Inter-Office Memorandum

To	IFS Project	Date	December 9, 1981
From	Ed Taft	Location	PARC/CSL
Subject	IFS Directory Operations (version 1.35)	File	[Maxc1]<IFS>IFSDirOps.bravo

XEROX

This document describes the BCPL interface to the IFS directory modules. Familiarity with the IFS file structure is assumed; see the memo IFS File Structure''. The low-level file operations have changed substantially since the previous version of IFS, particularly with regard to directory locking.

Organization

The directory package consists of the following modules:

IFSDirs.decl	Parameter and structure declarations used within the directory package and needed when calling many of its procedures.
IFSDirOpen.bcpl	Procedures to open and close IFS files and create streams to which normal Alto disk stream operations may be applied.
IFSDirDelRen.bcpl	Procedures to delete and rename IFS files.
IFSDirParse.bcpl	Procedures to parse IFS filenames and build File Descriptors (FDs).
IFSDirLookup.bcpl	Procedures to look up files in the IFS directory.
IFSDirUtil.bcpl	Miscellaneous utility procedures needed by the other modules.
IFSDirKey.asm	The compare key routine' and length routine' passed to the B-Tree package. (These routines depend on some special IFS microcode; the equivalent BCPL code is included as comments in the source file.)
IFSDirAdmin.bcpl	Procedures to create and destroy user Directory Information Files (DIFs).
IFSDirCheck.bcpl	A procedure to check the consistency of the directory B-Tree.

These modules call the B-Tree package and make use of other facilities provided in the IFS environment. There is a total of about 9000 words of code (including the B-Tree package), divided into 12 overlays of less than 1024 words each.

The directory package provides facilities at several levels. At the highest level are procedures implementing functions analogous to those in the standard Alto directory package. For example, the IFSOpenFile procedure translates directly from a file name to a stream in a manner similar to the Alto OpenFile.

At lower levels, specific file operations are passed a handle called a File Descriptor (FD). Procedures exist to translate a file name into an FD, to look up the FD in the directory, to open or close a file given its FD, and so on. An FD may designate multiple files (due to wildcard *' characters appearing in the name), and a procedure exists to step the FD from one such file to the next.

Directory and File Locks

Since the IFS is providing a multiple-access service, mutual exclusion mechanisms are required to maintain consistency of shared data structures. These mechanisms are relatively automatic when the directory package is accessed at its highest level. At the lower levels, however, callers must be aware of the resources that are locked at any given time.

Two kinds of locks are implemented, file locks and directory locks. An open file is locked in one of the following modes (defined in IFSDirs.decl):

modeRead	read-only, nonexclusive
modeWrite	write-only, exclusive
modeReadWrite	read-write, exclusive
modeAppend	append-only, exclusive
modeReadWriteShared	read-write, non-exclusive

In the exclusive modes, only one client at a time may access the file; in the non-exclusive modes, multiple clients may access the file simultaneously (but all clients must lock the file in the same mode). A file lock is set at the time the file is opened, deleted, or renamed, and the operation will fail if the lock cannot be set.

In addition to the locks on individual files, there is a lock controlling access to the directory itself (recall that the entire directory is a single B-Tree). Directory access conforms to a simple readers and writers' discipline whose lock modes are analogous to modeRead and modeReadWrite; but inability to set the directory lock immediately causes the process to wait rather than resulting in failure of the file operation. Note that the directory lock controls access only to the directory itself; operations on files that are individually locked may proceed without regard to the directory lock.

Since the directory is shared among all users, it is essential that a process lock it for as little time as possible. In particular, operations that can take arbitrarily long (such as deleting a file) should not be performed while keeping the directory locked. Most directory operations (lookup and update) are completed quickly. A process performing a potentially lengthy directory operation, such as enumerating it, is expected to check periodically for occurrences of lock conflicts (other processes waiting to use the directory) and to release and reacquire the lock when conflicts occur.

