**Portable Systems Group**

**NT OS/2 IRP Language Definition**

**Author:** *Gary D. Kimura*

*Revision 1.0x, December 15, 1989*

## 1. Introduction

The purpose of this chapter is to define the semantic contents of an I/O Request Packet (IRP). The information contained here is intended for use mainly by Device Driver and File System developers. The I/O system sends to the various Device Drivers[1] a stream of multiple IRPs that the drivers must interpret and respond to. Figure 1 shows the relationship between the device driver and the I/O system. Communication between the I/O system and the Device Driver is through IRPs. This chapter concentrates on the IRP language.

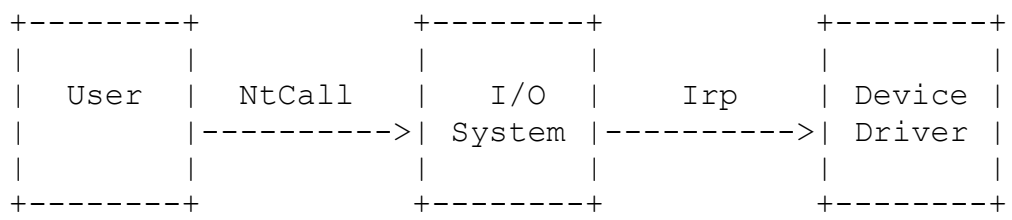



Figure 1
Logical control flow from user to Device Driver

Each IRP has a well defined format and semantic meaning, and the order in which they are sent must adhere to certain rules. The ordering of IRPs and responses form a context sensitive language.

Each IRP contains a common header section followed by one or more function specific records (also called IRP stack locations). From a Device Drivers viewpoint each IRP request is a single record describing one function to perform. That is, the drivers only interpret one function specific record. The additional stack locations are for use when a driver issues subsequent IRPs to a lower level driver and wishes to reuse the original IRP.

Each IRP function is identified by a major and minor function field in the IRP stack location record. The list of possible function combinations are listed below. Each line lists a major function code followed (in paranthesis) by a minor function code. Note that some major functions (e.g., CREATE) do not make use the minor function field.

CLOSE()
CONFIGURATION_CONTROL(...)
CREATE()
DEVICE_CONTROL(...)
DIRECTORY_CONTROL(NOTIFY_CHANGE_DIRECTORY)
DIRECTORY_CONTROL(QUERY_DIRECTORY)
FILE_SYSTEM_CONTROL(DISMOUNT_VOLUME)

---

[1]For clarity we will use the term Device Driver to refer to both Device Drivers and File systems.

FILE_SYSTEM_CONTROL(LOCK_VOLUME)
FILE_SYSTEM_CONTROL(MOUNT_VOLUME)
FILE_SYSTEM_CONTROL(QUERY_INFO_FILE_SYSTEM)
FILE_SYSTEM_CONTROL(SET_INFO_FILE_SYSTEM)
FILE_SYSTEM_CONTROL(UNLOCK_VOLUME)
FILE_SYSTEM_CONTROL(VERIFY_VOLUME)
INTERNAL_DEVICE_CONTROL(...)
LOCK_CONTROL(LOCK)
LOCK_CONTROL(UNLOCK_ALL)
LOCK_CONTROL(UNLOCK_SINGLE)
QUERY_ACL()
QUERY_EA()
QUERY_INFORMATION()
QUERY_VOLUME_INFORMATION()
READ()
READ_TERMINAL()
SET_ACL()
SET_EA()
SET_INFORMATION()
SET_NEW_SIZE()
SET_VOLUME_INFORMATION()
WRITE()

/* We need to define the minor function codes for the configuration, device, and internal device function codes. */

Each Device Driver will only receive a combination of the preceding function codes based on the drivers device type. This means that a file system device driver can expect to receive different functions than the keyboard device driver, or a disk driver. The possible device driver types are:

Disk Driver,
File System (including network redirector),
Keyboard Driver,
Mouse Driver,
Network Drivers,
Sound Driver,
Tape Driver,
Terminal Driver, and
Video Driver,

/* We will need to futher expand on the different network device drivers */

The remainder of this chapter describes the valid combination of IRP function codes that each different device driver can expect to receive. This is followed by a section listing every IRP function code along with a description of the function's parameters, semantics, and I/O completion status codes.

## 2. Valid IRP Function Combinations

The section contains an individual table for each device driver type that lists the set of valid IRP functions that can be sent to the driver and under what conditions the functions are sent.

