

AmigaMail

COLLABORATORS

	<i>TITLE :</i> AmigaMail		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 19, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	AmigaMail	1
1.1	II-49: Directory Scanning	1
1.2	Atomic Directory Scanning	1
1.3	MultiDirectory Assigns	3
1.4	Filename Matching	4

Chapter 1

AmigaMail

1.1 II-49: Directory Scanning

by Ewout Walraven

Prior to release 2.0, examining the contents of directories using `dos.library` required the use of two functions: `Examine()` and `ExNext()`. Although these routines perform the task for which they were intended, they have limitations. One is that these functions require stepping through a directory one entry at a time. For most applications that need to do directory scanning, it would more efficient to scan a directory in one pass rather than many. This would significantly reduce the time spent scanning. Also, these functions don't know anything about the AmigaDOS wildcards, so any wildcard processing must be done by the application, not by the OS.

Atomic Directory Scanning
MultiDirectory Assigns
Filename Matching

1.2 Atomic Directory Scanning

The `dos.library` function `ExAll()` is a powerful, V37 replacement for the `Examine()` and `ExNext()` functions:

```
BOOL ExAll(BPTR mydirlock, UBYTE *mybuffer,  
           LONG mybuffersize, LONG mydatatype,  
           struct ExAllControl *myexall);
```

`ExAll()` performs a one pass directory scan on a directory lock (`mydirlock` from the prototype above). `ExAll()` fills a buffer (`mybuffer` from the above prototype) with partial or whole `ExAllData` structures (from `<dos/exall.h>`):

```
struct ExAllData {  
    struct ExAllData *ed_Next; /* Pointer to the next structure */
```

```

    UBYTE *ed_Name;           /* File name */
    LONG  ed_Type;           /* File type */
    ULONG ed_Size;           /* File size */
    ULONG ed_Prot; /* Protection bits (FIBF_ definitions in <dos/dos.h> */
    ULONG ed_Days; /* Date in three longwords, forming a DateStamp. */
    ULONG ed_Mins;
    ULONG ed_Ticks;
    UBYTE *ed_Comment;        /* File comment. Cannot be used */
};

```

ExAll() copies an ExAllData structure into mybuffer for each entry in the directory. The size of the ExAllData structure depends on the value in mydatatype. The mydatatype parameter can have the following values:

ED_NAME	- ed_Name - file name
ED_TYPE	- ed_Type - file type (directory, file, soft link, etc.)
ED_SIZE	- ed_Size - file size
ED_PROTECTION	- ed_Prot - file protection bits
ED_DATE	- ed_Days, ed_Mins, ed_Ticks - file date in long words
ED_COMMENT	- ed_Comment - file comment (not currently supported).

Each of the possible mydatatype values corresponds to a field (or in the case of ED_DATE, a set of three fields) in the ExAllData structure. When ExAll() writes an ExAllData structure to mybuffer, it writes only the field that corresponds to mydatatype plus the fields that precede that datatype in the ExAllData structure. For example, if mydatatype is ED_SIZE, ExAll() would write ed_Next, ed_Type, and ed_Size to the buffer, ignoring the fields that follow ed_Size in the the ExAllData structure.

The ExAllData structures in mybuffer are organized into a linked list. The ExAllData structure's ed_Next field either points to the next directory entry's ExAllData structure or is NULL if no directory entries follow. Applications should only access the ExAllData structures using this link list.

As ExAll() scans a directory, it copies partial or whole ExAllData structure into mybuffer until it either runs out of directory entries or until it runs out of room in mybuffer. If ExAll() runs out of room, it will return a non-zero value, indicating that your application needs to perform more passes to finish scanning the directory.

To keep track of everything, ExAll() uses an application-supplied structure called ExAllControl (from <dos/exall.h>):

```

struct ExAllControl {
    ULONG eac_Entries; /* The number of entries returned in the buffer */
    ULONG eac_LastKey; /* Used to keep track of position in directory. */
    /* Do not change this value! */
    UBYTE *eac_MatchString; /* Optional parsed pattern for pattern match */
    struct Hook *eac_MatchFunc;
    /* Optional application hook to be called for each entry */
    /* Can be used to individually allow entries in the      */
    /* buffer or not                                          */
};

```

The ExAllControl structure must be allocated and freed using AllocDosObject() and FreeDosObject():

```
myexallcontrol = AllocDosObject(DOS_EXALLCONTROL, NULL);
FreeDosObject(DOS_EXALLCONTROL, myexallcontrol);
```

The ExAllControl structure's eac_Entries field contains the number of directory entries that ExAll() wrote into mybuffer. The eac_LastKey field is used by the file system to keep track of its place in the directory. An application must set this field to zero before calling ExAll() and must not make any changes to this field between scans of a directory.

