Advanced AmigaDOS Routines

COLLABORATORS

	TITLE : Advanced AmigaDOS Routines	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 12, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Advanced AmigaDOS Routines

1.1 Chapter 7 - Advanced AmigaDOS Routines

```
                Previous Chapter:                                 Next Chapter:
 6. Handlers
---------------------------------------------------------------
```

CHAPTER 7 - ADVANCED AMIGADOS ROUTINES

1.2 Introduction

INTRODUCTION

In this chapter I will describe some advanced routines in AmigaDOS. Although this chapter is mainly intended for experienced programmers I have tried to make it as easy as possible to read even if you are new C programmer and have not worked so much with AmigaDOS.

1.3 Get Information about Files and Directories

GET INFORMATION ABOUT FILES AND DIRECTORIES

In some occations you might need to get some information about a file or directory. You might want to see which protection bits are currently set, how big the it is, on what date it was created, what comment is attached and so on... To get this type of information you should use the Examine()

Examine() allows you to examine files, directories and volumes. In the following sections I will now and then refer to all of these types as "objects".

There are two things you have to do before you can call Examine():

1. You must lock the object you want to examine. Simply use the Lock() function as previously explained:

Lock the Object
2. Create a FileInfoBlock structure. The problem with this structure is that it is used by AmigaDOS directly and must therefore be long word aligned! (Since you have to create the structure you must make sure that it is long word aligned. Structures that are created by AmigaDOS itself will always be long word aligned.)

Create a File Info Block
You can now call Examine() and check the file/directory:

Call Examine()

Examine the File Info Block

Clean Up

1.4 Lock the Object

LOCK THE OBJECT

First you have to lock the object you want to examine with help of the Lock()
(read some values) it is enough with a shared lock ("read lock").

Here is an example:

```
  /* A "BCPL" pointer to our lock: */
  BPTR my_lock;

  - - -
  /* Try to lock the object we later will examine: */
  my_lock = Lock( "RAM:Highscore.dat", SHARED_LOCK );

  /* Check if we have successfully locked the object or not: */
  if( my_lock == NULL )
    printf( "Could not lock the object!\n" );
```

1.5 Create a File Info Block

```
                CREATE A FILE INFO BLOCK

Before we can call  Examine()
FileInfoBlock structure. This structure must, as already
explained, be long word aligned ("a present from the wonderful
BCPL language"). How we should create this structure depends
on if your program should be compatible with the old dos
libraries (WB1.2 and WB1.3) or not.


                 Old Dos Versions (WB1.3 or WB1.2)

                 New Release 2 (WB2.04) or Higher
```

1.6 Old Dos Versions (WB1.3 or WB1.2)

```
OLD DOS VERSIONS (WB1.3 OR WB1.2)

If your program should be able to run on the old systems (dos
libraries older than V37) you should allocate the
FileInfoStructure with help of the AllocMem() function.

Here is an example:

  /* Declare a pointer to a FileInfoBlock structure: */
  struct FileInfoBlock *my_fib_ptr;

  - - -

  /* Allocate enough memory for a FileInfoBlock structure: */
  /* (OK! This memory will be long word aligned.)          */
  my_fib_ptr = (struct FileInfoBlock *)
    AllocMem( sizeof( struct FileInfoBlock ), MEMF_ANY | MEMF_CLEAR );

  /* Check if we have allocated the memory successfully: */
  if( my_fib_ptr == NULL )
    printf( "Could not allocate enough memory!\n" );
```

1.7 New Release 2 (WB2.04) or Higher

NEW RELEASE 2 (WB2.04) OR HIGHER

If your program only should be used on systems with dos library V37 or higher (WB2.04 or higher) you should use the new AllocDosObject()

Here is a simple example:

```
  /* Declare a pointer to our FileInfoBlock */
  /* which we will allocate:                */
  struct FileInfoBlock *my_fib;

  - - -

  /* Create a FileInfoBlock structure with help */
  /* of the new AllocDosObject() function:      */
  my_fib = AllocDosObject( DOS_FIB, NULL );

  /* Check if we have allocated the memory successfully: */
  if( !my_fib )
    printf( "Could not allocate the FileInfoBlock!\n" );
```

