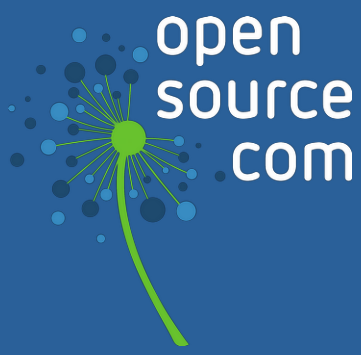


Advanced FreeDOS

by Jim Hall

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

Jim Hall

Jim Hall is an open source software advocate and developer, best known for usability testing in GNOME and as the founder + project coordinator of FreeDOS. At work, Jim is CEO of Hallmentum, an IT executive consulting company that provides hands-on IT Leadership training, workshops, and coaching.

Configure FreeDOS in plain text

By Jim Hall

The main configuration file for FreeDOS is a file in the root directory called `FDCONFIG.SYS`. This file contains a series of lines, each setting a value such as `LASTDRIVE=Z` or `FILES=40`. For example, the default `FDCONFIG.SYS` in FreeDOS looks like this:

```
SET DOSDIR=C:\FDOS
!COUNTRY=001,858,C:\FDOS\BIN\COUNTRY.SYS
!LASTDRIVE=Z
!BUFFERS=20
!FILES=40
!MENUCOLOR=7,0
MENUDEFAULT=1,5
MENU 1 - Load FreeDOS with JEMMEX, no EMS (most UMBs), max RAM free
MENU 2 - Load FreeDOS with JEMM386 (Expanded Memory)
MENU 3 - Load FreeDOS low with some drivers (Safe Mode)
MENU 4 - Load FreeDOS without drivers (Emergency Mode)
12?DOS=HIGH
12?DOS=UMB
12?DOSDATA=UMB
1?DEVICE=C:\FDOS\BIN\JEMMEX.EXE NOEMS X=TEST I=TEST NOVME NOINVLPG
234?DEVICE=C:\FDOS\BIN\HIMEMX.EXE
2?DEVICE=C:\FDOS\BIN\JEMM386.EXE X=TEST I=TEST I=B000-B7FF NOVME NOINVLPG
34?SHELL=C:\FDOS\BIN\COMMAND.COM C:\FDOS\BIN /E:1024 /P=C:\FDAUTO.BAT
12?SHELLHIGH=C:\FDOS\BIN\COMMAND.COM C:\FDOS\BIN /E:1024 /P=C:\FDAUTO.BAT
```

But what do all those lines mean? Why do some have a question mark (`?`) or an exclamation point (`!`), while other lines do not?

A simple configuration

Let's start with a simple configuration, so we can see what does what. Assume this very brief `FDCONFIG.SYS` file:

```
LASTDRIVE=Z
BUFFERS=20
FILES=40
DEVICE=C:\FDOS\BIN\HIMEMX.EXE
SHELL=C:\FDOS\BIN\COMMAND.COM C:\FDOS\BIN /E:1024 /P=C:\FDAUTO.BAT
```

This configuration file contains only a few instructions:

1. `LASTDRIVE=Z`
2. `BUFFERS=20`
3. `FILES=40`
4. `DEVICE=C:\FDOS\BIN\HIMEMX.EXE`
5. `SHELL=C:\FDOS\BIN\COMMAND.COM C:\FDOS\BIN /E:1024 /P=C:\FDAUTO.BAT`

The first instruction tells FreeDOS how many drive letters to reserve in memory. (DOS uses letters to represent each drive attached to the system, and `LASTDRIVE=Z` says to reserve drive letters from "A" to "Z."). `LASTDRIVE` affects the number of *logical drives* that your system can recognize. You probably don't have any logical drives; the FreeDOS installer doesn't set these up by default. In any case, it is safe to set `LASTDRIVE=Z` on any FreeDOS system.

The `BUFFERS` line reserves memory for file buffers. A *buffer* helps to speed up certain processes that require storage, such as copying files. If you set a larger value for `BUFFERS`, FreeDOS will reserve more memory—and vice versa for smaller values. Most users will set this to `BUFFERS=20` or `BUFFERS=40`, depending on how often they need to read and write files on the system.

The `FILES` setting determines how many files DOS allows you to open at one time. If you run an application that needs to open many files at once, such as a Genealogy database, you may need to set `FILES` to a larger value. For most users, `FILES=40` is a reasonable value.

`DEVICE` is a special instruction that loads a *device driver*. DOS requires device drivers for certain hardware or configurations. The line `DEVICE=C:\FDOS\BIN\HIMEMX.EXE` loads the *HimemX* device driver, so DOS can take advantage of expanded memory beyond the first 640 kilobytes.

The last line tells the FreeDOS kernel where to find the command-line shell. By default, the kernel will look for the shell as `COMMAND.COM`, but you can change it with the `SHELL` instruction. In this example, `SHELL=C:\FDOS\BIN\COMMAND.COM` says the shell is the `COMMAND.COM` program, located in the `\FDOS\BIN` directory on the `C` drive.

The other text at the end of the `SHELL` indicate the options to the `COMMAND.COM` shell. The FreeDOS `COMMAND.COM` supports several startup options to modify its behavior, including:

- **`C:\FDOS\BIN`** - The full path to the `COMMAND.COM` program
- **`/E:1024`** - The environment (E) size, in bytes. `/E:1024` tells `COMMAND.COM` to reserve 1024 bytes, or 1 kilobyte, to store its environment variables.
- **`/P=C:\FDAUTO.BAT`** - The `/P` option indicates that the shell is a permanent (P) shell, so the user cannot quit the shell by typing `EXIT` (the extra text `=C:\FDAUTO.BAT` tells `COMMAND.COM` to execute the `C:\FDAUTO.BAT` file at startup, instead of the default `AUTOEXEC.BAT` file)

With that simple configuration, you should be able to interpret some of the `FDCONFIG.SYS` file that's installed by FreeDOS.

Boot menu

FreeDOS supports a neat feature—multiple configurations on one system, using a "boot menu" to select the configuration you want. The `FDCONFIG.SYS` file contains several lines that define the menu:

```
!MENUCOLOR=7,0
MENUDEFAULT=1,5
MENU 1 - Load FreeDOS with JEMMEX, no EMS (most UMBs), max RAM free
MENU 2 - Load FreeDOS with JEMM386 (Expanded Memory)
MENU 3 - Load FreeDOS low with some drivers (Safe Mode)
MENU 4 - Load FreeDOS without drivers (Emergency Mode)
```

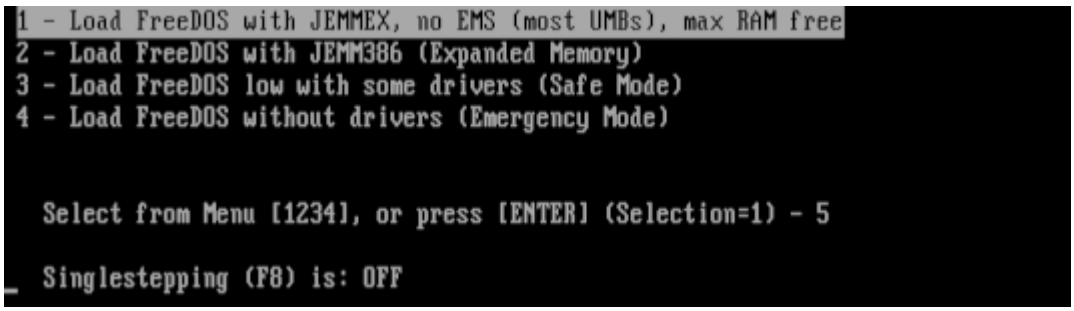
The `MENUCOLOR` instruction defines the text color and background color of the boot menu. These values are typically in the range 0 to 7, and represent these colors:

- 0 Black
- 1 Blue
- 2 Green
- 3 Cyan
- 4 Red
- 5 Magenta
- 6 Brown
- 7 White

So the `MENUCOLOR=7,0` definition means to display the menu with white (7) text on a black (0) background. If you instead wanted to use white text on a blue background, you could define this as `MENUCOLOR=7,1`.

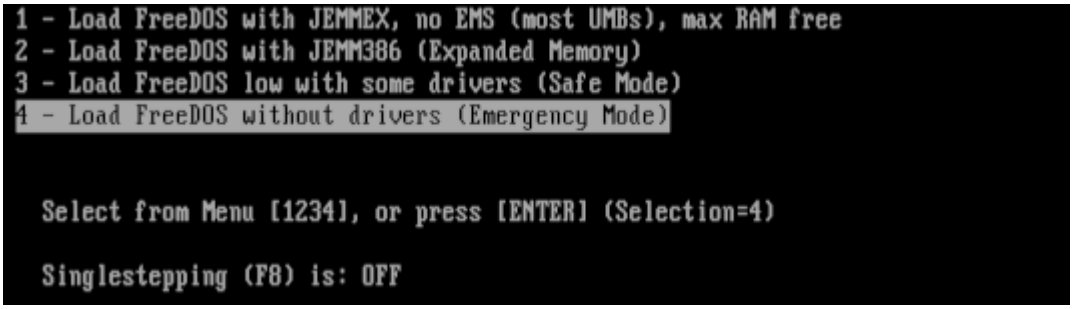
The exclamation point (`!`) at the start of the line means that this instruction will always be executed, no matter what menu option you choose.

The `MENUDEFAULT=1,5` line tells the kernel how long to wait for the user to select a boot menu entry, or what default menu entry to use if the user did not select one. `MENUDEFAULT=1,5` indicates the system will wait for 5 seconds; if the user did not attempt to select a menu item within that time, the kernel will assume boot menu "1" instead.



(Jim Hall, CC-BY SA 4.0)

The `MENU` lines after that are labels for the different boot menu configurations. These are presented in order, so menu item "1" is first, then "2," and so on.



(Jim Hall, CC-BY SA 4.0)

In the lines that follow in `FDCONFIG.SYS`, you will see numbers before a question mark (?). These indicate "for this boot menu entry, use this line." For example, this line with 234? will only load the HimemX device driver if the user selects boot menu entries "2," "3," or "4."

```
234?DEVICE=C:\FDOS\BIN\HIMEMX.EXE
```

There are lots of ways to use `FDCONFIG.SYS` to configure your FreeDOS system. We've only covered the basics here, the most common ways to define your FreeDOS kernel settings. For more information, explore the FreeDOS Help system (type `HELP` at the command line) to learn how to use all of the FreeDOS `FDCONFIG.SYS` options:

- **SWITCHES** Boot time processing behavior
- **REM** and **;** Comments (ignored in FDCONFIG.SYS)
- **MENUCOLOR** Boot menu text color and background color
- **MENUDEFAULT** Boot menu default value
- **MENU** Boot menu entry
- **ECHO** and **EECHO** Display messages
- **BREAK** Sets extended **Ctrl+C** checking on or off
- **BUFFERS** or **BUFFERSHIGH** How many disk buffers to allocate
- **COUNTRY** Sets international behavior
- **DOS** Tell the FreeDOS kernel how to load itself into memory
- **DOSDATA** Tell FreeDOS to load kernel data into upper memory
- **FCBS** Set the number of file control blocks (FCBs)
- **KEYBUF** Reassign the keyboard buffer in memory
- **FILES** or **FILESHIGH** How many files to have open at once
- **LASTDRIVE** or **LASTDRIVEHIGH** Set the last drive letter that can be used
- **NUMLOCK** Set the keyboard number pad lock on or off
- **SHELL**, **SHELLHIGH**, or **COMMAND** Set the command line shell
- **STACKS** or **STACKSHIGH** Add stacks to handle hardware interrupts

- **SWITCHAR** Redefines the command line option switch character
- **SCREEN** Set the number of lines on the screen
- **VERSION** Set what DOS version to report to programs
- **IDLEHALT** Activates energy saving features, useful on certain systems
- **DEVICE** and **DEVICEHIGH** Load a driver into memory
- **INSTALL** and **INSTALLHIGH** Load a "terminate and stay resident" (TSR) program
- **SET** Set a DOS environment variable

Configuring in plain text

Like Linux and BSD, FreeDOS configuration happens in plain text. No special tools for editing are required, so dive in and see what options suit you best. It's easy but powerful!

How FreeDOS boots

By Jim Hall

One thing I appreciate from growing up with DOS computers is that the boot process is relatively easy to understand. There aren't a lot of moving parts in DOS. And today, I'd like to share an overview of how your computer boots up and starts a simple operating system like FreeDOS.

Initial bootstrapping

When you turn on the power to your computer, the system performs several self-checks, such as verifying the memory and other components. This is called the **Power On Self Test** or "POST." After the POST, the computer uses a hard-coded instruction that tells it where to find its instructions to load the operating system. This is the "boot loader," and usually it will try to locate a Master Boot Record or (MBR) on the hard drive. The MBR then loads the primary operating system; in this case, that's FreeDOS.

This process of locating one piece of information so the computer can load the next part of the operating system is called "bootstrapping," from the old expression of "picking yourself up by your bootstraps." It is from this usage that we adopted the term "boot" to mean starting up your computer.

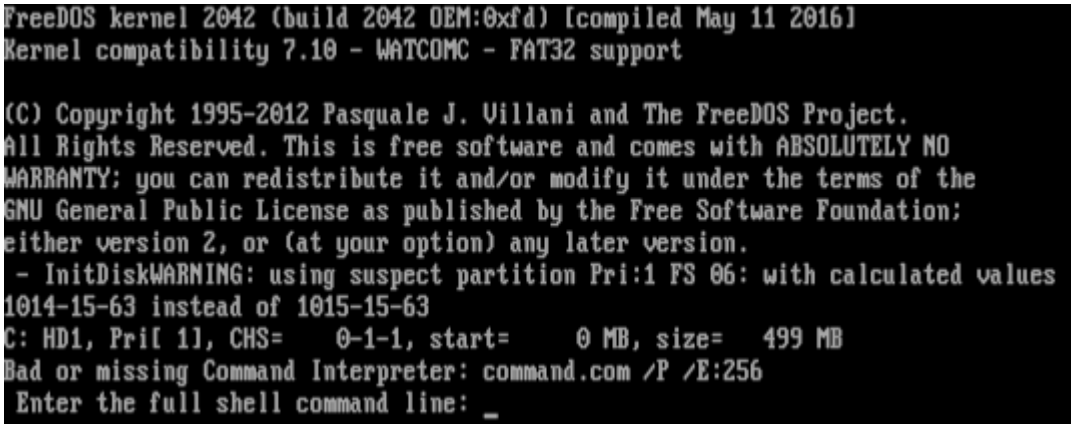
The kernel

When the computer loads the FreeDOS kernel, one of the first things the kernel does is identify any parameters the user has indicated to use. This is stored in a file called `FDCONFIG.SYS`, stored in the same root directory as the kernel. If `FDCONFIG.SYS` does not exist, then the FreeDOS kernel looks for an alternate file called `CONFIG.SYS`.

If you used DOS in the 1980s or 1990s, you may be familiar with the `CONFIG.SYS` file. Since 1999, FreeDOS looks for `FDCONFIG.SYS` first in case you have a DOS system that is *dual booting* FreeDOS with some other DOS, such as MS-DOS. Note that MS-DOS only uses the `CONFIG.SYS` file. So if you use the same hard drive to boot both FreeDOS and MS-DOS, MS-DOS uses `CONFIG.SYS` to configure itself, and FreeDOS uses `FDCONFIG.SYS` instead. That way, each can use its own configuration.

`FDCONFIG.SYS` can contain a number of configuration settings, one of which is `SHELL=` or `SHELLHIGH=`. Either one will instruct the kernel to load this program as the interactive shell for the user.

If neither `FDCONFIG.SYS` nor `CONFIG.SYS` exist, then the kernel assumes several default values, including where to find the shell. If you see the message "Bad or missing Command Interpreter" when you boot your FreeDOS system, that means `SHELL=` or `SHELLHIGH=` is pointing to a shell program that doesn't exist on your system.



(Jim Hall, CC-BY SA 4.0)

You might debug this by looking at the `SHELL=` or `SHELLHIGH=` lines. Failing that, make sure you have a program called `COMMAND.COM` in the root directory of your FreeDOS system. This is the *shell*, which I'll talk about next.

The shell

The term "shell" on a DOS system usually means a command-line interpreter; an interactive program that reads instructions from the user, then executes them. In this way, the FreeDOS shell is similar to the Bash shell on Linux.

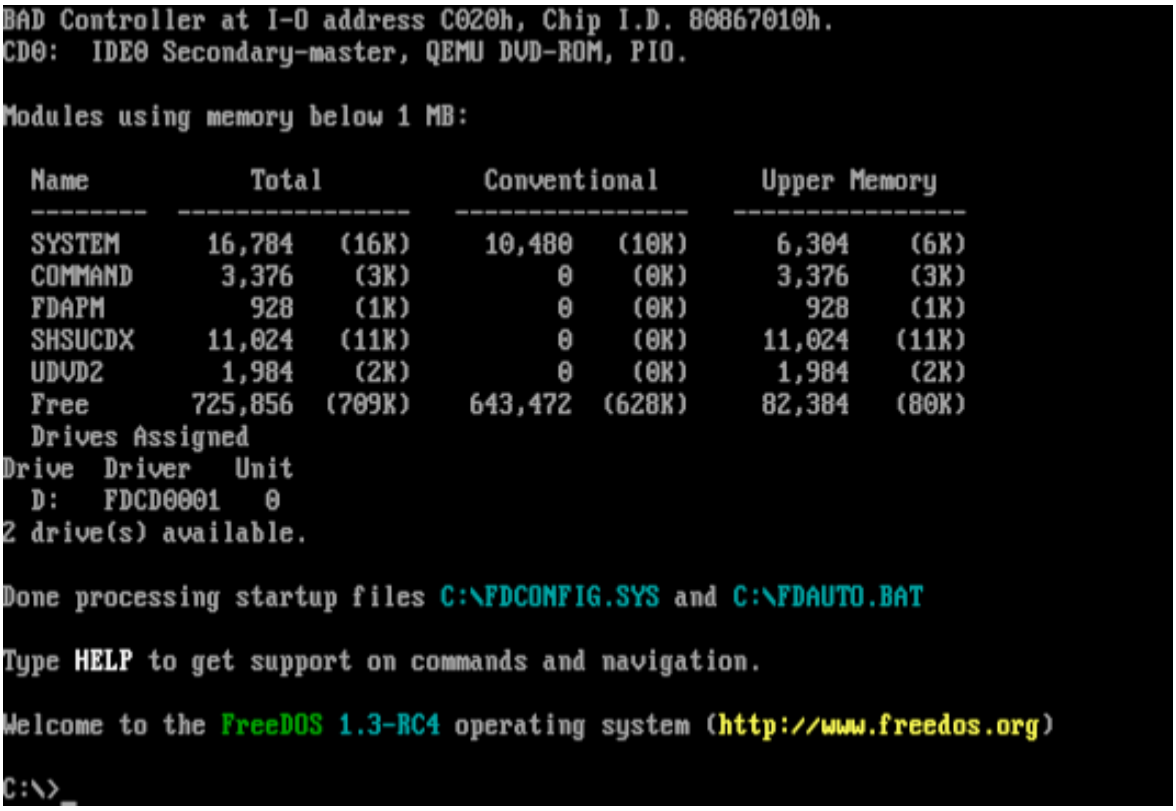
Unless you've asked the kernel to load a different shell using `SHELL=` or `SHELLHIGH=`, the standard command-line shell on DOS is called `COMMAND.COM`. And as `COMMAND.COM` starts

up, it also looks for a file to configure itself. By default, `COMMAND.COM` will look for a file called `AUTOEXEC.BAT` in the root directory. `AUTOEXEC.BAT` is a "batch file" that contains a set of instructions that run at startup, and is roughly analogous to the `~/.bashrc` "resource file" that Bash reads when it starts up on Linux.

