Handlers

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

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Handlers

1.1 Chapter 6 - Handlers

```
                Previous Chapter:                                    Next Chapter:
 5. Parsing the Command Line
-------------------------------------------------------------
```

```
               CHAPTER 6 - HANDLERS


                 Introduction

                 Available Handlers

                 Console Windows                - CON: & RAW:

                 Printer                        - PRT:

                 Use the Parallel Port          - PAR:

                 Use the Serial Port            - SER:

                 The AUX: Handler               - AUX:

                 Use the Speech Synthesizer     - SPEAK:

                 Transfer Data Between Programs - PIPE:

                 Output to NIL:                 - NIL:

                 Examples
```

1.2 Introduction

```
INTRODUCTION

AmigaDOS can use a lot of different devices or handlers as they
```

are also called. The most commonly used handler is the "file handler" which we have worked with in the previous chapters. The other handlers will be described in this chapter, but as you will soon notice, these other handlers are controlled in much the same manner.

The other handlers give you access to the printer, parallel and serial port, console windows, the speech synthesizer, and will even let you send data between two or more programs.

1.3 Available Handlers

AVAILABLE HANDLERS

Here is a list of all available AmigaDOS handlers: (See picture Handlers.pic as an illustration)

1. File handler. Read and write files, unbuffered as well as buffered. Picture: FileHandlers.pic
   Devices: (4 disk drives) "DF0:", "DF1:", "DF2:", "DF3:"
   (Hard disks) "DHx:" (x = 0, 1, 2, 3...)
   (Other file devices) "RAM:"...

2. Buffered and translated console handler (input/output in a "Console window"). Translation means that you can included special commands in the data which will be translated and executed by the device.
   Device: "CON:"

3. Unbuffered and untranslated (raw) console handler (input/output in a "Console window").
   Device: "RAW:"

4. Printer handler. With help of Preferences translated and buffered printer output to either the parallel or serial port depending on the preference settings.
   Device: "PRT:"

5. Parallel handler. Buffered parallel input/output.
   Device: "PAR:"

6. Serial handler. Buffered serial input/output.
   Device: "SER:"

7. Unbuffered and untranslated serial input/output.
   Device: "AUX:"

8. Translated output to the Amiga's speech synthesizer.
   Device: "SPEAK:"

9. Buffered data input/output transferring with other programs.
   Device: "PIPE:"

10. Output to nothing.

Device: "NIL:"

1.4 Console Windows - CON: & RAW:

CONSOLE WINDOWS - CON: & RAW:

One very useful thing is the Amiga's "Console windows". It is a window in which your program can communicate with the user. You can send text to the window and collect input from the user. The communication can be both buffered and unbuffered.

If the text in a console window does not fit it will be scrolled, and if the user then makes the window larger the previously hidden text parts will be displayed again. Console windows allows the user to edit the text line with help of the cursor keys, and the console window also stores the text lines the has typed so they can later be reused.

If the clipboard function is turned on (the "ConClip" program has been started, automatically done in the startup-sequence) the console windows even use the Clipboard to copy and paste data from and to other applications. (This clipboard support was introdused in Release 2).

There exist two types of console windows. First we have the normal "CON:" device which uses a buffer and translates some of the characters that are sent into special commands that will for example turn on bold, italic, change pen colour, flash the screen etc... The other type of console windows is the "RAW:" device which does not use any buffer nor will the characters be translated.

1.5 Open Console Windows

OPEN CONSOLE WINDOWS

To open a console window you simply use the Open() as you do with files. But instead of giving the Open() function a file name you give it a "console description" which tells the AmigaDOS how you want the window to be opned, what size, position etc.. The syntax for the console window is:

"device:x/y/width/height/title/option(s)"

```
  device:  Can either be "CON:" (buffered and translated) or
           "RAW:" (unbuffered and untranslated).

  x:       X position of the window.

  y:       Y position of the window.

  width:   The width of the window.

  height:  The hight of the window.

  title:   A window title (may contain spaces).

  options: You can set zero or more of the below sepcified
           flags. Note that you must put a slash (/=) betweem
           each flag.

  Flag       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]"
```

If you could successfully open the console window the Open() function will return a BCPL pointer to your console window. This pointer should be used as you normally do with simple file pointers. Actualy there is no difference between a file and a console window pointer. This is why you can use the normal file functions on console windows and any other type of handler as well.

