Introduction

COLLABORATORS			
	TITLE : Introduction		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 12, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Introduction

1.1 Chapter 1 - Introduction

Previous Chapter: Next Chapter:
0. Contents

CHAPTER 1 - INTRODUCTION

1.2 Commonly Used AmigaDOS Words

COMMONLY USED AMIGADOS WORDS

Before you can continue with the other chapters you need to know the meaning of some commonly used words, and how AmigaDOS work. For more information about AmigaDOS and the Shell see the manuals which were included with your computer when you bought it.

Logical Devices
Physical/Logical Devices & Volumes

1.3 Physical Devices

PHYSICAL DEVICES

Physical devices are parts of the Amiga to which data can either be sent to, read from or both. The most commonly used physical device is undoubtedly the internal disk drive "DF0:", but there exist a lot of other physical devices.

Here is the list of the standard AmigaDOS physical devices:

```
DF0:   File IO on disk drive 0
DF1:   File IO on disk drive 0
DF2:   File IO on disk drive 0
DF3:   File IO on disk drive 0

RAM:   File IO on RAM disk
DHx:   File IO on hard drive x (DH0, DH1, DH2, and so on...)

SER:   Buffered serial IO
PAR:   Buffered parallel IO
PRT:   Output to printer (trough Preferences)

CON:   Buffered translated window IO
RAW:   Unbuffered untranslated window IO

PIPE:  Buffered IO between programs
AUX:   Unbuffered serial IO
SPEAK: Output to the narrator (speech) device

NIL:   Output to nothing
```

(IO = Input / Output)
(Note that all device-names end with a colon)
(The listed devices are also usually called as "handlers")

If you want to copy a file called "program.c" from the internal disk drive (DF0:) to the second disk drive (DF1:) you write:

```
1.Prog:>
1.Prog:> copy from DF0:program.c to DF1:
```

If you want to print the file you can do it by simply copying the file to the "PRT:" device. The data in the file will be translated with help of the current settings in Preferences and outputed to the printer.

```
1.Prog:>
1.Prog:> copy from DF0:program.c to PRT:
```

Since AmigaDOS treats all devices equal you can even let it read your file out loud with help of the "speak" device. You only have to copy the file to the "SPEAK:" device, and the Amiga will immediately start to read the file out loud.

```
1.Prog:>
1.Prog:> copy from DF0:program.c to SPEAK:
```

AmigaDOS allows you also to use special windows (CON: or RAW:) for input/output. The only thing you need to add are some extra arguments just after the device name. These extra arguments tells AmigaDOS what size and position of the window you want, and if you want to use any special options. The syntax for the console window is:

```
CON:x/y/width/height/title/option(s)
```

If you want to use spaces in the title you have to put quotation marks around the whole expression. With "Release 2" (WB2.xx) some special options where added to the "CON:" device:

Option	Description
/AUTO:	Opens the window first when there is some IO
/BACKDROP:	Should be a "backdrop" window (no other window can be moved behind it)
/CLOSE:	Add a close window gadget
/NOBORDER:	Draw no borders around the window
/NODRAG:	Remove the drag gadget
/NOSIZE:	Remove the size gadget
/SCREEN:	Open on a specified public screen "/SCREEN [name]"
/SIMPLE	The window should use "simple refresh" mode (see Intuition manual for more information)
/SMART:	The window should use the "smart refresh" mode (see Intuition manual for more information)
/WAIT:	Waits with closing the window until the user types "Ctrl-\" or clicks on the close window gadget
/WINDOW:	Use the specified window (address in hexadecimal) "/WINDOW [pointer to window]"

To copy a file called "prg.c" from the current device to a "CON:" window you simply write:

```
1.Prog:>
1.Prog:> copy prg.c to CON:10/20/320/100/Text/CLOSE/WAIT/
```

This will copy the data in the file to a window positioned at x position 10, y position 20, width 320, height 100, with the title "Text", and it will be closed first when you click on the close window gadget which has been added. If you want to use spaces in the title you need to put quotes around the whole expression:

```
1.Prog:>
1.Prog:> copy prg.c to "CON:10/20/320/100/My Text/CLOSE/WAIT/"
```

All these types of "physical devices" which have been listed
are also sometimes called "handlers". The handler is actually
the process behind the device, and will be fully explained in
the chapter Handlers

