NOSTACK)
 #pragma linkage(V_REL     , OS64_NOSTACK)
 #pragma linkage(V_LOOKUP  , OS64_NOSTACK)
 #pragma linkage(V_RDWR    , OS64_NOSTACK)
```

```
 #pragma linkage(V_READDIR , OS64_NOSTACK)
 #pragma linkage(V_READLINK, OS64_NOSTACK)
 #pragma linkage(V_CREATE  , OS64_NOSTACK)
 #pragma linkage(V_MKDIR   , OS64_NOSTACK)
 #pragma linkage(V_SYMLINK , OS64_NOSTACK)
 #pragma linkage(V_GETATTR , OS64_NOSTACK)
 #pragma linkage(V_SETATTR , OS64_NOSTACK)
 #pragma linkage(V_LINK    , OS64_NOSTACK)
 #pragma linkage(V_RMDIR   , OS64_NOSTACK)
 #pragma linkage(V_REMOVE  , OS64_NOSTACK)
 #pragma linkage(V_RENAME  , OS64_NOSTACK)
 #pragma linkage(V_FSTATFS , OS64_NOSTACK)
 #pragma linkage(V_LOCKCTL , OS64_NOSTACK)
 #pragma linkage(V_EXPORT  , OS64_NOSTACK)
 #pragma linkage(V_ACCESS  , OS64_NOSTACK)
 #pragma linkage(W_PIOCTL  , OS64_NOSTACK)
 #pragma linkage(V_PATHCONF, OS64_NOSTACK)
#else  /* XPLINK */
 #pragma linkage(V_REG     , OS_UPSTACK)
 #pragma linkage(V_RPN     , OS_UPSTACK)
 #pragma linkage(V_GET     , OS_UPSTACK)
 #pragma linkage(V_REL     , OS_UPSTACK)
 #pragma linkage(V_LOOKUP  , OS_UPSTACK)
 #pragma linkage(V_RDWR    , OS_UPSTACK)
 #pragma linkage(V_READDIR , OS_UPSTACK)
 #pragma linkage(V_READLINK, OS_UPSTACK)
 #pragma linkage(V_CREATE  , OS_UPSTACK)
 #pragma linkage(V_MKDIR   , OS_UPSTACK)
 #pragma linkage(V_SYMLINK , OS_UPSTACK)
 #pragma linkage(V_GETATTR , OS_UPSTACK)
 #pragma linkage(V_SETATTR , OS_UPSTACK)
 #pragma linkage(V_LINK    , OS_UPSTACK)
 #pragma linkage(V_RMDIR   , OS_UPSTACK)
 #pragma linkage(V_REMOVE  , OS_UPSTACK)
 #pragma linkage(V_RENAME  , OS_UPSTACK)
 #pragma linkage(V_FSTATFS , OS_UPSTACK)
 #pragma linkage(V_LOCKCTL , OS_UPSTACK)
 #pragma linkage(V_EXPORT  , OS_UPSTACK)
 #pragma linkage(V_ACCESS  , OS_UPSTACK)
 #pragma linkage(W_PIOCTL  , OS_UPSTACK)
 #pragma linkage(V_PATHCONF, OS_UPSTACK)
#endif                                                          /*@P7A*/
#endif                                                          /*@P7A*/

 /*-----------------------------------------------------------------*/
 /* Macros to allow the calls by either the v_ or bpx1 names         */
 /*-----------------------------------------------------------------*/
#ifndef _BPXRTL_VFSI                                            /*@P7A*/
 #define  bpx1vrg  v_reg
 #define  bpx1vrp  v_rpn
 #define  bpx1vgt  v_get
 #define  bpx1vrl  v_rel
 #define  bpx1vlk  v_lookup
 #define  bpx1vrw  v_rdwr
 #define  bpx1vrd  v_readdir
 #define  bpx1vra  v_readlink
 #define  bpx1vcr  v_create
 #define  bpx1vmd  v_mkdir
 #define  bpx1vsy  v_symlink
 #define  bpx1vga  v_getattr
 #define  bpx1vsa  v_setattr
 #define  bpx1vln  v_link
 #define  bpx1vre  v_rmdir
 #define  bpx1vrm  v_remove
 #define  bpx1vrn  v_rename
 #define  bpx1vsf  v_fstatfs
 #define  bpx1vlo  v_lockctl
```

```
 #define  bpx1vex  v_export                                      /*@DCA*/
 #define  bpx1vac  v_access                                      /*@DDA*/
 #define  bpx1pio  w_pioctl                                      /*@P2A*/
 #define  bpx1vpc  v_pathconf                                    /*@P2A*/
#endif                                                           /*@P7A*/

 #endif                             /* End of VFSI Structures   */
```

## BPXYPFSI—PFS interface definitions

```
#ifndef __BPXYPFSI
  #define __BPXYPFSI
 /*****START OF SPECIFICATIONS****************************************
*
*    $MAC (BPXYPFSI) COMP(SCPX4) PROD(FOM):
*
*01* MACRO NAME: BPXYPFSI
*
*01* DSECT NAME: N/A
*
*01* DESCRIPTIVE NAME: Physical File System Interface Definition for C
*
*02*   ACRONYM: N/A
*                                                                   */
 /*01* PROPRIETARY STATEMENT=                                       */
/****PROPRIETARY_STATEMENT********************************************/
/*                                                                   */
/*                                                                   */
/* LICENSED MATERIALS - PROPERTY OF IBM                              */
/* THIS MACRO IS "RESTRICTED MATERIALS OF IBM"                       */
/* 5694-A01 (C) COPYRIGHT IBM CORP. 1993, 2003                       */
/*                                                                   */
/* STATUS= HOT7709                                                   */
/*                                                                   */
/****END_OF_PROPRIETARY_STATEMENT*************************************/
 /*
*01* EXTERNAL CLASSIFICATION:  PI
*01* END OF EXTERNAL CLASSIFICATION:
*
*01* FUNCTION: Provide a C language header file for the PFS Interface.
*
*     Defines C structures for the control blocks and tokens that
*     are used by the vfs_ and vn_ operations.
*
*     Defines C prototypes for the PFS entry points
*     of the vfs_ and vn_ operations.
*
*     Defines C structures and prototypes for the osi_  services
*     and the macros used to implement the calling linkages.
*
*     Defines C structures and prototypes for the File Exporter Exit.
*
*     The definition of the following can be suppressed, see below.
*       Defines C functions for the following Kernel Extension services
*        bcopy() - copy data from source to destination
*        bzero() - zero out bytes starting at a destination
*
*       Defines C functions for the following internal services
*        _memmove() - copy characters from one data object to another
*                     with checks for data overlap. This is invoked
*                     from the bcopy() function
*
*   The following structures are defined here:
*
*      O_VNTOK - Output Vnode Token
*      WPTOK   - Wait/Post Token for osi_post
*      SELTOK  - Vn_select Token for osi_selpost
```

```
*      TOKSTR  - First Parameter of a Vnode or VFS Operation
*      OSI     - Operating System Information - Second Parameter
*      CRED    - Security Auditing Information - Third Parameter
*      PFSPARM - Text from PARM operand of FILESYSTYPE and MOUNT.
*      MTAB    - vfs_mount parameter
*      NETW    - vfs_network parameter
*      PFSI    - PFS Initialization Block and related structures,
*                including the vnode and vfs operations tables.
*      PFSNAME - Name of the PFS from TYPE operand of FILESYSTYPE.
*      OSIT    - Operating System Interface Table with related
*                structures, macros, and OSI function prototypes.
*      GXPL    - File Exporter Exit parameter structure
*      OTHDPRM - osi_thread parameter
*      OTHDCRCV- osi_thread called routine recovery block
*      OGCDPRM - osi_getcred input structure
*      BSIC    - vfs_batsel input array
*      vncanflags - vn_cancel input flags
*
*
*   The following structures are automatically included from BPXYVFSI:
*
*      GTOK    - General Eight Byte Token
*      FID     - File Identifier
*      CBHDR   - General Control Block Header
*      ATTR    - File Attribute Structure
*      UIO     - User I/O Structure
*      DIRENT  - Directory Entries for v_readdir/vn_readdir.
*      FSATTR  - File System Attributes of v_fstatfs/vfs_statfs
*
*   The following parts of the interface are defined in other
*   headers as specified:
*
*      open_flags      for vn_open, vn_rdwr, etc. are in fcntl.h,
*                      except for O_EXEC which is defined here.
*      access_intent   for vn_access is in unistd.h
*      unmount_options for vfs_umount are in stat.h
*      pathconf_option for vn_pathconf is in unistd.h
*                        Except for PC_CASE and its return values  @DHA
*                        which are defined in this header.         @DHA
*      sigval          for osi_signal  is in signal.h
*
*      socket structures are in the various standard headers as used
*                        by the sockets applications.
*
*      ioctl commands for vn_ioctl are usually in ioctl.h.
*                     Those used with Common Inet for initialization
*                     and route changes are also included here.
*
*  The following symbols provide for replaceable features:
*
*     _SOCKADDR - defines the socket address structure used in
*                 the prototypes of the socket oriented vnode ops.
*
*                 Default: #define _SOCKADDR char
*                 Example: #define _SOCKADDR struct mysocketaddr
*
*
*     _OSIT_PTR - defines the name of the variable or structure member
*                 that holds the OSIT table address that was saved
*                 during PFS initialization.  This is used to call
*                 the OSI services.
*
*                 Default: #define _OSIT_PTR osit_ptr
*
*                 Examples: There are two ways this can be used:
*
*                   (1) Declare and set osit_ptr to the saved value:
```

```
*
*                        OSIT *osit_ptr;
*                        osit_ptr = saved_address;
*                 or
*                   (2) Change the #define for _OSIT_PTR:
*
*                        #undef  _OSIT_PTR
*                        #define _OSIT_PTR saved_address
*
*     _OSICALL  - internal macro for invoking the OSI_ services.
*                 This macro is not normally replaced, refer to
*                 its definition for details on how it works.
*
*     __ADDR64  - Controls definition of the ADDR64 data type.
*                 ADDR64 is an 8-byte data type used to deal with
*                 64-Bit user pointers.   If __ADDR64 is #defined
*                 then ADDR64 may be defined by the PFS else it will
*                 be defined here based on _LP64.
*
*     __FSPL    - Exposes the Fast Socket Parameter List.
*                 This requires inclusion of socket.h.
*
*
*  Conditional Processing is controlled by the following symbol:
*
*     _NO_PFS_KES - suppresses the Kernel Extension Services.
*                   Default: Include the service definitions.
*                   Example use: #define _NO_PFS_KES
*
*******************************************************************
*
*01* METHOD OF ACCESS:
*
*02*   C/370:
*
*        #include <string.h>
*        #include <bpxypfsi.h>
*
*01* NOTES:
*
*     This header file is consistent with the following mappings:
*
*           BPXZBSIC
*           BPXYSEL
*           BPXZCJAR
*           BPXZGXPL
*           BPXZMTAB
*           BPXZNETW
*           BPXZOSI
*           BPXZOSIT
*           BPXZPFSI
*           BPXZTPRM
*           BPXZCPRM
*           BPXZVFSO
*           BPXZVNOP
*           BPXZFSPL
*           IRRPCRED
*
*01* COMPONENT: OpenMVS (SCPX4)
*
*01* DISTRIBUTION LIBRARY:  AFOMHDR1
*
****END OF SPECIFICATIONS*********************************************/

 /*-------------------------------------------------------------------*/
 /* Include the common data areas                                     */
 /*-------------------------------------------------------------------*/
```

```
#define __BPXYVFSI_Common_Only
#include <bpxyvfsi.h>
#undef  __BPXYVFSI_Common_Only

#pragma page()
/*------------------------------------------------------------------*/
/* Opaque Tokens                                                    */
/*------------------------------------------------------------------*/
 typedef struct s_o_vntok {      /* Output Vnode Token              */
                 char o_vntok[8];
       }  O_VNTOK ;

 typedef struct s_wptok {        /* Wait/Post Token for osi_post    */
                 char wptok[24];
       }  WPTOK ;

 typedef struct s_seltok {       /* vn_select Token for osi_selpost */
                 char seltok[16];
       }  SELTOK ;


/*------------------------------------------------------------------*/
/* TOKSTR - Token Structure                              (BPXZCJAR)*/
/*   This is the first parameter on all Vnode/VFS Operations        */
/*------------------------------------------------------------------*/
 typedef struct s_tokstr {
     CBHDR  ts_hdr;              /*+00 Id & Length                  */
                                 /*    PFS's Tokens:                */
     GTOK   ts_init;             /*+08  Init Token   (Vnode & VFS)  */
     GTOK   ts_mount;            /*+10  Mount Token  (Vnode & VFS)  */
     GTOK   ts_file;             /*+18  File Token   (Vnode Only)   */

     char   ts_LFS[24];          /*+20 LFS specific fields          */
     int    ts_sysd1;            /*+38 System Data 1                */
     int    ts_sysd2;            /*+3C System Data 2                */

 } TOKSTR ;


/*------------------------------------------------------------------*/
/* 64-Bit User Buffer Address Considerations                   @POA*/
/*------------------------------------------------------------------*/
/* An attempt is being made to accommodate C PFSes that are compiled
 *  with LP64, those that are not, and those that aren't even compiled
 *  with the 2.6 level of Language Extended (for long long).
 * A non-exploiting PFS mostly needs to be able to copy the 64-bit
 *  u_buff64vaddr field and its own 31-bit buffer address into the
 *  64-bit fields of the copy64_struct.
 *
 * The PFS may typedef ADDR64 to an 8-byte data type of its own
 * choice and #define __ADDR64 to bypass the default typedef.      */

    #ifndef __ADDR64                                          /*@POM*/
      #ifdef _LP64                       /* Compiler Flag  */
         typedef char * ADDR64;          /* 64-bit pointer */
      #else
         typedef struct {                /* 64-bit area    */
                  int   HW;              /*   High Word    */
                  char *LW;              /*   Low Word 31-bit ptr */
                    }   ADDR64;
      #endif
    #endif

/*------------------------------------------------------------------*/
/* OSI - Operating System Information - Second Parameter   (BPXZOSI)*/
/*------------------------------------------------------------------*/
 typedef struct s_osi {
```