Here is an exmple on how to open a console window:

```
  /* A "BCPL" pointer to our console window: */
  BPTR my_console;

  - - -

  /* Open a console window: */
  my_console =
   Open( "CON:10/50/320/100/My Console!/CLOSE/WAIT", MODE_NEWFILE );
```

```
  /* Could we open the console window? */
  if( my_console == NULL )
  {
    /* Inform the user: */
    printf( "Error! Could not open the console window!\n" );

    /* Exit with an error code: */
    exit( 20 );
  }
```

1.6 Output to Console Windows

```
OUTPUT TO CONSOLE WINDOWS

Once you have successfully opened a console window window you
can start to write (send, output) and read (collect, input)
data with help of the normal  Read()  functions.

If you have used the "CON:" device you can include "ANSI
Console Control Sequences" which will be translated into
different types of commands like "turn on bold", "set pen
colour to 4", "flash the screen", etc.. See Appendix
 ANSI Console Control Sequences  for a complete list of all
available console commands.

Here is an example on how to send text to a console window:
(Output)
```

```
  /* Store the number of characters (bytes) */
  /* actualy written here:                  */
  int characters_written;

  /* The text we want to print. Note the new line */
  /* character "\n" and the tab character "\t".   */
  /* The reult will be:                           */
  /*                                              */
  /* Dear User,                                   */
  /*                                              */
  /* Bla Bla...                                   */
  /*                                              */
  /*         Yours Sincerely,                     */
  /*         -AB-                                 */

  UBYTE *my_text =
    "Dear User,\n\nBla Bla...\n\n\tYous Sincerely,\n\t-AB-";

  - - -

  /* Write the text: */
  characters_written =
    Write( file, my_text, strlen( my_text ) );

  /* Could we write all the text? */
  if( characters_written != strlen( text ) )
```

```
    printf( "Error! Could not write all text!\n" );
```

1.7 Input from Console Windows

```
INPUT FROM CONSOLE WINDOWS

When you use the  Read()
the user hits the Return or Enter key. Note that the number of
characters (bytes) collected depends on how many the user
typed before he/she pressed the Enter or Return key.

Note that there will not be any NULL sign at the end of the
collected string! There will however be an "Enter" sign (since
the used has to hit Enter or Return to continue). If you want
to use the collected data you must therefore add a NULL sign,
and possible also remove the Enter sign (not necessary, but if
you print the string the Enter sign will also be printed and
consequently move down the cursor one line, and this is
probably not wanted).

  If the user entered four characters and then pressed Return
  or Enter, 5 characters will be collected.

  If the user did not enter any characterrs at all and simply
  pressed Return or Enter on an empty line, 1 character will be
  collected.

  If the user did not enter anything but instead closed the
  window, 0 characters will be collected. (If the user entered
  some characters before he/she pressed enter those characters
  will be collected, but then there will of course not be any
  enter sign at the end of the string.)

Here is an example on how to collect text from a console
window: (Input)

  /* Size of the buffer: */
  #define MAX_LENGTH 256

  /* Store the number of characters (bytes) */
  /* actualy read here:                     */
  int characters_read;

  /* Declare a small buffer in which the collected */
  /* data can be stored.                           */
  UBYTE my_buffer[ MAX_LENGTH ];

  - - -

  /* Collect some text: */
  characters_read = Read( file, text, MAX_LENGTH );

  /* In the buffer there will now probably be some text, and   */
  /* at the end of the text there will be an "Enter" sign. The */
  /* Read() function will first return when the user hits the  */
```

```
  /* enter key (or closes the window). What comes after the    */
  /* enter sign is rubbish and should not be used.             */
  /*                                                           */
  /* To be able to print the text we need to put a NULL sign   */
  /* at the end of the string. (All strings must end with a    */
  /* NULL sign, otherwise the functions would not know when    */
  /* the string ends.) To also get rid of the Enter sign we    */
  /* simply substitute it with a NULL sign:                    */

  /* Put the NULL ('\0') sign at the end of the string: */
  if( characters_read > 0 )
  {
    /* Substitute the Enter sign with a NULL sign: */
    text[ characters_read - 1 ] = NULL;
  }
  else
  {
    /* Nothing was entered, not even an Enter sign! */
    /* The console window has been closed!          */
    printf( "The console window was closed!\n" );

    /* Clear the string: (Set a NULL sign in the begining) */
    text[ characters_read ] = NULL;
  }
```