### 2.1 Disk Driver IRPs

The set of possible IRPs that can be sent to a disk driver are:

| IRP Function | When sent |
|---|---|
| CLOSE | Anytime. |
| CREATE | Anytime. |
| DEVICE_CONTROL (...) | Anytime. |
| READ | Anytime. |
| WRITE | Anytime. |

### 2.2 File System IRPs

The set of possible IRPs that can be sent to a file system are:

| IRP Function | When sent |
|---|---|
| CLOSE | Only after a successful CREATE and then only on an opened file. This closes the file so no other operation can be performed on the file other than CREATE. |
| CREATE | Only after a successful MOUNT_VOLUME and then only on a mounted volume that is not locked. If successful the file is considered opened. |
| DIRECTORY_CONTROL (NOTIFY_CHANGE_DIRECTORY) | Only after a successful CREATE and then only on an opened directory file. |
| DIRECTORY_CONTROL (QUERY_DIRECTORY) | Only after a successful CREATE and then only on an opened directory file. |
| FILE_SYSTEM_CONTROL (DISMOUNT_VOLUME) | Only after a successful MOUNT_VOLUME and then only on a mounted volume. This |

| | |
|---|---|
| | dismounts the volume, so no other operation can be performed on the volume other than MOUNT_VOLUME. |
| FILE_SYSTEM_CONTROL (LOCK_VOLUME) | Only after a successful CREATE and then only on an opened file. This locks the volume containing the file such that no other creates using the same volume will succeed until the volume is unlocked. To be successful, the file used to lock the volume must also be the only opened file on the volume. |
| FILE_SYSTEM_CONTROL (MOUNT_VOLUME) | Anytime. If the operation is successful then a new device object for the volume is created and the volume is considered mounted and not locked. |
| FILE_SYSTEM_CONTROL (QUERY_INFO_FILE_SYSTEM) | Only after a successful CREATE and then only on an opened file. |
| FILE_SYSTEM_CONTROL (SET_INFO_FILE_SYSTEM) | Only after a successful CREATE and then only on an opened file. |
| FILE_SYSTEM_CONTROL (UNLOCK_VOLUME) | Only after a successful CREATE and then only on a opened file. The file system must handle the situation where the user is attempting to unlock a volume that is not locked. If successful this operation unlocks a previously locked volume so that other creates using the volume can now succeed. |
| FILE_SYSTEM_CONTROL (VERIFY_VOLUME) | Only after a successful MOUNT_VOLUME and then only on a mounted volume. |
| LOCK_CONTROL (LOCK) | Only after a successful CREATE and then only on an opened file. If successful this operation locks a range of bytes within a file. The locks remain in affect until they are explicitly unlocked or the file is closed. |
| LOCK_CONTROL (UNLOCK_ALL) | Only after a successful CREATE and then only on an opened file. The file system must handle the situation where an unlock is received even though there are no outstanding locks for that user. |

| | |
|---|---|
| LOCK_CONTROL (UNLOCK_SINGLE) | Only after a successful CREATE and then only on an opened file. The file system must handle the situation where an unlock is received even though there is not a corresponding lock. |
| QUERY_ACL | Only after a successful CREATE and then only on an opened file. |
| QUERY_EA | Only after a successful CREATE and then only on an opened file. |
| QUERY_INFORMATION | Only after a successful CREATE and then only on an opened file. |
| QUERY_VOLUME_INFORMATION | Only after a successful CREATE and then only on an opened file. |
| READ | Only after a successful CREATE and then only on an opened file. |
| SET_ACL | Only after a successful CREATE and then only on an opened file. |
| SET_EA | Only after a successful CREATE and then only on an opened file. |
| SET_INFORMATION | Only after a successful CREATE and then only on an opened file. |
| SET_NEW_SIZE | Only after a successful CREATE and then only on an opened file. |
| SET_VOLUME_INFORMATION | Only after a successful CREATE and then only on an opened file. |
| WRITE | Only after a successful CREATE and then only on an opened file. |

### 2.3 Keyboard Driver IRPs

The set of possible IRPs that can be sent to the Keyboard driver are:

| IRP Function | When sent |
|---|---|

| | |
|---|---|
| CLOSE | Anytime. |
| CREATE | Anytime. |
| DEVICE_CONTROL (...) | Anytime. |
| QUERY_INFORMATION | Anytime. |
| READ | Anytime. |
| SET_INFORMATION | Anytime. |
| WRITE | Anytime. |

### 2.4 Mouse Driver IRPs

The set of possible IRPs that can be sent to the Mouse driver are:

| IRP Function | When sent |
|---|---|
| CLOSE | Anytime. |
| CREATE | Anytime. |
| DEVICE_CONTROL (...) | Anytime. |
| QUERY_INFORMATION | Anytime. |
| READ | Anytime. |
| SET_INFORMATION | Anytime. |
| WRITE | Anytime. |