```
   CBHDR    osi_hdr;            /*+00 Id & Length             */

   char    *osi_ascb;           /*+08 ASCB ptr (set by osi_wait) */
   long    *osi_ecb;            /*+0C ECB ptr  (set by osi_wait) */
   int      osi_pid;            /*+10 Caller's PID for osi_signal*/
   char     osi_lfs[8];         /*+14 LFS data                */
                                /*    SMF I/O Counts Set by PFS: */
   int      osi_diribc;         /*+1C   Directory I/O block cnt  */
   int      osi_readibc;        /*+20   Read I/O block cnt       */
   int      osi_writeibc;       /*+24   Write I/O block cnt      */
                                /*+28   Read Bytes (double word) */
   int      osi_bytesrd_h;      /*                               */
   int      osi_bytesrd;        /*+2C    Read bytes (single wd)  */
                                /*+30   Write Bytes (double word)*/
   int      osi_byteswr_h;      /*                               */
   int      osi_byteswr;        /*+34    Written bytes (one wd)  */

   char    *osi_fsp;            /*+38 Opt ptr to output FSP  @P7M*/
   int      osi_pfsid;          /*+3C PFS identifier         @D6A*/
   struct osirtoken             /*+40 Ptr to Recovery Token      */
           *osi_rtokptr;
                                /*+44 Flags                      */
   BIT      osi_LFSrsvd     :2; /*    Reserved by LFS        @PMA*/
   BIT      osi_extcaller   :1; /*    External Caller        @PMA*/
   BIT                      :1; /*    Reserved for LFS Use   @E0A*/
   BIT      osi_qnowait     :1; /*    No wait on quiesce     @E0A*/
   BIT      osi_proctrm     :1; /*    In process termination @PKA*/
   BIT      osi_quiesce     :1; /*    On behalf of quiesce   @PKA*/
   BIT      osi_sharedread  :1; /*    Shared read            @DJC*/
   BIT      osi_asy1        :1; /*    AsyncIO Part 1         @DGA*/
   BIT      osi_asy2        :1; /*    AsyncIO Part 2         @DGA*/
   BIT      osi_ok2compimd  :1; /*    May Complete Immed     @DGA*/
   BIT      osi_compimd     :1; /*    Did Complete Immed     @DGA*/
   BIT      osi_timedwait   :1; /*    Timed Wait Requested   @DJC*/
   BIT      osi_usersync    :1; /*    sync requested by user @DJC*/
   BIT      osi_remount     :1; /*    Call is for remount    @D9A*/
   BIT      osi_privileged  :1; /*    User is Privileged     @D9A*/
   short    osi_workarealen;    /*+46 Work Area Length           */
   char    *osi_workarea;       /*+48 Work Area for PFS Usage    */
   ATTR    *osi_attr;           /*+4C Optional Ptr to Output Attr*/
   WPTOK    osi_token;          /*+50 Token for osi_post         */
   char     osi_rsvd2[8];       /*+68 reserved for LFS       @P7C*/
   GTOK     osi_asytok;         /*+70 AsyncIO LFS/PFS Token  @DGA*/
   char     osi_rsvd3[4];       /*+78 reserved for LFS       @P7C*/
   char    *osi_xmib;           /*+7C Ptr to XMIB            @PMA*/
                                /*+80 Original End of OSI    @PFA*/
                                /*+80 Flags2                     */
   BIT      osi_vfsexcl     :1; /*    VFS Latch is held EXCL @PMA*/
   BIT      osi_onktask     :1; /*    Running on Kernel Task @PMA*/
   BIT                      :6;
   BIT      osi_commbuff    :1; /*    Buffers in Common      @DWA*/
   BIT      osi_fsmoving    :1; /*    File System is moving  @E0A*/
   BIT                      :6;
   char     osi_rsvd4[6];       /*+82 unused yet             @PMA*/
   ADDR64   osi_uaiocb64;       /*+88 User's Aiocb Addr      @DWA*/
   int      osi_LFSrsvd5;       /*+90 reserved for LFS       @PMA*/
   char     osi_rsvd5[12];      /*+94 unused yet             @PMA*/
                                /*+A0 End of OSI             @PFA*/
} OSI ;

   #define  osi_uaiocb  osi_uaiocb64.LW  /* User's Aiocb Addr @DWA*/

   /*-------------------------------------------------------------*/
   /* PFS Recovery Token                                          */
   /*   Set and Cleared by the PFS during a VNODE/VFS operation.  */
   /*   If this is non-zero when an abend in the PFS is percolated*/
   /*     to the LFS's ESTAE the PFS will be invoked for          */
```

```
    /*     VN_RECOVERY to clean up its resources.              @D5A*/
    /*   If this is non-zero during user EOM processing the        */
    /*     PFS will be invoked for VFS_RECOVERY to clean up        */
    /*     whatever was recorded with the token.                   */
    /*-----------------------------------------------------------*/
     struct osirtoken {
            void *osirt_ptr[2];
      } ;

        /* Extended recovery token area passed to vn_recovery @PCA*/
     struct osirtokenx {                                     /*@PCA*/
            struct osirtoken osirtx_rtoken;  /* Original osirtoken */
                   char      osirtx_rsv[16]; /* Reserved           */
                   void     *osirtx_sdwa;    /* Ptr to SDWA or 0   */
            struct vnrcvydumplist *osirtx_dumplist;          /*@PKA*/
      } ;                                                    /*@PCA*/
        /* The fourth parameter to vn_recovery may be considered
           as either osirtoken or osirtokenx.  For migration
           purposes the prototype is not being changed.       @PCA*/

        /* vn_recovery output dumplist                         @PKA*/
        struct vnrcvydumplist {                              /*@PKA*/
                   int  vnrcvydumpcount;                     /*@PKA*/
                struct  vnrcvydumparea {                     /*@PKA*/
                          char  vnrcvydumpstoken[8];         /*@PKA*/
                          void *vnrcvydumpaddr;              /*@PKA*/
                          int   vnrcvydumplength;            /*@PKA*/
                          BIT   vnrcvydumpsumm :1;           /*@PKA*/
                          BIT                 :31;           /*@PKA*/
                 } vnrcvydumpareas[1];                       /*@PKA*/
          };                                                 /*@PKA*/

 /*--------------------------------------------------------------*/
 /* vn_recovery retval flags                                     */
 /*--------------------------------------------------------------*/
 #define VNR_RETERRNO   1  /* Return -1 with retcode and rsncode   */
 #define VNR_RETSUCCESS 2  /* Return retcode as retval to user     */
 #define VNR_NODUMP     4  /* Suppress the SDUMP for this abend    */


/*----------------------------------------------------------------*/
/* CRED - Security Auditing Information - Third Parameter (IRRPCRED)*/
/*       This parameter is generally just passed to SAF.          */
/*                                                                */
/*       Refer to SAF documentation for details on security       */
/*       related interfaces and structures.                       */
/*----------------------------------------------------------------*/
 /* length, alet, ptr set used by CREDACLINFO  6@DSA*/
 typedef struct s_credacl {
    int  len;             /* cred_aclinfo[].len   */
    int  alet;            /* cred_aclinfo[].alet  */
    int  rsv;
    int  ptr;             /* cred_aclinfo[].ptr   */
    } CREDACL;

 /* aclinfo area pointed to from cred  3@DSA*/
 typedef struct s_credaclinfo {
    CREDACL cred_aclinfo[5];
    } CREDACLINFO;

  /* constants used to access an aclinfo slot for an acl type   6@DSA*/
  #define CREDACCESSACL  0
  #define CREDFMODELACL  1
  #define CREDDMODELACL  2
  #define CREDPFMODELACL 3
  #define CREDPDMODELACL 4
```

```
 /* Constants for cred_utype:      */
 #define CRED_UREGULAR    1       /* Regular User                  */
 #define CRED_USYSTEM     2       /* System User, like a superuser */

 /* Constants for cred_function:   */
 #define AFC_ACCESS       1       /* Use Real UID/GID on checks    */

typedef struct s_cred {
    CBHDR    cred_hdr;            /*+00 Id & Length                */
    char     cred_ver;            /*+08 Version                    */
    char     cred_utype;          /*+09 User Type                  */
    short    cred_function;       /*+0A User Function              */
    char     rsv4;                /*+0C Reserved               @DYA*/
    BIT      cred_rsv_bit8   :1;  /*+0D reserved bit #8        @DYC*/
    BIT      cred_seclablactive :1; /*  seclabel class active  @DYA*/
    BIT      cred_SeclRequired  :1; /*  mlfsobj option active  @DZA*/
    BIT                    :5;      /*  reserved bits          @DZC*/
    char     cred_info[50];       /*+0E Security Audit Info    @DYC*/
    int      rsv1;                /*+40                       6@DSA*/
    int      cred_aclalet;        /*+44                            */
    int      rsv2;                /*+48                            */
    void     *cred_aclptr;        /*+4C points to an ACL for access*/
    #define  cred_aclinfoptr cred_aclptr /* for makefsp and setfacl*/
    char     cred_seclabel[8];    /*+50 security label         @DYA*/
    void    *cred_aceeptr;        /*+58 ACEE for SRB requests  @DXA*/
    char     cred_ROSeclabel[8];  /*+5C Seclabel for RO Files  @DYA*/
    char     rsv5[28];            /*+64                            */
 } CRED ;

#pragma page()
/*-----------------------------------------------------------------*/
/* PFSPARM - Text from PARM operand of FILESYSTYPE and MOUNT.      */
/*                                                                 */
/*  The parmtext field is of variable length, from 0 to 1024 bytes, */
/*  with the actual length passed in the parmlen field.            */
/*-----------------------------------------------------------------*/
 typedef struct s_pfsparm {
         short parmlen;              /* Length of the text       */
         char  parmtext[1024];       /* Text, not null terminated.*/
   } PFSPARM ;

/*-----------------------------------------------------------------*/
/* MTAB - vfs_mount parameter                            (BPXZMTAB)*/
/*                                                                 */
/*   This structure passes to the PFS the parameters that were     */
/*   specified on a ROOT or MOUNT command and provides for the     */
/*   exchange of information between the LFS and PFS.              */
/*                                                                 */
/*   The PFS is expected to set the fields marked with an S,       */
/*   if appropriate.                                               */
/*                                                                 */
/*-----------------------------------------------------------------*/

    typedef char mt_aggname[45]; /*   Aggregate Name           @DUA*/

 typedef struct s_mtab {
    CBHDR    mt_hdr;            /* +00 Id & Length                */
    int         rsvd1;          /* +08 Reserved                   */
    char     mt_filesys[44];    /* +0C Name of the file system    */
    char     mt_ddname[8];      /*S+38 DD name of the file system if
                                        mt_filesys is an MVS DSN   */
    char     mt_filesystype[8]; /* +40 Type name of the PFS that
                                        owns this file system.     */
                                /* +48 Mount mode for this file sys*/
    BIT      mt_readonly  :1;   /*      Read only specified       */
    BIT      mt_readwrite :1;   /*      Read/Write specified      */
    BIT      mt_nosuid    :1;   /*      no setuid             @D8A*/
```

```
BIT      mt_nosec     :1;    /*      no security           @DNA*/
BIT      mt_noauto    :1;    /*      no automove           @DPA*/
BIT      mt_aunmount  :1;    /*      Unmount during recovery@DTA*/
BIT                   :2;    /*      Reserved         @DTC @DPC*/
                             /* +49 Lfs specific flags         */
BIT      mt_internalcall :1; /*      Mount from an internal
                                      module - no authority check
                                      is needed.               */
BIT      mt_nowait       :1; /*S     If requests are made of this
                                      file system while it is
                                      quiesced, don't wait for the
                                      unquiesce, give error rc. */
BIT      mt_remount      :1; /*     mount is a remount      @DCA*/
BIT                      :5; /*     Reserved                @DCC*/
short    mt_syncinterval;    /*S+4A Interval to use for sync    */
PFSPARM *mt_parmaddr;        /* +4C Address of PARM specified
                                    on MOUNT or ROOT.           */
int      mt_ccsid;           /* +50 TAG Ccsid value         @DRC*/
                             /*     Mount Point: (for info only)*/
char    *mt_mountptaddr;     /* +54   Address of the pathname   */
int      mt_pathlen;         /* +58   Length of the pathname    */
                             /*                                 */
int      mt_stdev;           /* +5C The unique ID assigned to
                                    this filesystem.  This value
                                    must be returned in at_dev. */
                             /* +60 Pathconf values for File Sys*/
int      mt_linkmax;         /*S+60   PFS: link_max             */
int      mt_namemax;         /*S+64   LFS & PFS: name_max       */
                             /* +68 Pathconf flags              */
BIT      mt_notrunc   :1;    /*S      LFS & PFS: posix_No_trunc */
BIT      mt_chownrstd :1;    /*S      Security: chown restricted*/
BIT      mt_caseinsensitive   :1; /*S 0=sensitive,1=not     @DHA*/
BIT      mt_casenonpreserving :1; /*S 0=perserving,1=not    @DHA*/
BIT                   :4;    /*       Reserved                  */
char        rsvd3[3];        /* +69   Reserved              @DRC*/
BIT      mt_nullFS    :1;    /* Null value for FILESYSTEM   @DRC*/
BIT      mt_nullMP    :1;    /* Null value for MOUNTPOINT   @DRC*/
BIT      mt_TagText   :1;    /* TAG TEXT value. Auto conversion */
                             /* is allowed for every untagged   */
                             /* file & mt_ccsid is the implicit */
                             /* charset id. When off, auto      */
                             /* conversion is precluded.    @DRC*/
BIT                   :13;   /*       Reserved              @DRA*/
                             /* +6E PFS communication flags     */
BIT      mt_asynchmount :1;  /*S      Asynchronous  mount in
                                      progress.
                                       - Set by PFS to indicate
                                         to LFS that mount will
                                         complete asynchronously
                                       - Set by LFS to indicate
                                         to PFS that this call
                                         is to complete an
                                         asynchronous mount @DCC*/
BIT      mt_synchonly   :1;  /*       Mount must be completed
                                      synchronously.  That is,
                                      vfs_mount must not return
                                      +1                    @D7A*/
BIT      mt_noclient  :1;    /*S      Mount must not be completed
                                      by establishing a client -
                                      server relationship with
                                      owning system. Set by PFS
                                                           @01C*/
BIT      mt_ininit    :1;    /*       Set by the LFS to allow PFS
                                      to know mount was done
                                      during initialization @PAA*/
BIT      mt_nevermove :1;    /*S      Sysplex environment only:
                                      file system cannot be moved
```

```
                                        to another system     @DPA*/
    BIT      mt_secacl    :1;    /*S      Security product supports
                                        ACLS.                 @DSA*/

    BIT      mt_aggattachrw :1;  /*S      Agg is attached R/W   @DUA*/
    BIT      mt_agghfscomp  :1;  /*S      Agg is HFS Compatible @DUA*/
    BIT                   :3;  /*       Reserved for HFS @04C @03A*/
    BIT                 :5;    /*       Reserved              @04C*/
    char       rsvd4[8];         /*       Reserved              @DPA*/
    char     mt_sysname[8];      /*       system to be mounted
                                        on                    @DPA*/
              /*+80  End of Ver1 Mtab --------------------- @DUA*/
    char       rsvd5[32];                                    /*@DUA*/
 mt_aggname *mt_aggnameptr;      /*+A0    Ptr to AggName Area   @DUA*/
    char       rsvd6[12];                                    /*@DUA*/
              /*+B0  End of Ver2 Mtab --------------------- @DUA*/
  } MTAB ;