You can change the shell, and the startup file for the shell, in the `FDCONFIG.SYS` file, with `SHELL=` or `SHELLHIGH=`. The FreeDOS installer sets up the system to read `FDAUTO.BAT` instead of `AUTOEXEC.BAT`. This is for the same reason that the kernel reads an alternate configuration file; you can dual-boot FreeDOS on a hard drive with another DOS. FreeDOS will use `FDAUTO.BAT` while MS-DOS will use `AUTOEXEC.BAT`..

Without a startup file like `AUTOEXEC.BAT`, the shell will simply prompt the user to enter the date and time.

And that's it. Once FreeDOS has loaded the kernel, and the kernel has loaded the shell, FreeDOS is ready for the user to type commands.



(Jim Hall, CC-BY SA 4.0)

Automate tasks with BAT files on FreeDOS

By Jim Hall

Even if you haven't used DOS before, you are probably aware of its command-line shell, named simply `COMMAND.COM`. The `COMMAND.COM` shell has become synonymous with DOS, and so it's no surprise that FreeDOS also implements a similar shell called "FreeCOM"—but named `COMMAND.COM` just as on other DOS systems.

But the FreeCOM shell can do more than just provide a command-line prompt where you run commands. If you need to automate tasks on FreeDOS, you can do that using *batch files*, also called "BAT files" because these scripts use the `.BAT` extension.

Batch files are much simpler than scripts you might write on Linux. That's because when this feature was originally added to DOS, long ago, it was meant as a way for DOS users to "batch up" certain commands. There's not much flexibility for conditional branching, and batch files do not support more advanced features such as arithmetic expansion, separate redirection for standard output vs error messages, background processes, tests, loops, and other scripting structures that are common in Linux scripts.

Here's a helpful guide to batch files under FreeDOS. Remember to reference environment variables by wrapping the variable name with percent signs (%) such as `%PATH%`. However, note that `FOR` loops use a slightly different construct for historical reasons.

Printing output

Your batch file might need to print messages to the user, to let them know what's going on. Use the `ECHO` statement to print messages. For example, a batch file might indicate it is done with a task with this statement:

```
ECHO Done
```

You don't need quotes in the `ECHO` statement. The FreeCOM `ECHO` statement will not treat quotes in any special way and will print them just like regular text.

Normally, FreeDOS prints out every line in the batch file as it executes them. This is usually not a problem in a very short batch file that only defines a few environment variables for the user. But for longer batch files that do more work, this constant display of the batch lines can become bothersome. To suppress this output, use the `OFF` keyword to the `ECHO` statement, as:

```
ECHO OFF
```

To resume displaying the batch lines as FreeDOS runs them, use the `ON` keyword instead:

```
ECHO ON
```

Most batch files include an `ECHO OFF` statement on the first line, to suppress messages. But the shell will still print `ECHO OFF` to the screen as it executes that statement. To hide that message, batch files often use an at sign (@) in front. This special character at the start of any line in a batch file suppresses printing that line, even if `ECHO` is turned on.

```
@ECHO OFF
```

Comments

When writing any long batch file, most programmers prefer to use *comments* to remind themselves about what the batch file is meant to do. To enter a comment in a batch file, use the `REM` (for *remark*) keyword. Anything after `REM` gets ignored by the FreeCOM shell.

```
@ECHO OFF
REM This is a comment
```

Executing a "secondary" batch file

Normally, FreeCOM only runs one batch file at a time. However, you might need to use another batch file to do certain things, such as set environment variables that are common across several batch files.

If you simply call the second batch file from a "running" batch file, FreeCOM switches entirely to that second batch file and stops processing the first one. To instead run the second batch file "inside" the first batch file, you need to tell the FreeCOM shell to *call* the second batch file with the `CALL` keyword.

```
@ECHO OFF
CALL SETENV.BAT
```

Conditional evaluation

Batch files do support a simple conditional evaluation structure with the `IF` statement. This has three basic forms:

1. Testing the return status of the previous command
2. Testing if a variable is equal to a value
3. Testing if a file exists

A common use of the `IF` statement is to test if a program returned successfully to the operating system. Most programs will return a zero value if they completed normally, or some other value in case of an error. In DOS, this is called the *error level* and is a special case to the `IF` test.

To test if a program called `MYPROG` exited successfully, you actually want to examine if the program returned a "zero" error level. Use the `ERRORLEVEL` keyword to test for a specific value, such as:

```
@ECHO OFF
MYPROG
IF ERRORLEVEL 0 ECHO Success
```

Testing the error level with `ERRORLEVEL` is a clunky way to examine the exit status of a program. A more useful way to examine different possible return codes for a DOS program is with a special variable FreeDOS defines for you, called `ERRORLEVEL`. This stores the error

level of the most recently executed program. You can then test for different values using the == test.

You can test if a variable is equal to a value using the == test with the `IF` statement. Like some programming languages, you use == to directly compare two values. Usually, you will reference an environment variable on one side and a value on the other, but you could also compare the values of two variables to see if they are the same. For example, you could rewrite the above `ERRORLEVEL` code with this batch file:

```
@ECHO OFF
MYPROG
IF %ERRORLEVEL%==0 ECHO Success
```

And another common use of the `IF` statement is to test if a file exists, and take action if so. You can test for a file with the `EXIST` keyword. For example, to delete a temporary file called `TEMP.DAT`, you might use this line in your batch file:

```
@ECHO OFF
IF EXIST TEMP.DAT DEL TEMP.DAT
```

With any of the `IF` statements, you can use the `NOT` keyword to *negate* a test. To print a message if a file *does not* exist, you could write:

```
@ECHO OFF
IF NOT EXIST TEMP.DAT ECHO No file
```

Branched execution

One way to leverage the `IF` test is to jump to an entirely different part of the batch file, depending on the outcome of a previous test. In the simplest case, you might want to skip to the end of the batch file if a key command fails. Or you might want to execute other statements if certain environment variables are not set up correctly.

You can skip around to different parts of a batch file using the `GOTO` instruction. This jumps to a specific line, called a *label*, in the batch file. Note that this is a strict "go-to" jump; batch file execution picks up at the new label.

Let's say a program needed an existing empty file to store temporary data. If the file did not exist, you would need to create a file before running the program. You might add these lines to a batch file, so your program always has a temporary file to work with:

```
@ECHO OFF
IF EXIST temp.dat GOTO prog
ECHO Creating temp file...
TOUCH temp.dat
:prog
ECHO Running the program…
MYPROG
```

Of course, this is a very simple example. For this one case, you might instead rewrite the batch file to create the temporary file as part of the `IF` statement:

```
@ECHO OFF
IF NOT EXIST temp.dat TOUCH temp.dat
ECHO Running the program...
MYPROG
```

Iteration

What if you need to perform the same task over a set of files? You can *iterate* over a set of files with the `FOR` loop. This is a one-line loop that runs a single command with a different file each time.

The `FOR` loop uses a special syntax for the iteration variable, which is used differently than other DOS environment variables. To loop through a set of text files so you can edit each one, in turn, use this statement in your batch file:

```
@ECHO OFF
FOR %%F IN (*.TXT) DO EDIT %%F
```

Note that the iteration variable is specified with only one percent sign (%) if you run this loop at the command line, without a batch file:

```
C:\> FOR %F IN (*.TXT) DO EDIT %F
```

Command-line processing

FreeDOS provides a simple method to evaluate any command-line options the user might have provided when running batch files. FreeDOS parses the command line, and stores the first nine batch file options in the special variables `%1`, `%2`, .. and so on until `%9`. Notice that the eleventh option (and beyond) are not directly accessible in this way. (The special variable `%0` stores the name of the batch file.)

If your batch file needs to process more than nine options, you can use the `SHIFT` statement to remove the first option and *shift* every option down by one value. So the second option becomes `%1`, and the tenth option becomes `%9`.

Most batch files need to shift by one value. But if you need to shift by some other increment, you can provide that parameter to the `SHIFT` statement, such as:

```
SHIFT 2
```

Here's a simple batch file that demonstrates shifting by one:

```
@ECHO OFF
ECHO %1 %2 %3 %4 %5 %6 %7 %8 %9
ECHO Shift by one ..
SHIFT 1
ECHO %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Executing this batch file with ten arguments shows how the `SHIFT` statement reorders the command line options, so the batch file can now access the tenth argument as `%9`:

```
C:\SRC>args 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9
Shift by one ..
2 3 4 5 6 7 8 9 10
C:\SRC>
```

Set and use environment variables in FreeDOS

By Jim Hall

A useful feature in almost every command-line environment is the *environment variable*. Some of these variables allow you to control the behavior or features of the command line, and other variables simply allow you to store data that you might need to reference later. Environment variables are also used in FreeDOS.

Variables on Linux

On Linux, you may already be familiar with several of these important environment variables. In the Bash shell on Linux, the `PATH` variable identifies where the shell can find programs and commands. For example, on my Linux system, I have this `PATH` value:

```
bash$ echo $PATH
/home/jhall/bin:/usr/lib64/ccache:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/
sbin
```

That means when I type a command name like `cat`, Bash will check each of the directories listed in my `PATH` variable, in order:

1. `/home/jhall/bin`
2. `/usr/lib64/ccache`
3. `/usr/local/bin`
4. `/usr/local/sbin`
5. `/usr/bin`
6. `/usr/sbin`

And in my case, the `cat` command is located in the `/usr/bin` directory, so the full path to that command is `/usr/bin/cat`.

To set an environment variable on Linux, you type the name of the variable, then an equals sign (=) and then the value to store in the variable. To reference that value later using Bash, you type a dollar sign ($) in front of the variable name.

```
bash$ var=Hello
bash$ echo $var
Hello
```

Variables on FreeDOS

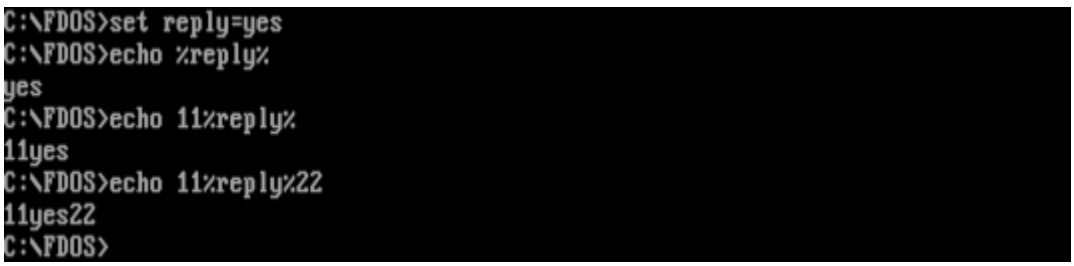
On FreeDOS, environment variables serve a similar function. Some variables control the behavior of the DOS system, and others are useful to store some temporary value.

To set an environment variable on FreeDOS, you need to use the SET keyword. FreeDOS is *case insensitive*, so you can type that using either uppercase or lowercase letters. Then set the variable as you might on Linux, using the variable name, an equals sign (=), and the value you want to store.

However, referencing or *expanding* an environment variable's value in FreeDOS is quite different from how you do it on Linux. You can't use the dollar sign ($) to reference a variable in FreeDOS. Instead, you need to surround the variable's name with percent signs (%).

(Jim Hall, CC-BY SA 4.0)

It's important to use the percent signs both before and after the name because that's how FreeDOS knows where the variable name begins and ends. This is very useful, as it allows you to reference a variable's value while immediately appending (or prepending) other text to the value. Let me demonstrate this by setting a new variable called `reply` with the value `yes`, then referencing that value with the text "11" before and "22" after it:



(Jim Hall, CC-BY SA 4.0)

Because FreeDOS is case insensitive you can also use uppercase or lowercase letters for the variable name, as well as the SET keyword. However, the variable's value will use the letter case as you typed it on the command line.

Finally, you can see a list of all the environment variables currently defined in FreeDOS. Without any arguments, the SET keyword will display all variables, so you can see everything at a glance:

(Jim Hall, CC-BY SA 4.0)

Environment variables are a useful staple in command-line environments, and the same applies to FreeDOS. You can set your own variables to serve your own needs, but be careful about changing some of the variables that FreeDOS uses. These can change the behavior of your running FreeDOS system:

- **DOSDIR**: The location of the FreeDOS installation directory, usually `C:\FDOS`
- **COMSPEC**: The current instance of the FreeDOS shell, usually `C:\COMMAND.COM` or `%DOSDIR%\BIN\COMMAND.COM`
- **LANG**: The user's preferred language
- **NLSPATH**: The location of the system's language files, usually `%DOSDIR%\NLS`
- **TZ**: The system's time zone

- **PATH**: A list of directories where FreeDOS can find programs to run, such as `%DOSDIR%\BIN`
- **HELPPATH**: The location of the system's documentation files, usually `%DOSDIR%\HELP`
- **TEMP**: A temporary directory where FreeDOS stores output from each command as it "pipes" data between programs on the command line
- **DIRCMD**: A variable that controls how the `DIR` command displays files and directories, typically set to `/OGNE` to order (O) the contents by grouping (G) directories first, then sorting entries by name (N) then extension (E)

If you accidentally change any of the FreeDOS "internal" variables, you could prevent some parts of FreeDOS from working properly. In that case, simply reboot your computer, and FreeDOS will reset the variables from the system defaults.

How to archive files on FreeDOS

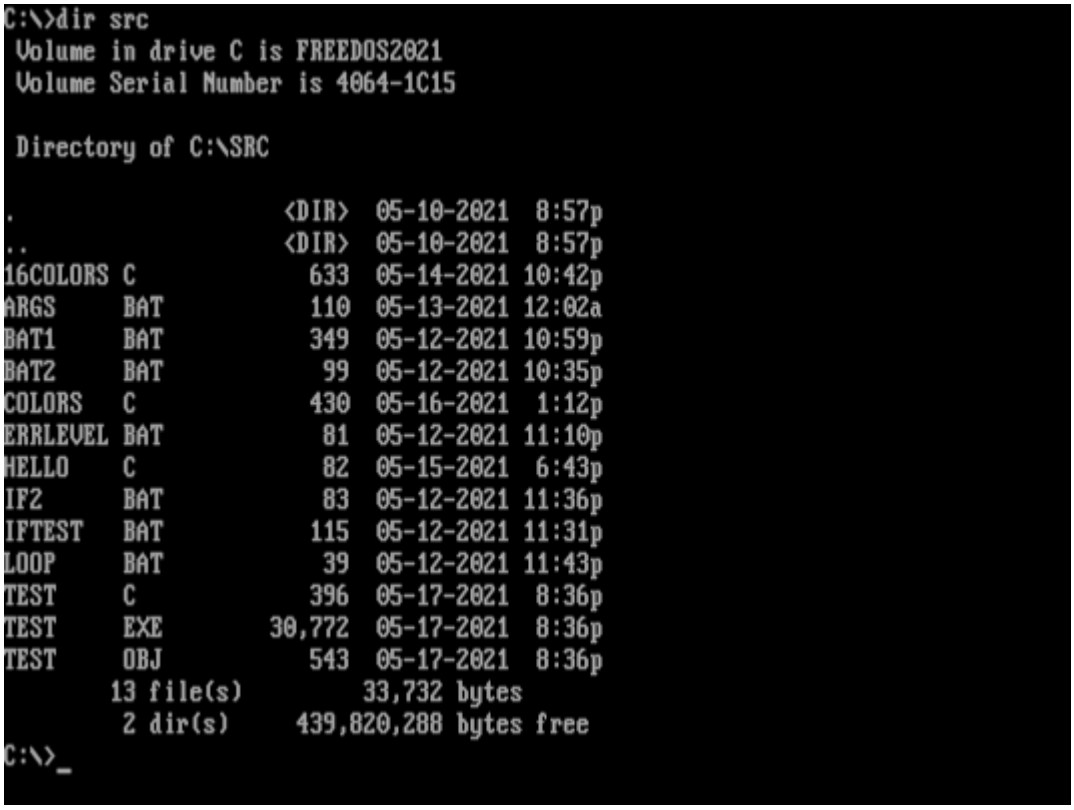
By Jim Hall

On Linux, you may be familiar with the standard Unix archive command: `tar`. There's a version of `tar` on FreeDOS too (and a bunch of other popular archive programs), but the de facto standard archiver on DOS is Zip and Unzip. Both Zip and Unzip are installed in FreeDOS by default.

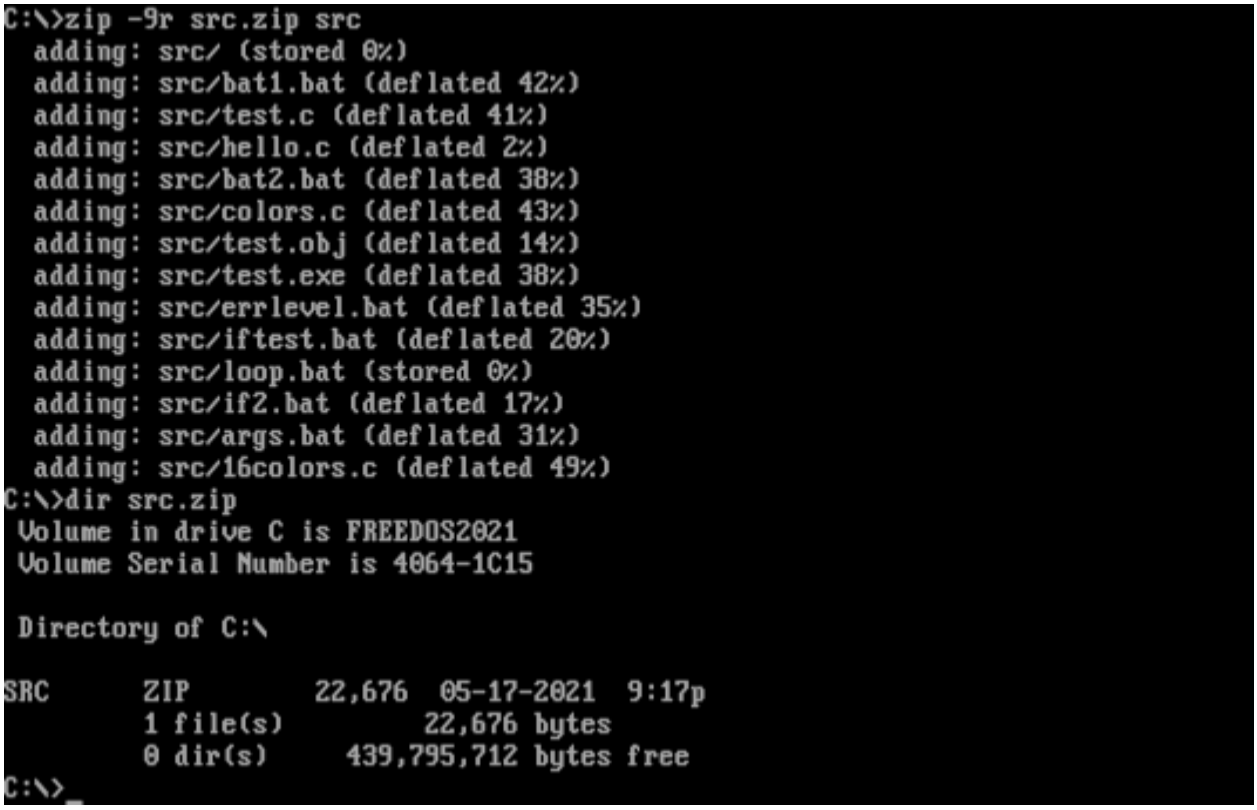
The Zip file format was originally conceived in 1989 by Phil Katz of PKWARE, for the PKZIP and PKUNZIP pair of DOS archive utilities. Katz released the specification for Zip files as an open standard, so anyone could create Zip archives. As a result of the open specification, Zip became a standard archive on DOS. The Info-ZIP project implements an open source set of `ZIP` and `UNZIP` programs.

Zipping files and directories

You can use `ZIP` at the DOS command line to create archives of files and directories. This is a handy way to make a backup copy of your work or to release a "package" to use in a future FreeDOS distribution. For example, let's say I wanted to make a backup of my project source code, which contains these source files:



`ZIP` sports a ton of command-line options to do different things, but the command line options I use most are `-r` to process directories and subdirectories *recursively*, and `-9` to provide the maximum compression possible. `ZIP` and `UNZIP` use a Unix-like command line, so you can combine options behind the dash: `-9r` will give maximum compression and include subdirectories in the Zip file.



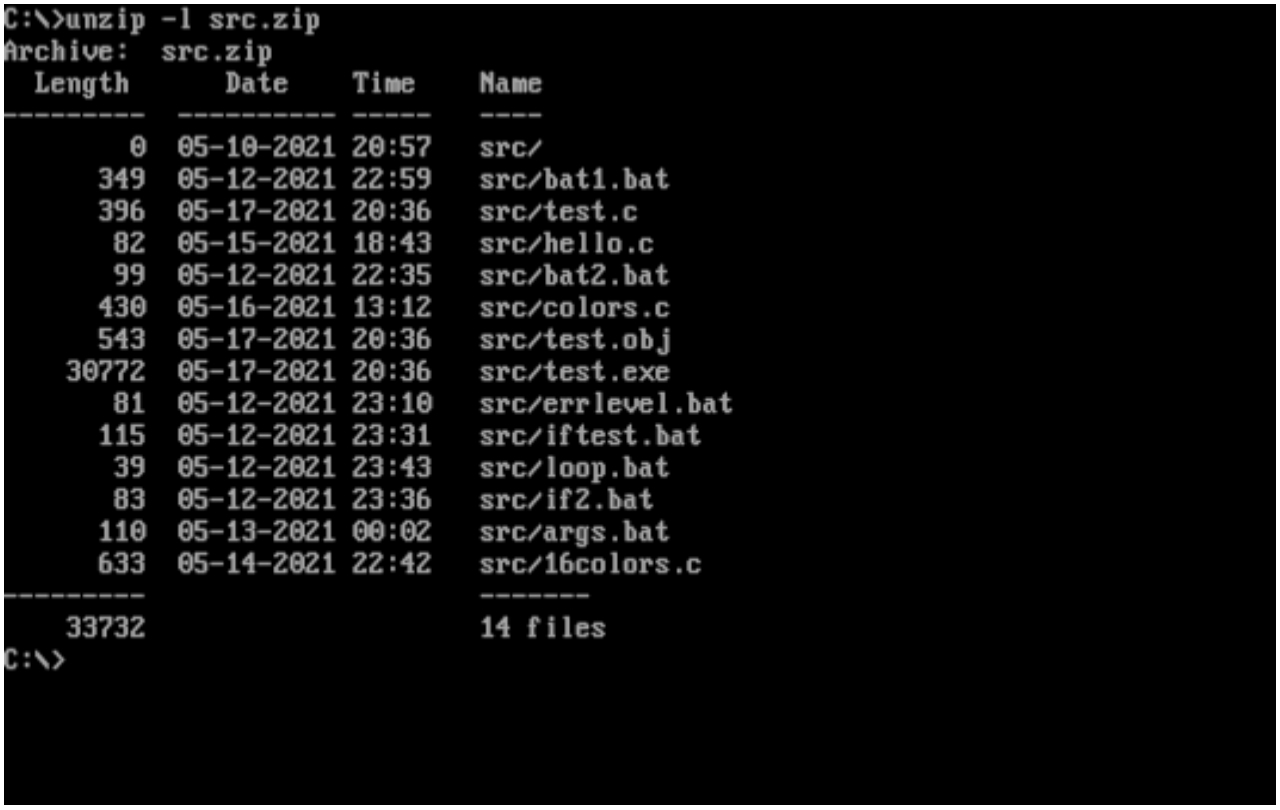
(Jim Hall, CC-BY SA 4.0)

In my example, `ZIP` was able to compress my source files from about 33 kilobytes down to about 22 kilobytes, saving me 11 kilobytes of valuable disk space. You might get different compression ratios depending on what options you give to `ZIP` or what files (and how many) you are trying to store in a Zip file. Generally, very long text files (such as source code) yield good compression—very small text files (like DOS "batch" files of only a few lines) are usually too short to compress well.

Unzipping files and directories

Saving files into a Zip file is great, but you'll eventually need to extract those files somewhere. Let's start by examining what's inside the Zip file we just created. For this, use the `UNZIP` command. You can use a bunch of different options with `UNZIP`, but I find I use just a few common options.

To list the contents of a Zip file, use the `-l` ("list") option:



(Jim Hall, CC-BY SA 4.0)

The output allows me to see the 14 entries in the Zip file: 13 files plus the `SRC` directory entry.

If I want to extract the entire Zip file, I could just use the `UNZIP` command and provide the Zip file as a command-line option. That extracts the Zip file starting at my current working directory. Unless I'm restoring a previous version of something, I usually don't want to overwrite my current files. In that case, I will want to extract the Zip file to a new directory. You can specify the destination path with the `-d` ("destination") command-line option:

```
C:\>unzip src.zip -d temp
Archive:  src.zip
   creating: temp/src/
  inflating: temp/src/bat1.bat
  inflating: temp/src/test.c
  inflating: temp/src/hello.c
  inflating: temp/src/bat2.bat
  inflating: temp/src/colors.c
  inflating: temp/src/test.obj
  inflating: temp/src/test.exe
  inflating: temp/src/errlevel.bat
  inflating: temp/src/iftest.bat
 extracting: temp/src/loop.bat
  inflating: temp/src/if2.bat
  inflating: temp/src/args.bat
  inflating: temp/src/16colors.c
C:\>
```

(Jim Hall, CC-BY SA 4.0)

Sometimes I want to extract a single file from a Zip file. In this example, let's say I wanted to extract `TEST.EXE`, a DOS executable program. To extract a single file, you specify the full path *from the Zip file* that you want to extract. By default, `UNZIP` will extract this file using the path provided in the Zip file. To omit the path information, you can add the `-j` ("junk the path") option.

You can also combine options. Let's extract the `SRC\TEST.EXE` program from the Zip file, but omit the full path and save it in the `TEMP` directory:

```
C:\>unzip -j src.zip src\test.exe -d temp
Archive:  src.zip
  inflating: temp/test.exe
C:\>_
```

(Jim Hall, CC-BY SA 4.0)

Because Zip files are an open standard, we continue to see Zip files today. Every Linux distribution supports Zip files using the Info-ZIP programs. Your Linux file manager may also have Zip file support—on the GNOME file manager, you should be able to right-click on a folder and select "Compress" from the drop-down menu. You'll have the option to create a new archive file, including a Zip file.

Creating and managing Zip files is a key skill for any DOS user. You can learn more about `ZIP` and `UNZIP` at the Info-ZIP website, or use the `-h` ("help") option on the command line to print out a list of options.

Copy files between Linux and FreeDOS

By Jim Hall

I run Linux as my primary operating system, and I boot FreeDOS in a virtual machine. Most of the time, I use QEMU as my PC emulator, but sometimes I'll run other experiments with GNOME Boxes (which uses QEMU as a back-end virtual machine) or with VirtualBox.

I like to play classic DOS games, and sometimes I'll bring up a favorite DOS application. I teach a Management Information Systems (MIS) class where I talk about the history of computing, and I'll sometimes record a demonstration using FreeDOS and a legacy DOS application, such as As-Easy-As (my favorite DOS spreadsheet—once released as "shareware" but now available for free from TRIUS, Inc).

But using FreeDOS this way means I need to transfer files between my FreeDOS virtual machine and my Linux desktop system. Let me show you how I do that.

Accessing the image with guestmount

I used to access my virtual disk image by calculating the offset to the first DOS partition, then calling the Linux `mount` command with the right mix of options to match that offset. This was always error-prone and not very flexible. Fortunately, there's an easier way to do it.
The `guestmount` program from the libguestfs-tools package lets you access or *mount* the virtual disk image from Linux. You can install `libguestfs-tools` using this command on Fedora:

```
$ yum install libguestfs-tools libguestfs
```

Using `guestmount` is not as easy as double-clicking the file from the GNOME file manager, but the command line isn't too difficult to use. The basic usage of `guestmount` is:

```
$ guestmount -a image -m device mountpoint
```

In this usage, *image* is the virtual disk image to use. On my system, I created my QEMU virtual disk image with the `qemu-img` command. The `guestmount` program can read this disk image format, as well as the QCOW2 image format used by GNOME Boxes, or the VDI image format used in VirtualBox.

The *device* option indicates the partition on the virtual disk. Imagine using this virtual disk as a real hard drive. You would access the first partition as `/dev/sda1`, the second partition as `/dev/sda2`, and so on. That's the syntax for `guestmount`. By default, FreeDOS creates one partition on an empty drive, so access that partition as `/dev/sda1`.

And *mountpoint* is the location to "mount" the DOS filesystem on your local Linux system. I'll usually create a temporary directory to work with. You only need the mount point while you're accessing the virtual disk.

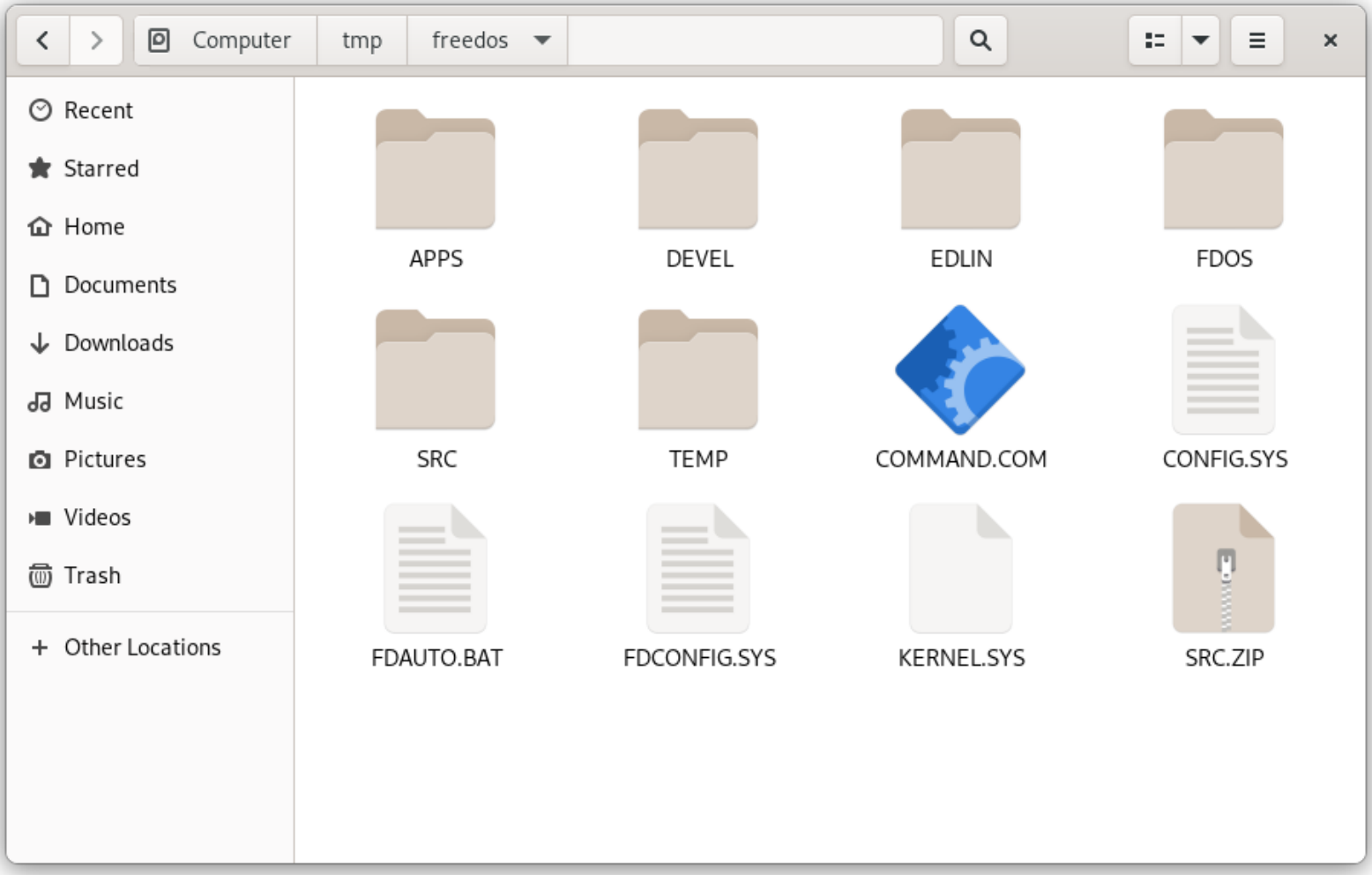
Putting that all together, I use this set of commands to access my FreeDOS virtual disk image from Linux:

```
$ mkdir /tmp/freedos
$ guestmount -a freedos.img -m /dev/sda1 /tmp/freedos
```

After that, I can access my FreeDOS files via the `/tmp/freedos` directory, using normal tools on Linux. I might use `ls /tmp/freedos` at the command line, or open the `/tmp/freedos` mount point using the desktop file manager.

```
$ ls -l /tmp/freedos
total 216
drwxr-xr-x.  5 root root  8192 May 10 15:53 APPS
-rwxr-xr-x.  1 root root 85048 Apr 30 07:54 COMMAND.COM
-rwxr-xr-x.  1 root root   103 May 13 15:48 CONFIG.SYS
drwxr-xr-x.  5 root root  8192 May 15 16:52 DEVEL
drwxr-xr-x.  2 root root  8192 May 15 13:36 EDLIN
-rwxr-xr-x.  1 root root  1821 May 10 15:57 FDAUTO.BAT
-rwxr-xr-x.  1 root root   740 May 13 15:47 FDCONFIG.SYS
drwxr-xr-x. 10 root root  8192 May 10 15:49 FDOS
-rwxr-xr-x.  1 root root 46685 Apr 30 07:54 KERNEL.SYS
```

```
drwxr-xr-x.  2 root root  8192 May 10 15:57 SRC
-rwxr-xr-x.  1 root root  3190 May 16 08:34 SRC.ZIP
drwxr-xr-x.  3 root root  8192 May 11 18:33 TEMP
```



Using GNOME file manager to access the virtual disk
(Jim Hall, CC-BY SA 4.0)

For example, to copy several C source files from my Linux `projects` directory into `C:\SRC` on the virtual disk image, so I can use the files under FreeDOS later, I can use the Linux `cp` command:

```
$ cp /home/jhall/projects/*.c /tmp/freedos/SRC
```

The files and directories on the virtual drive are technically *case insensitive*, so you can refer to them using uppercase or lowercase letters. However, I find it more natural to type DOS files and directories using all uppercase.

```
$ ls /tmp/freedos
APPS         CONFIG.SYS  EDLIN       FDCONFIG.SYS  KERNEL.SYS  SRC.ZIP

COMMAND.COM  DEVEL       FDAUTO.BAT  FDOS          SRC         TEMP
$ ls /tmp/freedos/EDLIN
EDLIN.EXE  MAKEFILE.OW
$ ls /tmp/freedos/edlin
EDLIN.EXE  MAKEFILE.OW
```

Unmounting with guestmount

You should always *unmount* the virtual disk image before you use it again in your virtual machine. If you leave the image mounted while you run QEMU or VirtualBox, you risk messing up your files.

The companion command to `guestmount` is `guestunmount`, to unmount the disk image. Just give the mount point that you wish to unmount:

```
$ guestunmount /tmp/freedos
```

Note that this command is spelled slightly differently from the Linux `umount` system command.

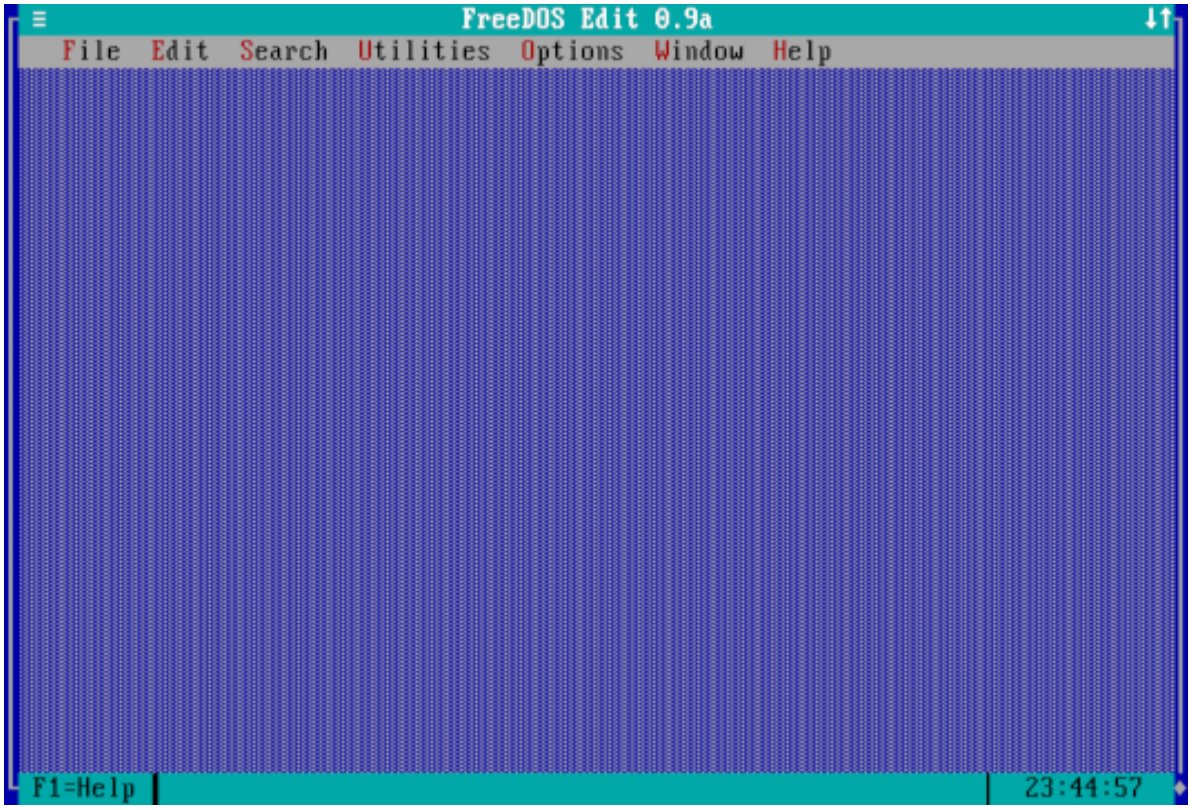
How to use the FreeDOS text editor

By Jim Hall

Editing files is a staple on any operating system. Whether you want to make a note about something, write a letter to a friend, or update a system configuration file—you need an editor. And FreeDOS provides a user-friendly text editor called (perhaps unimaginatively) "FreeDOS Edit."

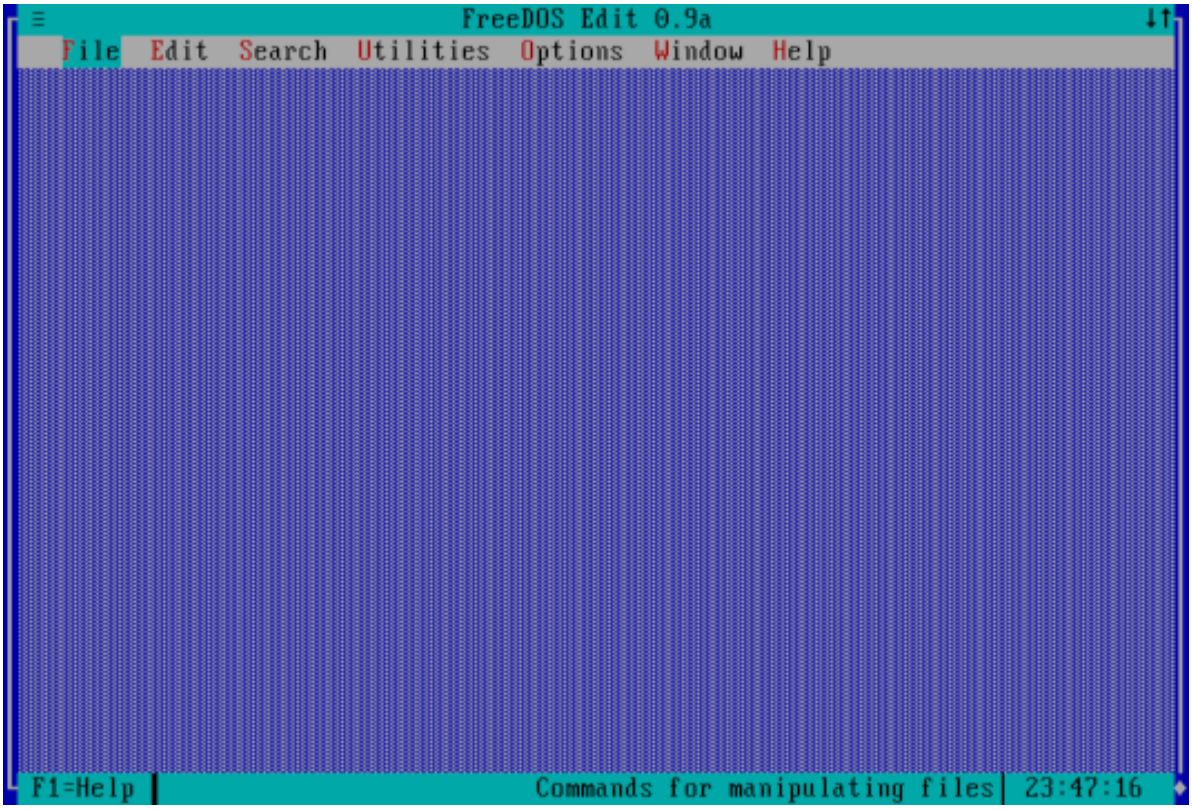
Editing files

The simplest invocation of FreeDOS Edit is just `EDIT`. This brings up an empty editor window. The patterned background suggests an empty "desktop"—a reminder that you aren't editing any files.



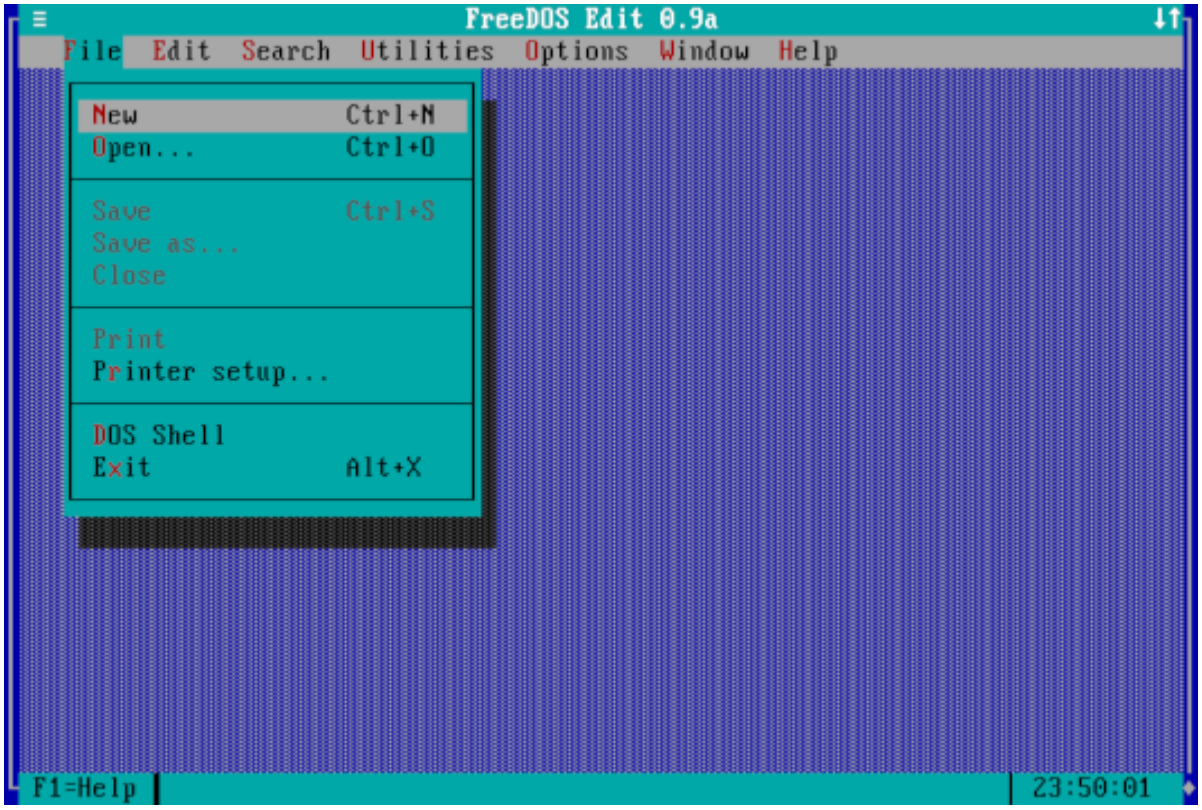
FreeDOS Edit without any files loaded
(Jim Hall, CC-BY SA 4.0)

Like most DOS applications, you can access the menus in Edit by tapping the `Alt` key once on your keyboard. This activates the menu. After hitting `Alt`, Edit will switch to "menu" access and will highlight the "File" menu. If you want to access a different menu on the menu bar, use the left and right arrow keys. Press the down arrow or hit the `Enter` key to go "into" the menu.



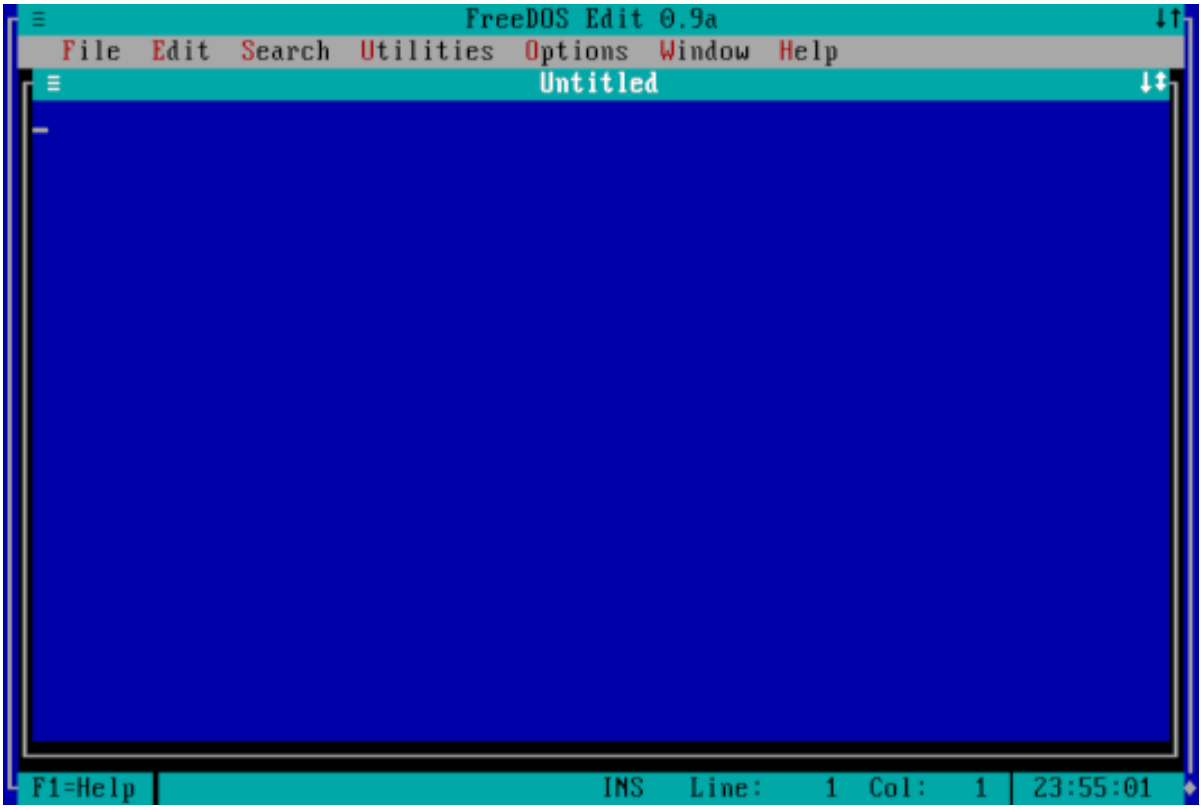
Highlighting the menu
(Jim Hall, CC-BY SA 4.0)

Do you notice that the first letter for each menu title is a different color? This highlight indicates a shortcut. For example, the "F" in the "File" menu is highlighted in red. So you could instead press `Alt`+F (`Alt` and F at the same time) and Edit will bring up the "File" menu.



The File menu
(Jim Hall, CC-BY SA 4.0)

You can use the "File" menu to either start a new (empty) file, or open an existing file. Let's start a new file by using the arrow keys to move down to "New" and pressing the `Enter` key. You can also start a new file by pressing `Ctrl+N` (`Ctrl` and `N` at the same time):



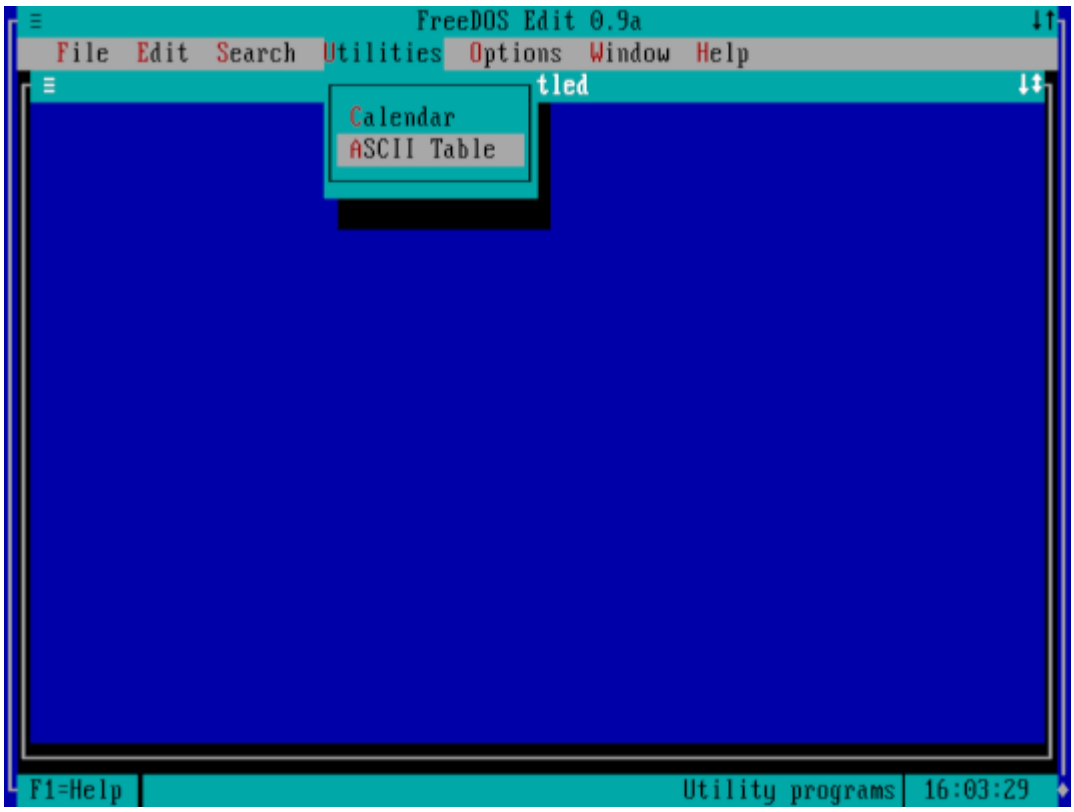
Editing a new file
(Jim Hall, CC-BY SA 4.0)

Editing files should be pretty straightforward after that. Most of the familiar keyboard shortcuts exist in FreeDOS Edit: `Ctrl+C` to copy text, `Ctrl+X` to cut text, and `Ctrl+V` to paste copied or cut text into a new location. If you need to find specific text in a long document, press `Ctrl+F`. To save your work, use `Ctrl+S` to commit changes back to the disk.

Programming in Edit

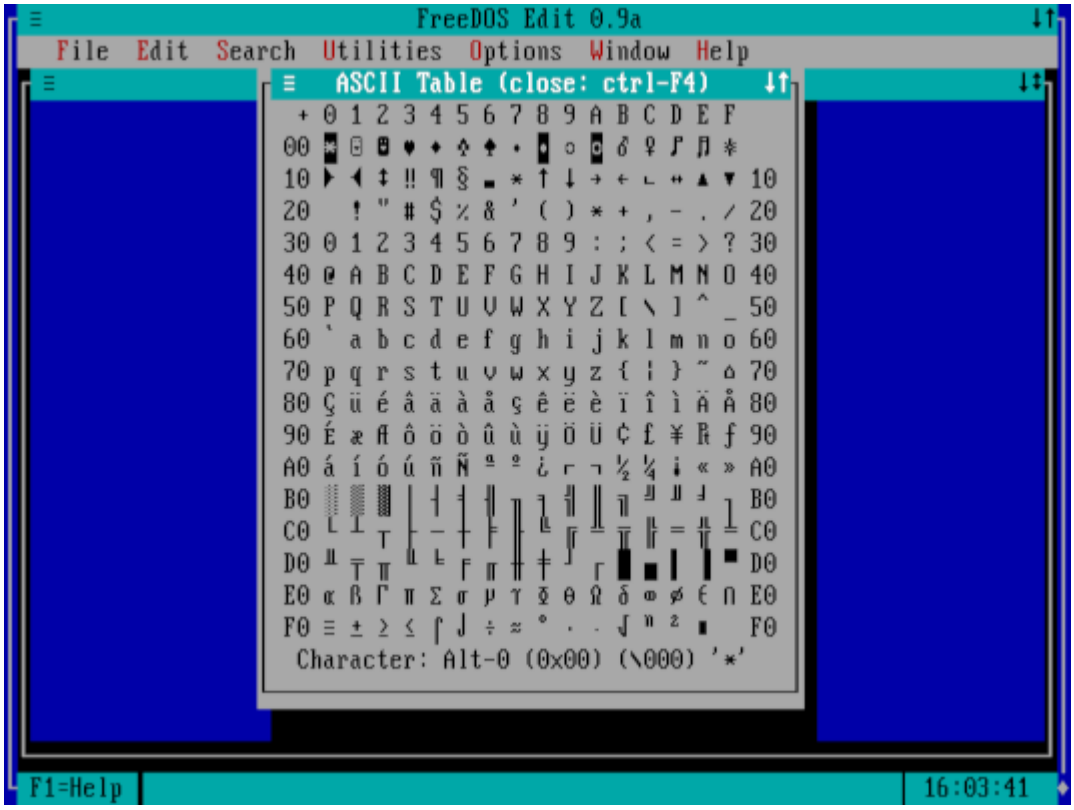
If you're a programmer, you may find the extended ASCII table a useful addition. DOS systems supported an "extended" ASCII character set commonly referred to as "code page 437." The standard characters between 0 and 127 include the letters A through Z (uppercase and lowercase), numbers, and special symbols like punctuation. However, the DOS extended characters from code 128 to code 255 included foreign language characters and "line drawing" elements. DOS programmers often made use of these extended ASCII characters, so FreeDOS Edit makes it easy to view a table of all the ASCII codes and their associated characters.

To view the ASCII table, use the "Utilities" menu and select the "ASCII Table" entry. This brings up a window containing a table.



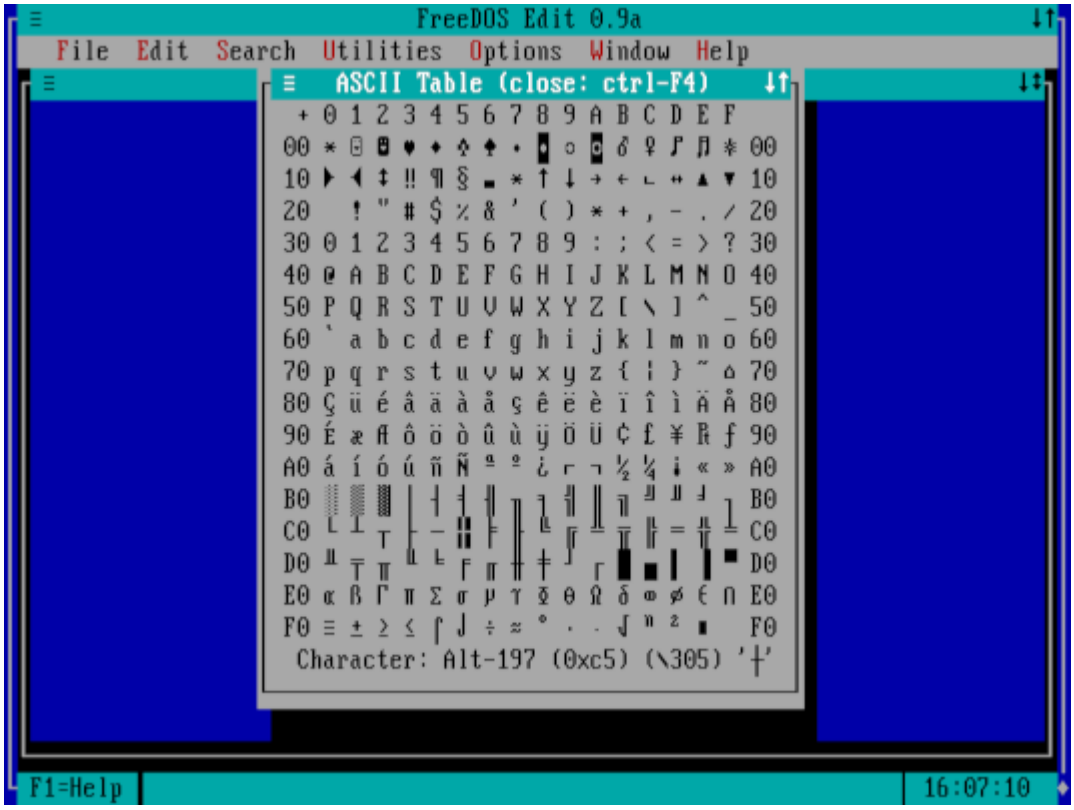
Find the ASCII Table in the Utilities menu
(Jim Hall, CC-BY SA 4.0)

Along the left, the table shows the hexadecimal values "00" through "F0," and the top shows the single values "0" through "F." These provide a quick reference to the hexadecimal code for each character. For example, the item in the first row (00) and the first column (0) has the hexadecimal value 00 + 0, or 0x00 (the "NULL" value). And the character in the fifth row (40) and the second column (1) has the value 40 + 1, or 0x41 (the letter "A").



The ASCII Table provides a handy reference for extended characters
(Jim Hall, CC-BY SA 4.0)

As you move the cursor through the table to highlight different characters, you'll see the values at the bottom of the table change to show the character's code in decimal, hexadecimal, and octal. For example, moving the cursor to highlight the "line intersection" character at row C0 and column 5 shows this extended character code as 197 (dec), 0xc5 (hex), and 305 (oct). In a program, you might reference this extended character by typing the hex value `0xc5`, or the octal "escape code" `\305`.



The "line intersection" character is 197 (dec), 0xc5 (hex), and 305 (oct)
(Jim Hall, CC-BY SA 4.0)

Feel free to explore the menus in Edit to discover other neat features. For example, the "Options" menu allows you to change the behavior and appearance of Edit. If you prefer to use a more dense display, you can use the "Display" menu (under "Options") to set Edit to 25, 43, or 50 lines. You can also force Edit to display in monochrome (white on black) or reversed mode (black on white).

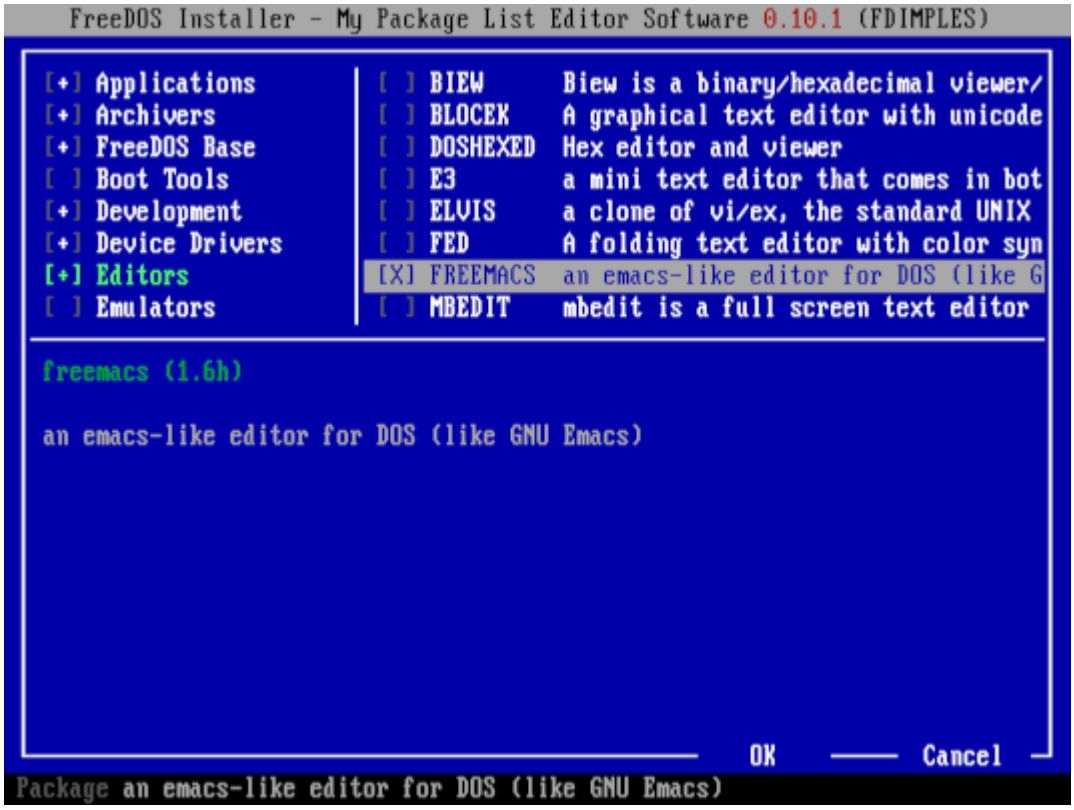
Edit text like Emacs in FreeDOS

By Jim Hall

On Linux, I often use the GNU Emacs editor to write the source code for new programs. I learned GNU Emacs long ago when I was an undergraduate student, and I still have the "finger memory" for all the keyboard shortcuts.

When I started work on FreeDOS in 1994, I wanted to include an Emacs-like text editor. You can find many editors similar to Emacs, such as MicroEmacs, but these all take some shortcuts to fit into the 16-bit address space on DOS. However, I was very pleased to find Freemacs, by Russell "Russ" Nelson.

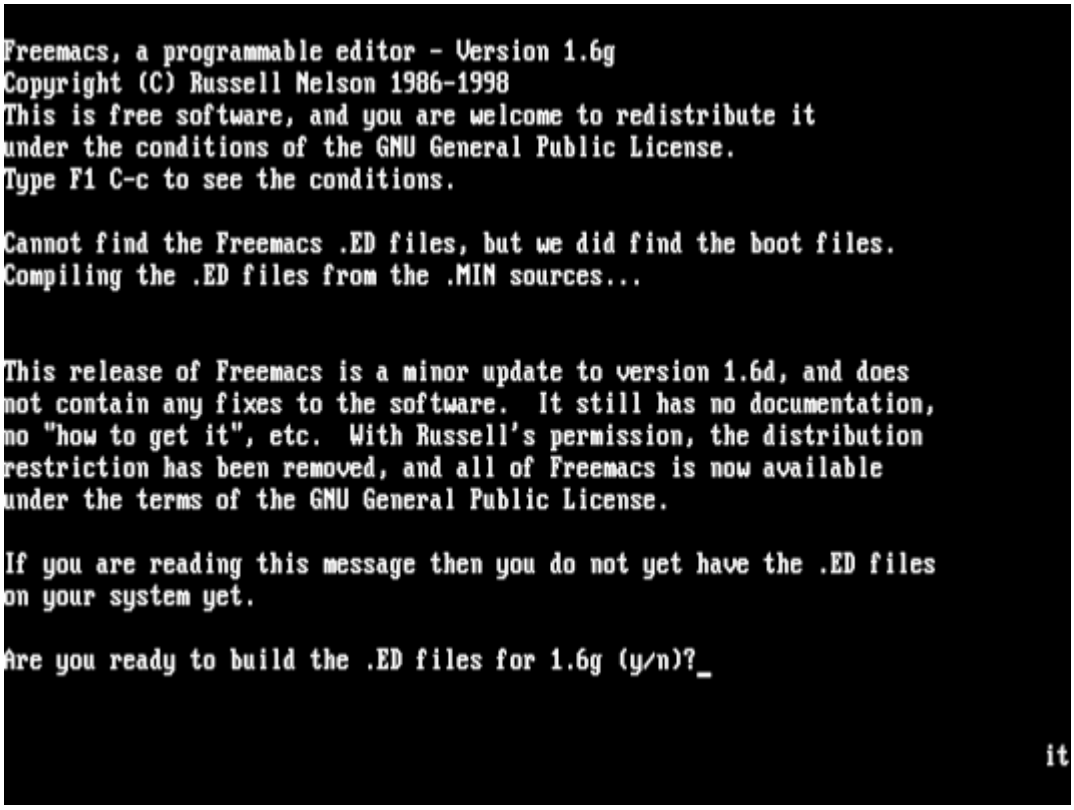
You can find Freemacs in FreeDOS, on the Bonus CD. You can use FDIMPLES to install the package, which will install to `\APPS\EMACS`.



Installing Freemacs from the FreeDOS Bonus CD
(Jim Hall, CC-BY SA 4.0)

Initial setup

The first time you run Freemacs, the editor will need to "compile" all of the setup files into a form that Freemacs can quickly process. This will take a few minutes to run, depending on your system's speed and memory, but fortunately, you only need to do it once.

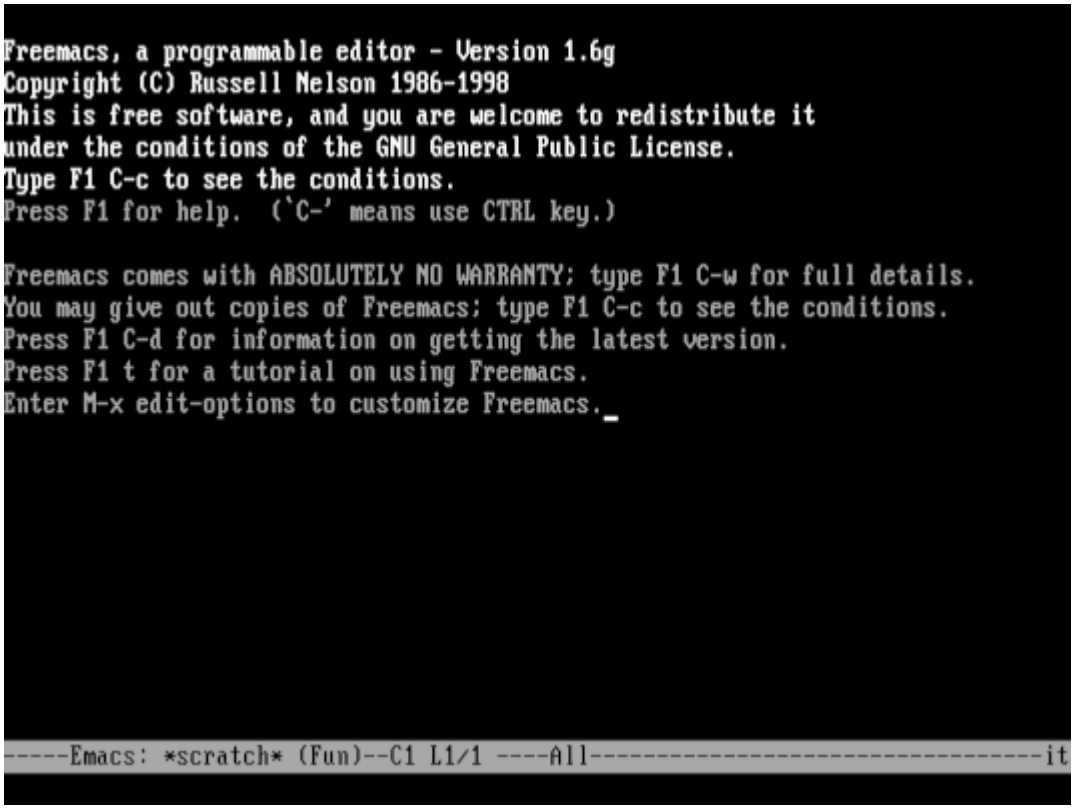


Press Y to build the Freemacs MINT files
(Jim Hall, CC-BY SA 4.0)

Freemacs actually processes the editor files in two passes. When Freemacs has successfully completed the first pass, it prompts you to restart the editor so it can finish processing. So don't be surprised that the process seems to start up again—it's just "part 2" of the compilation process.

Using Freemacs

To edit a file with Freemacs, start the program with the text file as an argument on the command line. For example, `emacs readme.doc` will open the Readme file for editing in Freemacs. Typing `emacs` at the command line, without any options, will open an empty "scratch" buffer in Freemacs.



Starting Freemacs without any files opens a "scratch" buffer
(Jim Hall, CC-BY SA 4.0)

Or, you can start Freemacs without any command-line options, and use the Emacs shortcuts `C-x C-f` (or `M-x find-file`). Freemacs then prompts you for a new file to load into the editor. The shortcut prefix `C-` means you should press the `Ctrl` key and some other key, so `C-x` is `Ctrl` and the `x` key together. And `M-x` is shorthand for "press the 'Meta' key (usually `Esc`) then hit `x`."

Opening a new file with C-x C-f
(Jim Hall, CC-BY SA 4.0)

Freemacs automatically detects the file type and attempts to load the correct support. For example, opening a C source file will also set Freemacs to "C-mode."

Editing a C source file in Freemacs
(Jim Hall, CC-BY SA 4.0)

If you also use GNU Emacs (like me), then you are probably curious to get Freemacs to match the C indentation that GNU Emacs uses (2 spaces.) Here is how to set Freemacs to use 2 spaces in C-mode:

1. Open a C source file in Freemacs.
2. Enter `M-x edit-options` to edit Freemacs settings.
3. Use the settings to change both "C-brace-offset" and "C-indent-level" to 2.
4. Save and exit Freemacs; you'll be prompted to save settings.

A few limitations

Much of the rest of Freemacs operates like GNU Emacs. If you're already familiar with GNU Emacs, you should feel right at home in Freemacs. However, Freemacs does have a few limitations that you might need to know:

The extension language is not LISP. The biggest difference between GNU Emacs on Linux and Freemacs on FreeDOS is that Freemacs uses a different extension language. Where GNU Emacs implements a LISP-like interpreter, Freemacs implements a different extension

language called MINT—based on the string processing language, TRAC. The name "MINT" is an acronym, meaning "MINT Is Not TRAC."

You shouldn't expect to evaluate LISP code in Freemacs. The MINT language is completely different from LISP. For more information on MINT, see the reference manual. We provide the full documentation via the FreeDOS files archive on Ibiblio, at /freedos/files/edit/emacs/docs. In particular, the MINT language is defined in mint.txt and mint2.txt.

Freemacs cannot open files larger than 64 kilobytes. This is a common limitation in many programs. 64kb is the maximum size of the data space for programs that do not leverage extended memory.

There is no "Undo" feature. Be careful in editing. If you make a mistake, you will have to re-edit your file to get it back to the old version. Also, save early and often. For very large mistakes, your best path might be to abandon the version you're editing in Freemacs, and load the last saved version.

The rest is up to you! You can find more information about Freemacs on Ibiblio, at /freedos/files/edit/emacs/docs. For a quick-start guide to Freemacs, read quickie.txt. The full manual is in tutorial.txt.

Use this nostalgic text editor on FreeDOS

By Jim Hall

In the very early days of DOS, the standard editor was a no-frills *line editor* called Edlin. Tim Paterson wrote the original Edlin for the first version of DOS, then called 86-DOS and later branded PC-DOS and MS-DOS. Paterson has commented that he meant to replace Edlin eventually, but it wasn't until ten years later that MS-DOS 5 (1991) replaced Edlin with Edit, a full-screen editor.

You may know that FreeDOS is an open source DOS-compatible operating system that you can use to play classic DOS games, run legacy business software, or develop embedded systems. FreeDOS has very good compatibility with MS-DOS, and the "Base" package group includes those utilities and programs that replicate the behavior of MS-DOS. One of those classic programs is an open source implementation of the venerable Edlin editor; Edlin is distributed under the GNU General Public License version 2.

Written by Gregory Pietsch, Edlin is a well-designed, portable editor. You can even compile Edlin on Linux. As Gregory described Edlin in the free ebook *23 Years of FreeDOS*, The top tier parses the input and calls the middle tier, a library called `edlib`, which calls the string and array-handling code to do the dirty work. But aside from its technical merits, I find Edlin is a joy to use when I want to edit text the "old school" way.

FreeDOS includes Edlin 2.18. That's actually one revision out of date, but you can download Edlin 2.19 from the FreeDOS files archive on Ibiblio. You'll find two files there—*edlin-2.19.zip* contains the source code, and *edlin-219exe.zip* is just the DOS executable. Download the *edlin-219exe.zip* file, and extract it to your FreeDOS system. I've unzipped my copy in `C:\EDLIN`.

Edlin takes a little practice to "get into" it, so let's edit a new file to show a few common actions in Edlin.

A walkthrough

Start editing a file by typing `EDLIN` and then the name of the file to edit. For example, to edit a C programming source file called `HELLO.C`, you might type:

```
C:\EDLIN> edlin hello.c
```

I've typed the FreeDOS commands in all lowercase here. FreeDOS is actually *case insensitive*, so you can type commands and files in uppercase or lowercase.
Typing `edlin` or `EDLIN` or `Edlin` would each run the Edlin editor. Similarly, you can identify the source file as `hello.c` or `HELLO.C` or `Hello.C`.

```
C:\EDLIN> edlin hello.c
edlin 2.19, copyright (c) 2003 Gregory Pietsch
This program comes with ABSOLUTELY NO WARRANTY.
It is free software, and you are welcome to redistribute it under the terms of
the GNU General Public License – either version 2 of the license, or, at your
option, any later version.
hello.c: 0 lines read
*
```

Once inside Edlin, you'll be greeted by a friendly `*` prompt. The interface is pretty minimal; no shiny "menu" or mouse support here. Just type a command at the `*` prompt to start editing, revise lines, search and replace, save your work, or exit the editor.

Since this is a new file, we'll need to add new lines. We'll do this with the *insert* command, by typing `i` at the `*` prompt. The Edlin prompt changes to `:` where you'll enter your new text. When you are done adding new text, type a period (`.`) on a line by itself.

```
*i
 : #include <stdio.h>
 :
 : int
 : main()
 : {
 :   puts("Hello world");
 : }
 : .
*
```

To view the text you've entered so far, use the *list* command by entering `l` at the * prompt. Edlin will display lines one screenful at a time, assuming 25 rows on the display. But for this short "Hello world" program, the source code fits on one screen:

```
*l
1: #include <stdio.h>
2:
3: int
4: main()
5: {
6:   puts("Hello world");
7:*}
*
```

Did you notice the * on line 7, the last line in the file? That's a special mark indicating your place in the file. If you inserted new text in the file, Edlin would add it at this location.

Let's update the C source file to return a code to the operating system. To do that, we'll need to add a line *above* line 7. Since that's where Edlin has the mark, we can use `i` to insert next text before this line. Don't forget to enter `.` on a line by itself to stop entering the new text.

By listing the file contents afterwards, you can see that we inserted the new text in the correct place, before the closing "curly brace" in the program.

```
*i
 :   return 0;
 : .
*l
1: #include <stdio.h>
2:
3: int
4: main()
5: {
6:   puts("Hello world");
7:   return 0;
8:*}
*
```

But what if you need to edit a single line in the file? At the * prompt,simply type the line number that you want to edit. Edlin works one line at a time, so you'll need to re-enter the full line. In this case, let's update the `main()` function definition to use a slightly different programming syntax. That's on line 4, so type 4 at the prompt, and re-type the line in full.

Listing the file contents afterwards shows the updated line 4.

```
*4
4:*main()
4: main(void)
*l
1: #include <stdio.h>
2:
3: int
4:*main(void)
5: {
6:   puts("Hello world");
7:   return 0;
8: }
*
```

When you've made all the changes you need to make, don't forget to save the updated file. Enter `w` at the prompt to *write* the file back to disk, then use `q` to *quit* Edlin and return to DOS.