1.4 Volumes

VOLUMES

To access a disk you can either use the physical device name of
the drive in which the disk is, or use the name of the disk
(the "volume name"). If you want to copy a file called
"program.c" from the RAM disk to a disk called "DOCUMENTS", and
the disk is in the second disk drive, you can either write:

```
  1.Prog:>
  1.Prog:> copy from RAM:program.c to DF1:
```

or

```
  1.Prog:>
  1.Prog:> copy from RAM:program.c to DOCUMENTS:
```

(Note the colon after the volume name!) The advantage with the
last example is that you do not need to bother about which
drive the DOCUMENTS disk is in. Furthermore, if the desired
disk was not in any drive, AmigaDOS will ask you to insert it,
and you do not need to worry about writing to the wrong disk.

1.5 Directories/Subdirectories/Files

DIRECTORIES/SUBDIRECTORIES/FILES

On a disk there may exist directories, subdirectories and files.
A subdirectory is a directory inside another directory. (There
may be subdirectories inside other subdirectories and so on...)

Picture: Dir&Sub.pic

If you want to access a file which is in the "current
directory" (the directory you are currently standing in) you
only have to use the file name.

```
  "file name"
```

If you want to access a file which is in a directory further
in you have to add the directory name, a slash (/) and then
the file name. If the file is several directories in you
have to add the name of each directory and a slash for every
directory (subdirectory).

```
  "directory name"/"file name"
```

```
  "directory name"/"subdirectory name"/"file name"
  and so on...

If you are standing in a directory and want to access a file
which is outside (one step back) you have to put a slash in
front of the file name. If the file is several directories back
you have to add a slash for every directory.

  /"file name"
  //"file name"
  ///"file name"
  and so on...

If you want to access a file on a specific device or volume you
simply add the volume name, a colon (:) and then the file name.
If the file is inside a directory on that volume you also have
to add the directory name(s) as explained before.

  "volume name":"file name"
  "volume name":"directory name"/"file name"
  "volume name":"directory name"/"sub directory name"/"file name"
  and so on...
```

1.6 Logical Devies

```
LOGICAL DEVICES

Logical devices is a simple way to find files, regardless of
where the file actually is. For example: If you have all your
C programs on a disk called "Programs", placed in directory
named "Examples", and you want to run the program "test1" you
need to write: ("volume name":"directory name"/"file name")

  1.Prog:>
  1.Prog:> Programs:Examples/test1

If you often want to access files in that directory you can
assign a "logical device" to it. You then only need to write
the logical device name and the file name, and AmigaDOS will
automatically look on the right disk and directory.

You assign logical devices by using the command "Assign"
which is called like this: (Note the space between the logical
device name and the path.)

  Assign "logical device name": [device/(directory/subdirectory)]

For example:

  1.Prog:>
  1.Prog:> Assign EX: Programs:Examples

To gain access to the file "test1" you then only need to write:

  1.Prog:>
```

```
  1.Prog:> EX:test1
```

When you boot up the computer it will automatically create some commonly used logical devices. A good example is the logical device "FONTS:". It is automatically assigned to the system disk's "fonts" directory. Here is the list of some of the most commonly used logical devices:

```
  Standard
  Logical
  Devices         Description
  ----------------------------------------------------------
  SYS:            The system disk which was used when the
                  computer started ("the boot disk")
  C:              All Shell (CLI) commands can be found here
  FONTS:          All fonts can be found here
  L:              All types of (file) handlers are stored here
  LIBS:           All disk libraries can be found here
                  [loaded when you call OpenLibrary().]
  S:              All batch files (files with Shell commands)
                  are stored here
  DEVS:           All "devices" (device is a commonly used
                  word...) can be found here.
  and so on...
  ----------------------------------------------------------
```

The logical device "C:" is assigned to the system disk's "c" directory, where all CLI commands are. If you have copied some CLI commands to the RAM disk, and you want AmigaDOS to look there instead of looking on the system disk's c directory you simply reassign the C: device. For example:

```
  1.Prog:>
  1.Prog:> Assign C: RAM:
```