/*-----------------------------------------------------------------*/
/* NETW - vfs_network parameter                        (BPXZNETW)*/
/*                                                                 */
/*   This structure passes to the PFS the parameters that were     */
/*   specified on a NETWORK command and provides for the           */
/*   exchange of information between the LFS and PFS.              */
/*                                                                 */
/*   The PFS is expected to set the fields marked with an S.       */
/*                                                                 */
/*-----------------------------------------------------------------*/
 typedef struct s_netw {
    CBHDR    nt_hdr;             /* +00 ID & Length                */
    int         rsvd1;           /* +08 Reserved                   */
    int      nt_domnum;          /* +0C Numeric value of the domain */
    char     nt_domname[16];     /* +10 Name of the domain         */
    char     nt_type[8];         /* +20 Filesystype of the PFS     */
    int      nt_maxsockets;      /* +28 Max number sockets         */
    int      nt_stdev;           /* +2C The unique ID assigned to
                                        this filesystem.  This value
                                        must be returned in at_dev. */
                                 /* +30 Parser Flags:              */
    BIT      nt_havename  :1;    /*       DOMAINNAME given         */
    BIT      nt_havenum   :1;    /*       DOMAINNUMBER given       */
    BIT      nt_havesock  :1;    /*       MAXSOCKETS given         */
    BIT      nt_havetype  :1;    /*       TYPE       given         */
    BIT                   :4;
    BIT      nt_invaname  :1;    /*       DOMAINNAME invalid       */
    BIT      nt_invanum   :1;    /*       DOMAINNUMBER invalid     */
    BIT      nt_invasock  :1;    /*       MAXSOCKETS invalid       */
    BIT      nt_invatype  :1;    /*       TYPE       invalid       */
    BIT                   :4;
                                 /* +32 Flags:                     */
    BIT      nt_localremote :1;  /*S      0=Local, 1=Remote        */
    BIT      nt_commoninet  :1;  /*       running under Cinet   @PGA*/
    BIT                     :6;
    char        rsvd2[1];        /* +33 Reserved                   */
    short    nt_iaaport;         /* +34 Starting Reserved Port  @PGA*/
    short    nt_iaacount;        /* +36 Number of Reserved Ports@PGA*/
    char     nt_parmmem[8];      /* +38 Parmlib Member name     @PGA*/
                                 /* +40 */
  } NETW ;

      /* nt_localremote values                                     */
      #define NETW_LOCAL    0      /*   Local (intra-system) socket */
      #define NETW_REMOTE   1      /*   Remote (network) socket     */


/*-----------------------------------------------------------------*/
/* O_EXEC flag for the open_flags parameter of vn_open    (BPXYOPNF)*/
```

```
/*--------------------------------------------------------------------*/
   #define O_EXEC   0x00800000     /* Do Open Access check for Exec */

/*--------------------------------------------------------------------*/
/* _SOCKADDR Dummy Value                                    (BPXYSOCK)*/
/*--------------------------------------------------------------------*/
  #ifndef _SOCKADDR         /* This macro can be externally set @PIM*/
     #define _SOCKADDR char /*  to the desired sockaddr struct. @PIM*/
  #endif                                                      /*@PIM*/

/*--------------------------------------------------------------------*/
/* Select Parameters - vn_select and vfs_batsel             (BPXYSEL)*/
/*--------------------------------------------------------------------*/

  /* sel_function values                                             */

     #define SEL_QUERY      1        /* SELECT Query                 */
     #define SEL_CANCEL     2        /* SELECT Cancel                */
     #define SEL_BATSELQ    3        /* BATCH-SELECT Query           */
     #define SEL_BATSELC    4        /* BATCH-SELECT Cancel          */
     #define SEL_POLLQUERY  5        /* POLL Query               @P9A*/
     #define SEL_BATPOLLQ   6        /* BATCH-POLL Query         @P9A*/
     #define SEL_BATPOLLC   7        /* BATCH-POLL Cancel        @P9A*/
     #define SEL_POLLCANCEL 8        /* POLL Cancel              @P9A*/

  /* sel_options  values                                             */

     #define SEL_READ    0x40000000  /* Read                         */
     #define SEL_WRITE   0x20000000  /* Write                        */
     #define SEL_XCEPT   0x10000000  /* Exception                    */


  /* Batch Select Interface Control (BSIC) Block         (BPXZBSIC)*/

  typedef struct s_bsicent {       /* Individual Entry:              */
      int      bs_request;         /*  Status Request                */
      int      bs_response;        /*  Status Response               */
      GTOK     bs_file;            /*  File Token, same as ts_file  */
      char    *bs_workptr;         /*  Work Area Ptr for use by PFS */
      SELTOK   bs_seltok;          /*  Select Token for osi_selpost */
   } BSICENT;

  typedef struct s_bsic {          /* Main structure with array:    */
      char     bsh_id[4];          /*  Identifier                    */
      int      bsh_fdcount;        /*  Number of bsh_ents (files)   */
      char    *bsh_workptr;        /*  Work Area Ptr for use by PFS */
      BSICENT  bsh_ents[1];        /*  Entry array (1 per fdcount)  */
   } BSIC ;


/*--------------------------------------------------------------------*/
/* Direction parameter for vn_sockopt                             @P8A*/
/*--------------------------------------------------------------------*/

     #define GET_SOCKOPT    1        /* Get socket options       @P8A*/
     #define SET_SOCKOPT    2        /* Set socket options       @P8A*/
     #define SET_IBMSOCKOPT 3        /* SetIBMsockopt options    @PDA*/

  /*------------------------------------------------------------------*/
  /* vn_sockopt(SET_IBMSOCKOPT) Options                   (BPXYSOCK)*/
  /*------------------------------------------------------------------*/
   #define SOCK_SO_BULKMODE                 0x8000       /*@DKA,@DMC*/
   #define SOCK_SO_IGNOREINCOMINGPUSH            1       /*@DKA,@DMC*/
   #define SOCK_SO_NONBLOCKLOCAL            0x8001       /*@DKA,@DMC*/
   #define SOCK_SO_IGNORESOURCEVIPA              2       /*@DKA,@DMC*/
   #define SOCK_SO_OPTMSS                   0x8003       /*@DKA,@DMC*/
   #define SOCK_SO_OPTACK                   0x8004       /*@DKA,@DMC*/
```

```
/*----------------------------------------------------------------*/
/* vn_getname Name_type values                               @PHA*/
/*----------------------------------------------------------------*/
   #define gnm_getpeername        1                        /*@PHA*/
   #define gnm_getsockname        2                        /*@PHA*/

/*----------------------------------------------------------------*/
/* vn_cancel Flags                                           @DGA*/
/*----------------------------------------------------------------*/
 struct  vncanflags  {             /* vn_cancel flags          @DGA*/
     BIT               :8;         /*  Reserved                    */
     BIT               :23;        /*  Also reserved               */
     BIT  vncanforce    :1;        /*  Forced Cancel           @DGA*/
  } ;


/*----------------------------------------------------------------*/
/* PathConf Extensions - vn_pathconf                    (BPXYPCF)*/
/*----------------------------------------------------------------*/
     #define PC_CASE          100    /* pathconf_option value   @DHA*/

     #define CASE_INSENSITIVE   2    /*   Ret if not sensitive  @DHA*/
     #define CASE_NONPRESERVING 1    /*   Ret if not preserving @DHA*/

/*----------------------------------------------------------------*/
/* Accept_and_Receive structure - vn_anr                (BPXZOSI)*/
/*----------------------------------------------------------------*/
 struct anr_struct {                                       /*@PHA*/
           int         remote_sockaddr_length;
           _SOCKADDR  *remote_sockaddr_ptr;
           int         local_sockaddr_length;
           _SOCKADDR  *local_sockaddr_ptr;
           int         msg_flags;
  } ;                                                      /*@PHA*/

/*----------------------------------------------------------------*/
/* 64-Bit Versions of the Iovec and MsgHdr      (BPXYMSGH & BPXYIOV)*/
/*----------------------------------------------------------------*/

   struct iov64 {                                          /*@POA*/
         ADDR64 iov64_base;               /* 64-Bit Ptr              */
            int iov64_lenh;               /* Required to be Zero     */
     signed int iov64_len;                /* Length, < 2G            */
    };


   struct msg64hdr {                                       /*@POA*/
         ADDR64 msg64_name;               /* 64-Bit sockaddr ptr     */
         ADDR64 msg64_iov;                /* 64-Bit iov ptr          */
         ADDR64 msg64_control;            /* 64-Bit ancillary ptr    */
         int    msg64_flags;              /* MSG_ flags              */
         int    msg64_namelen;            /* 31-Bit sockaddr length  */
         int    msg64_iovlen;             /* 31-Bit number of iovecs */
         int    msg64_controllen;         /* 31-Bit ancillary len    */
    };


/*----------------------------------------------------------------*/
/* Fast Sockets Parameter List - VN_FSR/FSRF/FSRM        (BPXZFSPL)*/
/*----------------------------------------------------------------*/
#ifdef __FSPL
   struct fs_sr  {             /* FSP1 - vn_fsr               @POA*/
                                          /*+30*/
        int     sr_ibufflen;              /*   buffer length         */
        int     sr_ibufferalet;           /*   buffer alet           */
        int     sr_iflags;                /*   flags                 */
```

```
        char   *sr_ibufferptr;          /*   31-bit ptr           */
                                        /*+40*/
        ADDR64  sr_ibufferptr64;        /*   64-bit buff ptr      */
      } ;

    struct fs_srf {            /* FSP2 - vn_fsrf               @POA*/
                                        /*+30*/
        int     srf_ibufflen;           /*   buffer length        */
        int     srf_ibufferalet;        /*   buffer alet          */
        int     srf_iflags;             /*   flags                */
        int     srf_isockaddrlen;       /*   sockaddr length      */
                                        /*+40*/
        char   *srf_isockaddrptr;       /*   ptr to sockaddr in pri*/
        char   *srf_ibufferptr;         /*   ptr to buffer        */
        ADDR64  srf_ibufferptr64;       /*   64-bit buff ptr      */
      } ;

    struct fs_srm {            /* FSP3 - vn_fsrm               @POA*/
                                        /*+30*/
        int     srm_iflags;             /*   flags                */
        int     srm_iiovalet;           /*   iov structure alet   */
        int     srm_iiovbufalet;        /*   alet for iov buffers */
        union {
            struct msghdr srm_imsghdr;       /* 31-bit msghdr     */
            struct {  int     rsvd;
             struct msg64hdr  srm_imsghdr64;  /* 64-bit msghdr     */
                   } srm_imsghdr64u;
              } srm_imsgh;
      } ;

  struct s_fspl {                                            /*@POA*/
     CBHDR    fs_hdr;            /*  ID & Length                   */
     char       rsvd1[3];        /*  Reserved                      */
                                 /*  Flags:                        */
     BIT      fs_key       :4;   /*   user's key                   */
     BIT      fs_addr64    :1;   /*   64-bit buffer addrs          */
     BIT                   :1;   /*                                */
     BIT      fs_shutd     :1;   /*   send & shutdown (msg_eof)    */
     BIT      fs_rwind     :1;   /*   0=read, 1=write              */
     CRED    *fs_cred;           /*   ptr to cred              @DXA*/
                                 /*+10*/
     GTOK     fs_pfstok;         /*   pfs token from vnode (ts_file)*/
     int      fs_openflgs;       /*   open flags                   */
     OSI     *fs_osi;            /*   ptr to osi                   */
                                 /*+20*/
     int      fs_rv;             /*   return value                 */
     int      fs_rc;             /*   return code (errno)          */
     int      fs_rsn;            /*   reason code (errnojr)        */
     int      fs_sockdes;        /*   common socket descriptor     */
                                 /*+30*/
     union {                     /*   call specific parms          */
            struct fs_sr  fs_isr;
            struct fs_srf fs_isrf;
            struct fs_srm fs_isrm;
      } fs_parms;
   } ;

 #endif


 /*-----------------------------------------------------------------*/
 /* Inactive buffer structure (IAB)  -  vfs_inactive (BPXZOSI)      */
 /*-----------------------------------------------------------------*/
    typedef struct s_iabent  {        /* Individual Entry           */
        char     *iab_vnode;          /* Vnode pointer              */
        char      iab_pfs[8];         /* Pfs token                  */
        char     *iab_server_vnode;   /* Server's vnode ptr         */
```

```
       FID       iab_fid;            /* Fid for validation      @PPA*/
       int       iab_return_value;   /* Return value                 */
    } IABENT;                                                   /*@PLA*/
   typedef struct s_iab  {           /* Main structure with array:  */
       int       iab_devno;          /*   Device number             */
       IABENT    iab_ents[1];        /*   Entry array (1 per vnode) */
    } IAB;                                                      /*@PLA*/

#ifndef SIOCSETRTTD
   /*---------------------------------------------------------------*/
   /* Ioctl commands used during initialization of a PFS           */
   /* when using Common Inet                                   @P8A*/
   /* NOTE: Values of the form 000013xx can only be used           */
   /*       with the w_ioctl() function, not with ioctl().         */
   /*---------------------------------------------------------------*/

     #define SIOCSETRTTD        0x8008C981 /* Set TD - Left bookend */
     #define IOCC_TCCE          0x0000138e /* (5006) - Right bookend*/

   /*---------------------------------------------------------------*/
   /* Ioctl commands used during normal processing of route        */
   /* changes when using Common Inet                           @P8A*/
   /*---------------------------------------------------------------*/

     #define SIOCMSDELRT        0x0000138f /* (5007) - Delete Route */
     #define SIOCMSADDRT        0x00001390 /* (5008) - Add Route    */
     #define SIOCMSSIFADDR      0x00001391 /* (5009) - Set Interface
                                                      Address      */
     #define SIOCMSSIFFLAGS     0x00001392 /* (5010) - Set Interface
                                                      Flags        */
     #define SIOCMSSIFDSTADDR   0x00001393 /* (5011) - Set pt-to-pt
                                                  interface address*/
     #define SIOCMSSIFBRDADDR   0x00001394 /* (5012) - Set broadcast
                                                      Address      */
     #define SIOCMSSIFNETMASK   0x00001395 /* (5013) - Set Interface
                                                      Network Mask */
     #define SIOCMSSIFMETRIC    0x00001396 /* (5014) - Set Interface
                                                     Routing Metric*/
     #define SIOCMSRBRTTABLE    0x00001397 /* (5015) - Rebuild Routing
                                                      Table        */
     #define SIOCMSMETRIC1RT    0x00001398 /* (5016) - Set Metric1  */
     #define SIOCMSICMPREDIRECT 0x00001399 /* (5017) - ICMP Redirect*/
#endif