### 2.5 Network Drivers IRPs

The set of possible IRPs that can be sent to the Network drivers are:

| IRP Function | When sent |
|---|---|

/* This table needs to be filled in */

### 2.6 Sound Driver IRPs

The set of possible IRPs that can be sent to the Sound driver are:

| IRP Function | When sent |
|---|---|

/* This table needs to be filled in */

### 2.7 Tape Driver IRPs

The set of possible IRPs that can be sent to the Tape driver are:

| IRP Function | When sent |
|---|---|

/* This table needs to be filled in */

### 2.8 Terminal Driver IRPs

The set of possible IRPs that can be sent to the Terminal driver are:

| IRP Function | When sent |
|---|---|

/* This table needs to be filled in */

### 2.9 Video Driver IRPs

The set of possible IRPs that can be sent to the Video driver are:

| IRP Function | When sent |
|---|---|
| CLOSE | Anytime. |
| CREATE | Anytime. |
| DEVICE_CONTROL (...) | Anytime. |
| QUERY_INFORMATION | Anytime. |
| READ | Anytime. |
| SET_INFORMATION | Anytime. |
| WRITE | Anytime. |

## 3. IRP Function Descriptions

This section describes the input parameters and semantics for each IRP function code. It also discusses the interactions between the parameters and lists possible return status codes.

The parameter descriptions list all the fields that are used within the IRP by the operation being described. Each parameter is either Read (i.e., used as input to the operation), Set (i.e., used as output for the operation), or Ignored. To help distinguish the parameters we will also use the two terms IrpFlags and FunctionFlags to denote the flags field of the IRP header and the I/O stack location respectively.

In the description of the return status codes we do not include generic values such as STATUS_PENDING or STATUS_INVALID_PARAMETER which can be returned for any IRP. We also do not describe values that can be returned by a lower level device drivers such as STATUS_PARITY_ERROR.

### 3.1 Close

The close function is used to close a previously opened file or directory. Its two input parameters are a device object and an IRP. The device object parameter points to a volume previously mounted by the Device Driver and is where the file opened file exists. The IRP contains the close function parameters (and are listed below).

Besides closing the file, this function will optionally deletes the file based upon the disposition specified by the caller (See the SET_INFORMATION operation). If this is the last file object with the file opened and the disposition is *delete on close* then the file is removed from the on-disk structure.

**Close** (
**IN PDEVICE_OBJECT** *DeviceObject*,
**IN PIRP** *Irp*
);

Parameters within the IRP:

| Parameter type and name | Description |
|---|---|
| **PMDL** *MdlAddress* | Ignored. |
| **ULONG** *IrpFlags* | Ignored. |
| **STRING** *FileObject->FileName* | Ignored. |

| | |
|---|---|
| **ULONG**<br>*FileObject->RelatedFileObject* | Ignored. |
| **PVOID**<br>*FileObject->FsContext* | Read and Set. The driver uses this field to retrieve any private data (established by the CREATE function) that needs to be processed in order to close the file. It is set to NULL upon return from the close function. |
| **PVOID**<br>*FileObject->FsContext2* | Read and Set. The driver uses this field to retrieve any private data (established by the CREATE function) that needs to be processed in order to close the file. It is set to NULL upon return from the close function. |
| **PVOID**<br>*FileObject->SectionObjectPointer* | Set. The close function must set this field to NULL. |
| **IO_STATUS_BLOCK**<br>*IoStatus* | Set. This receives the final return status of the operation. The possible return status values are listed later. |
| **PEPROCESS**<br>*AlternateProcess* | Ignored. |
| **KPROCESSOR_MODE**<br>*RequestorMode* | Ignored. |
| **PVOID**<br>*SystemBuffer* | Ignored. |
| **PIO_STATUS_BLOCK**<br>*UserIosb* | Ignored. |
| **PKEVENT**<br>*UserEvent* | Ignored. |
| **LARGE_INTEGER**<br>*AllocationSize* | Ignored. |
| **PVOID**<br>*UserBuffer* | Ignored. |

Parameters within the IRP Stack:

| Parameter type and name | Description |
|---|---|
| **UCHAR** *MajorFunction* | Read. Must be equal to IRP_MJ_CLOSE. |
| **UCHAR** *MinorFunction* | Ignored. |
| **UCHAR** *FunctionFlags* | Ignored. |
| **UCHAR** *Control* | Ignored. |

Iosb Return Status and Information:

The following status codes are used to complete the CLOSE function.

| Return status followed by information field of IOSB | Description |
|---|---|
| STATUS_SUCCESS<br>Ignored | Indicates that the opened file has been closed. |

### 3.2 Create

The create function is used to create or open a file or a directory. Its two input parameters are a device object and an IRP. The device object parameter points to a volume previously mounted by the Device Driver and is where the file will exist. The IRP contains the create function parameters (and are listed below).