1.7 Physical/Logical Devices & Volumes

PHYSICAL/LOGICAL DEVICES & VOLUMES

To list all physical and logical devices as well as all currently accessable volumes you can use the "Assign" command. Simply type "Assign" without any arguments, and you will see something like this:

```
  1.Prog:>
  1.Prog:> Assign
  Volumes:
  Ram Disk [Mounted]
  HD4 [Mounted]
  HD3 [Mounted]
  HD2 [Mounted]
  DH1 [Mounted]
  HD0 [Mounted]
```

```
  Directories:
  GPFax           HD0:GPFax
  CProg:          HD3:CPrograms
  ACE:            HD4:AmigaCEncyclopedia
  DOC:            HD2:Documents
  BACKUP:         HD4:Backup
  REXX            HD0:RexxPrograms
  CLIPS           Ram Disk:Clipboards
  T               Ram Disk:T
  ENV             Ram Disk:env
  ENVARC          HD0:prefs/env-archive
  SYS             HD0:
  C               HD0:c
  S               HD0:s
  LIBS            HD0:libs
  DEVS            HD0:devs
  FONTS           HD0:Fonts
  L               HD0:l

  Devices:
  PIPE AUX SPEAK RAM CON
  RAW SER PAR PRT DH0
  DF0 DF1 DH1 DH2 DH3
```

At the top you will see all currently accessable volumes, and if they are "mounted" (in a drive) or not. Then all logical devices are listed (called "Directories"), and finally the physical devices are listed.

1.8 Dos Library

DOS LIBRARY

Before you can access any function in a library you normally have to open it with help of the OpenLibrary() function. The AmigaDOS library is however opened automatically for you when your program is started. You do therefore not need to (nor should) open or close the dos library yourself as you do with the other libraries.

There is one thing you must look out for. Although the AmigaDOS library (the dos library as it is usually called) will have been opened for you it is not sure that you may use all the functions described in this manual. Many of the functions were first included with Release 2, and only exist in dos library version 36 or higher. If you try to use a new function on an old Amiga the system will crash!

If you use any of the new functions you must first check that the user really has the needed library version or higher. If the user does not have the needed library version your program must immediately tell the user that he/she has a too old dos library, and then immediately terminate, or at least not use the new functions.

Whenever I describe a function I will always tell you when the function was first included in the library. For example, when you see the sign "V36+" it means that the function may only be used if the user has version 36 or higher. If you see the text "All versions" it means that the function can be used with any version of that library.

Normally when you open a library you can tell the Amiga what is the lowest library version you accept. The question is now how you can check the version number of the dos library if it is already open. The answer is simple. You only have to declare the global dos library pointer as an external pointer, and it will automatically be initialized for you. The library pointer must be called "DOSBase". Here is an example: (Simply include this line at the beginning of your code, and the rest will be done automatically for you.)

```
  extern struct DosLibrary *DOSBase;
```

Once you have a pointer to the dos library you can check the current version number. The version number is found in the "lib_Version" field of the DosLibrary structure. (See header file "dos/dosextens.h" for a complete description of the DosLibrary structure.)

```
  /* We need dos library version 36 or higher: */
  if( DOSBase->dl_lib.lib_Version < 36 )
    printf( "The dos library is too old!\n" );
  else
    printf( "The user has the new functions in Release 2!\n" );
```

Once you are sure that the user has the needed version or higher you can start to use the new functions.

See Example 1 for more information: Read! Run! Edit!

```
  ************************************************************
  *                                                          *
  *  REMEMBER! If you use any function which has been marked *
  *  as a new function (V36, V37, V39, V38...) you MUST      *
  *  check that the user really has the needed library       *
  *  version or higher!                                      *
  *                                                          *
  ************************************************************
```

1.9 BCPL

BCPL

AmigaDOS was not, as everything else on the Amiga, written in C. Instead they used the BCPL programming language which is the predecessor of C. As new dos library versions are released more and more of the old BCPL language is removed. There are however

still some parts left which can not be taken away due to
compatibility problems.

The reason why you need to know about this BCPL is because the
language only used the data type "long word" ("LONG"). The
problem is that all BCPL data must therefore be "long word
aligned" - the data must start on an even word address (on
a four byte boundary).

When you normally declare a structure the memory which is
reserved can start on any byte address. If you would try to use
such a structure together with BCPL there would be a lot of
problems since BCPL would not be able to address it (use it).