#pragma page()
/*******************************************************************/
/*                                                                 */
/* Physical File System Initialization Interface Structures        */
/*                                                                 */
/*   These structures are used during the activation of a PFS.     */
/*   The pfsinit routine is invoked with the following parameters: */
/*                                                                 */
/*     pfsinit(PFSI *P, PFSNAME *N, PFSPARM *M, void *V, OSIT *O)   */
/*                                                                 */
/*   The variable names, P,N,M,V, and O are used in the examples.   */
/*******************************************************************/

/*-----------------------------------------------------------------*/
/* PFSI - PFS Initialization Block                       (BPXZPFSI)*/
/*                                                                 */
/*   This structure is used to exchange information between        */
/*   the LFS and PFS during initialization.                        */
/*                                                                 */
/*   The PFS is expected to set the fields marked with an S.       */
/*                                                                 */
/*-----------------------------------------------------------------*/
 typedef struct s_pfsi {
```

```
CBHDR   pfsi_hdr;         /* +00 ID and Length                  */
short   pfsi_ver;         /* +08 Version number                 */
char    pfsi_rsvd1;       /* +0A Reserved                       */
char    pfsi_tdindex;     /* +0B Cinet Td Index passed to PFS @DVA*/
GTOK    pfsi_pfsanchor;   /*S+0C The PFS init token that will be
                                 passed to the PFS on all calls. */
struct vfsotab
       *pfsi_vfso;        /*S+14 Address of the VFS ops table    */
                          /* +18 Flags                           */
BIT     pfsi_ook     :1;    /*  File system is running outside
                                 the kernel                      */
BIT     pfsi_alone   :1;    /*  File system is the only PFS in
                                 this A.S. outside the kernel    */
BIT     pfsi_new     :1;    /*  File system is being intialized
                                 for the 1st time in this AS @P0C*/
BIT     pfsi_estaeexits :1; /*  osi_thread called routine
                                 permanent ESTAE supported   @DMA*/
BIT     pfsi_memcritical:1; /*  LFS supports osi_memcritical
                                 in this release             @02A*/
BIT     pfsi_sysplex :1;    /*  USS started SYSPLEX(YES)     @E0A*/
BIT                  :2;    /*  Reserved                     @E0C*/
BIT     pfsi_commbuff   :1; /*  Common Buffers Supported     @DWA*/
BIT                     :7; /*  Reserved                     @DGC*/
BIT     pfsi_osync   :1;    /*S vn_open does fsync for O_SYNC @PMA*/
BIT     pfsi_srb     :1;    /*S SRM Mode supported           @DGA*/
BIT     pfsi_asyio   :1;    /*S Async I/O supported          @DGA*/
BIT     pfsi_rddplus :1;    /*S ReadDirPlus supported        @DHA*/
BIT     pfsi_64datoff  :1;  /*S 64-Bit Real Page Supported   @PMA*/
BIT     pfsi_nolgfile  :1;  /*S O_NOLARGEFILE size checking  @PMA*/
BIT     pfsi_addr64    :1;  /*S 64-Bit User areas supported  @PMA*/
BIT     pfsi_ipv6      :1;  /*S IPv6 Capable                 @DVA*/
BIT     pfsi_romntclient :1; /*S=1: Read-only mounts on other than
                                   owner should be client
                                   (i.e. served)
                               =0: Such mounts should be local
                                   (i.e. file system is sysplex
                                   aware)                    @PEC*/
BIT     pfsi_rwmntclient:1;  /*S=1: Read-write mounts on other than
                                   owner should be client
                               =0: Such mounts should be local
                                                             @D0C*/
BIT     pfsi_usethreads :1; /*S File system requests support for
                                 the osi_thread function     @D7A*/
BIT     pfsi_disableLLA :1; /*S File system requests no lookup
                                 look aside support          @D5A*/
BIT     pfsi_stayalone  :1; /*S File system requests no other PFS
                                 be started in this A.S.         */
BIT     pfsi_immeddel   :1; /*S Removed files are deleted if, or
                                 when, their open count is 0 @D6A*/
BIT     pfsi_cpfs       :1; /*S File system is written in C. Invoke
                                 w/ a preinit. C environment     */
BIT     pfsi_datoffmove :1; /*S File system supports DATOFF move
                                 for page read operations        */
struct vnoptab
       *pfsi_vnop;        /*S+1C Address of the Vnode ops table  */
int     pfsi_tcbaddr;     /* +20 Address of the TCB for this PFS */
long    pfsi_initcompecb; /* +24 ECB that the PFS posts when
                                 initialization is complete.     */
char    pfsi_pfstype;     /*S+28 The type of the PFS             */
char    pfsi_rsvd2[3];    /* +29 Reserved                        */
long    pfsi_pfsecb;      /* +2C ECB that is posted when the Kernel
                                 is terminating.  The PFS should
                                 be waiting on this ECB.         */
                          /*     Pathconf() values as applicable: */
int     pfsi_pipebuf;      /*S+30   pipe_buf                      */
int     pfsi_maxcanon;     /*S+34   max_canon                     */
int     pfsi_maxinput;     /*S+38   max_input                     */
```

```
                          /* +3C   Flags:                     */
 BIT     pfsi_chownrstd :1; /*S       POSIX_Chown_restr         */
 BIT                    :7; /*        Reserved                  */
 char    pfsi_rsvd3[2];     /* +3D   Reserved                   */
 char    pfsi_vdisable;     /*S+3F   _posix_vdisable            */

 char   *pfsi_restart;     /* +40 Address of Restart Option Byte   */
 struct dmpinf
        *pfsi_dumpptr;     /* +44 Address of Dump Information      */

 char    pfsi_asname[8];   /* +48 Address Space Name of PFS    @D1A*/
 char    pfsi_ep[8];       /* +50 Entry point attached during
                                  initialization               @D1A*/
 int     pfsi_pfsid;       /* +58 Pfs Identifier               @P5A*/
 struct ot_statflags
        *pfsi_otstatptr;   /* +5C osi_thread status flags      @P5A*/
 char    pfsi_rsvd4[8];    /* +60 Reserved                     @P5C*/
                           /*     Inserts for Dump Titles:         */
 char    pfsi_compon[3];   /*S+68   This PFS's Component Prefix    */
 char    pfsi_compid[5];   /*S+6B   This PFS's Component ID        */
 char    pfsi_startname[8];/* +70 Start name for PFS           @DAA*/
 int     pfsi_pfspc;       /* +78 PfsPc Number, Colony Only    @PMA*/
 char    pfsi_rsvd6[36];   /* +7C Reserved                     @D7C*/

} PFSI ;

    /*-----------------------------------------------------------*/
    /* pfsi_restart - Restart Option Values                      */
    /*   Example usage:   *(P->pfsi_restart) = RESTART_NONE;      */
    /*-----------------------------------------------------------*/
     #define RESTART_WTOR     0       /* Prompt operator first     */
     #define RESTART_AUTO     1       /* Restart automatically     */
     #define RESTART_NONE     2       /* Do not restart this PFS   */
     #define RESTART_KILL     3       /* Terminate OMVS too        */

     #define RESTART_RCWTOR   4       /* Restart Colony and Prompt
                                      operator for PFS restart @D1A*/
     #define RESTART_RCAUTO   5       /* Restart Colony and
                                         Automatic PFS restart @D1A*/
     #define RESTART_RCNONE   6       /* Bring down Colony and
                                         No PFS restart tried  @D1A*/
     #define RESTART_PFSCTL   7       /* Wait for pfsctl(Restart)
                                                               @PJA*/

    /*-----------------------------------------------------------*/
    /* pfsi_pfstype - PFS Type Values                            */
    /*   Example usage:    P->pfsi_pfstype = MNT_FSTYPE_REMOTE;    */
    /*-----------------------------------------------------------*/
          /* These are defined with the common structures in      */
          /*   BPXYVFSI as the constants starting with MNT_FSTYPE_ */

    /*-----------------------------------------------------------*/
    /* pfsi_ver - Version Values                                 */
    /*-----------------------------------------------------------*/
     #define PFSI_VER0        0       /* Initial Version           */
     #define PFSI_VER1        1       /* Second Version            */
     #define PFSI_VER2        2       /* Second Version + HOTC @D4A*/

    /*-----------------------------------------------------------*/
    /* pfsi_otstatptr - pointer to osi_thread status flags     @P5A*/
    /*-----------------------------------------------------------*/
    struct  ot_statflags    {     /* osi_thread status flags      */
        BIT   ot_available :1;    /*  Thread services are available*/
        BIT                :7;
     } ;

    /*-----------------------------------------------------------*/
```

```
/* pfsi_vnop - VNODE Operations Table                (BPXZVNOP)*/
/*   This table is built by the PFS and returned to the LFS    */
/*------------------------------------------------------------*/
 #define VN_OPEN             0
 #define VN_CLOSE            1
 #define VN_RDWR             2
 #define VN_IOCTL            3
 #define VN_GETATTR          4
 #define VN_SETATTR          5
 #define VN_ACCESS           6
 #define VN_LOOKUP           7
 #define VN_CREATE           8
 #define VN_REMOVE           9
 #define VN_LINK            10
 #define VN_RENAME          11
 #define VN_MKDIR           12
 #define VN_RMDIR           13
 #define VN_READDIR         14
 #define VN_SYMLINK         15
 #define VN_READLINK        16
 #define VN_FSYNC           17
 #define VN_TRUNC           18
 #define VN_INACTIVE        19
 #define VN_AUDIT           20
 #define VN_PATHCONF        21                         /*@D5A*/
 #define VN_RECOVERY        22                         /*@D5A*/
 #define VN_LOCKCTL         23      /* for File Exp Exit only */
 #define VN_CANCEL          24                         /*@DGA*/
 #define VN_SELECT          25
 #define VN_ACCEPT          26
 #define VN_BIND            27
 #define VN_CONNECT         28
 #define VN_GETNAME         29
 #define VN_SOCKOPT         30
 #define VN_LISTEN          31
 #define VN_READWRITEV      32
 #define VN_SNDRCV          33
 #define VN_SNDTORCVFM      34
 #define VN_SRMSG           35
 #define VN_SHUTDOWN        37

 #define VN_FSR             38                         /*@DLA*/
 #define VN_FSRF            39                         /*@DLA*/
 #define VN_FSRM            40                         /*@DLA*/
 #define VN_SRX             42                         /*@DLA*/
 #define VN_ANR             43                         /*@DLA*/

 #define MAX_VNOPS          44


 typedef void VNOP_OP();           /* Generalized Vnode Op    */
 #pragma linkage(VNOP_OP, OS)      /* Is called with OS lnkg  */

struct vnoptab {                   /* The Vnode Op Table      */
          CBHDR   vnop_hdr;

        VNOP_OP  *vnop_op[MAX_VNOPS];
 };

 #define  VNOP_ID   "VNOP"
 #define  VNOP_HDR  {{VNOP_ID}, sizeof(struct vnoptab)}

    /* Example initialization of this table:
     |              /* Get storage, init hdr & zero out rest  *
     |    struct vnoptab pfstab = { VNOP_HDR };
     |              /* Set the address of each supported op   *
     |    pfstab.vnop_op[VN_OPEN]   = pfs_open;
```

```
      |     pfstab.vnop_op[VN_CLOSE]  = pfs_close;
      |     pfstab.vnop_op[VN_RDWR]   = pfs_rdwr;
      |       . . . etc.
      |               /* Return the table address to the LFS    *
      |     P->pfsi_vnop = &pfstab;
      */

 /*-----------------------------------------------------------*/
 /* pfsi_vfso - VFS Operations Table                (BPXZVFSO)*/
 /*   This table is built by the PFS and returned to the LFS    */
 /*-----------------------------------------------------------*/
  #define  VFS_MOUNT        0
  #define  VFS_UMOUNT       1
  #define  VFS_SYNC         2
  #define  VFS_INACT        3                              /*@PPA*/
  #define  VFS_STATFS       4
  #define  VFS_VGET         6
  #define  VFS_RECOVERY     7
  #define  VFS_BATSEL       9
  #define  VFS_GETHOST     10
  #define  VFS_SOCKET      11
  #define  VFS_NETWORK     12
  #define  VFS_PFSCTL      13
  #define  MAX_VFSOPS      14


  typedef void VFS_OP();             /* Generalized VFS Op      */
  #pragma linkage(VFS_OP, OS)        /* Is called with OS lnkg  */

 struct vfsotab {                    /* The VFS Op Table        */
          CBHDR   vfso_hdr;

         VFS_OP  *vfso_op[MAX_VFSOPS];
  };

  #define   VFSO_ID   "VFSO"
  #define   VFSO_HDR  {{VFSO_ID}, sizeof(struct vfsotab)}

 /*-----------------------------------------------------------*/
 /* Dump Information - used by the PFS to add LFS address space */
 /* and data space areas to the dumps that are taken by the PFS.*/
 /*-----------------------------------------------------------*/
    struct pfsi_dumpent {        /* Individual Dump Area Entry: */
       char  pfsi_dumpstoken[8];    /* Stoken of the space      */
       int   pfsi_dumpalet;         /* Reserved             @P8C*/
       int   pfsi_dumpflag;         /* Reserved             @P8C*/
       char *pfsi_dumpstart;        /* Starting address         */
       char *pfsi_dumpend;          /* Ending address           */
    } ;

  struct dmpinf {            /* Area pointed to by pfsi_dumpptr */
    int     pfsi_dumpents;     /* Number of Dump Area Entries   */
    char    pfsi_dumpid[4];    /* EBCDIC ID - FDUM          @P8A*/
    char    pfsi_rsvd7[8];     /* Reserved                  @P8C*/
    struct pfsi_dumpent        /* Array of Dump Areas, actual   */
            pfsi_dumpdata[16]; /*   number of entries is in     */
  } ;                          /*   pfsi_dumpents.              */


/*------------------------------------------------------------------*/
/* PFSNAME - Name of the PFS from TYPE operand of FILESYSTYPE.      */
/*    This string is blank padded and not null terminated.          */
/*------------------------------------------------------------------*/
 typedef struct s_pfsname {
          char  pfsname[8];           /* PFS Type or Name           */
   } PFSNAME ;
```

```
#pragma page()
/*****************************************************************/
/*                                                               */
/* File Exporter Exit Interfaces                                 */
/*                                                               */
/*****************************************************************/