1.8 Call Examine()

CALL EXAMINE()

Once you have locked the object you want to examine and you have allocated a FileInfoBlock structure you may call the Examine()

1.9 Examine the File Info Block

EXAMINE THE FILE INFO BLOCK

If you have successfully examined the object you may look at the different fields in the FileInfoBlock structure. The structure is defined in header file "dos/dos.h" like this:

```
  struct FileInfoBlock
  {
    LONG fib_DiskKey;
    LONG fib_DirEntryType;
    char fib_FileName[108];
    LONG fib_Protection;
    LONG fib_EntryType;
    LONG fib_Size;
    LONG fib_NumBlocks;
    struct DateStamp fib_Date;
    char fib_Comment[80];
    char fib_Reserved[36];
  };

fib_DiskKey:       Key number for the disk. Usually of no
                  interest for us.

fib_DirEntryType: If the number is smaller than zero it is a
                  file. On the other hand, if the number is
                  larger than zero it is a directory (or volume
                  or device).

                    /* Is it a file or directory (etc)? */
                    if( my_fib->fib_DirEntryType < 0 )
                      printf( "File\n" );
                    else
                      printf( "Directory or Volume\n" );

fib_FileName:      Null terminated string containing the file-
                  name. (File names may not be longer than 30
                  characters.)

fib_Protection:    Field containing the protection flags:
                  (if set)

                  FIBF_DELETE  : the file/directory can not be
                                 deleted.
                  FIBF_EXECUTE : the file can not be executed.
                  FIBF_WRITE   : you can not write to the file.
                  FIBF_READ    : you can not read the file.
                  FIBF_ARCHIVE : Archive bit.
                  FIBF_PURE    : Pure bit.
                  FIBF_SCRIPT  : Script bit.

                    /* Is the object protected: */
                    if( my_fib->fib_Protection & FIBF_DELETE )
                      printf( "Protected!" );
                    else
                      printf( "Not protected!" );

fib_EntryType:     File/Directory entry type number. Usually of no
                  interest for us.

fib_Size:          Size of the file (in bytes).

fib_NumBlocks:     Number of blocks in the file.

fib_Date:          Structure containing the date when the file
                  was latest updated/created. See below for
                  more information.

fib_Comment:       Null terminated string containing a comment.
                  (Max 80 characters including the NULL sign.)

fib_Reserved:      This field is for the moment reserved, and may
                  therefore not be used.
```

The DateStamp structure which is a part of the FileInfoBlock structure is also defined in header file "dos/dos.h", and looks like this:

```
  struct DateStamp
  {
    LONG ds_Days;
    LONG ds_Minute;
    LONG ds_Tick;
  };
```

```
ds_Days:   Number of days since 01-Jan-1978.

ds_Minute: Number of minutes past midnight.

ds_Tick:   Number of ticks past the last minute. There are 50
           ticks / second. (50 * 60 = 3000 ticks / minute.)
```

See the examples for a complete list of how to examine the FileInfoBlock structure.

1.10 Clean Up

CLEAN UP

Once you have examined the FileInfoBlock structure and do not want to use it any more you should deallocate it. If you allocated it with help of AllocMem() you must free it with help of FreeMem():

```
  /* Deallocate the memory when we do not need it any more: */
  FreeMem( my_fib_ptr, sizeof( struct FileInfoBlock ) );
```

However, if you allocated the structure with help of the new AllocDosObject() function you have to use the FreeDosObject() function to deallocate the structure.

You must of course also unlock the object when you do not need it any more. Simply use the UnLock()

```
  /* Unlock the object: */
  UnLock( my_lock );
```

1.11 Examine Files/Subdirectories in a Directory/Device

EXAMINE FILES/SUBDIRECTORIES IN A DIRECTORY/DEVICE

A directory, volume or device (I will refer to them all as "directories") can contain several files as well as several (sub)directories. If you want to examine all objects in a

directory you should first use the Examine()
examine the directory.

If you discover that the current object is a directory you
can use a function called ExNext()
the objects in the directory. The first time you call ExNext()
you will get information about the first object in the
directory. You can then call ExNext() again to get information
about the next object and so on until there are no more objects
left to examine and the ExNext() function fails.

