# An Introduction to DOS

## Contents

## 1. Introduction
With the advent of graphical user interfaces (GUIs) like Microsoft Windows, the idea of actually using a command line prompt and typing in commands at the keyboard strikes many people as quite old-fashioned. Why go to the trouble to memorise commands and their syntax when you can just select them from menus, fill in dialog boxes with parameters, or drag files onto application icons? In fact, there are many things that are easier to do from a command prompt than in a GUI; how easy is it in a GUI to rename all files with a ".cxx" extension so that they have a ".cpp" extension instead?

Anyway, let's begin at the beginning. **DOS** stands for "**D**isk **O**perating **S**ystem", and is a generic name for the basic IBM PC operating system. Several variants of DOS are available, including **Microsoft**'s version of DOS (**MS-DOS**), **IBM**'s version (**PC-DOS**), and several others. There's even a free version of DOS called OpenDOS.

There are actually several levels to DOS. At the lowest level is the **BIOS** (**B**asic **I**nput/**O**utput **S**ystem) which is responsible for managing devices like the keyboards and disk drives at the simplest possible level (e.g. the BIOS lets you say things like "get me sector 5 of track 3 from disk drive 1"). This is done partly by software in **ROM** (**r**ead-**o**nly **m**emory) and partly by BIOS extensions which are loaded when the system first starts up (with MS-DOS, these are in a file called IO.SYS; on PC-DOS they're in IBMBIO.COM).

The second layer provides a set of higher level services implemented using the low-level BIOS services; you can now refer to disk drive 1 as "drive A:" and instead of referring to specific sectors or tracks you can refer to files by name (e.g. `LETTER.TXT`). You can also treat devices as if they were named files, so that for example you can use the name PRN: to refer to the printer. In other words, this level provides a file system (you can refer to files by name and let DOS worry about translating the name into a physical location) as well as some device independence (you don't have to differentiate storing text in a file from sending it to the printer). This layer of the system is implemented by another file which is loaded when the system first starts up (the file is called MSDOS.SYS on MS-DOS systems, and IBMDOS.COM on PC-DOS systems).

The third layer is the command interpreter (or shell), which is what most people think of as DOS (it's not, but it's what you interact with when you use the computer, so it's an understandable confusion). This is contained in another file called COMMAND.COM, which is just an ordinary program that is started automatically (and you can replace it with something better, for example 4DOS, a shareware shell included on this CD). The shell's job is to display a command prompt on the screen to let you know you're supposed to type something, then to read a line of text that you type, and to interpret it as a command, which usually involves loading another program into memory and running it. When it's finished doing this, it displays another prompt and waits for you to type in another command.

## 2. The file system
Before we go any further, it would be a good idea to look at the DOS file system. The file system lets us store information in named files. You can call a file anything you like which might help you remember what it contains as long as you follow certain basic rules:

**1. File names** can be up to 8 **characters long**. You can use letters and digits but only a few punctuation marks (! $ % # ~ @ - ( ) _ { }). You can't exceed 8 characters or use spaces or characters like * or ? or +. Names are case-insensitive, i.e. it doesn't matter whether you use capitals or lowercase letters; "`A`" and "`a`" are treated as the same thing.

**2. File names** can also have an **extension** of up to three characters which describes the type of file. There are some standard extensions, but you don't have to use them. Examples include `COM` and `EXE` for executable programs, `TXT` for text files, `BAK` for backup copies of files, or `CPP` for C++ program files. The extension is separated by a dot from the rest of the filename.

For example, a file called `FILENAME.EXT` has an 8-character name (`FILENAME`) followed by a three-character extension (`.EXT`). You could also refer to it as filename.txt since case doesn't matter, but I'm going to use names in capitals for emphasis throughout this document.

Files are stored in directories; a directory is actually just a special type of file which holds a list of the files within it. Since a directory is a file, you can have directories within directories. Directory names also follow the same naming rules as other files, but although they can have an extension they aren't normally given one (just an 8-character name).

The system keeps track of your current directory, and if you just refer to a file using a name like `FILENAME.EXT` it's assumed you mean a file of that name in the current directory. You can specify a pathname to identify a file which includes the directory name as well; the directory is separated from the rest of the name by a backslash ("`\`"). For example, a file called `LETTER1.TXT` in a directory called `LETTERS` can be referred to as `LETTERS\LETTER1.TXT` (assuming that the current directory contains the `LETTERS` directory as a subdirectory). If `LETTERS` contains a subdirectory called `PERSONAL`, which in turn contains a file called `DEARJOHN.TXT`, you would refer to this file as `LETTERS\PERSONAL\DEARJOHN.TXT` (i.e. look in the `LETTERS` directory for `PERSONAL\DEARJOHN.TXT`, which in turn involves looking in the `PERSONAL` subdirectory for the file `DEARJOHN.TXT`).