 /*-------------------------------------------------------------*/
 /* Exit Parameter Structure                        (BPXZGXPL)*/
 /*-------------------------------------------------------------*/
 typedef struct s_gxpl {
      char     gx_id[4];          /*+00 EBCDIC ID                */
      short    gx_ver;            /*+04 Gxpl Version number      */
      short    gx_len;            /*+06 Length of Gxpl structure */

      short    gx_op;             /*+08 Operation Code           */
                                  /*+0A Flags:                   */
      BIT      gx_postop    :1;   /*     1=PostOp Call           */
      BIT      gx_readwrite :1;   /*     0=Read mode, 1=Write mode */
      BIT      gx_eom       :1;   /*     1=Called from user EOM  */
      BIT      gx_trunc     :1;   /*     1=File Size Change      */
      BIT                  :12;
      OSI     *gx_osi;            /*+0C OSI address              */

      int      gx_volhdl[4];      /*+10 VolHdl from v_export     */
      int      gx_anchor[2];      /*+20 Exit Anchor              */
      int      gx_state[2];       /*+28 Exit State Area          */
                                  /*    File Identifiers:        */
      FID      gx_fid1;           /*+30   The principal target   */
      FID      gx_fid2;           /*+38   The secondary target   */
      FID      gx_fid3;           /*+40   for rename, the to-dir */
      FID      gx_fid4;           /*+48   for rename, the to-file */

      int      gx_opretval;       /*+50 Op Return Value to PostOp */
      int      gx_retcode;        /*+54 Exit Return Code         */
      int      gx_rsncode;        /*+58 Exit Reason Code         */
      char    *gx_optparm;        /*+5C Optional Parameter       */

      char     gx_lfs[8];         /*+60 Reserved for the LFS     */
      char        rsvd1[8];       /*+68 Reserved for expansion   */
  } GXPL ;

    #define  GXPL_ID       "GXPL"
    #define  GXPL_VERSION   1

   /*-----------------------------------------------------------*/
   /* Constants for gx_op                                       */
   /*-----------------------------------------------------------*/
    #define  GXPL_INIT      0x1001     /* Initialization Call       */
    #define  GXPL_EXPCMD    0x1002     /* Exporter Command          */
    #define  GXPL_RECOVERY  0x1003     /* Recovery Call             */
    #define  GXPL_UNMOUNT   0x1004     /* Unmount  Call        @DBA*/
    #define  GXPL_UNEXPORT  0x1005     /* Unexport Call        @DBA*/
    #define  GXPL_EXPTERM   0x1006     /* Exporter has terminated   */
    #define  GXPL_TERM      0x1007     /* Termination Call          */
    #define  GXPL_MTPTCHG   0x1008     /* Mount Point Change   @PNA*/

    /* The Vnode operation values are the same as the pfsi_vnop
     | constants listed above, i.e. VN_OPEN, VN_RDWR, etc.
     */

    /*----------------------------------------------------------*/
    /* Byte Range Lock Parameters                               */
    /*----------------------------------------------------------*/
    struct gxlk {
         int   gxl_version;        /*  gxlk version number         */
         int   gxl_lckcmd;         /*  Lock Cmd: F_SETLK, etc.     */
```

```
        int   gxl_lcktype;        /*  Lock Type: F_RDLCK, etc.     */
        int   gxl_brbh;           /*  Range Beginning, high word   */
        int   gxl_brbl;           /*  Range Beginning, low word    */
        int   gxl_breh;           /*  Range End, high word         */
        int   gxl_brel;           /*  Range End, low word          */
        int   gxl_blkpid;         /*  Blocking PID                 */
        int       rsvd[2];
    };

      /* gxl_lckcmd and gxl_lcktype values are defined in fcntl.h. */

      #define GXL_VER0   0            /* First gxlk version        */
      #define GXL_EOFH   0x7FFFFFFF   /* End-Of-File High word     */
      #define GXL_EOFL   0xFFFFFFFF   /*             Low word      */

      /* gx_optparm values for GXPL_MTPTCHG                    @05A*/
      /*   The field must be cast to an (int) to be used here. @05A*/
                                                             /*@05A*/
      #define GXPL_MTPT_UNMOUNT  0   /* Mount Point Unmounted  @05A*/
      #define GXPL_MTPT_MOUNT    1   /* Mounting on Mt Pt      @05A*/
      #define GXPL_REMOUNT_RO    3   /* File Sys ReMount(RO)   @05A*/
      #define GXPL_REMOUNT_RW    4   /* File Sys ReMount(RW)   @05A*/

 /*------------------------------------------------------------------*/
 /* Exit Routine Prototype - as called by the LFS                    */
 /*------------------------------------------------------------------*/
  #pragma linkage(gx_exitrtn, OS)
  void gx_exitrtn (GXPL *);

#pragma page()
/*********************************************************************/
/*                                                                   */
/* Operating System Interface (OSI) Services                         */
/*                                                                   */
/*********************************************************************/

/*-------------------------------------------------------------------*/
/* Macros used to invoke the OSI services                            */
/*                                                                   */
/*   The OSIT table address must be saved during initialization     */
/*     and made available at the time of an OSI service call.        */
/*   Refer to the prolog for details on using this macro.            */
/*-------------------------------------------------------------------*/
  #ifndef _OSIT_PTR                 /* Establish the default osi_ptr */
     #define _OSIT_PTR osit_ptr
  #endif

/*-------------------------------------------------------------------*/
/* OSI Service Names                                                 */
/*   The OSI services are called with these names and the macros     */
/*     use the OSIT table to find the associated routine.            */
/*                                                                   */
/*   For example:  osi_wait(OSI_SETUP, osiaddr, &rc);                */
/*-------------------------------------------------------------------*/
  #define  osi_getvnode    _OSICALL(GETVNODE)
  #define  osi_mountstatus _OSICALL(MOUNTSTATUS)             /*@D4A*/
  #define  osi_ctl         _OSICALL(CTL)                     /*@DAA*/
  #define  osi_selpost     _OSICALL(SELPOST)
  #define  osi_wait        _OSICALL(WAIT)
  #define  osi_post        _OSICALL(POST)
  #define  osi_signal      _OSICALL(SIGNAL)
  #define  osi_sleep       _OSICALL(SLEEP)                   /*@D6A*/
  #define  osi_wakeup      _OSICALL(WAKEUP)                  /*@D6A*/
  #define  osi_kmsgget     _OSICALL(KMSGGET)                 /*@D6A*/
  #define  osi_kmsgsnd     _OSICALL(KMSGSND)                 /*@D6A*/
  #define  osi_kmsgrcv     _OSICALL(KMSGRCV)                 /*@D6A*/
  #define  osi_kmsgctl     _OSICALL(KMSGCTL)                 /*@D6A*/
```

```
  #define  osi_kipcget     _OSICALL(KIPCGET)                /*@DDA*/
  #define  osi_uiomove     _OSICALL(UIOMOVE)          /*      @D7A*/
  #define  osi_copyin      _OSICALL(COPYIN)           /*      @D7A*/
  #define  osi_copyout     _OSICALL(COPYOUT)          /*      @D7A*/
  #define  osi_thread      _OSICALL(THREAD)           /*      @D7A*/
  #define  osi_getcred     _OSICALL(GETCRED)          /*      @P6A*/
  #define  osi_upda        _OSICALL(UPDA)                   /*@DGA*/
  #define  osi_sched       _OSICALL(SCHED)                  /*@DGA*/
  #define  osi_lkfs        _OSICALL(LKFS)                   /*@DJA*/
  #define  osi_ctrace      _OSICALL(CTRACE)                 /*@DIA*/
  #define  osi_socket      _OSICALL(SOCKET)                 /*@DKA*/
  #define  osi_copy64      _OSICALL(COPY64)                 /*@PMA*/

    /* Internal Macro used to invoke the OSI_ service from the OSIT */
  #ifndef   _OSICALL
    #define _OSICALL(op) ((_OSIT_PTR) -> osit_ ## op)
  #endif

/*-----------------------------------------------------------------*/
/* OTHDPRM - Parameter structure input to osi_thread      (BPXZTPRM)*/
/*-----------------------------------------------------------------*/
 typedef struct s_othdprm {
          CBHDR ot_hdr;           /*+00 Id & Length               */
          char  ot_modname[64];   /*+08 Name of module to fetch   */
          void  *ot_parms;        /*+48 Pointer to parms to pass
                                        to module and(maybe) exit */
          char  ot_exitname[64];  /*+4C Name of exit routine      */
                                  /*+8C Input option flags        */
          BIT   ot_sigwait :1;    /*     Signal enabled wait      */
          BIT   ot_nowait  :1;    /*     no wait                  */
          BIT   ot_releasemods:1; /*     release modules when done */
          BIT   ot_rsvrd1:29;     /*     reserved                 */
          char  ot_rsrvd2[8];     /*+90 reserved              @DDC*/
   } OTHDPRM ;                                            /* @D7A*/

      #define  OTHDPRM_ID   "TPRM"
      #define  OTHDPRM_HDR  {{OTHDPRM_ID}, sizeof(OTHDPRM)}

/*-----------------------------------------------------------------*/
/* OTHDCRCV - osi_thread called routine recovery block             */
/*                                                                 */
/* This is the second parameter passed to the routine specified    */
/* in ot_modname and the "PARAM" for the ESTAEX routine.           */
/*-----------------------------------------------------------------*/
 typedef struct s_othdcrcv {
          void  *otr_rcvyrtn;     /*+00 Pointer to called module's
                                        recovery routine          */
          void  *otr_parms;       /*+04 Pointer to parms to pass
                                        to called module's
                                        recovery routine          */
          long  reserved1;        /*+08 Reserved                  */
          long  reserved2;        /*+0C Reserved                  */
          char  work_area[496];   /*+10 Work area for ESTAEX rtn  */
   } OTHDCRCV ;                                           /* @DMA*/

/*-----------------------------------------------------------------*/
/* OGCDPRM - Parameter structure input to osi_getcred     (BPXZCPRM)*/
/*-----------------------------------------------------------------*/
 typedef struct s_ogcdprm {
          CBHDR oc_hdr;           /*+00 Id & Length           I   */
          int   oc_real_uid;      /*+08 Real uid              O   */
          int   oc_effective_uid; /*+0C Effective uid         O   */
          int   oc_saved_uid;     /*+10 Saved uid             O   */
          int   oc_real_gid;      /*+14 Real gid              O   */
          int   oc_effective_gid; /*+18 Effective gid         O   */
          int   oc_saved_gid;     /*+1C Saved gid             O   */
          int   oc_maxsgids;      /*+20 Maximum number of
```

```
                                        supplementary gids
                                        there is room for.
                                        Set to actual number
                                        if not room for all   I/O */
        int   oc_numsgids;      /*+24 Number of supplementary
                                        gids returned        O   */
        int   *oc_gid_list;     /*+28 Pointer to array of
                                        supplementary gids    I   */
   } OGCDPRM ;                                           /* @P6A*/

      #define  OGCDPRM_ID   "CPRM"
      #define  OGCDPRM_HDR  {{OGCDPRM_ID}, sizeof(OGCDPRM)}

/*------------------------------------------------------------------*/
/* Time Interval - Input to osi_sleep and osi_wait             @P8A*/
/*   Double word S/390 timer units, or (time[0]*1.04) sec. approx.  */
/*------------------------------------------------------------------*/
   struct time_int  {
           unsigned long time[2];
     };

/*------------------------------------------------------------------*/
/* OSI LkFs Parameter                                               */
/*   Passed to osi_LkFs service.                                @DJA*/
/*------------------------------------------------------------------*/
 typedef struct s_osilparm {   /* osi LkFs parameter block      @DJA*/
          int osil_length;    /* Length                        @DJA*/
          TOKSTR *osil_tokstr;/* Cjar pointer                  @DJA*/
          int osil_devno;     /* Device number                 @DJA*/
          int osil_cmdcode;   /* Command Code (lock or unlk)   @DJA*/
          GTOK osil_handle;   /* Vfs lock handle               @DJA*/
  } OSILPARM;                                             /* @DJA*/

  #define OSIL_LOCK  1       /* Lock cmd code for osil_parm     @DJA*/
  #define OSIL_UNLK  2       /* Unlock cmd code for osil_parm   @DJA*/


/*------------------------------------------------------------------*/
/* osi_copy64 Parameter                                         @PMA*/
/*------------------------------------------------------------------*/
    struct copy64_struct {                                 /*@PMA*/
        int         c64_length;            /* Struct Length       */
        BIT                          :20;   /* Flags               */
        BIT         c64_dontincrsrc   :1;   /* 0=Add Len to Source */
        BIT         c64_dontincrdest  :1;   /* 0=Add Len to Dest   */
        BIT         c64_gotrecovery   :1;   /* 1=PFS has own FRR   */
        BIT         c64_direction     :1;   /* 0=Out, 1=In         */
        BIT         c64_keybits      :4;    /* User's storage key  */
        BIT                          :4;
        ADDR64      c64_sourcebuff;         /* Source              */
        ADDR64      c64_destbuff;           /* Destination         */
        int         c64_CLrsvd;             /* (reserved)          */
        int         c64_copylen;            /* Move length         */
        int         c64_sourcealet;
        int         c64_destalet;
        int         c64_rc;
        int         c64_rsn;
        char        c64_workarea[64];
    };

      #define  C64_OUT   0
      #define  C64_IN    1


/*------------------------------------------------------------------*/
/* OSI Services Prototypes                                          */
/*------------------------------------------------------------------*/
```

```
typedef void OSI_GETVNODE(int ent,      /* Entry Code              */
                     TOKSTR *,     /* Object's Parent's Tokstr*/
                       ATTR *,     /* Attr of the new object  */
                       GTOK *,     /* PFS File Token for obj  */
                    O_VNTOK *,     /* Object's vnode token    */
                  int *retval, int *retcode, int *rsncode);

typedef void OSI_MOUNTSTATUS(int ent,   /* Entry Code          @D4A*/
                          int devno, /* Devno (mt_stdev)    @D5C*/
                  int *retval, int *retcode, int *rsncode);

typedef void OSI_WAIT(int ent,          /* Entry Code              */
                    OSI *,          /* Caller's (waiter's) OSI */
                  int *rc,
                   ...   );         /* waitx parms:        @P8A
                                           int wait_flags
                                        struct time_int *    */

typedef void OSI_POST(WPTOK *,          /* osi_token of waiter     */
                  int *rc);

typedef void OSI_SIGNAL(OSI *,          /* Caller's OSI            */
               int    pid,          /* Target's osi_pid value  */
               int sigval,          /* Signal to issue         */
               int sigopt,          /* Signal options          */
              int *retval, int *retcode, int *rsncode);

typedef void OSI_SELPOST(SELTOK *,      /* Vn_select's select token*/
                 int *retval, int *retcode, int *rsncode);

typedef void OSI_SLEEP(OSI *,           /* Caller's OSI        @D6A*/
                  int resid,        /* Resource id             */
           struct time_int *,       /* Timeout interval    @P8C*/
                int *retval, int *retcode, int *rsncode);

typedef void OSI_WAKEUP(int resid,      /* Resource id         @D6A*/
                   int pfsid,       /* Pfs id                  */
                 int *retval, int *retcode, int *rsncode);

typedef void OSI_KMSGGET(int msgqkey,   /* Message Q Id   @D6A @D7C*/
                    int msgflag,    /* Flag field              */
                 int *retval, int *retcode, int *rsncode);

typedef void OSI_KMSGSND(int msgqkey,   /* Message Q Id   @D6A @D7C*/
                   void *msgaddr,   /* Message address         */
                    int msgalet,    /* Message alet            */
                    int msgsize,    /* Message size            */
                    int msgflag,    /* Flag field              */
                 int *retval, int *retcode, int *rsncode);

typedef void OSI_KMSGRCV(int msgqkey,   /* Message Q Id   @D6A @D7C*/
                   void *msgaddr,   /* Message address         */
                    int msgalet,    /* Message alet            */
                    int msgsize,    /* Message size            */
                    int msgtype,    /* Message type            */
                    int msgflag,    /* Flag field              */
                 int *retval, int *retcode, int *rsncode);

typedef void OSI_KMSGCTL(int msgqkey,   /* Message Q Id   @D6A @D7C*/
                    int msgcmd,     /* Message command         */
                   void *msgbuff,   /* Message bufffer         */
                 int *retval, int *retcode, int *rsncode);

typedef void OSI_KIPCGET(int ipctoken,  /* IPC token           @DDA*/
                   void *ipcbuff,   /* Output bufffer          */
                    int bufflen,    /* IPC buffer length       */
                    int ipccmd,     /* IPC command             */
```