It it important to remember that ExNext() can only be called
after you first have successfully called Examine(), and you
must use the same FileInfoBlock structure each time.

1.12 Get Information About a Disk

GET INFORMATION ABOUT A DISK

To get information about a disk you must first create an
"InfoData" structure in which all information will be stored.
As you probably already have guessed this structure must be
long word aligned. Once you have created the structure you
should lock the disk you want to examine and finally you can
check it:

1. Create the InfoData Structure

2. Lock the Disk

3. Call the Info() Function

4. InfoData Structure

1.13 Create the InfoData Structure

CREATE THE INFODATA STRUCTURE

Since the
InfoData structure
has to be long word aligned you
have to use AllocMem() to create it. (AllocDosObject()
support this type of object.)

Here is an example:

```
  /* Declare a pointer to our */
  /* info data block:         */
  struct InfoData *my_info_data;

  - - -
  /* Allocate memory for an InfoData structure: */
  my_info_data = (struct InfoData *)
    AllocMem( sizeof( struct InfoData ), MEMF_ANY );

  /* Have we successfully allocated the memory? */
  if( !my_info_data )
    printf( "Could not allocate enough memory!\n" );
```

1.14 Lock the Disk

```
                LOCK THE DISK

Once you have created an
                 InfoData structure
                 you should lock
the disk you want to examine with the help of the  Lock()
function. (Of course the order does not matter and you could
equally well have first locked the disk and then allocated the
InfoData structure.) Since we will only look at the disk it is
enough with a shared lock.

Here is an example:

  /* A "BCPL" pointer to our lock: */
  BPTR my_lock;

  - - -

  /* Lock the disk (in this case "df0:"): */
  my_lock = Lock( "df0:", SHARED_LOCK );

  /* Have we successfully locked the disk? */
  if( !my_lock )
    printf( "Could not lock the disk!\n" );
```

1.15 Call the Info() Function

```
CALL THE INFO() FUNCTION

At last we can call the  Info()
```

1.16 InfoData Structure

```
INFODATA STRUCTURE

If you have successfully called the  Info()
start to examine the fields in the InfoData structure which has
now been initialized. The InfoData structure is defined in
header file "dos/dos.h" like this:
  struct InfoData
  {
     LONG    id_NumSoftErrors;
     LONG    id_UnitNumber;
     LONG    id_DiskState;
     LONG    id_NumBlocks;
     LONG    id_NumBlocksUsed;
     LONG    id_BytesPerBlock;
     LONG    id_DiskType;
     BPTR    id_VolumeNode;
     LONG    id_InUse;
  };

id_NumSoftErrors: Number of soft errors on the disk. (Number
                  of damaged areas.)

id_UnitNumber:    In which unit the disk is in. (Note that it
                  might have been removed after you have called
                  the Info().)

id_DiskState:     The disk can have one of the following three
                  different states:

                  ID_VALIDATING      The disk has just been
                                     inserted and AmigaDOS is
                                     trying to see what type of
                                     disk it really is. (This
                                     flag can also be set if
                                     the disk is damaged, or
                                     there are internal problems
                                     in the filing system.)

                                     The disk can for the
                                     moment not be used when
                                     it is in this state.

                  ID_VALIDATED       The disk has been
                                     validated, and the disk
                                     is NOT write protected.

                  ID_WRITE_PROTECTED The disk has been
                                     validated, and the disk
                                     is write protected.

id_NumBlocks:     Number of blocks on the disk.

id_NumBlocksUsed: Number of blocks used.

id_BytesPerBlock: Size (in bytes) of each block.

id_DiskType:      There exist several different types of disks:

                  ID_NO_DISK_PRESENT No disk in the drive.
                                     (Interesting type of disk.)

                  ID_UNREADABLE_DISK The disk contains corrupted
```

```
                                      data and can not be used.

                    ID_DOS_DISK         It is a normal disk.

                    ID_FFS_DISK         The disk is using the
                                        "Fast Filing System" (FFS)

                    ID_INTER_DOS_DISK   It is a normal int. disk.

                    ID_INTER_FFS_DISK   The int. disk is using the
                                        "Fast Filing System" (FFS)

                    ID_NOT_REALLY_DOS   Not a dos (normal) disk.

                    ID_KICKSTART_DISK   It is a "Kickstart" disk.
                                        Special type of disk used
                                        on A1000 to load the
                                        system which is on the
                                        other Amiga models included
                                        in the Kickstart ROMs.

                    ID_MSDOS_DISK       It is an (IBM) MS dos
                                        disk. (The special program
                                        "CrossDos" which is
                                        included with WB 2.1 allows
                                        the user to also work with
                                        MS dos disks - 720 kB.)

id_VolumeNode:      Pointer to the volume node list which is
                    rarely used.

id_InUse:           If this field is non zero ths disk is in use.
                    (Since you must have locked the disk before
                    you could examine it, this field will always
                    be non zero since at least your program is
                    using the disk.)
```