Every disk has a root directory which is the main directory that everything else is part of. The **root directory** is called "`\`", so you can use absolute pathnames which don't depend on what your current directory is. A name like `\LETTERS\LETTER1.TXT` always refers to the same file regardless of which directory you happen to be working in at the time; the "`\`" at the beginning means "start looking in the root directory", so `\LETTERS\LETTER1.TXT` means "look in the root directory of the disk for a subdirectory called `LETTERS`, then look in this subdirectory for a file called `LETTER1.TXT`". Leaving out the "`\`" at the beginning makes this a relative pathname whose meaning is relative to the current directory at the time.

If you want to refer to a file on another disk, you can put a letter identifying the disk at the beginning of the name separated from the rest of the name by a colon ("`:`"). For example, `A:\LETTER1.TXT` refers to a file called `LETTER1.TXT` in the root directory of drive A. DOS keeps track of the current directory on each disk separately, so a relative pathname like `A:LETTER1.TXT` refers to a file called `LETTER1.TXT` in the currently-selected directory on drive `A`.

For convenience, all directories (except root directories) contain two special names: "x" refers to the **directory itself**, and "`..`" refers to the **parent directory** (i.e. the directory that contains this one). For example, if the current directory is `\LETTERS\PERSONAL`, the name ".." refers to the directory `\LETTERS`, "`..\BUSINESS`" refers to `\LETTERS\BUSINESS`, and "`..\..`" refers to the root directory "\".

## 3. Simple commands

If you start up a computer running DOS (or select the "MS-DOS Prompt" icon in Windows), after a bit of initialisation you will end up with a prompt to let you knwo that the command interpreter is waiting for you to type in a command. The prompt might look like this:

```
    C>
```

This prompt consists of the name of the current disk that you're using (`A` for the **main floppy disk drive**, B for the secondary floppy disk, C for the primary hard disk drive, and so on) followed by ">". You can get a list of the files in the current directory on that disk by typing the command **`DIR`** (short for "directory"):

```
    C>DIR
```

(The text in bold above indicates what you type in.) When you press the **ENTER** key, a list of files will be displayed on the screen, together with their sizes and the dates when they were last modified. If you want to see a list of the files on disk in the main floppy drive (drive A), you can do it by first selecting A as the current drive:

```
    C>A:
```

Just type the letter of the drive you want to select followed by a colon, and then press ENTER. The prompt will then appear like this:

```
    A>
```

and you can then type `DIR` as before:

```
A>DIR
```

Alternatively, you can specify the directory you want to list as a parameter separated by one or more spaces from the command name:

```
C>DIR A:
```

which lists the contents of the current directory on drive A, or

```
C>DIR A:\
```

which lists the root directory of drive **`A`**, or

```
C>DIR ..
```

which lists the directory above the one you're currently in. Note that for convenience you can arrange matters so the prompt tells you the current directory as well as the currently-selected disk drive, so that it might appear like this:

```
C:\WINDOWS>
```

meaning that `C:\WINDOWS` is the current directory. Now imagine that you type "`DIR` ..":

```
C:\WINDOWS>DIR ..
```

You should be able to see that this will list the contents of `C:\`, i.e. the root directory on drive `C`. (From now on I won't show the prompt in my examples, only what you need to type in.) The trouble with a command like `DIR` is that the list of files scrolls off the screen if there are more than a few files in the directory. Most commands have a set of options that you can use to modify their behaviour, which are specified after the command name and separated from it by a slash "`/`"; the `DIR` command has a `/P` (pause) option which makes it pause after each full screen of output, and waits for you to press a key before displaying the next screenful. Most commands support a "**help**" option called "`/?`"; typing `DIR/?` will give you a brief list of the options that `DIR` recognises, for instance.