```
                      int *retval, int *retcode, int *rsncode);

typedef void OSI_UIOMOVE(OSI *,         /* OSI struct          @D7A*/
                         char *uiomworkarea, /* work area for use by
                                                 uiomove       @D7A*/
                         char *pfsbuf , /* Pfs buffer          @D7A*/
                         int pfsbufalet,/* Alet for the PFS buf@D7A*/
                         int movelen,   /* number of bytes to move */
                         UIO *,         /* Uio structure           */
                         int *retval, int *retcode, int *rsncode);

typedef void OSI_COPYIN(char *desbuf,   /* destination buffer  @D6A*/
                        int desbufalet, /* destination buffer alet */
                        char *srcbuf,   /* source buffer           */
                        int srcbufalet, /* source buffer alet      */
                        int srckey,     /* source storage key      */
                        int movelen,    /* length to move          */
                        int *retval, int *retcode, int *rsncode);

typedef void OSI_COPYOUT(char *desbuf,  /* destination buffer  @D6A*/
                         int desbufalet,/* destination buffer alet */
                         char *srcbuf,  /* source buffer           */
                         int srcbufalet,/* source buffer alet      */
                         int deskey,    /* destination storage key */
                         int movelen,   /* length to move          */
                         int *retval, int *retcode, int *rsncode);

typedef void OSI_THREAD(OSI *,          /* OSI                 @D7A*/
                        OTHDPRM *,      /* Osit_Thread parm struct */
                        int *retval, int *retcode, int *rsncode);

typedef void OSI_GETCRED(OSI *,         /* OSI                 @P6A*/
                         char *workarea, /* 3K work area for
                                            use by getcred         */
                         int   alet,    /* alet for getcred parm and
                                           supplementary gid list */
                         OGCDPRM *,     /* Osit_Getcred parm struct*/
                         int *retval, int *retcode, int *rsncode);

typedef void OSI_CTL   (int  cmd,       /* Command Code        @DAA*/
                        int  arglen,    /* Argument Length         */
                        char *arg,      /* Argument Length         */
                        int  *retval, int *retcode, int *rsncode);

typedef void OSI_UPDA  (GTOK *lfs_asytok,   /* LFS's Token      @DGA*/
                        GTOK *pfs_asytok ); /* PFS's Token          */

typedef void OSI_SCHED (GTOK *lfs_asytok,   /* LFS's Token      @DGA*/
                        int  *retval, int *retcode, int *rsncode);

typedef void OSI_CTRACE(char *pfs_name, /* name of the PFS     @DIA*/
                        char *workarea, /* 3K work area for
                                           use by osi_ctrace       */
                        int  arglen,    /* Argument Length         */
                        char *arg,      /* Argument Length         */
                        int  *retval, int *retcode, int *rsncode);

typedef void OSI_SOCKET(char *function, /* socket function     @DKA*/
                        ...);           /* args for equiv BPX1xxx  */


typedef void OSI_LKFS (OSILPARM *,             /* LkFs parm     @PDC*/
                        int  *retval, int *retcode, int *rsncode);

typedef void OSI_COPY64(struct copy64_struct *,             /*2@PMA*/
                        char *workarea  );   /* 512 Byte work area */
```

```
                     /* OS linkage pragmas for the Services      */
  #pragma linkage(OSI_GETVNODE,OS)
  #pragma linkage(OSI_MOUNTSTATUS,OS)                        /*@D4A*/
  #pragma linkage(OSI_CTL,OS)                                /*@DAA*/
  #pragma linkage(OSI_SELPOST,OS)
  #pragma linkage(OSI_WAIT,OS)
  #pragma linkage(OSI_POST,OS)
  #pragma linkage(OSI_SIGNAL,OS)
  #pragma linkage(OSI_SLEEP,OS)                              /*@D6A*/
  #pragma linkage(OSI_WAKEUP,OS)                             /*@D6A*/
  #pragma linkage(OSI_KMSGGET,OS)                            /*@D6A*/
  #pragma linkage(OSI_KMSGSND,OS)                            /*@D6A*/
  #pragma linkage(OSI_KMSGRCV,OS)                            /*@D6A*/
  #pragma linkage(OSI_KMSGCTL,OS)                            /*@D6A*/
  #pragma linkage(OSI_KIPCGET,OS)                            /*@DDA*/
  #pragma linkage(OSI_UIOMOVE,OS)                            /*@D6A*/
  #pragma linkage(OSI_COPYIN,OS)                      /*       @D7A*/
  #pragma linkage(OSI_COPYOUT,OS)                     /*       @D7A*/
  #pragma linkage(OSI_THREAD,OS)                      /*       @D7A*/
  #pragma linkage(OSI_GETCRED,OS)                     /*       @P6A*/
  #pragma linkage(OSI_UPDA,OS)                               /*@DGA*/
  #pragma linkage(OSI_SCHED,OS)                              /*@DGA*/
  #pragma linkage(OSI_CTRACE,OS)                             /*@DIA*/
  #pragma linkage(OSI_LKFS,OS)                               /*@DJA*/
  #pragma linkage(OSI_SOCKET,OS)                             /*@DKA*/
  #pragma linkage(OSI_COPY64,OS)                             /*@PMA*/

/*-------------------------------------------------------------------*/
/* OSIT - Operating System Interface Table               (BPXZOSIT)*/
/*-------------------------------------------------------------------*/
 typedef struct s_osit {
           CBHDR  osit_hdr;              /*+00 ID & Length         */
           short  osit_ver;              /*+08 Version             */
           short  osit_rsvd1;
                                         /* Function Pointers      */
           OSI_GETVNODE    *osit_GETVNODE;    /* +0C                 */
           OSI_MOUNTSTATUS *osit_MOUNTSTATUS; /* +10     Ver3        */
           OSI_CTL         *osit_CTL;         /* +14    Ver2         */
           void            *osit_intern1;     /* +18                 */
           OSI_SELPOST     *osit_SELPOST;     /* +1C                 */
           OSI_WAIT        *osit_WAIT;        /* +20                 */
           OSI_POST        *osit_POST;        /* +24                 */
           OSI_SIGNAL      *osit_SIGNAL;      /* +28                 */
           OSI_SLEEP       *osit_SLEEP;       /* +2C     Ver3        */
           OSI_WAKEUP      *osit_WAKEUP;      /* +30     Ver3        */
           OSI_KMSGGET     *osit_KMSGGET;     /* +34     Ver3        */
           OSI_KMSGSND     *osit_KMSGSND;     /* +38     Ver3        */
           OSI_KMSGRCV     *osit_KMSGRCV;     /* +3C     Ver3        */
           OSI_KMSGCTL     *osit_KMSGCTL;     /* +40     Ver3        */
           OSI_KIPCGET     *osit_KIPCGET;     /* +44     Ver3        */
           OSI_UIOMOVE     *osit_UIOMOVE;     /* +48     Ver3        */
           OSI_COPYIN      *osit_COPYIN;      /* +4C    Ver2         */
           OSI_COPYOUT     *osit_COPYOUT;     /* +50    Ver2         */
           OSI_THREAD      *osit_THREAD;      /* +54     Ver3        */
           OSI_GETCRED     *osit_GETCRED;     /* +58     Ver3  @P6C */
           OSI_SCHED       *osit_SCHED;       /* +5C           @DGA*/
           OSI_UPDA        *osit_UPDA;        /* +60           @DGA*/
           OSI_LKFS        *osit_LKFS;        /* +64     Ver4  @DJA*/
           OSI_CTRACE      *osit_CTRACE;      /* +68           @DIA*/
           OSI_SOCKET      *osit_SOCKET;      /* +6C           @DKA*/
           void            *osit_rsvdB;       /* +70               */
                                              /*--- End of Ver4 @PMA*/
           OSI_COPY64      *osit_COPY64;      /* +74           @PMA*/
           void            *osit_rsvdC;       /*               @PMA*/
           void            *osit_rsvdD;       /*               @PMA*/
           void            *osit_rsvdE;       /* +80           @PMA*/
           void            *osit_rsvdF;       /*               @PMA*/
```

```
        void           *osit_rsvdG;      /*               @PMA*/
        void           *osit_rsvdH;      /*               @PMA*/

  } OSIT;

     /* Version numbers           */
     #define OSIT_VER1  1        /* Rel 1 and Rel 2 Base           */
     #define OSIT_VER2  2        /* Rel 2 with copyin,copyout,ctl   */
     #define OSIT_VER3  3        /* Rel 3 sleep,wkup,kmsg,uiom,thrd */
     #define OSIT_VER4  4        /* Rel 4 lkfs                 @DJA*/
     #define OSIT_VER5  5        /* Ver 5 copy64               @PMA*/
     #define OSIT_VER6  6        /* Ver 6 sysplex zfs          @E0A*/

/*----------------------------------------------------------------*/
/* Constants used with the Service calls                          */
/*----------------------------------------------------------------*/

  /* Input Entry Codes for osi_getvnode                          */
  #define OSI_BUILDVNOD       1      /* Build Vnode               */
  #define OSI_BUILDVNODNL     2      /* Build Vnode without locks   */
  #define OSI_RTNVNOD         3      /* Return unused Vnode         */
  #define OSI_BUILDVNODXL     4      /* Build Vnode-excl locks  @P3A*/
  #define OSI_UPDATEVNODE     5      /* Update PFS Area in Vnode@DGA*/
  #define OSI_ASSOCIATE       7      /* Update PFS Area in Vnode@DIA*/
  #define OSI_ASSOCIATENL     8      /* Update PFS Area in Vnode@DIA*/
  #define OSI_MEMCRITICAL     9      /* Crit PFS Storage Cond   @05A*/
  #define OSI_INACTASAP       10     /* Inact vnod asap         @E0A*/

  /* Input Entry Codes for osi_mountstatus                     @D4C*/
  #define OSI_MOUNTCOMPLETE   1      /* Asynchronous mount complete
                                                             @D4C*/

  /* Input Entry Codes for osi_wait                              */
  #define OSI_SETUP           1      /* Setup request              */
  #define OSI_SETUPSIG        4      /* Setup with signals         */
  #define OSI_SUSPEND         2      /* Wait request               */
  #define OSI_WAITX           5      /* Wait ext request with Latch
                                        and Timer control      @D6A*/
  #define OSI_INIT            6      /* Init OSI for a Task     @DGA*/
  #define OSI_INIT2           7      /* Init OSI with Length    @PFA
                                      osi_hdr.cblen=sizeof(OSI) @PFA*/

  /* Output Return Codes from osi_wait                           */
  #define OSI_POSTED          0      /* Osi_post was called.       */
  #define OSI_SIGNALRCV       4      /* Signal has been received.  */
  #define OSI_SHUTDOWN        8      /* OMVS is shutting down.     */
  #define OSI_UNMOUNTED      16      /* File System was unmounted  */
  #define OSI_POSTERTRM      18      /* Poster has terminated   @DBA*/
  #define OSI_TIMEOUT        28      /* Timer interval expired  @D7C*/
  #define OSI_ABEND          32      /* Abend occurred.            */
  #define OSI_BADPARM        34      /* Bad parm passed on call.   */
  #define OSI_ESTAEF         36      /* Estae setup failure occurred*/
  #define OSI_SYSTEMERR      38      /* System Error occurred.     */
  #define OSI_FRRACTIVE      40      /* FRR Active when signals
                                        enabled                @D6A*/
  /* Output Return Codes from osi_post  (in addition to above)  @DDA*/
  #define OSI_NOTWAITING      4      /* Waiter has gone         @DDA*/

  /* Flag values for ot_option_flags on osi_thread call        @D7A*/
  #define OSI_SIGWAIT     0x80000000 /* Wait caller's task with
                                        signals enabled            */
  #define OSI_NOWAIT      0x40000000 /* Don't wait caller's task    */
  #define OSI_RELEASEMODS 0x20000000 /* Release modules when done   */

  /* Flag values for wait_flags on osi_wait(waitx) calls       @P8A*/
   #define OSI_WTDROPLOCKS 0x00000001  /* Drop Locks over wait  @P8A*/
```

```
/*---------------------------------------------------------------------*/
/* Information used for loading the OSIT into a separate addr space */
/*---------------------------------------------------------------------*/
  #define OSIT_INIT  "BPXVOSIT"      /* The module to load & call   */

  typedef void OSIT_INITMOD (        /* Prototype for the call:     */
                         OSIT **,  /*  Output is a ptr to an OSIT */
                         int *retcode, int *rsncode);
  #pragma linkage(OSIT_INITMOD,OS)   /*  Called with OS linkgage    */

/*---------------------------------------------------------------------*/
/* Prototype of the PFS Initialization Routine                       */
/*   This routine is attached as an MVS task and invoked by the      */
/*   system with the following parameters:                           */
/*---------------------------------------------------------------------*/
  void pfsinit (PFSI *, PFSNAME *, PFSPARM *, void *, OSIT *);

  #pragma linkage(pfsinit,OS)         /* Is invoked with OS linkage */

/*---------------------------------------------------------------------*/
/* Prototypes of the Vnode and VFS operation routines.               */
/*   These routines are called by the LFS to perform their functions*/
/*---------------------------------------------------------------------*/