See picture LongWordAligned.pic . You can there see an
illustration of a small part of the memory in the Amiga. The
first byte you see has the address 0x7D00 (7D00 HEX = 32000 DEC).
This byte is located on a "long word boundary" since it can be
divided by 4 (4 bytes = 1 long word). Any structure that starts
on such an address can be accessed by BCPL.

Structure 1 starts on byte 0x7D00 and is therefore long word
aligned and can be accessed by BCPL. However, structures 2, 3,
and 4 can not be used by BCPL since they start in the middle of
a long address. Structure 5 is OK since it starts on
address 0x7D04 which can be divided by 4, and so on...

1.10 Long Word Aligned Structures

LONG WORD ALIGNED STRUCTURES

To make sure that a structure really starts on a long word
address (even word address = on four byte boundary) you have to
allocate the memory yourself with help of AllocMem(). (Remember
to deallocate the memory once you do not need it any more!)

Some new special structures with dos library version 36 or
higher should be allocated with the AllocDosObject() function.
These structures will also be long word aligned. When you need
to use AllocDosObject() will be described in the following
chapters.

Instead of writing: ("FileInfoBlock structure is a special dos
structure which will be explained in the following chapters.)

```
/* Declare a FileInfoBlock: (WRONG!) */
struct FileInfoBlock fib;
```

```
You should write: (Thanks BCPL!)

  /* 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 = AllocMem( sizeof( struct FileInfoBlock ),
    MEMF_ANY | MEMF_CLEAR );

  /* Check if we have allocated the memory successfully: */
  if( my_fib_ptr == NULL )
  {
    printf( "Not enough memory!\n" );
    exit( 20 );
  }

  - - -

  /* Use the structure as much as you want... */

  - - -

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


See Example 2 for more information:  Read!    Run!    Edit!
```

1.11 BCPL Pointers (BPTR)

```
BCPL POINTERS (BPTR)

The pointers in the BCPL language are called "BPTR"s. Since
BCPL only uses long addresses the pointers consequently only
points to complete long words. The BCPL address 0x0000 is the
address to the first long word in the memory. The next address
is 0x0001 and it points to the second long word, and so on...
When you increase a BCPL pointer by one you move 32 bits
forward (32 bits = 1 long word). Anything a BPTR points to
must therefore be long word aligned (start on a 32 bit
address).

Normal C pointers are addressing single bytes (8 bits), and
if you increase a C pointer by one you only move 8 bits
foreward.

When you want to convert a BCPL pointer to a C pointer you
consequently multiply the BPTR with four. (The C pointer is
four times larger than the BPTR.)

When you want to convert a C pointer into a BPTR (BCPL pointer)
```

it is a little bit trickier. To get the addresses right you
simply have to divide the C pointer by four, but the pointer
must also be long word aligned as all things BCPL work with.
You must therefore allocate the memory for the pointer with
help of AllocMem().

When you have to convert C to and from BCPL pointers you should
use the special macros that are defined in header file
"dos/dos.h". BADDR() converts a BPTR into a C pointer, and
MKBADDR() converts a C pointer into a BPTR.

Synopsis: c_ptr = BADDR(bptr);

  c_ptr: (APTR) The function returns a C pointer.

  bptr: (BPTR) The BPTR (BCPL pointer) you want to convert
          into a C pointer.

Synopsis: bptr = MKBADDR(c_ptr);

  bptr: (APTR) The function returns a BPTR (BCPL pointer).

  c_ptr: (LONG) The C pointer you want to convert into a BPTR
          (BCPL pointer).

          NOTE! The address in the pointer will be correctly
          converted and can be used by the dos functions.
          However, if you ever have to give AmigaDOS the actual
          BPTR (not the address in the pointer, but the memory
          used to store the address in) it must be long word
          aligned! See example 3 for more information:
           Read!   Run!   Edit!

1.12 BCPL Strings (BSTR)

BCPL STRINGS (BSTR)

To make life even more interesting the BCPL language uses a
special type of strings called "BSTRs". A BSTR is a BCPL
pointer to some data. The very first byte of that data area
contains the length of the string, and the following bytes
contain the actual string. Note that these BCPL strings do not
terminate with a NULL sign, so you have to use the first byte
in the string to get the actual length. A BSTR can therefore
never be longer than 255 characters (0 - 255 = one byte).