A Lock is a two-word structure defined in IFS.decl. Lock.count contains a positive read lock count or 1 to denote a write lock, and Lock.ctx contains a pointer to the context that last set the lock. All locks are manipulated by means of the Lock and Unlock procedures in IFSResUtilb.bcpl. They are called as follows:

Lock(lock, write [false], returnOnFail [false]) = true or false

Attempts to set the specified lock (a write lock if *write* is true or a read lock if false or omitted), and returns true if successful. If *returnOnFail* is false or omitted, blocks until the lock can be set; if true, returns false if the lock cannot be set immediately.

A process should not attempt to set the same lock multiple times without an intervening Unlock. An attempt is made to detect occurrences of this error (resulting in a call to IFSError), but the check is not foolproof.

Unlock(lock)

Unlocks the specified lock, which must have been either read- or write-locked by the same process.

File and directory locks are ordinarily manipulated by higher-level procedures such as LockFile and LockDirFD, described later. However, the Lock and Unlock primitives are directly useful for controlling access to other shared objects.

Data Structures

Most IFS data structures are not operated upon directly by programs calling the directory package, but rather are simply passed as arguments. However, an understanding of the contents and function of the major data structures is helpful.

IFS data structures are divided into two classes, *file system* and *runtime*. The data structures actually stored on the disk are defined in IFSFiles.decl and are documented in IFS File Structure''. These will not be further discussed here.

The runtime structure IFS (defined in IFS.decl) designates an active file system. The software is capable of dealing with multiple, independent file systems simultaneously. All file operations are performed relative to a particular file system denoted by an argument *fs*. The default is the *primary* file system *primaryIFS*, which must be on-line when IFS is started and is the one used for swapping. Other file systems may be mounted and dismounted while IFS is running.

The IFS structure contains configuration information (in particular, a table mapping logical unit numbers to physical disk drives) and directory information, including pointers to the B-Tree structure and Open File Table (OFT), the directory lock, and two other interesting items relating to directory access.

The IFS.dirVersion word is a count of modifications to the directory: it is incremented every time the directory is modified in any way. This is useful to programs that wish to re-validate the results of an earlier lookup when the directory has been unlocked since that lookup. If, after locking the directory, the program finds that dirVersion has not changed since the last lookup, then the lookup information is still valid; otherwise, the lookup must be repeated (a relatively expensive operation). This feature is used by the LookupFD procedure, described later.

The IFS.dirLockConflict word is set to true whenever a directory lock conflict occurs, i.e., when a process attempts to set the lock and cannot because it is already locked in a conflicting way. A program intending to keep the directory locked for a long time should, after locking it, set dirLockConflict to false, then periodically poll it and, when a conflict occurs, briefly relinquish the lock so as to give the conflicting process a chance to proceed.

The Open File Table (OFT) contains the locks for all open files. It is a hash table, keyed on the virtual disk addresses of the open files.

The UserInfo block (defined in IFSFiles.decl) contains the identity of and information about the user for whom file operations are being performed. The directory package uses this information in order to check access to files and to record the creator of new files. It is the responsibility of other parts of IFS to create the UserInfo block, check passwords, and so on.

Most procedures in the directory package assume they are running within the confines of a Rendezvous Socket Context (RSCtx, defined in IFSRS.decl), which contains a pointer to the appropriate UserInfo block. A special UserInfo block, pointed to by the static *system*, exists to permit privileged, internal file operations that bypass access checking.

All lower-level directory operations are passed a structure called a File Descriptor (FD), defined in IFSDirs.decl. An FD is originally created by CreateFD, which parses a file name and sets up some auxiliary lookup information, such as the actual version number as an integer (if one was specified)

and the indices of the end of the directory name and the end of the name body. The FD carries with it information that remains fixed for the life of the FD, such as the file system and the lookup control parameter. As operations are performed on the FD, various parts of it are updated.

High-Level Operations

The following operations are similar to ones available in the Alto Operating System.

IFSOpenFile(name, lvErrorCode [], mode [modeRead], itemSize [charItem], lc [see below], fs [primaryIFS], dirName [connected]) = stream or 0

Opens an IFS file, translating directly from a name to a stream. If successful, returns an open stream upon which the standard Alto disk stream operations may be performed. If unsuccessful, stores an error code in *@lvErrorCode* and returns zero.