The eac_MatchString field is used to pattern match the names of directory entries using the standard AmigaDOS pattern matching functions (for more on AmigaDOS pattern matching see the article ``Using the AmigaDOS Pattern Matching Functions'' in the September/October issue of Amiga Mail. If eac_MatchString is not NULL, ExAll() will only create ExAllData structures for the directory entries whose names match the matching string. Note that this matching string must have been parsed by ParsePatternNoCase(). If eac_MatchString is NULL, ExAll() will not perform any pattern matching.

The eac_MatchFunc field points to an application-supplied hook function. If this hook is not NULL, ExAll() will call the hook function. If the hook function returns TRUE, ExAll() will copy an ExAllData structure for this directory entry into mybuffer. The hook is called in the following manner:

```
BOOL = MatchFunc(hookptr, exalldata, typeptr)
                A0         A1         A2
```

where:

hookptr is a pointer to the hook being called exalldata is a pointer to the exalldata structure of the current directory entry typeptr is a pointer to a longword indicating the type of the ExAllData structure (mydatatype from the ExAll() prototype above).

By supplying a hook, each entry in the directory can be accepted or rejected according to your applications needs. An application can use both the matching string and MatchFunc() to perform AmigaDOS pattern matching and custom matching on directory entries.

The code example ListDir.c at the end of this article is a simple example of how to use ExAll(). The example ListDir2.c is a more complicated example that uses pattern matching and a hook function.

1.3 MultiDirectory Assigns

As of V36 it is possible to have a logical device assignment to more than one directory (see the dos.library Autodocs for AssignLock()/AssignAdd()). Since the user can utilize this with the C:Assign command, it is good practice to support this feature. The shell itself supports multidirectory assigns, although not all C: commands do. In general, when your application is presented with only a device name to scan, you should

check if it is an assignment. If it is, use `GetDeviceProc()` to get the handler for it, process it, and loop until `GetDeviceProc()` returns `NULL`, indicating there are no more directories for this assign. See the Autodocs for details.

The program `Find.c` is a more realistic example of the usage of patterns and `ExAll()` and shows a method of supporting multidirectory assigns. It scans one or more directories or volumes for the occurrence of a particular pattern. This example recursively scans subdirectories which means that `Find.c` may need more stack space than normal to keep from overflowing the stack. `Find.c` has two required arguments, a pattern and one or more directories to examine. It has two keywords, `FILES` and `DIRS`, to tell it to scan only for files or directories, respectively. The `ALL` keyword instructs `Find.c` to recurse into subdirectories when it encounters them.

1.4 Filename Matching

The release 2.0 `dos.library` introduced several other functions for directory scanning:

```
LONG MatchFirst(UBYTE *mypattern, struct AnchorPath *myanchorpath);
LONG MatchNext(struct AnchorPath *myanchorpath);
VOID MatchEnd(struct AnchorPath *myanchorpath);
```

When using these functions, an application first calls `MatchFirst()`, which performs some initialization (like calling `ParsePattern()` on the pattern matching string, `mypattern`) and finds the first directory entry that matches the directory entry `mypattern`. This pattern is relative to the current directory. An application must use the `MatchNext()` call to find subsequent matching directory entries. After the application is done looking for matches or the application encounters an error, it must call `MatchEnd()` to release internal buffers.

Before using these functions, you need to set up an `AnchorPath` structure. This structure must be initialized by `MatchFirst()` and passed to `MatchNext()` and `MatchEnd()` so they can keep track of the directory scan. An application must not make any changes to this structure while in the middle of a directory scan (before calling `MatchEnd()`). This `AnchorPath` structure must be longword aligned and is defined in `<dos/dosasl.h>` as follows:

```
struct AnchorPath {
    struct AChain  *ap_Base; /* Pointer to the first anchor */
#define ap_First ap_Base    /* Compatibility synonym. Don't use */
    struct AChain  *ap_Last; /* Pointer to the last anchor */
#define ap_Current ap_Last  /* Compatibility synonym. Don't use */
    LONG          ap_BreakBits; /* Bit flags to stop scanning */
    LONG          ap_FoundBreak; /* Bits flags which caused the stop */
    BYTE          ap_Flags; /* Behaviour flags */
    BYTE          ap_Reserved; /* Reserved for now */
    WORD          ap_Strlen; /* Buffer size for path name */
                                /* This used to be ap_Length */
#define ap_Length ap_Flags /* Compatibility for LONGWORD ap_Length */
```

```
/* Don't use */
struct FileInfoBlock ap_Info; /* FileInfoBlock for matched entry */
UBYTE ap_Buf[1]; /* Application allocated buffer for full */
/* path name*/
};
```