1.8 Close Console Windows

```
CLOSE CONSOLE WINDOWS

To close the console window simply call the  Close()
The window will usually close immediately, but if you have set
the flag "\WAIT" the window will first close when the user hits
"Ctrl-\" (holds down the "Ctrl" key and the "\" key). If the
window has a close gadget the user can also use that one.

The advantage with the "\WAIT" flag is that your program can
print some information in a console window and then close it
and terminate, but the window will still be left on the screen
until the user has read your message and closes the window.

If the user tries to close the window before you have closed
it all remaining  Read()  requests will immediately
return, and no characters written or collected.

Here is an example on how to close a console window: (extremely
tricky...)

  /* Close the console window: */
  Close( my_console );
```

1.9 Printer - PRT:

PRINTER - PRT:

You can even use AmigaDOS to print text with a printer. You simply open the "PRT:" handler as any other handler, and everything you then write will be printed. Since the "PRT:" handler uses the settings in Preferences the output will automatically directed to the right port which the printer is connected to (the parallel or serial port), and the characters and commands are translated to fit the in Preferences specified printer.

You can of course only write data to the printer (only output).

Since Preferences is used to translate all data which are going to be printed you can use the commands listed in appendix "Printer Commands" to turn on bold, italics, NLQ, etc...

Here is an example on how to use the "PRT:" handler:

```
  /* The text we want to print: */
  UBYTE *my_text = "This will be printed!\f";

  /* A "BCPL" pointer to our Printer handler: */
  BPTR my_printer;

  /* Store here the number of characters actually printed: */
  long characters_printed;

  - - -

  /* Open the Printer handler ("PRT:"): */
  my_printer = Open( "PRT:", MODE_NEWFILE );

  /* Have we opened the Printer handler successfully? */
  if( !my_printer )
  {
    /* Inform the user: */
    printf( "Error! Could not open the Printer handler!\n" );

    /* Exit with an error code: */
    exit( 20 );
  }

  - - -

  /* Print the text: */
  characters_printed =
    Write( my_printer, my_text, strlen( my_text ) );

  /* Could we print all characters? */
  if( characters_printed != strlen( my_text[ loop ] ) )
    printf( "Error! Could not print all text!\n" );

  - - -
```

```
  /* Close the Printer handler: */
  Close( my_printer );
```

1.10 Use the Parallel Port - PAR:

```
USE THE PARALLEL PORT - PAR:

To use the parallel port is also very simple. You open it as
normal, with the name "PAR:", and everything you read or write
will be done on the parallel port. Note that no translation is
done (only raw unmodified values are sent/received). The
parallel port can be used for both input,  Read()
 Write()

Note! The PAR: handler should NOT be used to print things! Use
the PRT: handler instead!
```