**Create** (
**IN PDEVICE_OBJECT** *DeviceObject*,
**IN PIRP** *Irp*
);

Parameters within the IRP:

| Parameter type and name | Description |
|---|---|
| **PMDL** *MdlAddress* | Ignored. |
| **ULONG** *IrpFlags* | Ignored. |
| **STRING** *FileObject->FileName* | Read. This is the name of the file being opened. |
| **ULONG** *FileObject->RelatedFileObject* | Read. This field is used for path relative file names.<br><br>If it is null then the file name is relative to the root of the volume (e.g., "\CONFIG.SYS" is the name of the configuration file located in root directory).<br><br>If is it not null then it points to a previously opened file object representing a directory on the volume, and the file name is relative to the specified directory (e.g., if the related file object is "\NT\SDK" the file name can be "INC\NTIOAPI.H"). Note that path relative file names do not begin with a backslash. |
| **PVOID** *FileObject->FsContext* | Set. This is used by the Device Driver to store file object specific information that can be retrieved later when the driver is called to perform subsequent operations on the file. |

| | |
|---|---|
| | The FAT file system stores in this field a pointer to an internal File Control Block (FCB) structure. |
| **PVOID**<br>*FileObject->FsContext2* | Set. This is used by the Device Driver to store file object specific information that can be retrieved later when the driver is called to perform subsequent operations on the file.<br><br>The FAT file system only uses this field for directories. It is a pointer to an internal Context Control Block (CCB) structure. |
| **PVOID**<br>*FileObject->SectionObjectPointer* | Set. It is set to the longword context for the file. It is not used for directories. For every opened file the driver allocates a single longword of context for exclusive use by the memory management system. All file objects that denote the same file point to the same longword context.<br><br>In FAT this is done by reserving a longword field in the FCB and having each section object pointer point to this field. |
| **IO_STATUS_BLOCK**<br>*IoStatus* | Set. This receives the final return status of the operation. The possible return status values are listed later. |
| **PEPROCESS**<br>*AlternateProcess* | Ignored. |
| **KPROCESSOR_MODE**<br>*RequestorMode* | Read. This is the mode of the requestor. It is used for to help decide if the requestor has the proper access rights to the file.<br><br>/**** We also need to pass in the token of the requestor ****/ |
| **PVOID**<br>*SystemBuffer* | Read. This field is only used if the file is being created and then it only specifies the optional extended attributes for the file. If the field is null the file will not be created with extended attributes. The create operation must complete with an error if there are any |

| | |
|---|---|
| | problems with the extended attributes.<br><br>For FAT there is a 64K limit to the size of the extended attributes (as packed on the disk). The create operation will complete with an error if this limit is exceeded. |
| **PIO_STATUS_BLOCK**<br>*UserIosb* | Ignored. |
| **PKEVENT**<br>*UserEvent* | Ignored. |
| **LARGE_INTEGER**<br>*AllocationSize* | Read. This field is only used if the file is being created and is ignored for directories and for open operations. It specifies the initial file allocation in bytes to allocate to the file. This is not the same as the end-of-file location. |
| **PVOID**<br>*UserBuffer* | Ignored. |

Parameters within the IRP Stack:

| Parameter type and name | Description |
|---|---|
| **UCHAR**<br>*MajorFunction* | Read. Must be equal to IRP_MJ_CREATE. |
| **UCHAR**<br>*MinorFunction* | Ignored. |
| **UCHAR**<br>*FunctionFlags* | Ignored. |
| **UCHAR**<br>*Control* | Ignored. |
| **ULONG**<br>*DesiredAccess* | Read. This is the access mask that the user is trying to acquire to the file. If the user is trying to open a file the mask will be a combination of the following values: |

DELETE,
READ_CONTROL,
WRITE_DAC,
WRITE_OWNER,
SYNCHRONIZE,
FILE_READ_DATA,
FILE_WRITE_DATA,
FILE_APPEND_DATA,
FILE_READ_EA,
FILE_WRITE_EA,
FILE_EXECUTE,
FILE_READ_ATTRIBUTES, and
FILE_WRITE_ATTRIBUTES.

If the user is trying to open a directory the mask will be a combination of the following values:

DELETE,
READ_CONTROL,
WRITE_DAC,
WRITE_OWNER,
SYNCHRONIZE,
FILE_LIST_DIRECTORY,
FILE_ADD_FILE,
FILE_ADD_SUBDIRECTORY,
FILE_READ_EA,
FILE_WRITE_EA,
FILE_TRAVERSE,
FILE_DELETE_CHILD,
FILE_READ_ATTRIBUTES, and
FILE_WRITE_ATTRIBUTES.

The driver must ensure that the combination of the caller's privileges and requestor's mode grants all of the desired accesses that the user is trying to acquire.