## 4. Some common commands

Here's a list of some common commands and what they do:

```
X:                        -- select X: as the current drive
DIR                       -- list the current directory
DIR directory             -- list the specified directory
CLS                       -- clear the screen
CD directory              -- change current directory to directory
TYPE file                 -- display specified file on the screen
COPY file1 file2          -- copy the first file to the second
COPY file directory       -- copy the file to the specified directory
DEL file                  -- delete the specified file
EDIT file                 -- edit the specified file
REN file newname          -- rename file to newname
FORMAT drive              -- format the disk in the specified drive
MD directory              -- make a new directory called directory
RD directory              -- remove the specified directory
                             (which must be empty)
```

In the list above, file (or **`file1`** or **`file2`**) is the pathname of a file, directory is the pathname of a directory and drive is the name of a disk drive (e.g. **`A:`** or **`C:`**). In the case of the **`REN`** (rename) command, newname is the new file name (not a path name because this just renames the file, it can't move the file to a new directory).

## 5. Wildcards

In some situations it's useful to be able to specify sets of files to be operated on by a single command. For example, you might want to delete all files with the extension "`.TXT`". You can do this like so:

```
DEL *.TXT
```

The star "`*`" matches any name at all. You can read it as "`anything .TXT`", although if I had to read this command out loud I'd probably say "`del star-dot-text`". Anyway, what it means is that any file whose name matches the pattern **`*`**`.TXT` (any file with a `.TXT` extension) will be deleted. You can do more sophisticated things:

```
COPY NOTE*.TXT NOTES
```

will copy any file whose name begins with **NOTE** (i.e. `NOTE` followed by anything) and has a `.TXT` extension to the `NOTES` subdirectory. The star is referred to as a wildcard by analogy with the role of a Joker in some card games, where the Joker can be used as any other card. The command

```
DEL *.*
```

will delete all the files in the current directory (all files with any name and any extension). Since this is obviously risky, you'll be prompted with a message that says "Are you sure (Y/N)?". If you are sure, type Y for "yes"; if not type N for "no". A non-obvious use of this technique is allowed by the **REN** (rename) command:

```
REN *.TXT *.LST
```

will rename all files with a `.TXT` extension to files with the same name but a `.LST` extension instead. A limitation of this technique is that you can only use a star as a wildcard at the end of the name or extension part of a filename; you might think that you could delete all files whose name ends with `X` like this:

```
DEL *X.*
```

Unfortunately this has the same effect as

```
DEL *.*
```

since the star matches everything up to the end of the name, and the X after that is just ignored. You can use a question mark to match a single character:

```
DEL ?X.*
```

This will match any file whose name consists of any single character followed by an `X` with any extension. To delete any files whose name ends with `X` would actually require a whole sequence of commands:

```
DEL X.*
DEL ?X.*
DEL ??X.*
DEL ???X.*
DEL ????X.*
DEL ?????X.*
DEL ??????X.*
DEL ???????X.*
```

This deletes all files whose name is `X`, then all files whose name is any one character followed by `X`, then all files whose name is any two characters followed by `X`, and so on up to any seven characters followed by `X`.

## 6. How DOS identifies commands

When you type in the name of a command, the command interpreter has to decide which program you're referring to. Some commands like `DEL`, `DIR` and `COPY` are easy to identify because they're internal commands which are built into the command interpreter itself. However, most commands require another program to be loaded, and these are known as external commands. The way that the command interpreter searches for commands is as follows:

❑ Is the command an internal command? If so, just do it.

❑ Is there a file of the correct name in the current directory? If you type BLOWUP as the command to be executed, the command interpreter will look for a file called BLOWUP.COM, then BLOWUP.EXE, then BLOWUP.BAT. Either of the first two will be treated as a program to be loaded into memory and executed; the last of these is a batch file. (Batch files will be explained later.)

❑ If nothing's been found yet, each directory listed in the path is examined in turn for files called BLOWUP.COM, BLOWUP.EXE or BLOWUP.BAT (as in the previous step).

❑ If all else fails, display an "unknown command" error message.

You can display the list of directories that will be searched in the third step above (the path) by typing the command **PATH**. You can also use the **PATH** command to set the path; just separate the names of the directories you want to be searched by semicolons ("**;**"), like this:

```
PATH C:\DOS;C:\TOOLS;C:\GAMES
```

This will search the DOS, TOOLS and GAMES directories (in that order) for any command that isn't an internal command or a program in the current directory. Normally you would set up your path in your AUTOEXEC.BAT file which is executed when the system first starts up.

If you want to be absolutely sure which version of a command is being executed, you can type in the extension as well, or a full pathname:

```
    BLOWUP.EXE
    C:\DISASTER\BLOWUP.BAT
```

The first of these will search the path for a file called BLOWUP.EXE (and will ignore BLOWUP.COM or BLOWUP.BAT if it finds either of them first); the second will execute the file C:\DISASTER\BLOWUP.BAT without looking anywhere else. You'll need to do this if you have a program whose name matches an internal command:

```
  DIR.EXE
```

This looks for a program called DIR.EXE; if you just typed DIR, the internal DIR command would be executed instead.