The *name* must be a BCPL string whose complete form is *<dir>name!ver*', but in which the directory and version may be omitted (in which case they will be defaulted). The default directory is the BCPL string *dirName*, if supplied, or the connected directory obtained from the running context's UserInfo block otherwise. The default version is given as part of *lc* (see below). The version may also be one of !H', !L', or !N' designating highest existing version, lowest existing version, or next higher version. If the lookup control permits it, wildcard *'s may appear anywhere in the name to designate multiple files (discussed in more detail later).

The *mode* should be one of modeRead, modeWrite, modeReadWrite, or modeAppend (modeReadWriteShared is illegal). The first three modes are equivalent to the corresponding ksTypes in the Alto operating system, while modeAppend is equivalent to modeWrite except that the stream is initially positioned to end of file. (IFSOpenFile correctly checks the write or append protection of a file when it is opened; however, it is the caller's responsibility to prohibit overwriting existing parts of a file opened in modeAppend.)

The *itemSize* is one of charItem or wordItem, as in the Alto operating system.

The lookup control (*lc*) parameter contains several bits and fields controlling certain aspects of the directory lookup process. If the lcCreate bit is set, then IFSOpenFile may create a new file (protections permitting); otherwise, if the designated file does not exist an error will result. If the lcMultiple bit is set, the name is permitted to designate multiple files by means of *'; otherwise, occurrence of *' in the name will cause an error. The remainder of the lookup control word is a default version control specification, to be used in the absence of an explicit version number in the name. This may be one of:

lcVHighest	highest existing version
lcVNext	next higher version (highest+1)
lcVLowest	lowest existing version
lcVAll	all versions (same as !*')

The default lookup control specification depends on the mode in which the file is being opened, as follows:

modeRead	lcVHighest
modeWrite	lcCreate+lcVNext
modeReadWrite	lcCreate+lcVHighest
modeAppend	lcCreate+lcVHighest

If the name contains *'s (which are accepted only if the lookup control includes lcMultiple), then the first file whose name matches the pattern is opened. It is

expected that the caller will retain the FD and step it through all the other files matching the pattern, using the NextFD and OpenIFSStream primitives described later.

Closes(stream)

Performs the normal actions of cleaning up and destroying the stream, and also closes (i.e., unlocks) the file and destroys the FD.

IFSDeleteFile(name, lvErrorCode [], lc [lcVLowest], fs [primaryIFS], dirName [connected], deleteUndeletable [false]) = true or false

Deletes the specified file, returning true if successful and false if unsuccessful. The parameters are interpreted as for IFSOpenFile. The name may not designate multiple files. The user must have write access to the file.

A file may have an *undeletable* attribute that prevents it from being deleted even if all other conditions are satisfied; if *deleteUndeletable* is true, the *undeletable* attribute is ignored.

IFSDeleteOldVersions(name, lvErrorCode [], fs [primaryIFS], dirName [connected], deleteUndeletable [false]) = true or false

Deletes all but the highest-numbered version of all files designated by *name*, which may include *'s but must not have an explicit version number. Returns true normally and false if no file by that name exists or any of the delete operations fails; an error code for the last such failure is stored in @lvErrorCode. *deleteUndeletable* is interpreted as for IFSDeleteFile.

IFSRenameFile(oldName, newName, lvErrorCode [], lc [lcVHighest], fs [primaryIFS], oldDirName [ctxRunning.connName], newDirName [ctxRunning.connName]) = true or false

Renames the file *oldName* to be *newName*, returning true if successful and false if unsuccessful. The old file must exist and the new file must not exist. The *lc* parameter applies to *oldName*; the lookup control used for *newName* is lcCreate+lcVNext. The user must have write access to the old file and create access to the user directory in which the renamed file will reside.

Lower-Level Directory Operations

CreateFD(name, lc, lvErrorCode [], fs [primaryIFS], dirName [connected]) = fd or 0

Parses *name* and constructs an FD, returning the FD if successful and zero if unsuccessful. The only possible errors are syntax errors in the name; this procedure makes no references to the directory.