                 /* File and Directory oriented operations          */
  void vn_open    (TOKSTR *, OSI *, CRED *,
                     int *open_flags,
                     int *retval, int *retcode, int *rsncode);
  void vn_close   (TOKSTR *, OSI *, CRED *,
                     int *open_flags,
                     int *retval, int *retcode, int *rsncode);
  void vn_readdir (TOKSTR *, OSI *, CRED *,
                     UIO *,
                     int *retval, int *retcode, int *rsncode);
  void vn_readlink(TOKSTR *, OSI *, CRED *,
                     UIO *,
                     int *retval, int *retcode, int *rsncode);
  void vn_create  (TOKSTR *, OSI *, CRED *,
                     int *namelen, char *name, ATTR *, O_VNTOK *,
                     int *retval, int *retcode, int *rsncode);
  void vn_mkdir   (TOKSTR *, OSI *, CRED *,
                     int *namelen, char *name, ATTR *, O_VNTOK *,
                     int *retval, int *retcode, int *rsncode);
  void vn_symlink (TOKSTR *, OSI *, CRED *,
                     int *namelen, char *name,  ATTR *,
                     int *symlen,  char *symlink,
                     int *retval, int *retcode, int *rsncode);
  void vn_lookup  (TOKSTR *, OSI *, CRED *,
                     int *namelen, char *name, O_VNTOK *,
                     int *retval, int *retcode, int *rsncode);
  void vn_getattr (TOKSTR *, OSI *, CRED *,
                     ATTR *,
                     int *retval, int *retcode, int *rsncode);
  void vn_setattr (TOKSTR *, OSI *, CRED *,
                     ATTR *,
                     int *retval, int *retcode, int *rsncode);
  void vn_access  (TOKSTR *, OSI *, CRED *,
                     int *access_intent,
                     int *retval, int *retcode, int *rsncode);
  void vn_trunc   (TOKSTR *, OSI *, CRED *,
                     int *offset,
                     int *retval, int *retcode, int *rsncode);
  void vn_fsync   (TOKSTR *, OSI *, CRED *,
                     int *retval, int *retcode, int *rsncode);
  void vn_link    (TOKSTR *, OSI *, CRED *,
                     int *namelen, char *name, TOKSTR *,
                     int *retval, int *retcode, int *rsncode);
```

```
void vn_rmdir   (TOKSTR *, OSI *, CRED *,
                   int *namelen, char *name,
                   int *retval, int *retcode, int *rsncode);
void vn_remove  (TOKSTR *, OSI *, CRED *,
                   int *namelen, char *name,
                   int *retval, int *retcode, int *rsncode);
void vn_rename  (TOKSTR *, OSI *, CRED *,
                   int *oldlen, char *oldname,
                   int *newlen, char *newname, TOKSTR *,
                   int *retval, int *retcode, int *rsncode);
void vn_audit   (TOKSTR *, OSI *, CRED *,
                   int *retval, int *retcode, int *rsncode);

               /* File System oriented operations                 */
void vfs_mount  (TOKSTR *, OSI *, CRED *,
                   MTAB *, O_VNTOK *,
                   int *retval, int *retcode, int *rsncode);
void vfs_umount (TOKSTR *, OSI *, CRED *,
                   int *unmount_options,
                   int *retval, int *retcode, int *rsncode);
void vfs_statfs (TOKSTR *, OSI *, CRED *,
                   FSATTR *,
                   int *retval, int *retcode, int *rsncode);
void vfs_sync   (TOKSTR *, OSI *, CRED *,
                   int *retval, int *retcode, int *rsncode);
void vfs_inact  (TOKSTR *, OSI *, CRED *,                 /*@PPA*/
                   struct s_iab *, int *iablen,
                   int *retval, int *retcode, int *rsncode);
void vfs_vget   (TOKSTR *, OSI *, CRED *,
                   FID *, O_VNTOK *,
                   int *retval, int *retcode, int *rsncode);

               /* General operations                              */
void vn_select    (TOKSTR *, OSI *, CRED *,
                     SELTOK *,
                     int *sel_function,
                     int *sel_options,
                     char *pfsworkptr,
                     int *retval, int *retcode, int *rsncode);
void vfs_batsel   (TOKSTR *, OSI *, CRED *,
                     int *rsvd1,
                     int *sel_function,
                     BSIC *,
                     int *rsvd2,
                     int *retval, int *retcode, int *rsncode);
void vn_rdwr      (TOKSTR *, OSI *, CRED *,
                     int *open_flags, UIO *,
                     int *retval, int *retcode, int *rsncode);
void vn_readwritev(TOKSTR *, OSI *, CRED *,
                     int *open_flags, UIO *,
                     int *retval, int *retcode, int *rsncode);
void vn_inactive  (TOKSTR *, OSI *, CRED *,
                     int *retval, int *retcode, int *rsncode);
void vn_ioctl     (TOKSTR *, OSI *, CRED *,
                     int *open_flags,
                     int *cmd, int *arglen, char *arg,
                     int *retval, int *retcode, int *rsncode);
void vn_pathconf  (TOKSTR *, OSI *, CRED *,               /*@D5A*/
                     int *pathconf_option,
                     int *retval, int *retcode, int *rsncode);
void vn_recovery  (TOKSTR *, OSI *, CRED *,               /*@D5A*/
                     struct osirtoken *,
                     int *retval, int *retcode, int *rsncode);
void vfs_recovery (TOKSTR *, OSI *, CRED *,
                     struct osirtoken *,
                     int *retval, int *retcode, int *rsncode);
void vfs_pfsctl   (TOKSTR *, OSI *, CRED *,
```

```
                     int *cmd, UIO *,
                     int *retval, int *retcode, int *rsncode);
  void vn_cancel    (TOKSTR *, OSI *, CRED *,                 /*@DGA*/
                     struct vncanflags *,
                     GTOK   *pfs_asytok,
                     GTOK   *lfs_asytok,
                     int *retval, int *retcode, int *rsncode);

               /* Socket Network (domain) oriented operations     */
  void vfs_network  (TOKSTR *, OSI *, CRED *,
                     NETW *,
                     int *retval, int *retcode, int *rsncode);
  void vfs_socket   (TOKSTR *, OSI *, CRED *,   /* socket|socketpair*/
                     int *domain, int *type, int *protocol,
                     int *dim, O_VNTOK (*vntoks)[2],
                     int *retval, int *retcode, int *rsncode);
  void vfs_gethost  (TOKSTR *, OSI *, CRED *,   /* get host id|name */
                     int *namelen, char *name,
                     int *retval, int *retcode, int *rsncode);

               /* Socket oriented operations                      */

  void vn_accept    (TOKSTR *, OSI *, CRED *,
                     int *addrlen, _SOCKADDR *,
                     int *open_flags, O_VNTOK *,
                     int *retval, int *retcode, int *rsncode);
  void vn_bind      (TOKSTR *, OSI *, CRED *,
                     int *addrlen, _SOCKADDR *,
                     int *retval, int *retcode, int *rsncode);
  void vn_connect   (TOKSTR *, OSI *, CRED *,
                     int *addrlen, _SOCKADDR *,
                     int *open_flags,
                     int *retval, int *retcode, int *rsncode);
  void vn_getname   (TOKSTR *, OSI *, CRED *,   /* peername|sockname*/
                     int *function,
                     int *addrlen, _SOCKADDR *,
                     int *retval, int *retcode, int *rsncode);
  void vn_listen    (TOKSTR *, OSI *, CRED *,
                     int *backlog,
                     int *retval, int *retcode, int *rsncode);
  void vn_sndrcv    (TOKSTR *, OSI *, CRED *,
                     int *open_flags, UIO *, int *sr_flags,
                     int *retval, int *retcode, int *rsncode);
  void vn_sndtorcvfm(TOKSTR *, OSI *, CRED *,
                     int *open_flags, UIO *, int *sr_flags,
                     int *addrlen, _SOCKADDR *,
                     int *retval, int *retcode, int *rsncode);
  void vn_srmsg     (TOKSTR *, OSI *, CRED *,
                     int *open_flags, UIO *, int *sr_flags,
                     int *retval, int *retcode, int *rsncode);
  void vn_shutdown  (TOKSTR *, OSI *, CRED *,
                     int *how,
                     int *retval, int *retcode, int *rsncode);
  void vn_sockopt   (TOKSTR *, OSI *, CRED *,   /* Get|Set sockopt  */
                     int *function, int *level,
                     int *optname, int *optvallen, char *optval,
                     int *retval, int *retcode, int *rsncode);
  void vn_srx       (TOKSTR *, OSI *, CRED *,                 /*@PFA*/
                     int *open_flags, UIO *,
                     int *retval, int *retcode, int *rsncode);
  void vn_anr       (TOKSTR *, OSI *, CRED *,                 /*@PHA*/
                     struct anr_struct *, UIO *,              /*@PIC*/
                     int *open_flags, O_VNTOK *,
                     int *retval, int *retcode, int *rsncode);

#pragma page()
               /* The PFS operations are invoked with OS linkage  */
```

```
  #pragma linkage(vn_open     ,OS)
  #pragma linkage(vn_close    ,OS)
  #pragma linkage(vn_rdwr     ,OS)
  #pragma linkage(vn_readdir  ,OS)
  #pragma linkage(vn_readlink,OS)
  #pragma linkage(vn_create   ,OS)
  #pragma linkage(vn_mkdir    ,OS)
  #pragma linkage(vn_symlink  ,OS)
  #pragma linkage(vn_lookup   ,OS)
  #pragma linkage(vn_inactive,OS)
  #pragma linkage(vn_getattr  ,OS)
  #pragma linkage(vn_setattr  ,OS)
  #pragma linkage(vn_access   ,OS)
  #pragma linkage(vn_trunc    ,OS)
  #pragma linkage(vn_fsync    ,OS)
  #pragma linkage(vn_link     ,OS)
  #pragma linkage(vn_rmdir    ,OS)
  #pragma linkage(vn_remove   ,OS)
  #pragma linkage(vn_rename   ,OS)
  #pragma linkage(vn_audit    ,OS)
  #pragma linkage(vn_pathconf,OS)                            /*@D5A*/
  #pragma linkage(vn_recovery,OS)                            /*@D5A*/
  #pragma linkage(vn_cancel   ,OS)                           /*@DGA*/
  #pragma linkage(vn_ioctl    ,OS)
  #pragma linkage(vn_select   ,OS)
  #pragma linkage(vn_accept   ,OS)
  #pragma linkage(vn_bind     ,OS)
  #pragma linkage(vn_connect  ,OS)
  #pragma linkage(vn_getname  ,OS)
  #pragma linkage(vn_listen   ,OS)
  #pragma linkage(vn_sndrcv   ,OS)
  #pragma linkage(vn_srmsg    ,OS)
  #pragma linkage(vn_shutdown,OS)
  #pragma linkage(vn_sockopt  ,OS)
  #pragma linkage(vn_readwritev,OS)
  #pragma linkage(vn_sndtorcvfm,OS)
  #pragma linkage(vn_srx      ,OS)
  #pragma linkage(vn_anr      ,OS)

  #pragma linkage(vfs_mount    ,OS)
  #pragma linkage(vfs_umount   ,OS)
  #pragma linkage(vfs_statfs   ,OS)
  #pragma linkage(vfs_sync     ,OS)
  #pragma linkage(vfs_inact    ,OS)                          /*@PPA*/
  #pragma linkage(vfs_vget     ,OS)
  #pragma linkage(vfs_recovery,OS)
  #pragma linkage(vfs_batsel   ,OS)
  #pragma linkage(vfs_network  ,OS)
  #pragma linkage(vfs_socket   ,OS)
  #pragma linkage(vfs_gethost  ,OS)

/*-------------------------------------------------------------------*/
/* Ctrace utility                                                    */
/*------------------------------------------------------------- 4@DIA*/
struct ctrcvt  {char x[0x8c];struct ctrcve *cve;};
struct ctrcve  {char x[0xf0];struct ctrocvt *ocvt;};
struct ctrocvt {char x[0x130];unsigned int csptrace:1;};
#define TRACEISON ((*(struct ctrcvt**)0x10)->cve->ocvt->csptrace)

#pragma page()
 #ifndef _NO_PFS_KES
/*-------------------------------------------------------------------*/
/* Internal Services Prototypes                                      */
/*-------------------------------------------------------------------*/

  void * _memmove (void *, const void *, size_t);          /* @D7A */
```

```
 /*---------------------------------------------------------------*/
 /* Kernel Extension Services                                     */
 /* These functions are a subset of the OS/390 Language Environment */
 /* C functions.  LE functions are not available to PFSes and     */
 /* the functions included here may be called in their place.     */
 /*---------------------------------------------------------------*/
/*****************************************************************/
/*                                                               */
/* Name:    bcopy                                          @D7A */
/*                                                               */
/* Format:  #include <string.h>                                  */
/*          #include <bpxypfsi.h>                                */
/*          void bcopy( source, destination, length )            */
/*                                                               */
/* Description:                                                  */
/*          Copies 'length' bytes from 'source' to 'destination'. */
/*          Overlapping source and destination are handled       */
/*          correctly.                                           */
/*                                                               */
/* Returned Value:                                               */
/*          None                                                 */
/*                                                               */
/* External References: _memmove                                 */
/*                                                               */
/* Synopsis:                                                     */
/* void bcopy (const void *source, void *destination, size_t length) */
/*                                                               */
/* Related Information:                                          */
/*          <bpxypfsi.h>                                         */
/*          _memmove()                                           */
/*                                                               */
/*****************************************************************/

static                                                /*@DIA*/
void bcopy (const void *src, void *dst, size_t length)       /*@D7A*/
{
 /*
  * let _memmove do the work...
  */

       _memmove( dst, src, length );                  /*@D7A*/
}

 #pragma page()
/*****************************************************************/
/*                                                               */
/* Name:    bzero                                          @D7A */
/*                                                               */
/* Format:  #include <string.h>                                  */
/*          #include <bpxypfsi.h>                                */
/*          void bcopy( destination, length )                    */
/*                                                               */
/* Description:                                                  */
/*          Zeroes out 'length' bytes, starting at 'destination'. */
/*                                                               */
/* Returned Value:                                               */
/*          None                                                 */
/*                                                               */
/* External References:  memset                                  */
/*                                                               */
/* Synopsis:                                                     */
/* void bzero (const void *destination, size_t length)           */
/*                                                               */
/* Related Information:                                          */
/*          <bpxypfsi.h>                                         */
/*          memset()                                             */
/*                                                               */
```

```
/*****************************************************************/

static                                                  /*@DIA*/
void bzero (void *dest, size_t length)                  /*@D7A*/
{
 /*
  * let memset do the work ...
  */

        memset( dest, 0, length );                      /*@D7A*/
}

 #pragma page()
 /*--------------------------------------------------------------*/
 /* Internal Services                                            */
 /*--------------------------------------------------------------*/
/*---------------------------------------------------------------*/
/*  Name:    _memmove                                            */
/*                                                               */
/*                                                               */
/*  Purpose: Copies characters from one data object to another   */
/*           with check for overlap                              */
/*                                                               */
/*  Input: s1 - object to move the characters to                 */
/*         s2 - object to move the characters from               */
/*         n  - the number of characters to move                 */
/*                                                               */
/*  Output: Returns a pointer to object s1                       */
/*                                                               */
/*  External References:  None                                   */
/*                                                               */
/*  Description:                                                 */
/*                                                               */
/*   Copy n characters from object s2 to object s1.              */
/*   If overlay exists between s2 and s1, the move shall         */
/*   take place correctly.  A pointer to the object s1 shall     */
/*   be returned.                                                */
/*                                                               */
/*---------------------------------------------------------------*/

static                                                 /*@DIA*/
void *_memmove (register void *s1, register const void *s2,
                register size_t n) {

    register void *anchor = s1;   /* save s1 to return              */
    char         *p1;
    char         *p2;
    size_t        x;
    size_t        y;