## 7. Redirecting input and output

One nice thing about using a command prompt rather than a GUI is that it's much simpler to capture the output of a program in a file, or take the input for a program from somewhere other than the keyboard. All **programs have a number of standard input/output channels**, of which the most important are the **standard input** (normally the **keyboard**) and the **standard output** (normally the **screen**). There is also a **standard error** output which is also associated with the **screen** (and is intended for displaying error messages).

A few programs write their output directly to the screen (or elsewhere) but in many cases they will write results to the standard output. If you don't do anything unusual, this will be the screen. However, you can associate the standard output with a file instead of the screen so that when the program writes anything to the standard input, it'll end in the file you've specified rather than being displayed on the screen. All you have to do is specify "`> filename`" on the command line and the standard output will be routed to the file filename instead of the screen. For example,

```
    DIR
```

displays a directory listing on the screen, but

```
    DIR >DIRLIST.TXT
```

puts the directory listing in a file called DIRLIST.TXT, and doesn't display anything on the screen. (Read the ">" out loud as "to"; "dir to dirlist-dot-text".) If the file already exists, doing this will overwrite the existing contents. If you don't want this to happen, use ">>" instead of ">". This will append the program's output to the existing contents of the file rather than destroying it:

```
    DIR >>DIRLIST.TXT
```

This has the same effect as before, except that if the file DIRLIST.TXT already exists, the program's output will be tacked on to the end of it rather than replacing it. The standard input can be redirected using "<":

```
    MYPROG <TESTDATA.TXT
```

This will run the program MYPROG, but instead of reading its input from the keyboard it will read it from the file TESTDATA.TXT. This means that you can set up a file of test data for a program you're writing so that you can try the same test data over and over again while you're debugging without having to type it all in every time.

Redirecting the standard error output is more difficult; COMMAND.COM doesn't give you any way to do it, but you can either get an add-on utility to do it or use a more powerful shell like 4DOS that does allow this as a standard feature.

## 8. Pipes

Sometimes you want the output from one command to be processed by another command. For example, if you use the DIR command to display a long directory listing, it'll just scroll off the screen. Suppose you didn't know that the DIR command has a /P option; what could you do? There is a useful command called MORE which displays what it reads from the standard input onto the screen a screenful at a time, and then waits for you to press a key before continuing. So you could redirect the output of DIR into a temporary file and then use MORE to display it a screen at a time, like this:

```
    DIR >DIRLIST.TMP
    MORE <DIRLIST.TMP
```

```
   DEL DIRLIST.TMP
```

This happens often enough that there's a standard way to do it; you just separate the DIR and MORE commands with a "pipe" symbol ("|"), and the output of the first command is automatically routed to the input of the second command, so the above three lines can be written as a single line like this:

```
   DIR | MORE
```

Another example is a cunning way to avoid having to type in Y if you say "DEL *.*"; there is a standard command called ECHO which just echoes its parameters to the standard output (so ECHO HELLO WORLD displays the message HELLO WORLD on the screen), and you can use ECHO to give the DEL command the Y it needs:

```
   ECHO Y | DEL *.*
```

You'll still see the "Are you sure (Y/N)?" prompt, but the answer is read from the output of the ECHO command rather than from the keyboard (and you'll see how to overcome this in the next section).

Pipes (and all the other redirection facilities) are borrowed from the Unix world, where they've been commonplace for nearly twenty years now. Unix has accumulated a vast collection of filter programs, whose only purpose is to process the output from one command before passing it on to the next one. A classic example involves using a handful of simple programs connected in a continuous pipeline to process a text file of any size and list the ten commonest words together with the number of occurrences of each. This involves converting the input to lowercase (to ignore case differences), replacing all non-alphabetic characters with line breaks (so you end up with one word per line), sorting the result to give a list of words in alphabetical order, removing adjacent identical lines (so you end up with a list of unique words and the number of occurrences of each), sorting the list by number of occurrences and then displaying the first ten lines of the result.

## 9. Using device names as files

As I mentioned earlier, DOS provides names for most I/O devices and you can use these names in place of filenames. Here are some of the device names available:

```
   PRN:    -- the current printer
   LPT1:   -- the first parallel port (normally the printer)
   COM1:   -- the first serial port
   COM2:   -- the second serial port
   NUL:    -- the "null device"
```

For example, if you want to print a file you can just use one of the following commands:

```
   COPY FILE PRN:
   TYPE FILE >PRN:
```

The first of these copies the file FILE to the printer; the second one displays it to the standard output but redirects the standard output to the printer.