CreateFD sets the *fs*, *lc*, *lenDirString*, *lenSubDirString*, *lenBodyString*, and *version* fields in the FD structure. A skeleton Directory Record (DR, defined in IFSFiles.decl) is constructed and saved in the *dr* pointer; this record is of drTypeNormal and contains a copy of the name string, with an appropriate version number appended if appropriate (0 for lcVLowest and 65535 for lcVHighest or lcVNext).

If the lookup control includes lcMultiple and the name contains *'s (or the version control is lcVAll and no version is specified in the name), a template is constructed and stored in the *template* field in the FD for later pattern matches, the index of the first * is stored in the *iFirstStar* field, and the name in the DR is truncated at that point for use as a starting key by NextFD.

All remaining fields in the FD are zeroed. In particular, the *lookupStatus* field is set to lsNoLookup, indicating that this FD has not yet been looked up in the directory.

DestroyFD(fd) = 0

Destroys the FD (and the DR and template pointed to by it, if any). Zero is returned so as to permit use in contexts such as:

fd = DestroyFD(fd)

LookupFD(fd, dirLockMode [lockNone]) = 0 or error code

Looks up the file described by *fd*, applying the lookup control parameters (already stored in the FD) as appropriate. If the file already exists, replaces the DR pointed to by the FD with a copy of the actual entry that was found (including its type, length, and FP). If the file does not exist but lcCreate is set in the lookup control, generates a complete DR (except for the FP) which may be used when creating the file.

Returns zero if successful and an error code if unsuccessful. Failure to find the file in the directory is considered an error only if lcCreate is not set or directory *<dir>*' does not exist.

In the successful case, the *lookupStatus* field in the FD is set to one of the following:

lsNonexistent	The file does not exist, and no other version of that file exists either. In order to create the file, one must use directory-default file properties.
lsOtherVersion	The file doesn't exist, but another version of the file does exist. The FP of that file is stored in the DR to permit one to read its leader page and obtain is properties. (This facilitates the inheriting of properties from one version to the next.)
lsExists	The file exists, and a copy of its actual directory entry record is stored in the DR.

If the FD designates multiple files and this is the first time LookupFD has been called, LookupFD calls NextFD to find the first directory entry matching the pattern in the FD's template. If no such file is found, an error is returned.

LookupFD normally assumes that the directory is *not* locked at the time of the call, and leaves it unlocked upon return. This behavior is controlled by the value of *dirLockMode*:

lockNone	unlocked at call, unlocked upon return
lockRead	unlocked at call, read-locked upon return
lockWrite	unlocked at call, write-locked upon return
lockAlready	locked at call, locked the same way upon return

The directory locking actions are the same whether LookupFD succeeds or fails.

If the directory is locked upon return from LookupFD, then for as long as the directory remains locked the FD (in particular, the *lookupStatus*) is *valid*, i.e., it accurately reflects the state of the directory. Hence one may immediately perform operations that depend on the validity of the FD, such as modifying the directory entry.

However, once the directory has become unlocked, the FD is no longer valid, since some other process could change the directory in the meantime. In this case, before making use of the information in the FD one must *revalidate* it by calling LookupFD again, and one must be prepared for the possibility that its *lookupStatus* may change or even that the new call will fail despite the previous call having succeeded. (The revalidation operation is very cheap if the directory has not actually changed since the last LookupFD on the same FD.)

Most operations that take an FD as an argument expect the directory to be unlocked at the time of the call and perform the revalidation operation internally; the only important exception is TransferLeaderPage.

LookupIFSFile(name, lc, lvErrorCode [], fs [primaryIFS], dirName [connected]) = fd or 0

Combines the actions of CreateFD and LookupFD, returning an FD if successful and zero if unsuccessful. The directory should be unlocked at the time of the call and is unlocked upon return.

NextFD(fd, dirLockMode [lockNone]) = true or false

If *fd* designates multiple files, finds the next file matching the pattern and replaces *fd*'s DR with its directory record. If such a file is found, sets the *lookupStatus* to lsExists and returns true. If no such file is found, or *fd* does not designate multiple files, returns false. The *dirLockMode* argument is treated as in LookupFD.