If your application does not need a full path name to matching directory entries, it should initialize the `ap_Strlen` field to zero. In this case, your application can get what it needs from the `AnchorPath`'s `ap_Info` field. It can also get a lock on the directory from `ap_Current->an_Lock` field. If your application needs the full path name of matching directory entries, it must allocate a buffer at the end of the `AnchorPath` structure and put the size of the buffer, in bytes, into `ap_Strlen`.

The `ap_BreakBits` field allows the user to abort a directory scan in progress. The bits in this field correspond to the `SIGBREAKF_` bits defined in `<dos/dos.h>`. If the corresponding bit is set in `ap_BreakBits`, `MatchFirst()` or `MatchNext()` will stop a scan in progress if one of those signals occurs. If this occurs, the bit of the signal that caused the break will be set in `ap_FoundBreak`.

With this information alone it is possible to perform simple file pattern matching. As previously mentioned, the first match must be found with `MatchFirst()`. If `MatchFirst()` (or `MatchNext()`) cannot find a match or it encounters an error, it returns an error number, otherwise it returns a zero (which is unusual for a `dos.library` function). If `MatchFirst()` does not encounter any problems, the application should look for subsequent matches by calling `MatchNext()`. The application should call `MatchNext()` until it returns an AmigaDOS error value. `MatchNext()` accepts a pointer to the `AnchorPath` structure initialized by `MatchFirst()`.

Normally, the error that `MatchNext()` returns is `ERROR_NO_MORE_ENTRIES`, indicating that there are no more directory entries that match mypattern. `MatchEnd()` is used to release any resources that were allocated in the scanning process. Due to a number of bugs in the V36 implementation, these functions should only be used as of V37. `ListPattern.c` is a simple example of using `MatchFirst()/MatchNext()`.

For more complex matching, the `ap_Flags` field can be used to define the behavior of `MatchFirst()` and `MatchNext()`. Currently, there are several flags defined:

```
APF_ITSWILD
APF_DODIR
APF_DIDDIR
APF_NOMEMERR
APF_DirChanged
APF_FollowHLinks
```

The `APF_ITSWILD` flag will be set if a wildcard was present in the pattern after the call to `MatchFirst()`. It will be used to instruct `MatchNext()` but your application can check it too and perform an action depending on its status.

The `APF_DODIR` flag instructs `MatchFirst/Next()` to enter a directory if it encounters one. This flag can be set and reset on an individual basis.

Once `MatchNext()` is finished processing a directory, it will set the `APF_DIDDIR` bit and will change the `AnchorPath`'s directory back to the parent directory.

`APF_NOMEMERR` indicates that `MatchFirst/Next()` encountered a fatal out of memory error. Processing the directory should be aborted and an error returned to the user.

The `APF_DirChanged` flag indicates that `MatchNext()` noticed that directory lock has changed since the previous `MatchNext()` call.

The `APF_FollowHLinks` flag tells `MatchFirst/MatchNext()` to follow hard link directories if the `APF_DODIR` bit is set. This feature is in place to avoid confusing applications that do not know anything about hard links.

Most existing versions of the 2.0 include file `<dos/dosasl.h>` mention two other flags, `APF_DOWILD` and `APF_DODOT`. These are not currently in use by the system.

The `DirComp.c` example is a more complex example of using `MatchFirst()/Next()`. It takes a path, which may include wildcards, and compares the directory entries it finds with those in the target directory. If the user specifies the `DATE` keyword, `DirComp` will also compare the timestamps. The `ALL` keyword tells `DirComp` to do a recursive scan. For deeply nested directories the `BUFFER` keyword can enlarge the buffer that `DirComp` uses from its standard 256 bytes up to 4096 bytes (calling it the `''Joanne''` keyword might be more appropriate).
