

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-113: Handling Multiple Assigns with Conventional Directories	1

Chapter 1

AmigaMail

1.1 II-113: Handling Multiple Assigns with Conventional Directories

Handling Multiple Assigns with Conventional Directories

Staff

One of the features introduced by Release 2 is Multiple Assigning. This feature allows an AmigaDOS assign to carry over several directories which can be on different volumes. This makes it possible to split up assigns such as `libs:` and `fonts:`.

The article ``Directory Scanning'' on page II-49 contains an example called `find.c` that illustrates scanning a path that can contain a multiassign. However, besides being rather complicated, `find.c` makes a special case of scanning assigns, which isn't necessary (`find.c` also did something evil--`find.c` uses `DOSBase`'s private pointer to the `utility.library`, essentially using the `utility.library` without opening it). The method needed to scan a multiassign directory also works on conventional directories.

Scanning a multiassign requires calling the `dos.library` function `GetDeviceProc()` in a loop to see each directory of the multiassign. Using `GetDeviceProc()`, the application doesn't have to concern itself with the differences between assigns, multiassigns, and volumes. The application just keeps calling `GetDeviceProc()` until it gets back a `NULL`.

```
struct DevProc *GetDeviceProc( STRPTR name, struct DevProc *dp );
```

The name can be any valid dos path. If there is a device name present, `GetDeviceProc()` will find the device's entry in the dos list and copy some information into a `DevProc` structure:

```
struct DevProc {
    struct MsgPort *dvp_Port;    /* Device's Message port, also called a Process ↵
        identifier */
    BPTR          dvp_Lock;      /* Lock on root of assign or lock on root of ↵
        volume */
    ULONG          dvp_Flags;
    struct DosList *dvp_DevNode; /* DON'T TOUCH OR USE! */
};
```

The important fields here are `dvp_Lock` and `dvp_Port`. The `dvp_Lock` field is a lock on the root of the object named in `GetDeviceProc()`. It serves as a starting point in locating the named object. If the object name contains an assign (i.e. `''libs:''`), `dvp_Lock` is the root of the assign. For example, on a typical Release 2 system, the `libs:` assign refers to the `libs` directory on the `System2.0:` volume. Calling `GetDeviceProc()` on `''libs:''` in this case will yield a lock on `System2.0:libs`.

If the named object contains a dos volume, `dvp_Lock` is either a lock on the root of the dos volume or `NULL`. If the object named in `GetDeviceProc()` contains a non-filesystem device (i.e., `''ser:''`, `''par:''`, `''prt:''`, etc.) or it does not contain a device name, `dvp_Lock` is `NULL`.

The `dvp_Port` field points to a message port. This message port is connected to the handler process of a DOS device. The handler process controls a DOS device. DOS functions (like the `Lock()` function) use this message port to talk to the handler process of the named object. For example, from the `''libs:''` example above, the `dvp_Port` field refers to the message port of the handler process for the `System2.0:` volume.

Note that `dvp_Lock` is only a lock on the root of the named object. If the named object is a path several directories deep (for example, `libs:gadgets/colorwheel.gadget`), it's up to the application to handle the rest of the path. The application also has to handle the case where the named object is a path without a device name.

Although an application can send DOS packets directly to the message port (`dvp_Port`) of a handler process, normally it is easier to use functions from `dos.library`. The `multilist.c` example uses the `Lock()` function to lock the named object. `Multilist` has to do something a little unorthodox to use `Lock()`. `Lock()` accepts a path name to the object to lock. `Lock()` understands absolute paths (i.e. paths with a logical device name like `''df1:''` or `''libs:''`) and relative paths. If `Lock()` receives an absolute path name, `Lock()` can find the device's handler process using the logical device name in the absolute path. For a relative path, `Lock()` does not have enough information to find the named object, so it assumes the path is relative to the current directory and file system (each process has a current directory and file system).

This makes `Lock()` a little more difficult to use in `multilist.c` because, when processing an absolute path, `multilist` has to process the logical device name separately from the rest of the path. It has to use `GetDeviceProc()` to find the root of a logical device name (which can be an assign, multiassign, volume name, etc.) then it has to strip the logical device name from the absolute path. Without a logical device name, the path has become relative rather than absolute. The path is now relative to `dvp_Lock` and `dvp_Port`. In order for `Lock()` to work with this relative path, `multilist` must temporarily set the current directory and file system to the values in `dvp_Lock` and `dvp_Port`, respectively.

Note that the Autodoc for `GetDeviceProc()` says to check `IoErr()` for `ERROR_NO_MORE_ENTRIES` after receiving a `NULL` from `GetDeviceProc()`. Due to a bug, DOS does not set the error value correctly. Also note that the Autodoc says to check the `DevProc` structure's `dvp_Flags` field for the `DVPF_ASSIGN` flag. This was necessary in the 2.00 and 2.01 releases of the operating system due to a bug in DOS, but is no longer necessary.

The following example, `multilist.c`, accepts an arbitrary path name and lists the contents of it. The function `DoAllAssigns()` does all of the multiassign work. `DoAllAssigns()` accepts a path and a function pointer. It gets a lock on the object named in the path, and passes the lock to the function. This example is based on a Usenet posting by Randell Jesup.

`multilist.c`