**ULONG**
*Options*

Read. This field contains all of the different create options and create disposition flags that the user can specify in an NT call. The valid flags and their meanings are listed below:

FILE_CREATE_DIRECTORY

Read. Indicates that the user is creating a

| | |
|---|---|
| | new directory. |
| FILE_OPEN_DIRECTORY | Read. Indicates that the user is opening an existing directory. |
| FILE_WRITE_THROUGH | Ignored, but saved away for use by subsequent read and write operations to the file object. |
| FILE_SEQUENTIAL_ONLY | Ignored, but saved away for use by subsequent read and write operations to the file object. |
| FILE_MAPPED_IO | Ignored, but saved away for use by subsequent read and write operations to the file object. |
| FILE_DISABLE_CACHING | Ignored, but saved away for use by subsequent read and write operations to the file object. |
| FILE_SYNCHRONOUS_IO_ALERT | Ignored. |
| FILE_SYNCHRONOUS_IO_NONALERT | Ignored. |
| FILE_CREATE_TREE_CONNECTION | Read. Only used by the network. |
| | /**** need a complete description of this parameter ****/ |
| FILE_SUPERSEDE << 24[2] | Read. Indicates that if the file already exists it should be superseded, and if the file does not exist it should be created. |
| FILE_CREATE << 24 | Read. Indicates that if the file already exists it is an error, and if the file does not exist it should be created. |
| FILE_OPEN << 24 | Read. Indicates that if the file already exists it is to be opened, and if the file |

[2]To test if the flags FILE_SUPERSEDE, FILE_OPEN, FILE_CREATE, and FILE_OPEN_IF are in the options parameter the driver must first shift the flag 24 bits to the left and then do the test (e.g., Option & (FILE_SUPERSEDE << 24)).

does not exist it is an error.

FILE_OPEN_IF << 24

Read. Indicates that if the file already exists it is to be opened, and if the file does not exist it should be created.

/**** We need a list of the illegal flag combinations, and state that they will never be seen in an IRP ****/

**USHORT**
*FileAttributes*

Read. This field specifies the DOS file attributes to use when creating or superseding a file, and is ignored when opening an existing file. It is a combination of any of the following flags:

FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_CONTROL, and
FILE_ATTRIBUTE_NORMAL

The flag FILE_ATTRIBUTE_NORMAL overrides all other file attribute flags. (i.e., if the user specifies normal and readonly then the file is created as a normal file and not readonly).

**USHORT**
*ShareAccess*

Read. This field specifies the share mode access between processes trying to open the same file. All users that open a file for shared access must specify the exact same share flags. This is separate from their desired access. For example a file opened shared read, write, and delete, must be opened by all users as shared read, write, and delete even though the desired access might only specify read access.

The valid flags and their meanings are listed below:

FILE_SHARE_READ

Read. Indicates that the file can be opened by others for read access. If the

file is already opened for shared read access then other users can open it for read access.

FILE_SHARE_WRITE — Read. Indicates that the file can be opened by others for write access. If the file is already opened for shared write access then other users can open it for write access.

FILE_SHARE_DELETE — Read. Indicates that the file can be opened by others for delete access. If the file is already opened for shared delete access then other users can open it for delete access.

FILE_SHARE_RENAME — Read. Indicates that the file can be renamed by others. If the file is already opened for shared renamed access then other users can rename the file.

The test that a user requesting shared read, write, or delete can be done by the Device Driver during the create operation (i.e., a user is allowed read access to a shared file if the shared access flags match, shared read is specified, and the file's security protection allows for read access). The test for rename access must be deferred until the a rename IRP is processed (see the Set Information IRP description).

**ULONG** *EaLength* — Read. This parameter is specified only if the user is creating or superseding a file and has specified an EA for the file. This parameter is then the size, in bytes, of the EA set specified by the user. (i.e., it is the size of the system buffer parameter).

Iosb Return Status and Information:

The following status codes are used to complete the CREATE function.

