Files

COLLABORATORS

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

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Files **1**

1.1 Chapter 2 - Files . . . 1

1.2 Introduction . . . 1

1.3 Some General Information About Files . . . 2

1.4 Work With Files . . . 3

1.5 Open the File . . . 4

1.6 Read Data . . . 4

1.7 Write Data . . . 5

1.8 Move Inside a File . . . 5

1.9 Close the File . . . 6

1.10 Files and Multitasking . . . 7

1.11 The Lost Update Problem . . . 7

1.12 Solve the Lost Update Problem . . . 7

1.13 When Files Should be Locked . . . 8

1.14 Lock Files . . . 10

1.15 Unlock Files . . . 10

1.16 Open a Locked File . . . 10

1.17 OTHER FILE FUNCTIONS . . . 11

1.18 Examples . . . 11

Chapter 1

Files

1.1 Chapter 2 - Files

```
                Previous Chapter:                                 Next Chapter:
 1. Introduction
---------------------------------------------------------------
```

```
               CHAPTER 2 - FILES
```

Introduction

Some General Information About Files

Work With Files

Files and Multitasking

Other File Functions

Examples

1.2 Introduction

INTRODUCTION

In this chapter I will explain how to work with files. I will describe how you open files, read as well as save data, and finally close the file again. In this chapter I will also explain how to work with files in a multitasking environment.

I will concentrate this chapter to only discuss the most commonly used file functions. In the follwoing chapters you will find more advanced routines like the "buffered IO" as well as other types of useful file function.

1.3 Some General Information About Files

SOME GENERAL INFORMATION ABOUT FILES

If you have not worked with files before you might feel a bit confused when you hear someone mention "file cursors" and "exclusive locks" and so forth... However, do not dispair, I will try to do my best to explain these things.

A "file" is simply a collection of data which have been stored, and can later be retreived (read). Data can be stored on many types of devices, but the most commonly used place to store data is oundouptedly on a disk. Data can also be stored on tapes, hard disks, laser disks, and so on...

A program can create a new file and store data in it, or it can collect data from an already existing file. A program can also append new data or alter data in an existing file.

When you want to use a file you first have to "open" it. When you open a file AmigaDOS prepares the system and the file so they can start to send and receive data.

AmigaDOS is using a "file cursor" which is like a pointer that can be moved around inside the file like the pickup of a record player. When you open a file the file cursor is always positioned at the beginning of the file. Since all data in a file are treated as bytes will the file cursor simply point to the first byte in the file.

See picture ReadWrite.pic . At the top of the picture you can see a long rectangle filled with question marks. This is how an empty disk may be illustrated. Since the disk is empty we do not know what values are stored on it, if there are any, therefore the question marks.

1. When we "open" the file (with help of the Open() function which will be explained later on) the file cursor is positioned at the beginning of the file, here illustrated as a small arrow. You can see that the file cursor points to the first byte in the file.

2. We then write some data to the file (with help of the Write() function which will soon be explained). We send the text "HELLO", and it will be stored in the file. Remember that a file is like an array of bytes, and consequently each letter is stored in a byte. A letter (char) is treated as a byte by C, and you could therefore equally well have sent the numbers 72, 69, 76, 76 and 79 which are the ASCII values of "HELLO".

   When you send data to a file there is sometimes put a a special "End Of File" sign at the end of the file. This "EOF" sing (the constant "EOF" is defined as -1 in header file "stdio.h") is used by some of the standard routines in C, but actually NOT used by AmigaDOS. (AmigaDOS uses instead some other routines to keep track of the file length, but this will not be discussed here.)

However, although AmigaDOS does not use the EOF sign I have included it in the picture since the file routines acts like if there was a sign at the end of the file. The EOF sign simply illustrates the end of the file although it acutally is unknown byte as the following question marks.

As you can see on the picture the text "HELLO" has been stored in the file, one character (or byte if you so like) in each box after each other. You can also see that the file cursor now has moved and is pointing at the end of the file, ready to add more data if necessary.

3. We now sends some data to the file again, and the text " WORLD" is appended to the file. The file cursor is once again moved to the end of the file ready to append even more data.

4. If we now would like to read some of the data we have stored we must first move the file cursor back to the first character (byte) we want to read. (We move the file cursor with help of the Seek() function which will as all the other functions be explained in the following sections. Here we tell AmigaDOS to move the file curser three bytes foreward from the beginning of the file.)