1.17 The Internal Assign, Volume and Device List

THE INTERNAL ASSIGN, VOLUME AND DEVICE LIST

AmigaDOS has a list of all Assigns, Volumes and Devices it currently knows about. Whenever a disk is inserted or removed, a new assign is added etc... this list is automatically updated.

As a programmer you migh need to know which assigns, volumes or devices are currently available. A file requester should for example be able to display this list so the user can directly select the device, assign or volume he/she wants to go to.

If you want to get the names of all objects AmigaDOS currently knows about, and you want your program to be compatible with all dos library versions, you have to go deep down into the system. However, as long as you know what you are doing (or

```
follows my steps carefully) there is danger and we are not
breaking any "programming laws" by doing this.

This is what you have to do:

  1. Get a pointer to the dos library. We simply declare the
     global dos library pointer as external, and it will
     automatically be initialized for us as explained earlier.

       /* Declare an external global library */
       /* pointer to the Dos library:        */
       extern struct DosLibrary *DOSBase;


  2. In the DosLibrary structure you will find a pointer to
     a "RootNode" structure. This strucure contains some
     fundamental parts of AmigaDOS but shold not be used
     unless you really know what you are dowing.

       /* Declare a pointer to the RootNode structure: */
       struct RootNode *rootnode_ptr;

       - - -

       /* Get a pointer to the RootNode structure: */
       rootnode_ptr = DOSBase->dl_Root;


  3. In the RootNode structure we can find a BCPL pointer to
     a DosInfo structure.

       /* Declare a temporary BCPL pointer used */
       /* to convert BPTRs into C pointer with: */
       BPTR temp_bptr;

       - - -

       /* Get a BCPL pointer (BPTR) to */
       /* the DosInfo structure:       */
       temp_bptr = rootnode_ptr->rn_Info;


  4. Since you got a BPTR (a BCPL pointer) you must convert it
     into a normal C pointer before you can use it. (BCPL
     pointers are four times "smaller" than normal C pointers
     and it must therefore be multiplied by 4, which is done
     with help of the BADDR() macro.)

       /* Declare a pointer to a DosInfo structure: */
       struct DosInfo *dos_info_ptr;

       - - -

       /* Convert the BCPL pointer into a normal C pointer: */
       dos_info_ptr = (struct DosInfo *) BADDR( temp_bptr );
```

5. It is in this DosInfo structure you will find a linked list of "DosList" nodes. In each DosList node you will find the name of one device, assign or volume. However, before you may scan the linked list you have to "lock" it so it does not change while you are reading it. If the user inserts or removes a disk for example the list will be rebuilt and nodes may be added or taken away, and this must of course not happen while you are in the middle of the linked list!

   On the older dos libraries, prior to V36, there does not exist any function to directly lock the list. Instead you have to use the system function "Forbid()" which will turn off some parts of the multitasking. (Note that while you are in this "forbidden" mode you may not use any Wait() calls, and you should as quickly as possible return to normal state.)

   ```
   /* Turn off parts of the multitasking: */
   Forbid();
   ```

6. You can now scan the list of "DosList" nodes. Each DosList node (structure) has a pointer to the next node. In the last node this pointer is pointing NULL. (Note that while you are examining the nodes you have to convert a lot of BCPL pointers into normal C pointers.)