The null device NUL: is for situations where you want an empty input file or you want program output to be ignored. If you read from it, it looks like a file containing zero bytes of data; if you write to it, everything you write is cheerfully ignored and is just discarded. This is useful for throwing away unwanted output:

```
   COPY RESULT*.TXT PRN: >NUL:
   ECHO Y | DEL *.* >NUL:
```

The first example copies all files whose names begin with RESULT to the printer, and throws away all the messages that the COPY command normally displays on the screen ("1 file copied" or whatever). The second is a completely silent way of deleting all files in the current directory; the "Are you sure (Y/N)?" prompt is throw away, and the answer to this unasked question is then read from the output produced by the ECHO command.

## 10. Batch files

Sometimes you will want to execute the same sequence of commands over and over again. This is easy to do; just create a file containing those commands (one command per line) and give it a name with a .BAT extension (e.g. DO_IT.BAT). A file like this is known as a **batch file**. Once you've done this you can just type DO_IT at the command prompt, and (assuming that DO_IT.BAT is the first file that the command interpreter finds when it's looking for the command, as described above) the command interpreter will read this file and execute each line as a separate command. For example, assume that DO_IT.BAT contained the following lines:

```
   CLS
 ECHO HELLO, WORLD
```

Typing DO_IT would clear the screen and the display the message "HELLO, WORLD". By default, each line of the file is displayed on the screen ("echoed") before it's executed. This is useful for debugging but it ruins the effect in the batch file above. You can disable the echoing by prefixing the command with "@":

```
@CLS
@ECHO HELLO, WORLD
```

Another method is to disable echoing at the start of the batch file with the command "**ECHO OFF**" (which needs to be prefixed by "@", otherwise it will be echoed to the screen):

```
@ECHO OFF
CLS
ECHO HELLO, WORLD
```

Batch files can be quite sophisticated. For example, you can get at the command line parameters by referring to them as %1, %2, %3 and so on up to %9. The following batch file displays the second command line parameter followed by the first one:

```
@ECHO OFF
ECHO %2 %1
```

If this is saved as DO_IT.BAT, typing

```
DO_IT HELLO WORLD
```

will display the message

```
WORLD HELLO
```

There are also commands which give you the basics of a simple programming language: IF statements to choose between alternative courses of action, FOR loops to repeat a sequence of actions, tests to determine if a command executed successfully, and so on.

One nasty trap to watch out for is when you try to execute a batch file from inside another batch file. Imagine you've got this in DO_IT.BAT:

```
DO_OTHER.BAT
ECHO Finished!
```

The **ECHO** command on the second line won't be executed; if you refer to one batch file inside the other, the effect is a GOTO rather than a CALL (i.e. you go to the inner batch file but you don't remember where you came from, so at the end of the inner batch file you end up back at the command prompt). The solution is to use the internal CALL command instead:

```
CALL DO_OTHER.BAT
ECHO Finished!
```

This will work as expected; it'll execute DO_OTHER.BAT and then display the message "Finished!".

## 11. The AUTOEXEC.BAT file

The file C:\AUTOEXEC.BAT is a special one, because when the system is first started the command interpreter will automatically execute this file as a batch file (hence the name). This is where to put all your system startup commands; for example, setting the path, loading your keyboard and mouse driver, running a virus checker, or whatever. Whenever you install any new DOS software, you'll need to check whether you need to update this file (and if you do, you'll either need to restart the system or type C:\AUTOEXEC.BAT to re-execute it before any of the changes will take effect). You'll often need to update your path to include the directory where your new software was installed. Some software will do this automatically, and amazingly often it will get it wrong or do it in such a way that something else no longer works. What I always do is to have a separate file (C:\BOOT\AUTOEXEC.BAT) where all the "real" work is done, and then I have a one-line C:\AUTOEXEC.BAT file which says this:

```
@C:\BOOT\AUTOEXEC.BAT
```

Remember, C:\BOOT\AUTOEXEC.BAT will never go back to C:\AUTOEXEC.BAT when it's finished (I deliberately didn't use a CALL command). This means that anything that an auto-installation utility adds to the end of C:\AUTOEXEC.BAT will be ignored, and I can inspect it and decide whether it's made a sensible change or not. If so, I can then update my real AUTOEXEC.BAT file to include the changes.