5. We now read some data (with help of the Read() function - not very surprisingly will even this function be explained later on in this text) which we have previously stored. In this example we read seven characters (bytes) from the current position and consequently we get the text "LO WORLD".

6. When you do not want to use the file any more you have to "close" it so other programs can read and or modify it later on. (To close a file you use the Close() function.) The file cursor will then automatically be removed.

1.4 Work With Files

WORK WITH FILES

The procedure of working with files is relative simple. First you have to declare a BCPL pointer (A BPTR) which you have to use whenever you want to do something with the file.

```
  /* A "BCPL" pointer to our file: */
  BPTR my_file;
```

You should then "open" the file which will prepare the system so it later can use the file. To open files you use the "Open()"

function. Once you have opened the file you can start to read data with help of the "Read()" function, write data with help of the "Write()" function, as well as move the file cursor with help of the "Seek()" function. When you do not want to use the file any more you close it by calling the "Close()" function.

Sometimes when you have received an error you can call a special function named "IoErr()" which will give you some more information about the error. This function is, however, first explained in chapter Miscellaneous
bother about it yet.

Open The File

Read Data

Write Data

Move Inside a File

Close the File

1.5 Open the File

OPEN THE FILE

Before you can do anything with a file you have to ask AmigaDOS to "open" it. When you open a file you actualy tell the computer what you would like to do with the file. You might want to open a new file or an already existing one.

When you tell AmigaDOS that you want to open an old file the file cursor will be positioned in the beginning of the old file, pointing to the first byte. You can also open a file as "new", and a new file will be created for you and the file cursor is positioned in the beginning of the new file. If you open an already existing file as a "new" file the existing file will be deleted and a new one created for you.

To open files you should use the Open()

1.6 Read Data

READ DATA

Once you have successfully opened a file you can start to read from it or write to it. AmigaDOS consider files to be a stream of bytes, and every time you read a file you have to specify how many bytes you want to read. To read a single character you should read 1 byte since the size of a character is exactly one byte. To read an integer you have to read 4 bytes, and so on...

Since you most of the time want to read complete structures or arays of data it can be rather difficult to calculate the right number of bytes if you have to do it yourself. Luckily there exist a small function in C called "sizeof()" which simply returns the size in bytes of a specified object:

```
  Synopsis: size = sizeof( object );

      size: The size of the specified object (in bytes).

    object: The object you want to know the size of.
```

Example:

```
  /* Get the size (in bytes) of an object: */
  struct Screen my_screen;
  size = sizeof( struct Screen );
```

If you want to get the size of a string you should use another standard C function called "strlen()":

```
  Synopsis: length = strlen( string );

    length: The number of characters in the string. (Remember
            that one character is equal to one byte.)

    string: Pointer to a NULL terminated string you want to
            examine.
```

Example:

```
  /* Get the length (number of characters) in a string: */
  UBYTE my_string = "The Amiga C Encyclopedia!";
  length = strlen( my_string );
```

To read some bytes in a file you can use the Read() It will collect a specified number of bytes starting from the current position of the file cursor.

1.7 Write Data

WRITE DATA

To write some data (bytes) to a file you can use the Write() function. It will write a specified number of bytes starting from the current position of the file cursor.

1.8 Move Inside a File

MOVE INSIDE A FILE

The Read() functions will start their operations at the byte which the file cursor is currently pointing to. When you open a file the file curser is positioned at the beginning of the file. This may of course not be the position you desire.

If you, for example, want add data to the file you need to move the file cursor to the end of the file. If you on the other hand have just written some data to a file and would like to read it again you have to move the file cursor back to the positon you want to start to read at.

To move the file cursor you should use the Seek()
You tell it which position you want to move to. The position is relative to a specified "offest position" which can either be the beginning of the file, the current position, or the end of the file. See picture OffsetPosition.pic .

1.9 Close the File

CLOSE THE FILE

Once you have finished working with the file and do not want to use it any more you must "close" it. When you close a file will the file cursor be removed, and other programs can now start to use the file.

Note that if you opened the file with a "shared lock" (only "write lock") other programs have been able to read the file but have not been able to modify it. If you opened the file with an "exclusive lock" ("read and write lock") no other programs have been able to read it nor to modify it.