   ```
   /* Decalre pointer to the first DosList structure: */
   struct DosList *first_doslist_node;

   /* Decalre a pointer to the current (the one */
   /* we are working with) DosList structure:   */
   struct DosList *doslist_node;

   - - -

   /* Get a BCPL pointer (BPTR) to the */
   /* first "DosList" node:            */
   temp_bptr = dos_info_ptr->di_DevInfo;

   /* Convert the BPTR into a C pointer: */
   first_doslist_node = (struct DosList *)
     BADDR( temp_bptr );

   /* Start with the first node: */
   doslist_node = first_doslist_node;

   /* Stay in the loop until all */
   /* nodes have been checked:   */
   while( doslist_node )
   {

     - - -

     /* Examine the node... */
   ```

```
         - - -

         /* Go to next node: */

         /* Get a BPTR to the next node: */
         temp_bptr = doslist_node->dol_Next;

         /* Convert the BPTR into a C pointer: */
         doslist_node = (struct DosList *)
           BADDR( temp_bptr );
       }
```

  7. When all nodes have been examined you should as quickly as
     possible turn on the multitasking by calling the "Permit()"
     function.

```
       /* Turn the multitaskin ON again: */
       Permit();
```

easy as pie...

1.18 To Be Continued...

TO BE CONTINUED...

There are of course a lot of other advanced and interesting
things you can do with AmigaDOS, but that will be added in
coming updates... Remember to pay the registration fee so you
do not miss future updates!

                    TO BE CONTINUED....(!)

1.19 Examples

EXAMPLES

Example 1: Read!   Run!   Edit!
  This example demonstrates how to use the Examine() function.
  The program needs a file, directory or volume name as the
  only argument and it will print some interesting information
  about given the object.

  This example can be used with all versions of the dos
  library.

Example 2: Read!   Run!   Edit!
  This example does exactly the same thing as the previous one,
  it simply demonstrates how to use the Examine() function.
  However, this example uses the new AllocDosObject() and
  FreeDosObject() functions which were introduced in Release 2.
  You should use these new functions (if possible) instead of

```
  using the older method of allocating a fixed amount of memory
  for the dos object (the FileInfoBlock structure).

  This example can only be used with dos library V37 or higher.

Example 3:  Read!    Run!    Edit!
  This example demonstrates how to examine all objects in
  directory or volume. The program needs a directory or
  volume name as the only argument and it will then list
  all files and directories (subdirectories) in that
  directory or volume. This is a good example on how to
  use the Examine() and ExNext() functions.

  This example can be used with all versions of the dos
  library.

Example 4:  Read!    Run!    Edit!
  This program will examina all objects in a directory/device.
  The files will be listed, and if finds a directory it will
  with help of a recursive function examine all objects in
  that directory also and so on... Good example on how to use
  the Examine() and ExNext() functions in a recursive program.

  This example can be used with all versions of the dos
  library.

Example 5:  Read!    Run!    Edit!
  This example demonstrates how to use the Info() function
  to get some information about a disk. We will, among many
  things, check if the disk is write protected or not, what
  type of disk it is etc... In this example we examine the
  disk in "df0:".

  This example can be used with all versions of the dos
  library.

Example 6:  Read!    Run!    Edit!
  This example will examine some of the "lowest" parts in
  AmigaDOS. It will look up and print all Assigns, Volumes
  and Devices AmigaDOS knows about. Please note that we
  will dig fairly deep down into the system, and only
  experienced programmers are recommended to do this. I
  have added a lot of comments to help you, and if you cut
  out parts of this example carefully you should be able
  to use it in your own programs.

  This example can be used with all versions of the dos
  library.

Example 7:  Read!    Run!    Edit!
  This example will as the previous one examine the special
  lists of available Assigns, Volumes and Devices. This
  example will however add the volume name to the device
  in which the volume is. We will also only print the
  devices (with their volume name) which are currently
  available to access. We will for example not print the
  device "df0:" if there is not a disk in that drive.
```

```
However, if there is a disk in the drive we will both
print the device name and the name of the volume which is
in that device. We will therefore get a list which is
identical to the one used by the ASL file requeter.
```