Return status followed by

| information field of IOSB | Description |
|---|---|
| STATUS_SUCCESS<br>FILE_OPENED | Indicates that an existing file has been successfully located and opened. |
| STATUS_SUCCESS<br>FILE_SUPERSEDED | Indicates that an existing file has been successfully located and superseded. |
| STATUS_SUCCESS<br>FILE_CREATED | Indicates that an existing file (of the same name) does not exist and that a new file has been successfully created. |
| STATUS_ACCESS_DENIED<br>Ignored | Indicates that because of protection on the file, parent directory, or volume access has been denied to the file. This can also occur if the caller specified options or share access flags are not compatible with either the file or the previous share access that it was opened with. |
| STATUS_OBJECT_NAME_INVALID<br>Ignored | Indicates that the last name in the object's file name field does not contain a syntactically valid name (e.g., it's too long or contains invalid characters). |
| STATUS_OBJECT_NAME_NOT_FOUND<br>Ignored | Indicates that the last name in the object's file name field is not the name of an existing file. |
| STATUS_OBJECT_PATH_INVALID<br>Ignored | Indicates that a name within the path part of the object's file name field does not contain a syntactically valid name. |
| STATUS_OBJECT_PATH_NOT_FOUND<br>Ignored | Indicates that a name within the path part of the object's file name field does not contain the name of an existing directory. |
| STATUS_DISK_FULL_ERROR<br>Ignored | Indicates that because the disk is full the file cannot be created. This can occur when disk space cannot be allocated for the directory entry, file node, or the extended attributes. |
| STATUS_DISK_FULL_WARNING<br>FILE_SUPERSEDED | Indicates that the file has been superseded but because the disk is full the file cannot be given the user specified file allocation size. |
| STATUS_DISK_FULL_WARNING | Indictes that the file has been created but because |

| | |
|---|---|
| FILE_CREATED | the disk is full the file cannot be given ths user specified file allocation size. |
| STATUS_EA_INVALID<br>Ignored | Indicates that the EA structure passed into this function is syntactically invalid. |

### 3.3 Device Control

### 3.4 Directory Control(Notify Change Directory)

### 3.5 Directory Control(Query Directory)

### 3.6 File System Control(Dismount Volume)

### 3.7 File System Control(Lock Volume)

### 3.8 File System Control(Mount Volume)

The mount function is used mount a new disk volume. Its two input parameters are a device object and an IRP. The device object parameter points to the Device Drivers original device object that is created when the driver is initialized.

The mount operation can handle mounting new volume, and remounting a previously mounted volume. The parameter description that follows assumes that it is processing a new volume. At the end of the description we cover the updating required for the remount case.

**Mount** (
**IN PDEVICE_OBJECT** *DeviceObject*,
**IN PIRP** *Irp*
);

Parameters within the IRP:

| Parameter type and name | Description |
|---|---|
| **PMDL** *MdlAddress* | Ignored. |
| **ULONG** *IrpFlags* | Ignored. |
| **PFILE_OBJECT** *FileObject* | Ignored. |
| **IO_STATUS_BLOCK** *IoStatus* | Set. This receives the final return status of the operation. The possible return status values are listed later. |
| **PEPROCESS** *AlternateProcess* | Ignored. |
| **KPROCESSOR_MODE** *RequestorMode* | Ignored. |
| **PVOID** *SystemBuffer* | Ignored. |
| **PIO_STATUS_BLOCK** *UserIosb* | Ignored. |
| **PKEVENT** | Ignored. |

| | |
|---|---|
| *UserEvent* | |
| **LARGE_INTEGER**<br>*AllocationSize* | Ignored. |
| **PVOID**<br>*UserBuffer* | Ignored. |

Parameters within the IRP Stack:

| Parameter type and name | Description |
|---|---|
| **UCHAR**<br>*MajorFunction* | Read. Must be equal to IRP_MJ_FILE_SYSTEM_CONTROL. |
| **UCHAR**<br>*MinorFunction* | Read. Must be equal to IRP_MN_MOUNT_VOLUME. |
| **UCHAR**<br>*FunctionFlags* | Ignored. |
| **UCHAR**<br>*Control* | Ignored. |
| **PDEVICE_OBJECT**<br>*Vpb->DeviceObject* | Set. If the mount is successful this field is set the point to the newly allocated device object for the volume. If the mount is unsuccessful or this is a remount then this field is not updated. |
| **ULONG**<br>*Vpb->DeviceObject->Flags* | Set. If the mount is successful then the flag DO_DIRECT_IO is set in the newly created device objects flags field. Setting this flag allows the Device Driver to receive unbuffered I/O requests for this volume. |
| **ULONG**<br>*Vpb->SerialNumber* | Set. If the mount is successful this field is set to the serial number found on the volume. It is ignored if the mount is unsuccessful or in the case of a remount. |
| **CHAR**<br>*Vpb->VolumeName[20]* | Set. If the mount is successful this field is set to the label found on the volume. If the volume does not have a label then this field |

| | |
|---|---|
| | should be set to all spaces.<br><br>For FAT the volume label, if present, is found in the root directory as a special dirent. |
| **PDEVICE_OBJECT**<br>*DeviceObject* | Read. This is the device object that the Device Driver is to use when formulating IRPs to read or write to the volume. It is also called the target device object. If the volume is mounted successful this value must be remembered so the driver can handle subsequent requests to the volume. |