LockDirFD(fd, write [false])

Locks the directory referenced by *fd*. Sets a write lock if *write* is true and a read lock otherwise. If a lock conflict occurs, sets the IFS.dirLockConflict flag and waits until the directory becomes unlocked.

UnlockDirFD(fd)

Unlocks the directory referenced by *fd*.

ModifyDirFD(fd)

Declares the directory referenced by *fd* to have been modified, by incrementing its version number. This should be done whenever the directory is modified (i.e., an entry is created, deleted, or updated). The directory must be write-locked at the time of the call and remains write-locked upon return.

Lower-Level File Operations

OpenIFSFile(fd, mode) = 0 or error code

Opens the file designated by *fd*, returning zero if successful and an error code if unsuccessful. The directory should be unlocked at the time of the call and is unlocked upon return.

OpenIFSFile checks protections and allocations as appropriate, creates the file if necessary, and attempts to lock the file in the specified *mode*. If any of these operations fails, an appropriate error code is returned and the file is not locked.

Note: if *mode*=modeReadWriteShared, it is the caller's responsibility to maintain consistency among multiple clients accessing the file.

CreateIFSStream(fd, itemSize) = stream or 0

Creates and returns a stream for an open file designated by *fd*. Returns zero if unsuccessful (the only likely cause of failure is a disk error in the leader page). *fd* is saved in ST.par1, which must not be clobbered by anyone (par2 and par3 are ok to use, however). Positions the stream to end of file if the file was opened with modeAppend.

OpenIFSStream(fd, lvErrorCode [], mode [modeRead], itemSize [charItem]) = stream or 0

Combines the actions of OpenIFSFile and CreateIFSStream, returning the stream if successful and zero if unsuccessful. If any operation fails, an error code is stored in *@lvErrorCode* and the file is not locked. A *mode* of modeReadWriteShared is illegal.

StreamsFD(stream) = fd

Returns the FD designating the file associated with the open stream.

GetBufferForFD(fd) = buffer

Allocates (from sysZone) and returns a buffer capable of holding one page of the file designated by *fd*. It is the caller's responsibility to free this buffer.

CloseIFSFile(fd, dPages [0])

Closes (i.e., unlocks) the open file designated by *fd*, but does not destroy the FD. If *dPages* is supplied and nonzero, then the disk page utilization in the user's Directory Information File (DIF) is updated by the amount *dPages*, which may be positive or negative. *dPages* should be the amount by which the file's size changed while it was open.

The directory should *not* be locked at the time of the call and is not locked when CloseIFSFile returns.

CloseIFSStream(stream) = fd

Closes an open file given its associated stream handle, and destroys the stream, but does not destroy the FD. This procedure calls CloseIFSFile internally and takes care of the dPages computation. The FD is returned for convenience in enumerating multiple files. The Closes stream operation is identical to CloseIFSStream except that it also destroys the FD.

DeleteFileFromFD(fd, deleteUndeletable [false], fileLocked [false]) = 0 or error code

Deletes the file designated by *fd*, returning zero if successful and an error code if unsuccessful. This procedure deletes both the directory entry and the file itself. It differs from IFSDeleteFile in that it accepts an FD rather than a name and does not destroy the FD, so it is useful when deleting multiple files.

A file may have an *undeletable* attribute that prevents it from being deleted even if all other conditions are satisfied; if *deleteUndeletable* is true, the *undeletable* attribute is ignored.

DeleteFileFromFD assumes that the directory is unlocked at the time of the call and leaves it unlocked upon return. Ordinarily it assumes that the file is not locked; that is, if the file is currently open, DeleteFileFromFD will fail with ecFileBusy. However, if *fileLocked* is true, the caller asserts that the file is write-locked at the time of the call; DeleteFileFromFD unlocks the file while deleting it. (In this case, if the delete operation fails for some other reason, the file remains

locked upon return.)

RenameFileFromFD(fd, newName, newDirName [CtxRunning.connName]) = 0 or error code

Renames the file designated by *fd* to be *newName*, whose default directory is *newDirName*. This is similar to IFSRenameFile except that the old file is specified by an FD rather than by a string. When the rename is completed, *fd* still designates the file under its old name; this enables one to rename multiple files matching a single *fd*.