  /*************************************************************/
  /* check for destructive overlap and if it exists, move the end */
  /* of the string first.                                      */
  /*************************************************************/

   if ( ((char *)s1 > (char *)s2) && (((char *)s2 + n) > (char *)s1)) {
       p2 = (char *)s2 + n - 1; /* point to last character to move   */
       p1 = (char *)s1 + n - 1; /* point to last position in result  */
       x = (char *)s2 + n - (char *)s1; /* # of bytes colliding      */
       y = x;

       while ( y-- > 0 )
          *p1-- = *p2--;

       /********************************************************/
       /* can move the rest quickly                            */
```

```
         /*********************************************************/
         memcpy((char *)s1, s2, n - x);
      }
      else
         /*********************************************************/
         /* otherwise, regular move                               */
         /*********************************************************/
         memcpy (s1, s2, n);

      return anchor;
}
  #endif   /* Endif _NO_PFS_KES */
 #endif    /* Endif __BPXYPFSI  */
```

# Appendix E. Assembler and C-language facilities for writing a PFS in C

This appendix contains assembler routines that can be useful for writing a PFS in C.

Replacements for the C/370™ Systems Programming Facilities routines @@XGET and @@XFREE are included. **These routines must be included in your PFS.** They are supplied in the BPXFASM sample, included here, which must be assembled and link-edited with all PFS load modules.

C-function prototypes and assembler routines are also included for the following facilities:

| | |
|---|---|
| BPXT4KGT | Allocate a page of storage |
| BPXT4KFR | Free a page of storage |
| BPXTWAIT | Wait on an ECB list |
| BPXTPOST | Post an ECB |
| BPXTEPOC | Convert time-of-day clock format to seconds-since-the-epoch |

## Assembler replacements for @@XGET and @@XFREE

```
 TITLE 'BPXFASM: File System Assembler Utilities'
*/****START OF SPECIFICATIONS*******************************************
*
*    $MOD(BPXFASM) COMP(SCPX1) PROD(BPX):
*
*01* MODULE-NAME: BPXFASM
*
*01* CSECT NAME: @@XGET and @@XFREE
*
*01* DESCRIPTIVE-NAME: HOTC Replaceable Get/Free Storage for C PFSs
*
****END OF SPECIFICATIONS*********************************************/
*
BPXFASM  CSECT
BPXFASM  AMODE ANY
BPXFASM  RMODE ANY
BPXFASM  MODID BR=NO
*
**********************************************************************
*   CSECT-NAME:  @@XGET
*
*   DESCRIPTIVE-NAME:  Allocate storage for C/370
*
*   Input: R0 - length of storage to obtain  (high bit on for storage
*                above the line).
*          R14 - Return address
*
*   Output: R0 - length of storage obtained
*           R1 - address of memory obtained
*
*   No save area is provided.
*   R2 and R4 are used as work regs.
*   Regs and Access Regs 0, 1, 14, 15 may be modified.
*
**********************************************************************
@@XGET   CSECT
@@XGET   AMODE ANY
@@XGET   RMODE ANY
```

```
         ENTRY @@XGET
*
         LR    R2,R0             Save Input Length
         LR    R4,R14            Save Return Addr
         EPAR  R15               Extract Primary ASID
         LOCASCB ASID=(R15)      Locate the Primary ASCB, Ret in R1
         USING ASCB,R1
         L     R15,ASCBXTCB      Save Xmem Resource Owning TCB
         DROP  R1
         LR    R0,R2             Restore Input Length to R0
         BALR  R2,R0             Establish Addressability
         USING *,R2
*
         LTR   R0,R0             request for below?
         BNL   BELOW             yes
         SLL   R0,1              allocate anywhere
         SRL   R0,1              clear high bit
         LTR   R2,R2             are we running below the line
         BNL   BELOW             yes, get below instead of anywhere
         STORAGE OBTAIN,LENGTH=(R0),COND=YES,SP=3,TCBADDR=(R15)
         LTR   R15,R15           successful?
         BZR   R4                yes, return
         SR    R1,R1             R1=0, R15<>0 for failure
         BR    R4                Return
BELOW    DS    0H                Get memory below the line
         STORAGE OBTAIN,LENGTH=(R0),COND=YES,LOC=BELOW,               +
               SP=3,TCBADDR=(R15)
         LTR   R15,R15           Was it successful?
         BZR   R4                yes, return
         SR    R1,R1             R1=0, R15<>0 for failure
         BR    R4                Return
*
*
***********************************************************************
*   CSECT-NAME:  @@XFREE
*
*   DESCRIPTIVE-NAME:  Free allocated storage for C/370
*
*   Input: R0 - length of storage to free
*          R1 - address of storage to free
*          R14 - Return address
*
*   No save area is provided.
*   R2 and R4 are used as work regs.
*   Regs and Access Regs 0, 1, 14, 15 may be modified.
*
***********************************************************************
@@XFREE  CSECT
@@XFREE  AMODE ANY
@@XFREE  RMODE ANY
         ENTRY @@XFREE
*
         LR    R2,R1             Save Input Addr
         ST    R0,0(R2)          Save Input Length in the passed area
         LR    R4,R14            Save Return Addr
         EPAR  R15               Extract Primary ASID
         LOCASCB ASID=(R15)      Locate the Primary ASCB, Ret in R1
         USING ASCB,R1
         L     R15,ASCBXTCB      Save Xmem Resource Owning TCB
         DROP  R1
         L     R0,0(R2)          Restore Input Length to R0
         LR    R1,R2             Restore Input Addr   to R1
         BALR  R2,R0             Establish Addressability
         USING *,R2
*
         STORAGE RELEASE,LENGTH=(R0),ADDR=(R1),SP=3,TCBADDR=(R15)
         BR    R4
```

```
*
R0       EQU   0
R1       EQU   1
R2       EQU   2
R4       EQU   4
R14      EQU   14
R15      EQU   15
*
         PRINT OFF
         IHAASCB
         PRINT ON
*
         END
```

# BPXT4KGT—Get a page of storage

This function gets a 4KB page of storage.

## C function

```
#pragma linkage(BPXT4KGT,OS)
char *BPXT4KGT (long len,long key);
```

## Assembler routine

```
************************************************************************
*   CSECT-NAME:  BPXT4KGT
*
*   DESCRIPTIVE-NAME:  Allocate storage on a page boundary with key.
*                      Storage is allocated in subpool 229.
*
*   Input: R1 - Parm list
*                 length of storage to obtain
*                 key for storage
*
*   Output: R15 - address of storage obtained
************************************************************************
BPXT4KGT CSECT
BPXT4KGT AMODE ANY
BPXT4KGT RMODE ANY
         ENTRY BPXT4KGT
         EDCPRLG
         L     R2,0(R1)          get addr of length
         L     R0,0(R2)          get length
         L     R2,4(R1)          get addr of key
         L     R2,0(R2)          get key
         SLL   R2,4              put in bits 24-27
         STORAGE OBTAIN,LENGTH=(R0),BNDRY=PAGE,COND=YES,SP=229,KEY=(2)
         LTR   R15,R15           successful?
         BZ    OUT4KGT           yes, return
         SR    R1,R1             addr=0 for failure
OUT4KGT  LR    R15,R1            return storage address
         EDCEPIL
*
```

# BPXT4KFR—free a page of storage

This function frees a 4KB page of storage.

## C function

```
#pragma linkage(BPXT4KFR,OS)
void BPXT4KFR (long len,long key,char *stor);
```

## Assembler routine

```
***********************************************************************
*   CSECT-NAME:  BPXT4KFR
*
*   DESCRIPTIVE-NAME:  Free storage allocated by BPXT4KGT
*
*   Input: R1 - Parm list
*                length of storage to free
*                key for storage
*                address of storage
***********************************************************************
BPXT4KFR CSECT
BPXT4KFR AMODE ANY
BPXT4KFR RMODE ANY
         ENTRY BPXT4KFR
         EDCPRLG
         L     R2,0(R1)          get addr of length
         L     R0,0(R2)          get length
         L     R2,4(R1)          get addr of key
         L     R2,0(R2)          get key
         SLL   R2,4              put in bits 24-27
         L     R1,8(R1)          get storage addr
         STORAGE RELEASE,LENGTH=(R0),ADDR=(R1),SP=229,KEY=(R2)
         EDCEPIL
*
```

# BPXTWAIT—wait on an ECB list

This function waits for an ECB in a list to be posted.

## C function

```
#pragma linkage(BPXTWAIT,OS)
void BPXTWAIT (ECB *ecb1,...);
```

## Assembler routine

```
***********************************************************************
*   CSECT-NAME:  BPXTWAIT
*
*   DESCRIPTIVE-NAME:  Wait for an ECB in a list to be posted
*
*   NOTES:  This routine can be called from a PFS initialization
*           routine.  It will not run in cross memory mode.
*
*   Input: R1 - Address of ECBLIST passed in R1
***********************************************************************
BPXTWAIT CSECT
BPXTWAIT AMODE ANY
BPXTWAIT RMODE ANY
         ENTRY BPXTWAIT
         EDCPRLG
         LR    R4,R1             get pointer to ecb vector
         WAIT  1,ECBLIST=(R4),LINKAGE=SYSTEM,EUT=SAVE
         EDCEPIL
*
```

# BPXTPOST—post an ECB

This function posts an ECB.

## C function

```
#pragma linkage(BPXTPOST,OS)
void BPXTPOST (long ascb,ECB *ecb);
```

## Assembler routine

```
***********************************************************************
*   CSECT-NAME:  BPXTPOST
*
*   DESCRIPTIVE-NAME:  Post an ECB
*
*   Input: R1 - parm list:
*                ASCB address
*                Address of ECB
*
***********************************************************************
BPXTPOST CSECT
BPXTPOST AMODE ANY
BPXTPOST RMODE ANY
         ENTRY BPXTPOST
         EDCPRLG USRDSAL=POSTLN
         USING POSTDYN,R13
         MVC   POSTL(POSTLN),POSTS copy POST parmlist to dynamic area
         L     R2,0(,R1)         get addr of ascb addr
         L     R2,0(,R2)         get ascb addr
         L     R4,4(,R1)         get addr of ECB to post
         POST  (R4),ASCB=(R2),ERRET=POSTERR,ECBKEY=0,LINKAGE=SYSTEM,   X
               MF=(E,POSTL)
         EDCEPIL
POSTS    POST  0,ASCB=0,ERRET=0,ECBKEY=YES,MF=L
POSTERR  BR    R14
POSTDYN  EDCDSAD
POSTL    POST  0,ASCB=0,ERRET=0,ECBKEY=YES,MF=L
POSTLN   EQU *-POSTL
         IHAPSA
*
```

# BPXTEPOC—convert time-of-day to epoch time

This function converts time-of-day to seconds-since-the-epoch.

## C function

```
#pragma linkage(BPXTEPOC,OS)
void BPXTEPOC(char *tod, long *epoch);
```

## Assembler routine

```
***********************************************************************
*   CSECT-NAME:  BPXTEPOC
*
*   DESCRIPTIVE-NAME:  Convert TOD to Epoch time
*
*   Input: R1 ->
*             address of TOD value to convert (double word)
*             address of output epoch time (one word)
***********************************************************************
BPXTEPOC CSECT
BPXTEPOC AMODE ANY
BPXTEPOC RMODE ANY
         EDCPRLG
         L     R2,0(R1)           get tod address
         LM    R14,R15,0(R2)      get tod
```

```
         LTR   R14,R14            check high word for 0
         BNZ   EPOCTOD            if input tod is 0
         STCK  0(R2)              get current tod
         LM    R14,R15,0(R2)      get tod
EPOCTOD  L     R2,4(R1)           get output area
         LM    R0,R1,EPOCJ70      get epoch tod
         SLR   R15,R1
         BC    11,*+6
         BCTR  R14,0
         SLR   R14,0
         D     R14,EPOCST         divide by seconds per tod unit
         SLR   R14,R14
         LA    R1,2
         DR    R14,R1
         ST    R15,0(R2)
         EDCEPIL
EPOCJ70  DS    0D
         DC    X'7D91048BCA000000'
EPOCST   DC    X'7A120000'
*
```

# Appendix F. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

## Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

## Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

## z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

`www.ibm.com/servers/eserver/zseries/zos/bkserv/`

© Copyright IBM Corp. 1996, 2006

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

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

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the products and/or the programs described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 1996, 2006

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

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

## Programming Interface Information

This publication documents intended Programming Interfaces that allow the customer to write programs that use z/OS UNIX System Services (z/OS UNIX).

## Trademarks

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

AIX
C/370
DFS
IBM
ibm.com
IBMLink
Language Environment
Library Reader
MVS
RACF
z/Architecture
z/OS
z/VM

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

## Acknowledgments

This interface is a modification of the architecture outlined by S.R. Kleiman in the paper, “Vnodes: An Architecture for Multiple File System Types in Sun UNIX”, which was published in the *Proceedings: Summer Usenix Technical Conference & Exhibition* (June 1986).

# Index

## Special characters



## Numerics



## A



## B



© Copyright IBM Corp. 1996, 2006

## C



## D



## E

## S



## T



## U



## V

# W



# Z

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

**z/OS**
**UNIX System Services**
**File System Interface Reference**

**Publication No. SA22-7808-07**

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

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|---|---|---|---|---|---|
| Overall satisfaction | ☐ | ☐ | ☐ | ☐ | ☐ |

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

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|---|---|---|---|---|---|
| Accurate | ☐ | ☐ | ☐ | ☐ | ☐ |
| Complete | ☐ | ☐ | ☐ | ☐ | ☐ |
| Easy to find | ☐ | ☐ | ☐ | ☐ | ☐ |
| Easy to understand | ☐ | ☐ | ☐ | ☐ | ☐ |
| Well organized | ☐ | ☐ | ☐ | ☐ | ☐ |
| Applicable to your tasks | ☐ | ☐ | ☐ | ☐ | ☐ |

**Please tell us how we can improve this book:**

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

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

| | |
|---|---|
| Name | Address |
| Company or Organization | |
| Phone No. | |

**Readers' Comments — We'd Like to Hear from You**
SA22-7808-07

Cut or Fold
Along Line

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

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

# BUSINESS REPLY MAIL

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

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY
12601-5400

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



Cut or Fold
Along Line

Program Number: 5694-A01, 5655-G52

Printed in USA

SA22-7808-07