Since other programs are "locked out" when you open a file it is very important that you close the file later on when you do not need it so other programs can use the file. Note that every file you have opened must be closed!

```
  *********************************************************
  *                                                       *
  *  REMEMBER! Every file you have opened MUST be closed  *
  *  when you do not need it any more!                    *
  *                                                       *
  *********************************************************
```

It is very important that you close all files you have opened, and this must of course also be done if your program discovers an error and terminates. Sadly many programmers forget to clean up after themself if their program has to terminate because of an error. Please be careful with this!

To close a file you should use the Close()

1.10 Files and Multitasking

FILES AND MULTITASKING

Since the Amiga can have several programs running at the same time it can happen that several programs work with the same file. This can be very dangerous since two programs may work simultaneously with one file. What one program creats may be destroyed by the other program, and this type of problem is ften called "the lost update problem".

THE LOST UPDATE PROBLEM

SOLVE THE LOST UPDATE PROBLEM

WHEN FILES SHOULD BE LOCKED

LOCK FILES

UNLOCK FILES

OPEN A LOCKED FILE

1.11 The Lost Update Problem

THE LOST UPDATE PROBLEM

Imagine that you have two programs which are used alter some values in a file. Program "A" should multiply the values by 2. When "A" has updated the values program "B" should add 3.

If you look at the picture LostUpdate.pic you will see what can happen if you are very unlucky. Program "A" reads the value 10 and starts to calculate the answer. While "A" is busy working with the number program "B" starts and also reads the value before "A" has updated it! Program "B" will therefore also get the value 10.

Program "A" has now finished its calculations and writes the updated value 20 to the file. The program "B" finish its calculation and stores the value 13. As you can see has the update program "A" made been lost! This is why this type of error is called "the lost update problem".

1.12 Solve the Lost Update Problem

SOLVE THE LOST UPDATE PROBLEM

This type of problems with lost updates can luckily be avoided if the programs are "locking" the files. When a program "locks" a file no other program can use the file until the file has been unlocked. Whenever you are writing programs that may alter data in files other programs also work with you should be very careful to lock the files when necessary.

If the programs as described had locked the file while they were using it there would never have been any problems. See pricture UpdateOK.pic . The first thing program "A" does is this time to lock the file. Program "A" can then read any values and do its calculation with them, and can be absolute sure that no other program can use the file.

Program "B" is maybe trying to read the file, but will not be able to open it, so it has to wait. Once "A" has finished the calculations it writes the new value to the file and then unlocks the file. Prgram "B" can now open the file a and locks it to prevent other programs to use the file. Program "B" does its calculations and then stores the new value in the file, and the file is unlocked.

Since the programs were locking the file while they were using it there were no values lost. The problem with the lost update has been solved.

1.13 When Files Should be Locked

WHEN FILES SHOULD BE LOCKED

On large mainframe computers there exist many different types of locks with different priorities, but on the Amiga there exist only two different types of locks. You can lock a file so other programs may read it but not change it ("shared lock"), or if you do not want any other tasks to even read the file you set an "excluseive lock".

When you open a file with help of the Open() function you will automatically get a lock on the file. If you opened the file as a "new file" (MODE_NEWFILE) you will get an exclusive lock, and no other programs will be able to use the file until you close it.

However, if you have opened an "old" file (MODE_OLDFILE) or used the secial "read/write" option (MODE_READWRITE) the file will get a "shared lock". Other programs may therefore also read the file while you are working with it. (Whenever you save some data in the file it will temporarily be exclusive locked, but that will be converted into a normal shared lock as soon as all data has been witten.)

When you have opened a new file you will not need to lock the file yourself since it is already exclusive locked. If you have, on the other hand, opened an old file or used the read/ write mode you might need to lock the file to prevent other programs to read it while you are working with it.

The only problem is actually when you should lock a file and when not. A wordprocessors that works with a document usually never locks the actual dockument. Normally the wordprocessor even closes the file after it has read it, and opens it only temporarily while it saves the document. The file will therefore be completely "unlocked" while the wordprocessors is running!

If this is good or not can be discussed:

(+) The advantage is that the file can be deleted or altered by other programs while the wordprocessors is still running. The user is free to do what he or she wants with the file while the wordprocessors is running.

(-) The disadvantage is that the user can do whatever he/she likes with the file (some thinks this is an advantage as described above).