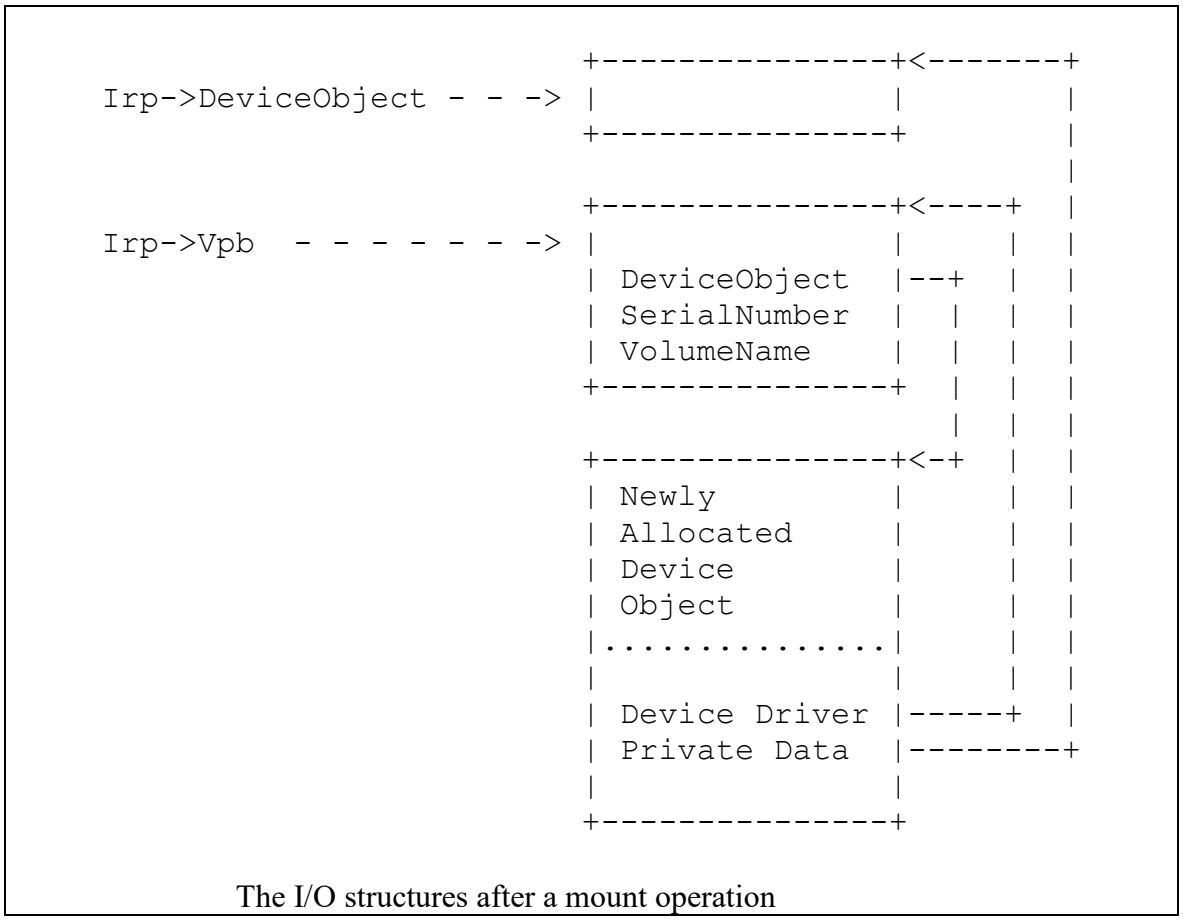
Iosb Return Status and Information:

The following status codes are used to complete the MOUNT function.

| Return status followed by information field of IOSB | Description |
|---|---|
| STATUS_SUCCESS<br>Ignored | Indicates that the volume has been successful mounted. |
| STATUS_WRONG_VOLUME<br>Ignored | Indicates that the volume cannot be mounted either because it does not recognize the on-disk structure or the on-disk structure has been currupted. |

Mounting a new volume:

The following figure shows the major I/O structures after processing a successful mount request.