1.11 Use the Serial Port - SER:

```
USE THE SERIAL PORT - SER:

The serail port is as equal simple as the parallel port. You
open it as normal, with the name "SER:", and everything you
read or write will be done on the serial port. Note that no
translation is done (only raw unmodified values are sent/
received). The serial port can be used for both input,  Read()
and output,  Write()

Note! The SER: handler should NOT be used to print things! Use
the PRT: handler instead!
```

1.12 The AUX: Handler - AUX:

```
THE AUX: HANDLER - AUX:

The AUX: handler is an unbuffered serial input/output handler
which will not be described here since it is never used by most
of us. (I have never used it myself so I prefer not to explain
it.)
```

1.13 Use the Speech Synthesizer - SPEAK:

```
USE THE SPEECH SYNTHESIZER - SPEAK:

The Amiga's speech synthesizer is a device which is sadly
rarely used, but can really be useful. With Release 2 it will
now even sound much better, so use it if you can. In fact, I
```

think most programs should have an option to turn on the
speech synthesizer so it will be used to read out loud
every single message that is printed. You should of course
never use the speech synthesizer only since many users can not
hear very good, but as an extra medium of communication it is
excellent!

To use the speech synthesizer handler is equally simple as all
other handlers. You only have to open the "SPEAK:" device, and
everything you sent do it with help of Write()
automatically be translated and read out loud. You can of course
only send text to the device and not read (output only).

When you open the speaker handler you can change some of the
default values in order to alter the sound: (The slash "/"
must be included before every option.)

```
  Syntax: SPEAK:OPT opt1/opt2/opt3...

  Option  Description
  ------------------------------------------------
  /n      Speak with a natural voice
  /m      Speak with a man voice
  /f      Speak with a "female" voice
  /r      Speak like a robot with a monototonous voice
  /sXXX   Speed, 40 - 400
  /pXXX   Pitch, 65 - 320
```

Here is an example on how to let the Amiga read some text:

```
  /* Here is the text we want the Amiga to read: */
  UBYTE *my_text = "Only the Amiga makes it possible!";

  /* A "BCPL" pointer to our Speaker handler: */
  BPTR my_translator;

  /* Store here the number of characters actually spoken: */
  long characters_spoken;

  - - -

  /* Open the Speaker handler, use "female" voice and fast: */
  my_translator =
   Open( "SPEAK:OPT/f/s250", MODE_NEWFILE );

  /* Have we opened the Speaker handler successfully? */
  if( my_translator == NULL )
  {
    /* Problems, inform the user: */
    printf( "Error! Could not open the Speaker handler!\n" );

    /* Exit with an error code: */
    exit( 20 );
  }

  - - -
```

```
  /* "And I could hear her say..." */
  characters_spoken =
    Write( my_translator, my_text, strlen( my_text ) );

  /* Was the whole line "read": */
  if( characters_spoken != strlen( my_text ) )
    printf( "Error! Could not \"read\" all text!\n" );

  - - -

  /* Close the Speaker handler: */
  Close( my_translator );
```

1.14 Transfer Data Between Programs - PIPE:

```
TRANSFER DATA BETWEEN PROGRAMS - PIPE:

Since there may be several progreams running at the same time
you sometimes may want to transfer a lot of data between to
programs. You could of course use a temporary file in which
the "sender" saves all data so the "receiver" later can read
it.

This is, however, not a very good solution since an unnecessary
file must be created, and if you want to transfer a lot of data
it would have to be very big. Secondly it is rather unefficient
since it takes some time to pen a file, save all data, close it
again so the other program can open it, read the data and then
finally close it again.

The programs should insted use is the special "PIPE:" handler
which allows programs to send data to each other at very high
speed. (See picture  PipeHandler.pic )

When you open the "PIPE:" handler you can give it a "piper
name" (the name of the temporary buffer where all data will be
temporarily stored) although it is not necessary. Since there
may be several programs using the piper handler I recommend
you to use some sort of piper name.

  Syntax: PIPE:[piper name]

The first program that opens the piper handler should open it
as "MODE_NEWFILE" and the other as "MODE_OLDFILE". However, you
can equaly well first open it as an old file and then as a new
file, but that might look a bit strange. The only really
important thing is that one program opens it as a new file and
the other as an old (both programs may not open it as new, or
both as old).

Once a program has opened a piper handler it can start to send
or collect data. (The other program does not have to open the
piper handler before the first program can use it.)

You send data with help of the  Write()
```