ChangeFileAttributes(fd, proc, arg) = 0 or error code

Facilitates changing leader page attributes (e.g., protections) of the file designated by *fd*. This procedure checks access, locks the directory, and calls *proc*(fd, buffer, arg), where *buffer* is a pointer to a buffer containing a copy of the file's leader page and *arg* is the *arg* passed to ChangeFileAttributes. If *proc* returns true, the leader page is rewritten.

The caller must either be the owner of the file (in the sense of UserOwns, below) or have write access to it. The directory must be unlocked at the time of the call and is unlocked upon return.

TransferLeaderPage(fd, buffer, write [false])

Transfers the leader page of the file designated by *fd* to or from the supplied page-size buffer. The page is written if *write* is true and read otherwise. The file need not be open, but if it isn't the directory must be locked and the FD must have been validated by LookupFD or NextFD.

LockTransferLeaderPage(fd, buffer, write [false]) = 0 or error code

Locks the directory, performs the same action as TransferLeaderPage, and unlocks the directory. This procedure calls LookupFD internally to revalidate *fd*; if that fails, an error code is returned.

UserOwns(fd, userInfo [CtxRunning.userInfo]) = true or false

Returns true if the user described by *userInfo* is the owner of the file described by *fd*. This determination is made strictly syntactically: if the user's login or connected directory name is the same as the directory name of the file then UserOwns returns true. No reference is made to the directory, and the file need not actually exist.

CheckAccess(prot, owner, userInfo [CtxRunning.userInfo]) = true or false

Returns true if the user described by *userInfo* has access to the object protected by *prot*, which is a Protection structure (defined in IFSFiles.decl). *owner* should be true if the user owns the object in question and false otherwise (e.g., for a file, the value of *owner* should be UserOwns(fd, userInfo)).

If the system is configured to use Grapevine for access control, CheckAccess will consult Grapevine when necessary and will update *userInfo* and the user's DIF to reflect any newly-discovered group memberships.

Directory Administration Operations

ReadDIF(name, fs [primaryIFS], lvErrorCode []) = dif or 0

Reads the Directory Information File (DIF) for the supplied directory name, and

returns a DIF structure (see IFSFiles.decl) which the caller must free when done with it. The initial DIFRec portion of the DIF is copied from the cached information in the DIF's directory entry rather than from the file itself so as to obtain the up-to-date value for the disk page usage. The caller must have read access to the DIF (which is ordinarily protected against all users except the owner).

WriteDIF(name, dif, fs [primaryIFS]) = 0 or error code

Creates or updates the DIF for the supplied directory name. *dif* must point to a completely filled-in DIF structure. This operation includes updating the information cached in the DIF's directory entry from the initial DIFRec portion of the DIF. The caller must have wheel' capability.

CreateUser(name, password, diskLimit [1000], owner [0], capabilities [0], worldRead [false], fs [primaryIFS]) = 0 or error code

Creates a new user directory (i.e., a DIF) with the parameters supplied and the remaining parameters set to default values. Returns zero if successful and an error code if unsuccessful. If *owner* is nonzero, a files-only directory is created and *owner* is taken to be a BCPL string specifying the directory's owner. The default file protection is set to give read access to the world if *worldRead* is true. The caller must have wheel' capability.

This procedure is useful primarily during file system creation for establishing the essential built-in directories (e.g., <System>). It provides means for setting only a subset of all possible directory parameters.

DestroyUser(name, fs [primaryIFS]) = 0 or error code

Destroys the named user directory, returning zero if successful and an error code if unsuccessful. All files whose names begin with *<name>*' are destroyed, including the DIF. Since the files are deleted using normal access methods, the caller should be prepared to retry the call after errors such as ecFileBusy. The caller must have wheel' capability.

WheelCall(proc, args ...) = result

Enables the current process's wheel' capability and then calls *proc* with the remainder of the argument list as its arguments; then restores the process's original capabilities and returns the result returned by *proc*. This has the effect of bypassing all access checks for operations performed during the call to *proc*; it is the caller's responsibility to determine whether this is reasonable.