For example: A user is editing a document and while the wordprocessor is running he/she tries to start a program that alters the file (the user is maybe using an extrnal spell checker, or some type of "reformatting" program) he/she will be allowed to do that. The problem comes if the user then (after he/she has spellchecked and reformatted the file) saves the document in the wordprocessors. All alterations which were done to the file will then be lost as described before!

Personally I believe that most programs should keep at least a shared lock on the file while the program is using it. The external spell checker as described would in this case still be able to read the file (some freedom for the user), but when the spelchecker tries to save the data it will fail and the user is alerted and forced to create a new file for the spell checked data.

One possible solution would be that the wordprocessors alerts the user whenever a file is going to be over written. In this case the user would also be warned that data might be lost. However, one problem remains. What would happen if the user deletes the file (a copy of it is in the wordprocessor the user thinks so it does not matter...) and then suddenly there is power cut?

There exist a simple "rule" of when you should use an exclusive lock on a file and when you can use a shared lock:

When files are altered automatically (without any action from the user) you should keep an exclusive lock on the file if possible. Automatic functions can very easily create unexpected situations and updated data might be lost (as described in our examle with the two programs "A" and "B").

When files are altered because of some action from the user it might be enough with a shared lock (if necesary). This depends of course on how much you trust the user (another "programming rule" is that you should never trust the user, and in such case you should actually use an exclusive lock).

1.14 Lock Files

LOCK FILES

When you want to put a lock on a file you should use the Lock() only need to use this function when you want to add an exclusive lock to a file. At least a shared lock will have automatically been added for you when you opened the file (if you open a new file you will even get an exclusive lock).

This Lock() function is, however, also needed when you are working with some special file functions which will be descrbed in the following chapters.

1.15 Unlock Files

UNLOCK FILES

Whenever you do not need a lock on a file you must unlock it.

```
  ***********************************************************
  *                                                         *
  *  REMEMBER! Every file you have locked MUST be unlocked  *
  *  when you do not need the lock any more!                *
  *                                                         *
  ***********************************************************
```

To unlock a file simply call the UnLock() it the lock that should be unlocked.

1.16 Open a Locked File

OPEN A LOCKED FILE

If you have open a file and want to put an exclusive lock on it the Lock() because the file is currently used by someone (in this case yourself since you have opened the file), and you can therefore not put an exclusive lock on the file. The problem is that you can not do the opposit eiter, you can not lock a file exclusively and then try open it since the Open() function will then fail.

Now it seems like it would be impossible to use an old file and still be able to put an exclusive lock on it. (New files will automaticaly get an exclusive lock.) The fact is that prior to

Release 2 it was impossible (unless you did some low level work). However, with Release 2 a new open function,
OpenFromLock()
36 or higher you can lock the file (exclusive or shared, although shared is unnecessary since that is done automatically when you open the file) and then use this new function to open the file with help of the lock.

Please remember to check that the user really have dos library V36 or higher before you try to call the OpenFromLock() function!

1.17 OTHER FILE FUNCTIONS

Other File Functions

Up to now we have discussed the elementary parts on how to work with files. We have looked at how to open, work with and close files, as well as how to use any necessary locks. This is enough for small applications, but you might need to do more things with the files. There exist a lot of useful support functions which will be described in the next chapter
File Functions

1.18 Examples

EXAMPLES

Example 1: Read!   Run!   Edit!

This program collects ten integer values from the user, and saves them in a file called "HighScore.dat" on the RAM disk.

Example 2: Read!   Run!   Edit!

This program will reads ten integer values from an already existing file called "HighScore.dat" which is located on the RAM disk. (This file was created by Example1.)

Example 3: Read!   Run!   Edit!

This program simply writes two strings to a file, moves the file cursor back some characters and then collects some characters in the middle of the file. This example does exactly what is explained in picture ReadWrite.pic .

Example 4: Read!   Run!   Edit!

This program will open an already existing file and update the values in it (we simply add 50 to each value). Since we do not want any other program to destroy our updated values we will lock the file exclusively while we are using it.

Since we want to put an exclusive lock on an already existing file we have to use the new "OpenFromLock()" function to open the file once we have successfully locked it. This example needs dos library V36 or higher.

Example 5: Read! Run! Edit!

This example demonstrates how you can write a (very) simple data base program. In this data base you can add names of persons and their telephone numbers. Whenever you want you can display the complete user list.

This example uses a "Console" window which has not been explained yet. It is therefore a bit difficult, and if you are unfamiliar with AmigaDOS you should skip this example for the moment and look at it later.