should note that the piper handler is using a small buffer of
around 4 kB, and once the buffer has been filled any further
write requests will be halted until someone empties the buffer.
(Your program will simply be put to sleep until someone
collects the data in the buffer and your read request can be
satisfied.)

You collect data with help of the Read()
However, if there is no data in the temporary buffer the read
request will be halted until all characters as requested has
been collected or the other program closes the handler:

  If program A sends 10 characters but you tried to read 15
  characters your program will be put to sleep until 5 more
  characters have been sent by A, or the piper handler was
  closed by A.

See programs "Example5A.c" and "Example5B.c" more information:

  Example 5A: Read!   Edit!

  Example 5B: Read!   Edit!

               Run Example 5A and 5B!

1.15 Output to NIL:

OUTPUT TO NIL:

The "NIL:" handler is often used when you write scripts for the
Shell or CLI and want all output to vanish. However, in a
program you hardly use it at all.

Everything you send to to NIL: will simply disappear. It is
used like all other handlers, but can of course only be used
for output.

1.16 Examples

EXAMPLES

Example 1: Read!   Run!   Edit!
  This is an example on how to open a console window on the
  default screen (usually the Workbench Screen). The window
  will first be closed when the user clicks on the close
  gadget or types CTRL-\.

  These types of console windows can be very handy when you
  want to give the user some information. Console windows
  are easy to use and look good.

  This program uses some flags which are only available in

```
  dos library version 36 or higher. The program can however
  still run on the older systems. The console window will
  then however be immediately closed when we close the
  file.

Example 2:  Read!    Run!    Edit!
  Here is a second example on how to use a console window.
  This time we will both write text to the window and and
  collect input from the user. As you can see, console
  windows can be very useful when you want to display text
  and collect input from the user.

  Since the console window uses a small buffer to store
  text lines the user has previously typed and since the
  console supports several useful editing commands it is
  very easy for the user to work with console windows.
  With Release 2 and above the user can even copy and
  paste data from and to the Clipboard device. Very handy!


Example 3:  Read!    Run!    Edit!
  This is an example on how to open a Printer handler and
  send some text to the printer. Since we use the "PRT:"
  handler the data will be translated accordingly to the
  settings in Preferences. Printing text is actually very
  simple when you can use the Printer handler!

Example 4:  Read!    Run!    Edit!
  This example demonstrates how to use the Speaker ("SPEAK:")
  handler. We will simply open the Speaker handler and set
  the speed to be very slow. Then we send some text to the
  handler and it will automatically be translated and read
  out loud for us.

  Yes, it was very late when I wrote this example...

Example 5A:  Read!    Edit!
  This example demonstrates how you can use the Pipe handler
  to copy (pipe) data to another program. This is the first
  part of the example, program A. To see how the Pipe handler
  work you have to start both program A and B. Once both are
  running can you enter some text in this (progra A's) console
  window. When you press enter the console will be closed and
  the text you have entered is piped to the other program.
  Program B will receive the text and prints it in it's own
  console window.

  If you want to transfer several lines of data you have to
  remember that the Pipe handler uses a small buffer (around
  4000 bytes, 4kB), and you will only be able to collect data
  when the buffer has been filled, or when the file is closed.
  In in this example I close the file once the data has been
  sent so it will immediately be available for program B.

Example 5B:  Read!    Edit!
  This example demonstrates how you can use the Pipe handler
  to copy (pipe) data to another program. This is the second
```

```
part of the example, program B. See example 5A for more
information.

 Run Example 5A and 5B!
```