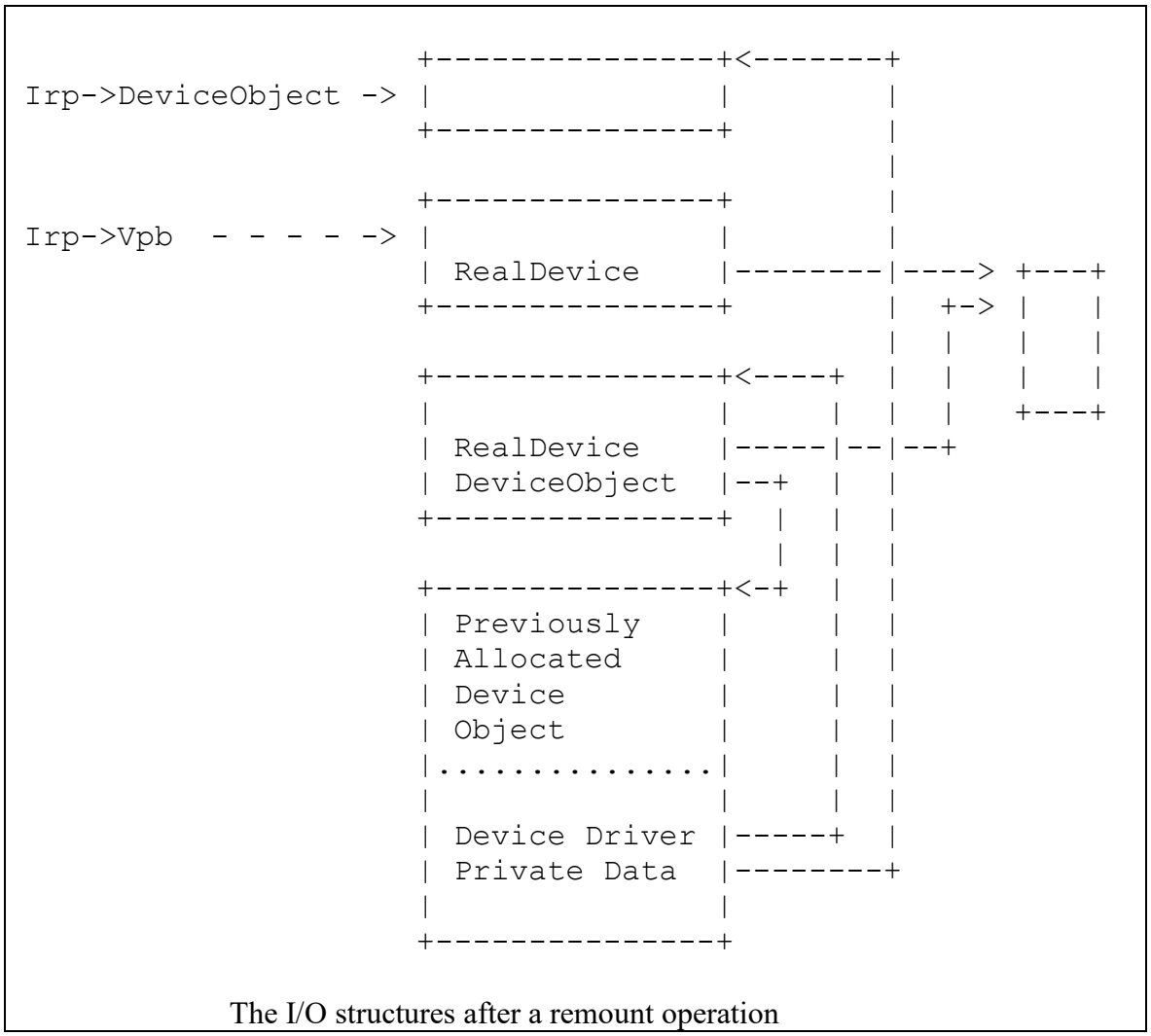



The I/O structures after a mount operation

In the preceding figure the newly allocated device object has immediately following it a Device Driver private data record that is for used only by the driver. This technique should be used in the driver to keep track of the VPB and the device object where it is to send its read and write requests. It should also be used to link together all of the mounted volumes serviced by the driver.

Remounting a volume:

By using the device driver private data record to maintain a link of all mounted volumes a Device Driver can determine if a mount request for a volume matches a previously mounted volume (They match if the both volume have the same serial number and volume label). The following figure shows the major I/O structure after processing a remount.




The I/O structures after a remount operation

The remount operation does not allocate any new structures, instead it it performs the following operations:

- The Device Drivers Private Data pointer to the target device object is changed to point to the new target device object.
- The RealDevice field of the Vpb that we previously mounted is set to the RealDevice field of the new Vpb that was passed in as a parameter in the IRP.
- The Irp->Vpb is deallocated from pool by the device driver, and complete the mount request with STATUS_SUCCESS.

**3.9 File System Control(Query Information File System)**

**3.10 File System Control(Set Information File System)**

**3.11 File System Control(Unlock Volume)**

**3.12 File System Control(Verify Volume)**

**3.13 Internal Device Control**

**3.14 Lock Control(Lock)**

**3.15 Lock Control(Unlock All)**

**3.16 Lock Control(Unlock Single)**

**3.17 Query Acl**

**3.18 Query Ea**

**3.19 Query Information**

**3.20 Query Volume Information**

### 3.21 Read

### 3.22 Read Terminal

### 3.23 Set Acl

### 3.24 Set Ea

### 3.25 Set Information

### 3.26 Set New Size

### 3.27 Set Volume Information

### 3.28 Write

**Revision History**

Original Draft 1.0, December 15, 1989