Picture: BSTR.pic

This BSTR gives you some problems since you can not use normal
string functions like printf() together with BCPL strings. Most
normal string functions expect a NULL sign at the end of the
string, and since BSTRs do not have that the functions simply
do not know when the string ends.

To help you I have therefore written a small function which

you can include in your programs when needed. The function will copy a BCPL string into a C string which you then can use as normal. The function needs a BCPL pointer to the string that should be converted, a pointer to a normal string where the data should be inserted, and finally the maximum length of the C string.

See Example 4 for more information: Read! Edit!

1.13 Boolean Values Used by AmigaDOS

BOOLEAN VALUES USED BY AMIGADOS

Many of the functions which will be described in the following chapters return boolean values. A boolean value is a value which is either true or false, and false usualy means that the function failed to do what it was supposed to do.

Whenever you want to use a boolean varaible in normal circomstances you should declare it as "BOOL". "BOOL" is defined in header file "exec/types.h", and is actually a "short". A boolean variable should then only be given the values "TRUE" or "FALSE" which are alse defined in the same header file ("exec/types.h") as 1 and 0.

A small demonstration on how to work with boolean values normally:

```
  /* Declare a boolean variable: */
  BOOL my_boolean_variable;

  /* Give the boolean variable a value: */
  /* (Only TRUE or FALSE may be used!)  */
  my_boolean_variable = TRUE;

  /* An example on how to use it: */
  if( my_boolean_variable )
    printf( "True!\n" );
  else
    printf( "False!\n" );
```

Well, this is how you work with boolean values normally. However, as life, nothing is only black and white, true or false. The boolean values used by AmigaDOS are NOT(!) of the type "short", nor is the "TRUE" value defined equally (sight...).

The boolean values used by the AmigaDOS functions are of the type "LONG" ("long"), and instead of using the normal TRUE or FALSE values you should use "DOSTRUE" and "DOSFALSE" which are defined in header file "dos/dos.h" as -1L and 0L. Note the difference between "TRUE" (defined as 1) and "DOSTRUE" (defined as -1L).

```
Whenever you use AmigaDOS functions which return boolean values
you must store the returned value in a "LONG", and be careful
to use the "DOSTRUE" and "DOSFALSE" names and not "TRUE" or
"FALSE"!

  /* Simple boolean variable used by AmigaDOS: */
  LONG ok;

  - - -

  /* Call some AmigaDOS function which */
  /* returns a boolean value:          */
  ok = SomeAmigaDOSFunction( xxx );

  /* INCORRECT! */
  if( ok == TRUE )
    xxx;

  /* Correct! */
  if( ok == DOSTRUE )
    xxx;

  /* INCORRECT (but will work since "FALSE" */
  /* and "DOSFALSE" are bot equal to 0.)    */
  if( ok == FALSE )
    xxx;

  /* Correct! */
  if( ok == DOSFALSE )
    xxx;

  /* Correct! (A simple solution to get */
  /* rid of most problems...)           */
  if( ok )
    xxx;
```

1.14 Examples

```
EXAMPLES

Example 1:  Read!    Run!    Edit!
  This short example examines the Dos library and prints the
  current version and revision number.

Example 2:  Read!    Run!    Edit!
  This example demonstrates how to allocate some memory which
  has to be long word aligned. We will allocate a FileInfoBlock
  structure in this example, but the procedure of allocating
  long word aligned memory is the same for all types of
  objects.

Example 3:  Read!    Run!    Edit!
  For experienced users only! This example demonstrates how
  to create a long word aligned BPTR. The actual BPTR (the
  memory used to store the BPTR address in) is long word
```

```
  aligned. This is rarely needed since you normally only
  work with addresses to data blocks which have to be long
  word aligned. However, if you ever have to give AmigaDOS
  an actual BPTR (not a BPTR address, but the BPTR itself)
  you need to allocate it as described in this example.

Example 4:  Read!    Edit!
  This example contains a useful function which converts hard
  to use BSTR (BCPL stings) into normal easy to use C strings.
  This example is not directly runnable and must instead be
  linked together with some other program.
```