```
*w
hello.c: 8 lines written
*q
C:\EDLIN>
```

Quick reference guide

Edlin does more than just "insert, edit, and save." Here's a handy cheat sheet showing all the Edlin functions, where *text* indicates a text string, *filename* is the path and name of a file, and *num* is a number (use `.` for the current line number, `$` for the last line number).

`?`	Show help
num	Edit a single line
`a`	Append a line below the mark
[*num*]`i`	Insert new lines before the mark
[*num*][`,`*num*]`l`	List the file (starting 11 lines above the mark)
[*num*][`,`*num*]`p`	Page (same as List, but starting at the mark)
[*num*]`,`[*num*]`,`*num*`,`[*num*]`c`	Copy lines
[*num*]`,`[*num*]`,`*num*`m`	Move lines
[*num*][`,`*num*][`?`]`s`*text*	Search for text
[*num*][`,`*num*][`?`]`r`*text*`,`*text*	Replace text
[*num*][`,`*num*]`d`	Delete lines
[*num*]`t`*filename*	Transfer (insert the contents of a new file at the mark)
[*num*]`w`[*filename*]	Write the file to disk
`q`	Quit Edlin
`e`[*filename*]	End (write and quit)

Programmers will be interested to know they can enter special characters in Edlin, using these special codes:

`\a`	alert
`\b`	backspace
`\e`	escape
`\f`	formfeed
`\t`	horizontal tab
`\v`	vertical tab
`\"`	double quote
`\'`	single quote
`\.`	period
`\\`	backslash
`\x`*XX*	hexadecimal number
`\d`*NNN*	decimal number
`\`*OOO*	octal number
`\^`*C*	control character

Why I like the FED text editor

By Jim Hall

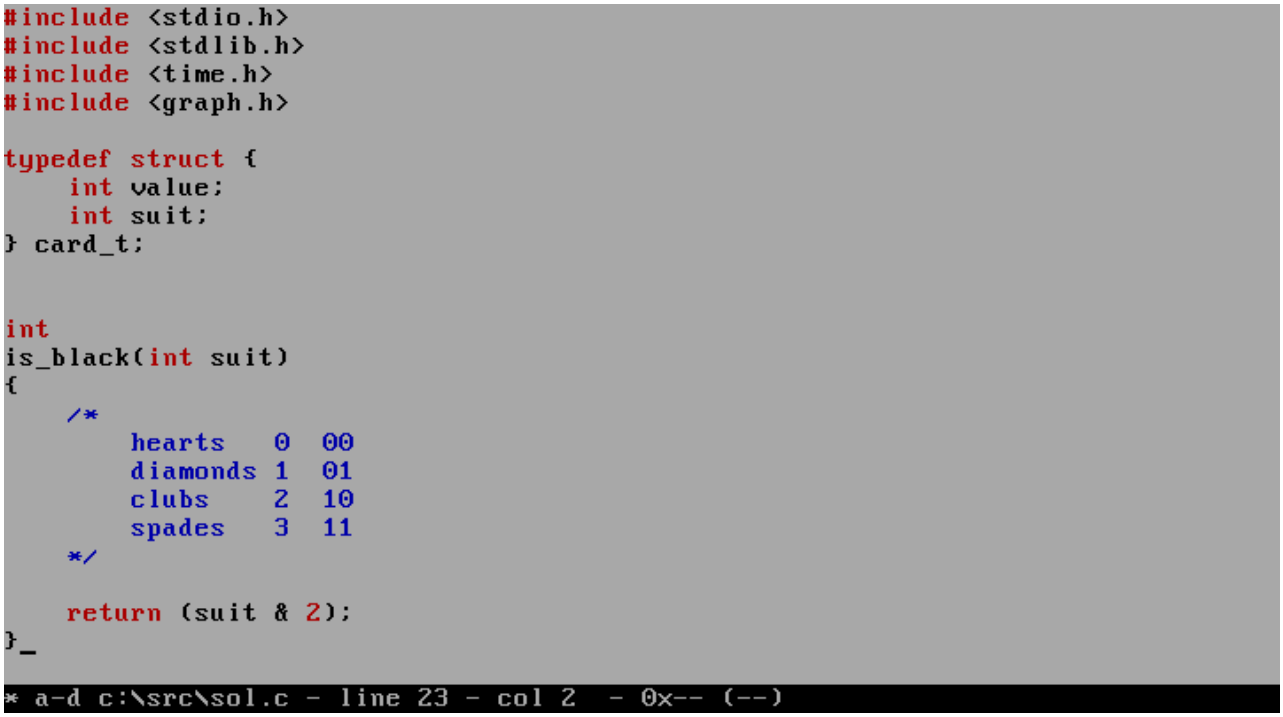
When I'm not at work on my Linux desktop, you can usually find me writing code for a legacy 16-bit system. FreeDOS is an open source DOS-compatible operating system that you can use to play classic DOS games, run legacy business software, or develop embedded systems. Any program that works on MS-DOS should also run on FreeDOS.

I grew up with DOS. My family's first personal computer was an Apple II clone, but we eventually upgraded to an IBM PC running DOS. I was a DOS user for over ten years, from the early 1980s until 1993, when I discovered Linux.

I was impressed by the freedom afforded by Linux and open source software. So when Microsoft announced the end of DOS in 1994, with the forthcoming Windows 95, I decided to write my own open source DOS. That's how FreeDOS started.

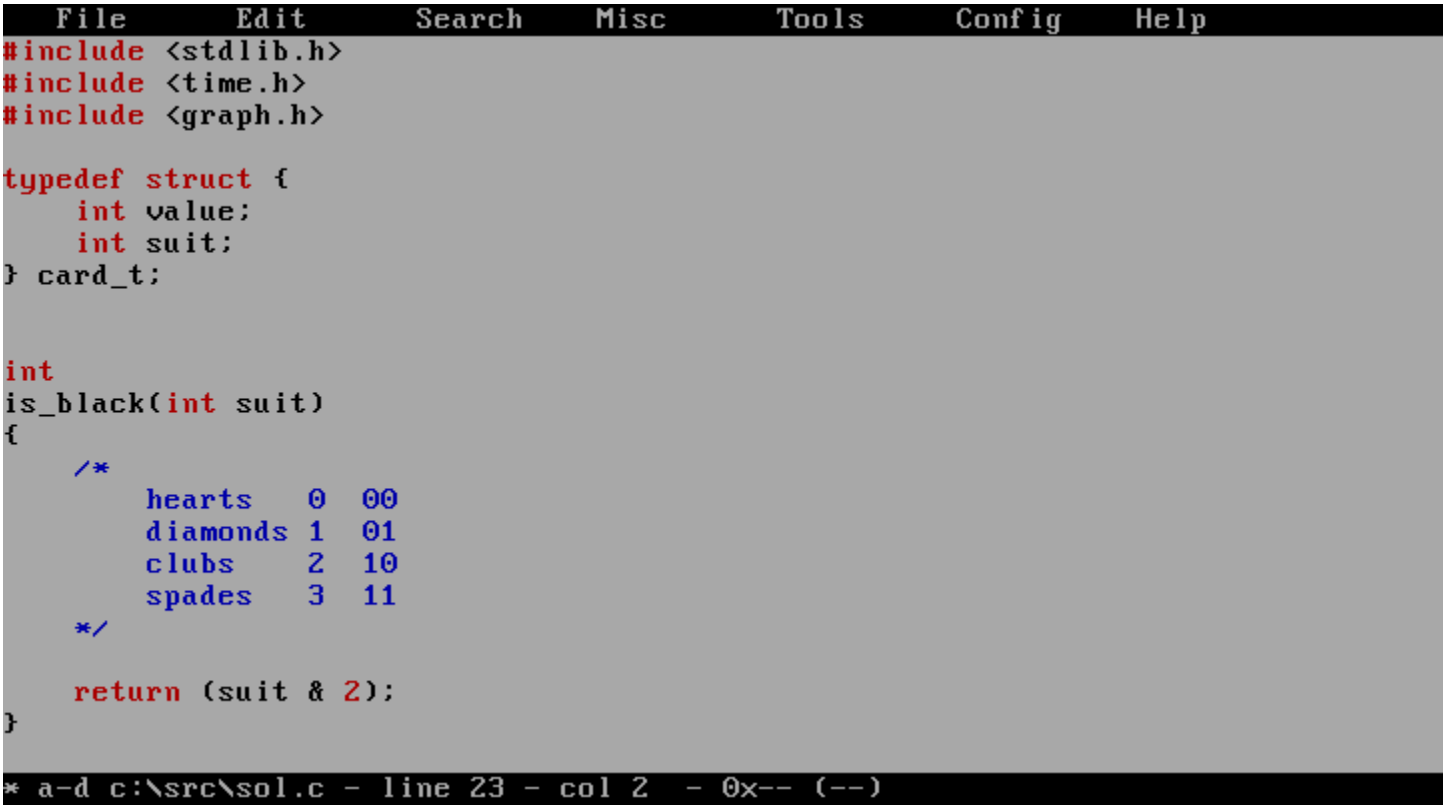
All these years later, and I continue working on FreeDOS. It is an excellent hobby system, where I can run my favorite DOS applications and games. And yes, I still write code for FreeDOS.

My favorite editor for DOS programming is the FED editor. FED is a minimal text editor without a lot of visual flair. This minimal approach helps me make the most of the standard 80x25 screen in DOS. When editing a file, FED displays a single status line at the bottom of the screen, leaving you the remaining 24 lines to write your code. FED also supports color syntax highlighting to display different parts of your code in various colors, making it easier to spot typos before they become bugs.



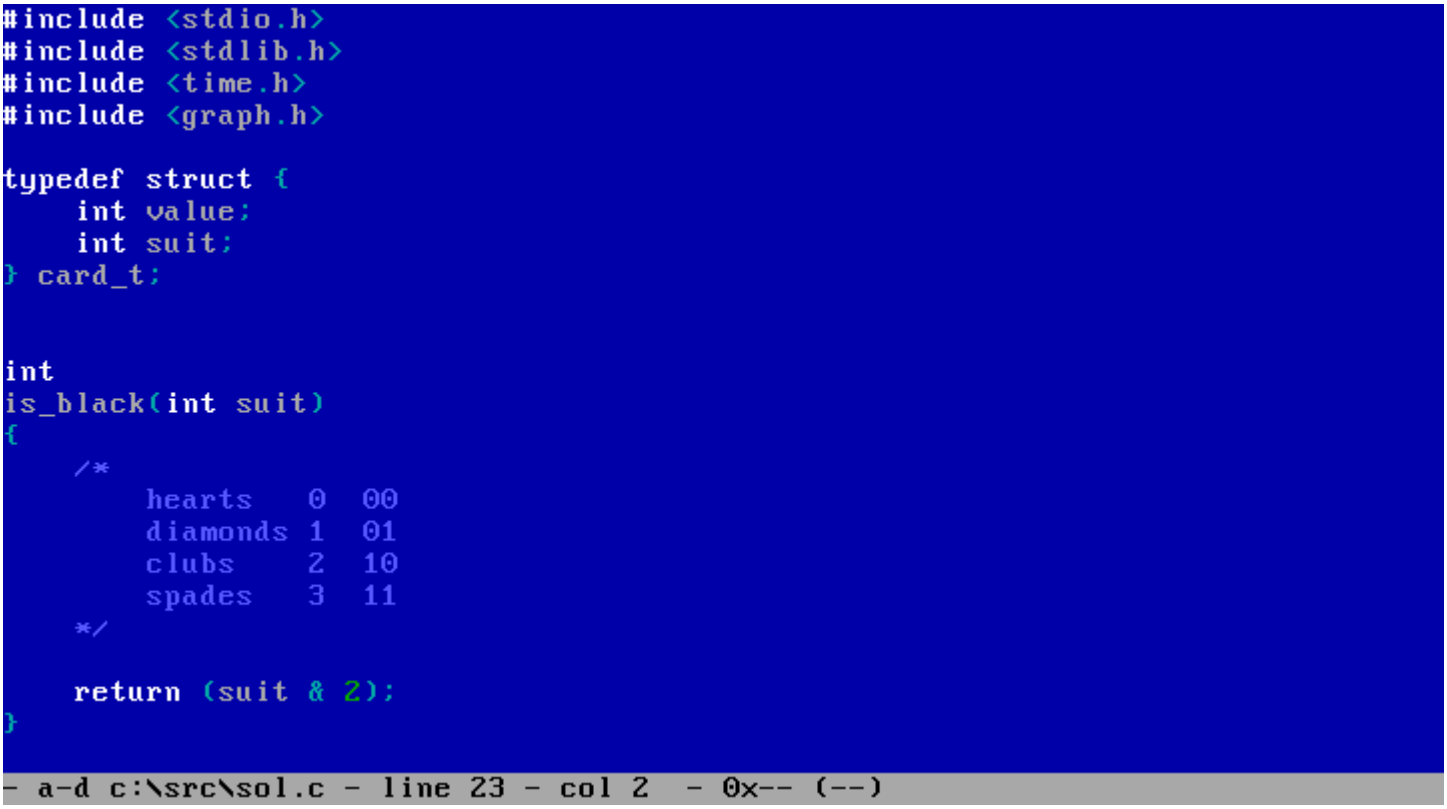
Writing a Solitaire game with FED - opensource.com

When you need to do something in the menus, press the **Alt** key on the keyboard, and FED displays a menu on the top line. FED supports keyboard shortcuts too, but be careful about the defaults. For example, **Ctrl-C** will close a file, and **Ctrl-V** will change the view. If you don't like these default keys, you can change the key mapping in the **Config** menu.



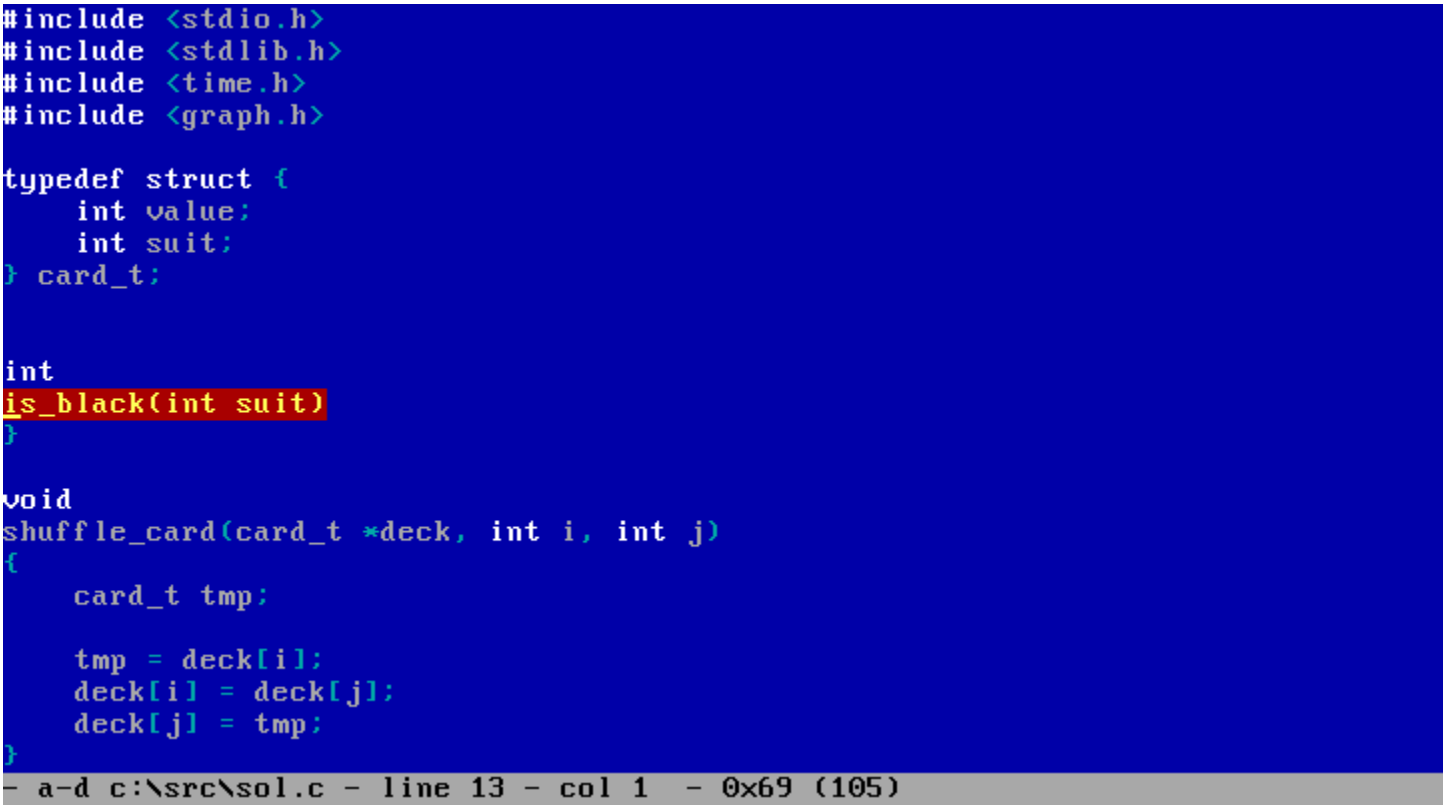
Tap the Alt key to bring up the menu - opensource.com

If you don't like the default black-on-white text display, you can change the colors under the **Config** menu. I prefer white-on-blue for my main text, with keywords in bright white, comments in bright blue, special characters in cyan, and numbers in green. FED makes it easy to set the colors you want.



My preferred colors when programming on DOS - opensource.com

FED is also a folding text editor, which means that it can collapse or expand portions of my code so that I can see more of my file. Tap **Ctrl-F** on a function name and FED will collapse the entire function. Folding works on other code, as well. I also use folding to hide **for** and **while** loops or other flow controls like **if** and **switch** blocks.



Folding a function lets you see more of the file - opensource.com

Shawn Hargreaves wrote and maintained FED from 1994 to 2004. Robert Riebisch has maintained FED since then. FED is distributed under the GNU GPL and supports DOS, Linux, and Windows.

Listen to music on FreeDOS

By Jim Hall

Music is a great way to relax. On Linux, I listen to music using Rhythmbox. But did you know you can listen to music on FreeDOS, as well? Let's take a look at two popular programs to listen to music:

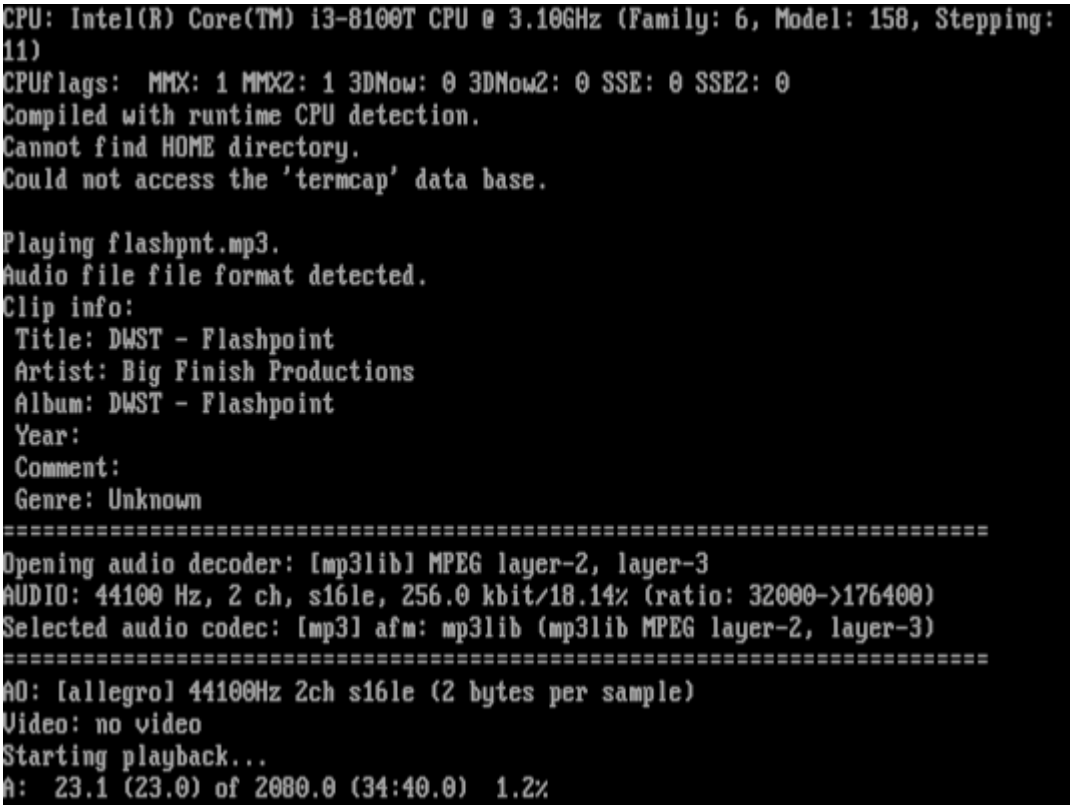
Listen to music with Mplayer

Mplayer is an open source media player that's usually found on Linux, Windows, and Mac—but there's a DOS version available, too. And that's the version we include in FreeDOS. While the DOS port is based on an older version (version 1.0rc2-3-3-2 from 2007) it is perfectly serviceable for playing media on DOS.

I use Mplayer to listen to music files on FreeDOS. For this example, I've copied one of my favorite audiobooks, *Doctor Who: Flashpoint* by Big Finish Productions, and saved it as `C:\MUSIC\FLASHPNT.MP3` on my FreeDOS computer. To listen to *Flashpoint* on FreeDOS, I launch Mplayer from the FreeDOS command line and specify the MP3 filename to play. The basic usage of Mplayer is `mplayer [options] filename` so if the default settings work well for you, then you can just launch Mplayer with the filename. In this case, I ran these commands to change my working directory to `\MUSIC` and then run Mplayer with my MP3 audiobook file:

```
CD \MUSIC
MPLAYER FLASHPNT.MP3
```

FreeDOS is *case insensitive*, so it will accept uppercase or lowercase letters for DOS commands and any files or directories. You could also type `cd \music` or `Cd \Music` to move into the Music directory, and that would work the same.



You can use Mplayer to listen to MP3 files
(Jim Hall, CC-BY SA 4.0)

Using Mplayer is a "no frills" way to listen to music files on FreeDOS. But at the same time, it's not distracting, so I can leave FreeDOS to play the MP3 file on my DOS computer while I use my other computer to do something else. However, FreeDOS runs tasks one at a time (in other words, DOS is a "single-tasking" operating system) so I cannot run Mplayer in the "background" on FreeDOS while I work on something else *on the same FreeDOS computer*.

Note that Mplayer is a big program that requires a lot of memory to run. While DOS itself doesn't require much RAM to operate, I recommend at least 16 megabytes of memory to run Mplayer.

Listen to audio files with Open Cubic Player

FreeDOS offers more than just Mplayer for playing media. We also include the Open Cubic Player, which supports a variety of file formats including Midi and WAV files.

In 1999, I recorded a short audio file of me saying, "Hello, this is Jim Hall, and I pronounce 'FreeDOS' as *FreeDOS*." This was meant as a joke, riffing off of a similar audio file (`english.au`, included in the Linux source code tree in 1994) recorded by Linus Torvalds

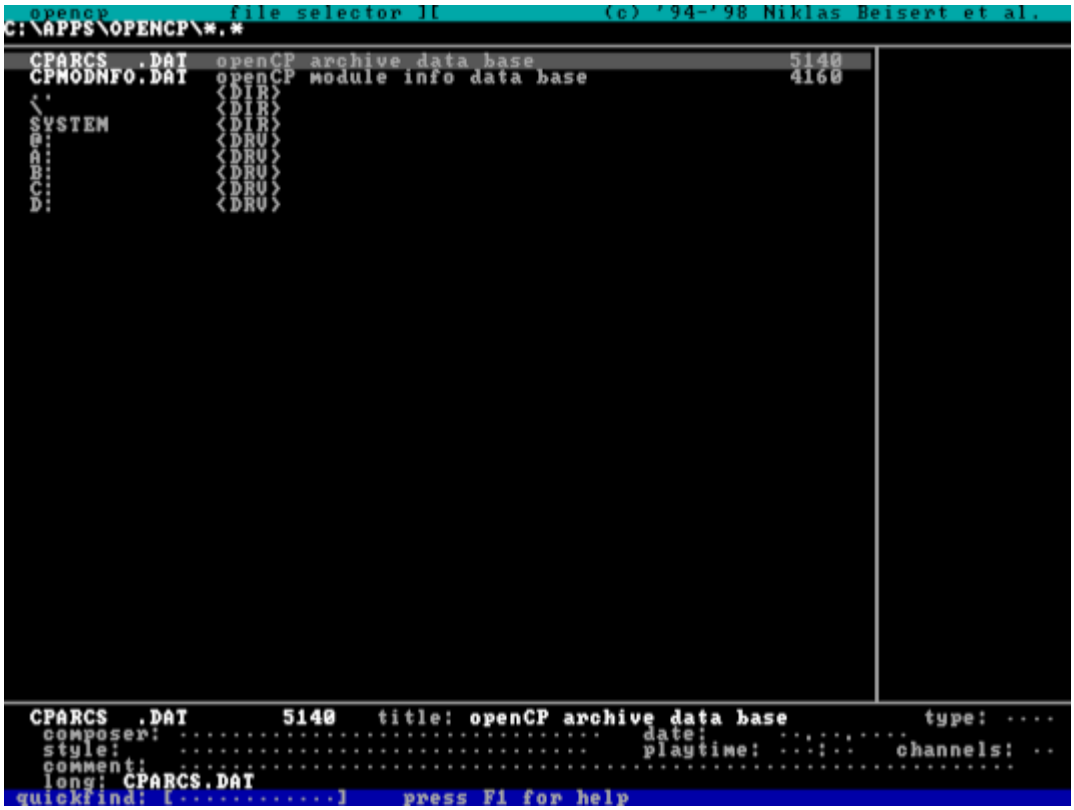
to demonstrate how he pronounces "Linux." We don't distribute the *FreeDOS* audio clip in FreeDOS itself, but you are welcome to download it from our Silly Sounds directory, found in the FreeDOS files archive at Ibiblio.

You can listen to the *FreeDOS* audio clip using the Open Cubic Player. To run Open Cubic Player, you normally would run `CP` from the `\APPS\OPENCP` directory. However, Open Cubic Player is a 32-bit application that requires a 32-bit DOS extender. A common DOS extender is DOS/4GW. While free to use, DOS/4GW is not an open source program, so we do not distribute it as a FreeDOS package.

Instead, FreeDOS provides another open source 32-bit extender called DOS/32A. If you did not install everything when you installed FreeDOS, you may need to install it using FDIMPLES. I used these two commands to move into the `\APPS\OPENCP` directory, and to run Open Cubic Player using the DOS/32A extender:

```
CD \APPS\OPENCP
DOS32A CP
```

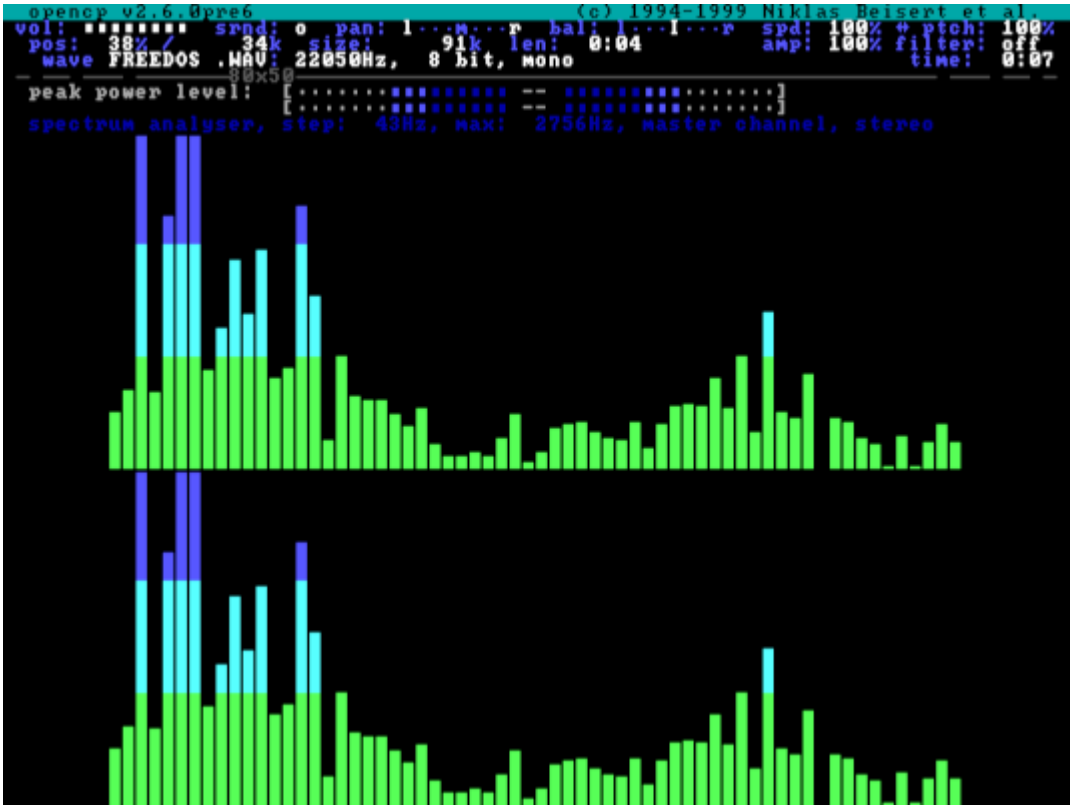
Open Cubic Player doesn't sport a fancy user interface, but you can use the arrow keys to navigate the *File Selector* to the directory that contains the media file you want to play.



Open Cubic Player opens with a file selector
(Jim Hall, CC-BY SA 4.0)

The text appears smaller than in other DOS applications because Open Cubic Player automatically changes the display to use 50 lines of text, instead of the usual 25 lines. Open Cubic Player will reset the display back to 25 lines when you exit the program.

When you have selected your media file, Open Cubic Player will play it in a loop. (Press the Esc key on your keyboard to quit.) As the file plays over the speakers, Open Cubic Player displays a spectrum analyzer so you can see the audio for the left and right channels. The *FreeDOS* audio clip is recorded in mono, so the left and right channels are the same.



Open Cubic Player playing the "FreeDOS" audio clip
(Jim Hall, CC-BY SA 4.0)

DOS may be from an older era, but that doesn't mean you can't use FreeDOS to run modern tasks or play current media. If you like to listen to digital music, try using Open Cubic Player or Mplayer on FreeDOS.

Program on FreeDOS with Bywater BASIC

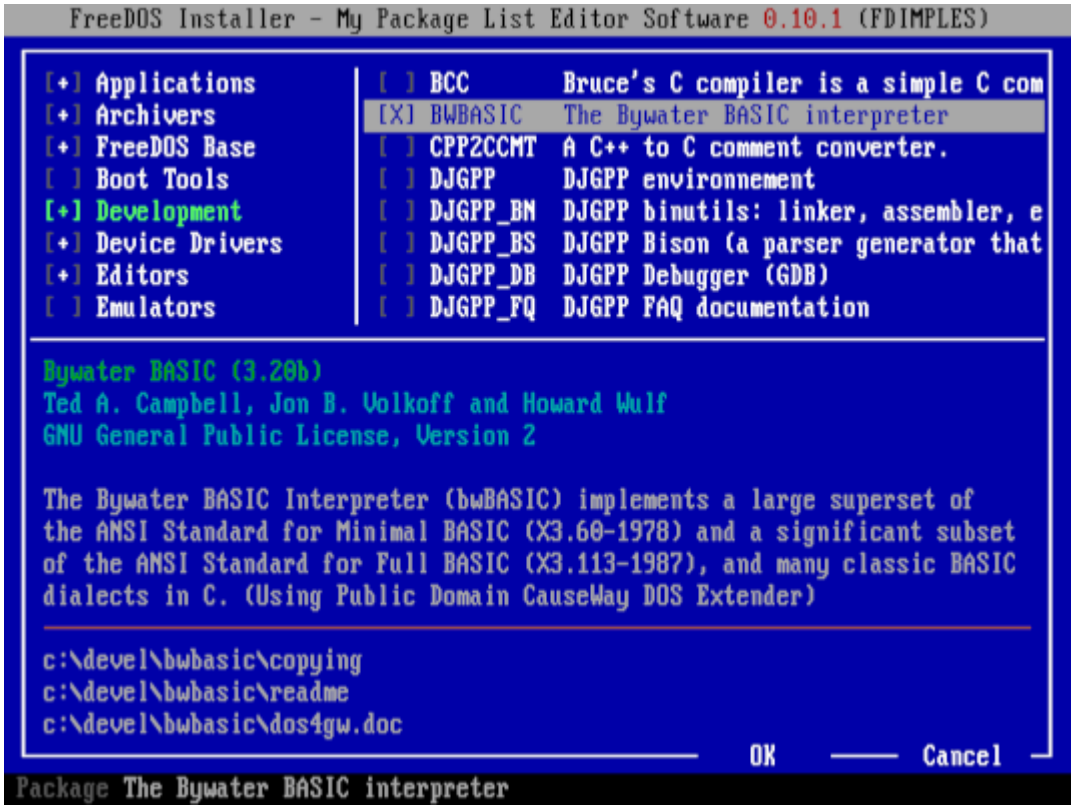
By Jim Hall

In the early days of personal computing—from the late 1970s and through the 1980s—many people got their start with BASIC programming. BASIC was a universal programming language that came built into most personal computers, from Apple to IBM PCs.

When we started the FreeDOS Project in June 1994, it seemed natural that we should include an open source BASIC environment. I was excited to discover one already existed in Bywater BASIC.

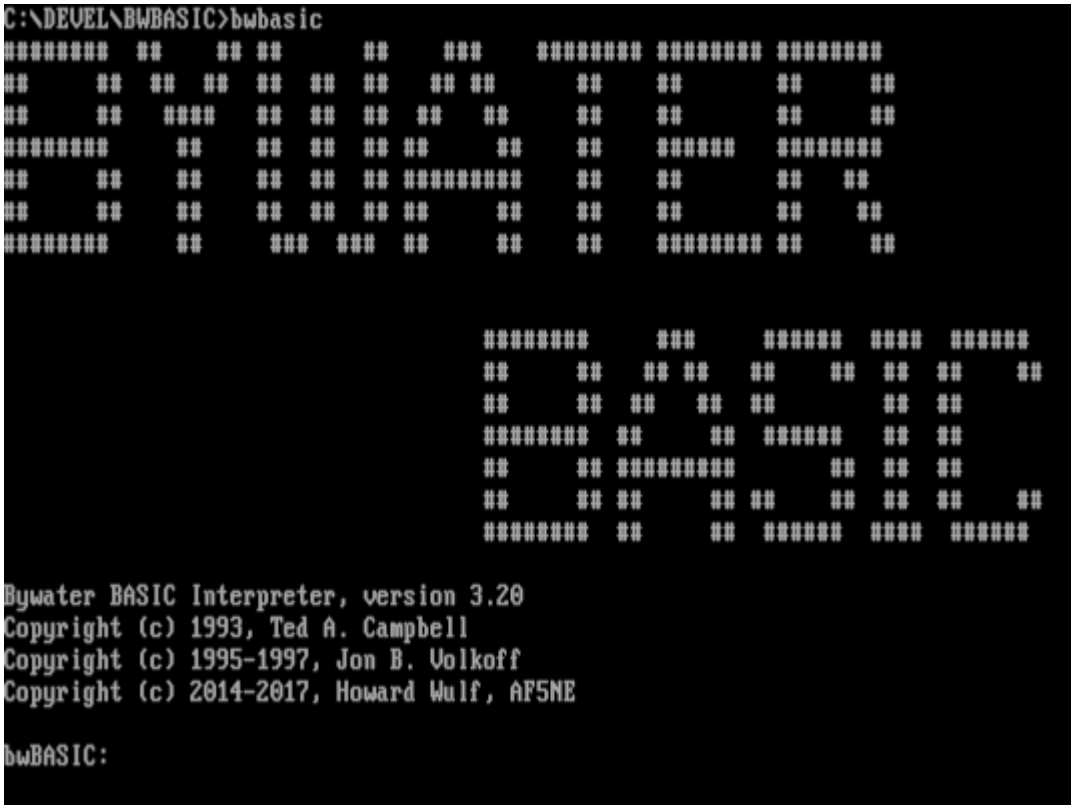
The Bywater BASIC website reminds us that Bywater BASIC implements a large superset of the ANSI Standard for Minimal BASIC (X3.60-1978) and a significant subset of the ANSI Standard for Full BASIC (X3.113-1987). It's also distributed under the GNU General Public License version 2, which means it's open source software. We only want to include open source programs in FreeDOS, so Bywater BASIC was a great addition to FreeDOS in our early days.

We've included Bywater BASIC since at least FreeDOS Alpha 5, in 1997. You can find Bywater BASIC in FreeDOS in the "Development" package group on the Bonus CD. Load this:



Installing Bywater BASIC on FreeDOS
(Jim Hall, CC-BY SA 4.0)

FreeDOS installs the Bywater BASIC package in the `\DEVEL\BWBASIC` directory. Change to this directory with `CD \DEVEL\BWBASIC` and type `BWBASIC` to run the Bywater BASIC interpreter.

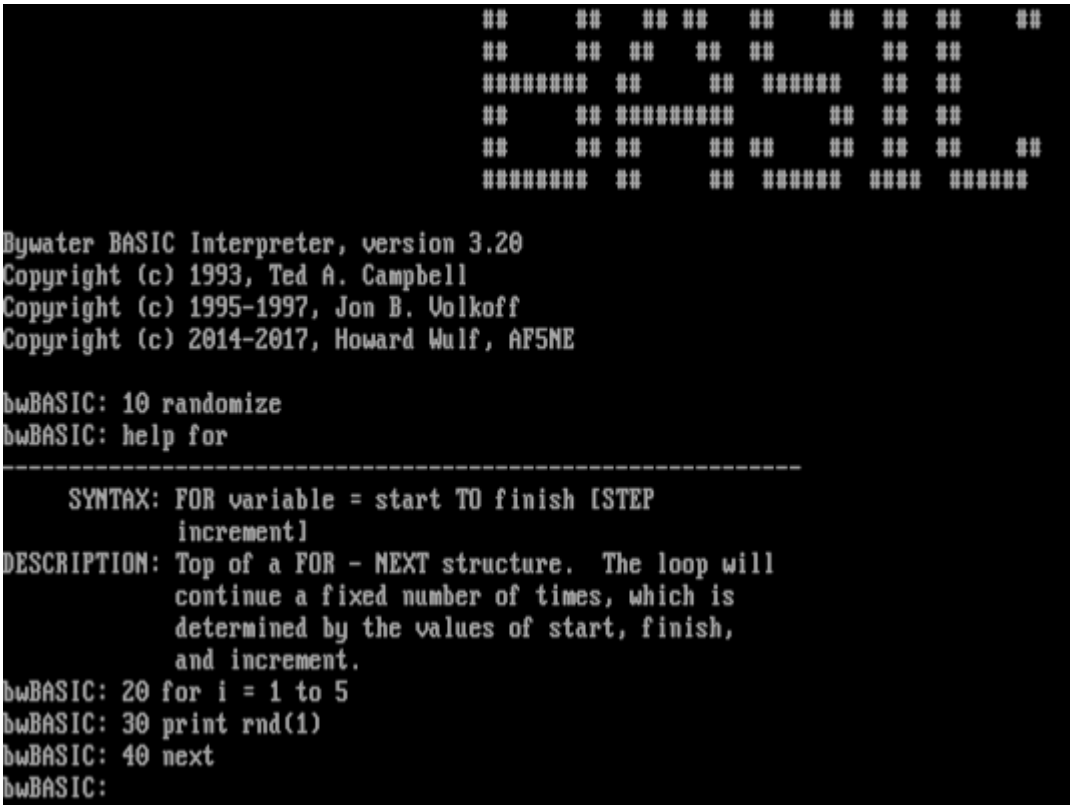


The Bywater BASIC intepreter
(Jim Hall, CC-BY SA 4.0)

Writing a sample program

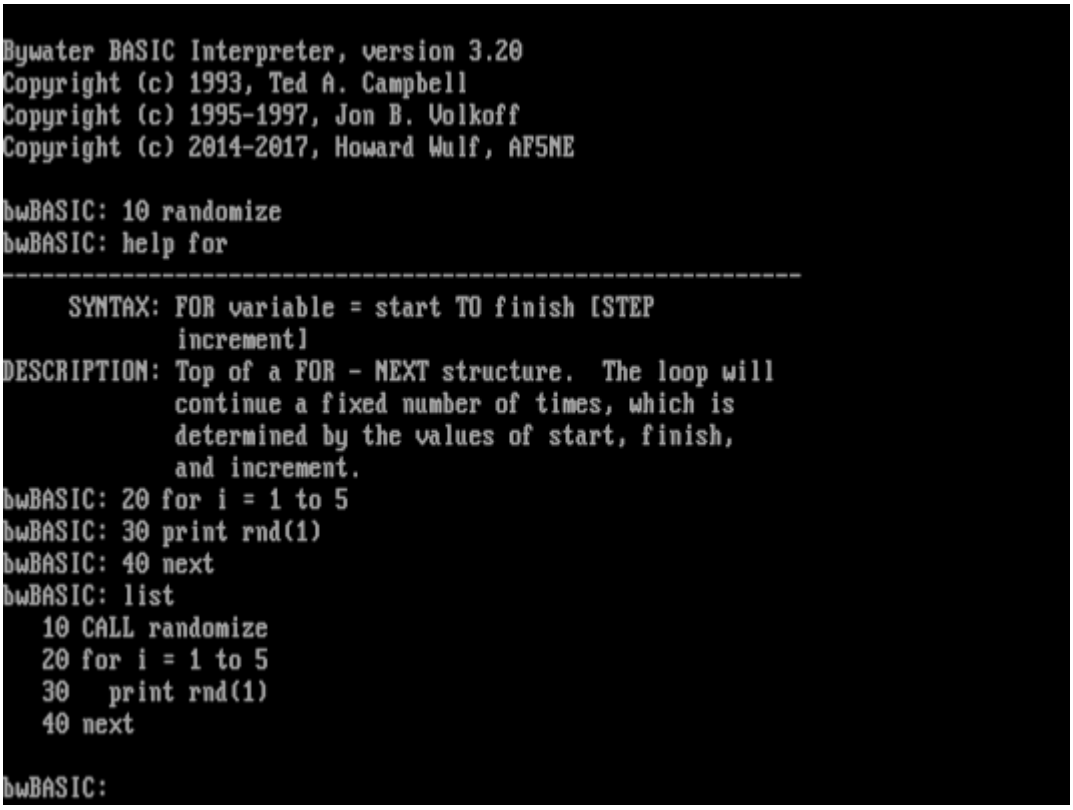
Let me demonstrate Bywater BASIC by writing a test program. We'll keep this simple—print five random numbers. This requires only a few constructs—a loop to iterate over five values and a random number generator. BASIC uses the `RND(1)` statement to generate a random value between 0 and 1. We can use `PRINT` to display the random number.

One feature I like in Bywater BASIC is the integrated "help" system. There's nothing more frustrating than forgetting the syntax for a BASIC statement. For example, I always forget how to create BASIC loops. Do I use `FOR I IN 1 TO 10` or `FOR I = 1 TO 10`? Just type `help FOR` at the Bywater BASIC prompt and the interpreter displays the usage and a brief description.



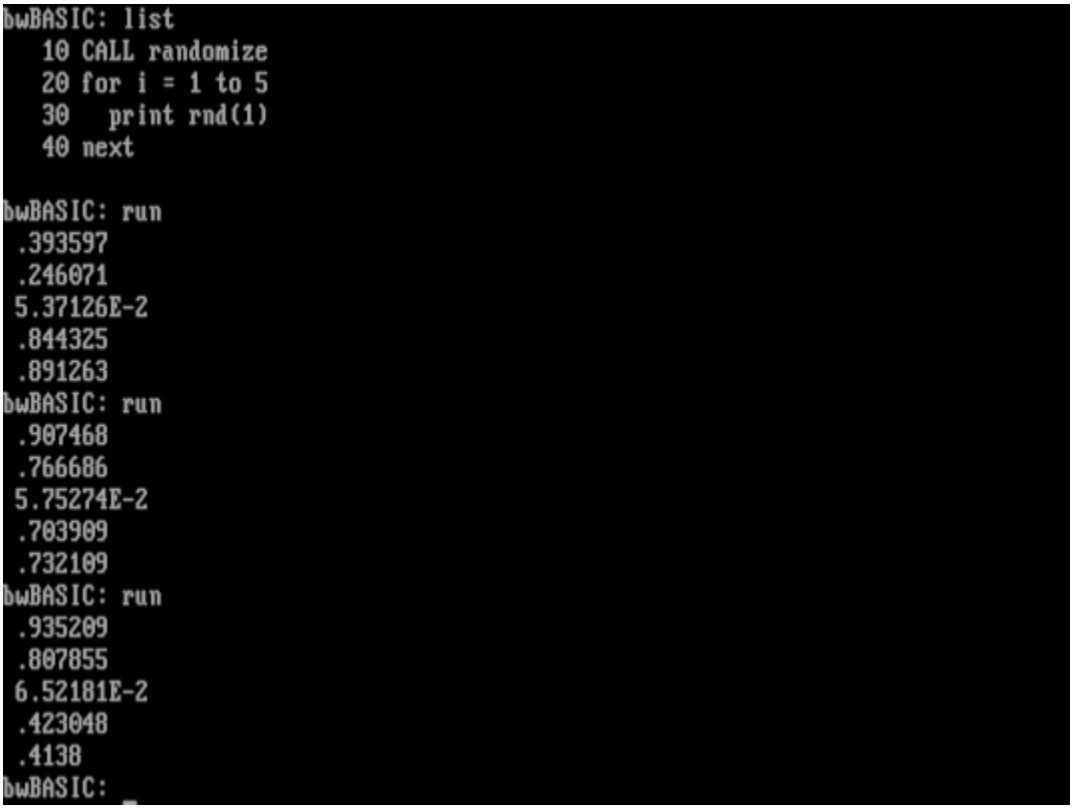
Use the "help" system as a quick-reference guide
(Jim Hall, CC-BY SA 4.0)

Another neat feature in Bywater BASIC is how it reformats your BASIC instructions, so they are easier to read. After typing my brief program, I can type `list` to see the full source listing. Bywater BASIC automatically adds the `CALL` keyword to my `RANDOMIZE` statement on line 10 and indents the `PRINT` statement inside my loop. These small changes help me to see loops and other features in my program, which can aid in debugging.



Bywater BASIC automatically reformats your code
(Jim Hall, CC-BY SA 4.0)

If everything looks okay, then type `RUN` to execute the program. Because I used the `RANDOMIZE` statement at the start of my BASIC program, Bywater *seeds* the random number generator with a random starting point. This ensures that my numbers are actually random values and don't repeat when I re-run my program.



Generating lists of random numbers
(Jim Hall, CC-BY SA 4.0)

Install Bywater BASIC on your FreeDOS system and start experimenting with BASIC programming. BASIC can be a great first programming language, especially if you are interested in getting back to the "roots" of personal computing. You can find more information about Bywater BASIC in the manual, installed in the `\DEVEL\BWBASIC` directory as `BWBASIC.DOC`. You can also explore the online "help" system by typing `HELP` at the Bywater BASIC prompt.

Why I love programming on FreeDOS with GW-BASIC

By Jim Hall

When I was growing up, it seemed every "personal computer" from the TRS-80 to the Commodore to the Apple let you write your own programs in the Beginners' All-purpose Symbolic Instruction Code (BASIC) programming language. Our family had a clone of the Apple II called the Franklin ACE 1000, which—as a clone—also ran AppleSoft BASIC. I took to AppleSoft BASIC right away and read books and magazines to teach myself about BASIC programming.

Later, our family upgraded to an IBM PC running DOS. Just like every personal computer before it, the IBM PC also ran its own version of DOS, called BASICA. Later versions of DOS replaced BASIC with an updated interpreter called GW-BASIC.

BASIC was my entry into computer programming. As I grew up, I learned other programming languages. I haven't written BASIC code in years, but I'll always have a fondness for BASIC and GW-BASIC.

Microsoft open-sources GW-BASIC

In May 2020, Microsoft surprised everyone (including me) by releasing the source code to GW-BASIC. Rich Turner (Microsoft) wrote in the announcement on the Microsoft Developer Blog:

Since re-open-sourcing MS-DOS 1.25 & 2.0 on GitHub last year, we've received numerous requests to also open-source Microsoft BASIC. Well, here we are! As clearly stated in the repo's readme, these sources are the 8088 assembly language sources from 10th Feb 1983 and are being open-sourced for historical reference and educational purposes. This means we will not be accepting PRs (Pull Requests) that modify the source in any way.

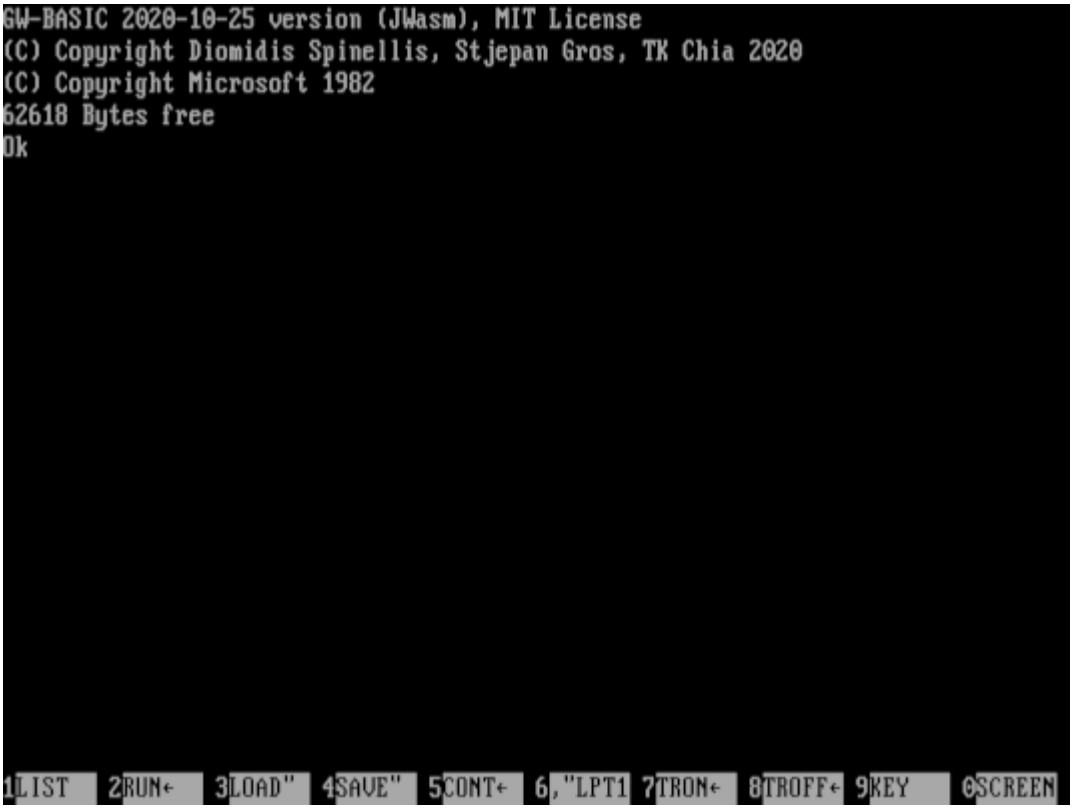
You can find the GW-BASIC source code release at the GW-BASIC GitHub. And yes, Microsoft used the MIT License, which makes this open source software.

Unfortunately, the GW-BASIC code was entirely in Assembly, which wouldn't build with modern tools. But open source developers got to work on that and adjusted the code to assemble with updated DOS assemblers. One project is TK Chia's GitHub project to update GW-BASIC to assemble with JWASM or other assemblers. You can find several source and binary releases on TK Chia's project. The notes from the latest version (October 2020) mention that this is a 'pre-release' binary of GW-BASIC as rebuilt in 2020 and that support for serial port I/O is missing. Light pen input, joystick input, and printer (parallel port) output need more testing. But if you don't need those extra features in GW-BASIC, you should be able to use this latest release to get back into BASIC programming with an open-sourced GW-BASIC.

FreeDOS doesn't include GW-BASIC, but installing it is pretty easy. Just download the `gwbas-20201025.zip` archive file from TK Chia's October 2020 GW-BASIC release, and extract it (unzip it) on your FreeDOS system. The binary archive uses a default path of `\DEVEL\GWBASIC`.

Getting started with GW-BASIC

To start GW-BASIC, run the `GWBASIC.EXE` program from the DOS command line. Note that DOS is *case insensitive* so you don't actually need to type that in all uppercase letters. Also, DOS will run any `EXE` or `COM` or `BAT` programs automatically, so you don't need to provide the extension, either. Go into the `\DEVEL\GWBASIC` and type `GWBASIC` to run BASIC.



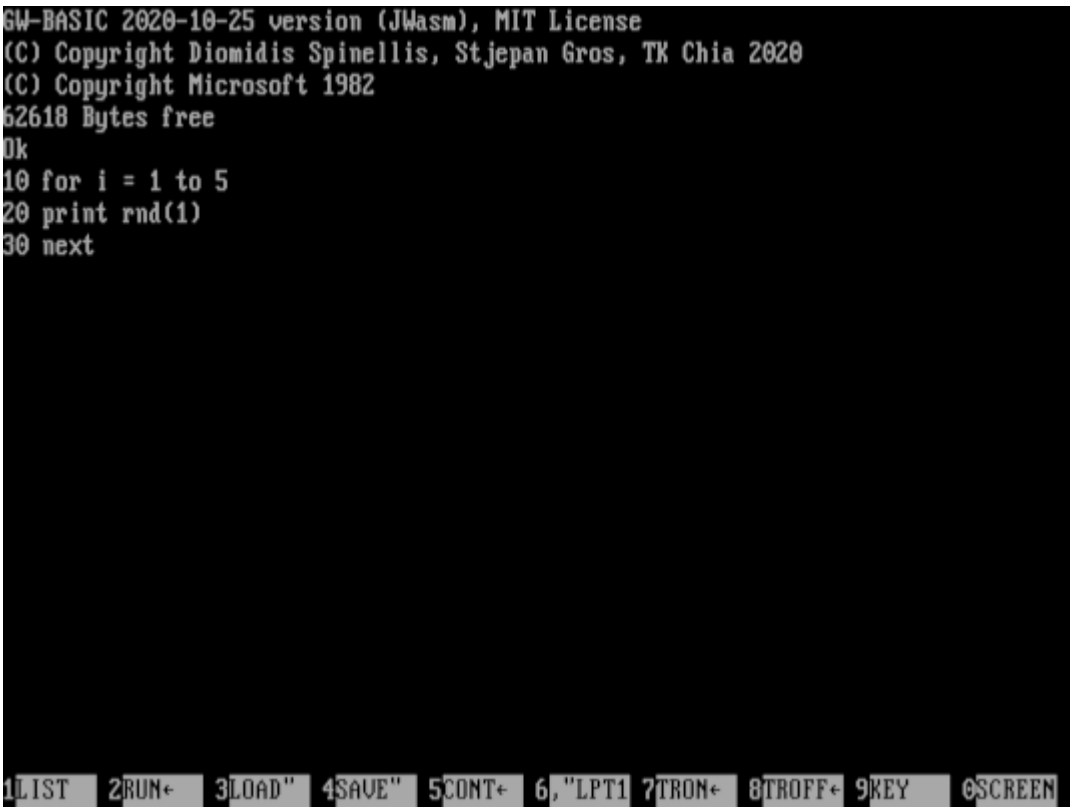
The GW-BASIC interpreter
(Jim Hall, CC-BY SA 4.0)

GW-BASIC is an *interpreted* programming language. The GW-BASIC environment is a "shell" that parses each line in your BASIC program *as it runs the code*. This is a little slower than *compiled* languages like C but makes for an easier coding-debugging cycle. You can test your code as you go, just by entering it into the interpreter.

Each line in a GW-BASIC program needs to start with a line number. GW-BASIC uses the line numbers to make sure it executes your program statements in the correct order. With these line numbers, you can later "insert" new program statements between two other statements by giving it a line number that's somewhere in between the other line numbers. For this reason, most BASIC programmers wrote line numbers that went up by tens so that line numbers would go like 10, 20, 30, and so on.

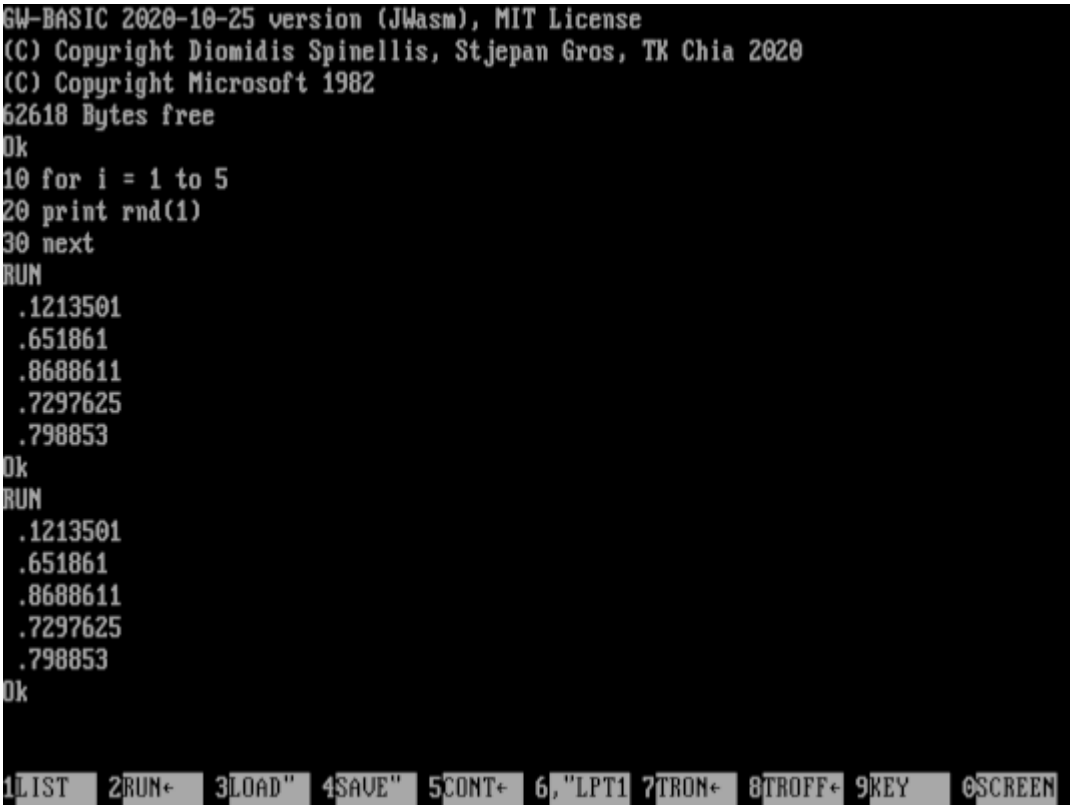
New to GW-BASIC? You can learn about the programming language by reading an online GW-BASIC reference. Microsoft didn't release a programming guide with the GW-BASIC source code, but you can search for one. Here's one reference that seems to be a copy of the original Microsoft GW-BASIC User's Guide.

Let's start with a simple program to print out a list of random numbers. The `FOR` statement creates a loop over a range of numbers, and `RND(1)` prints a random value between 0 and 1.



Entering our first program
(Jim Hall, CC-BY SA 4.0)

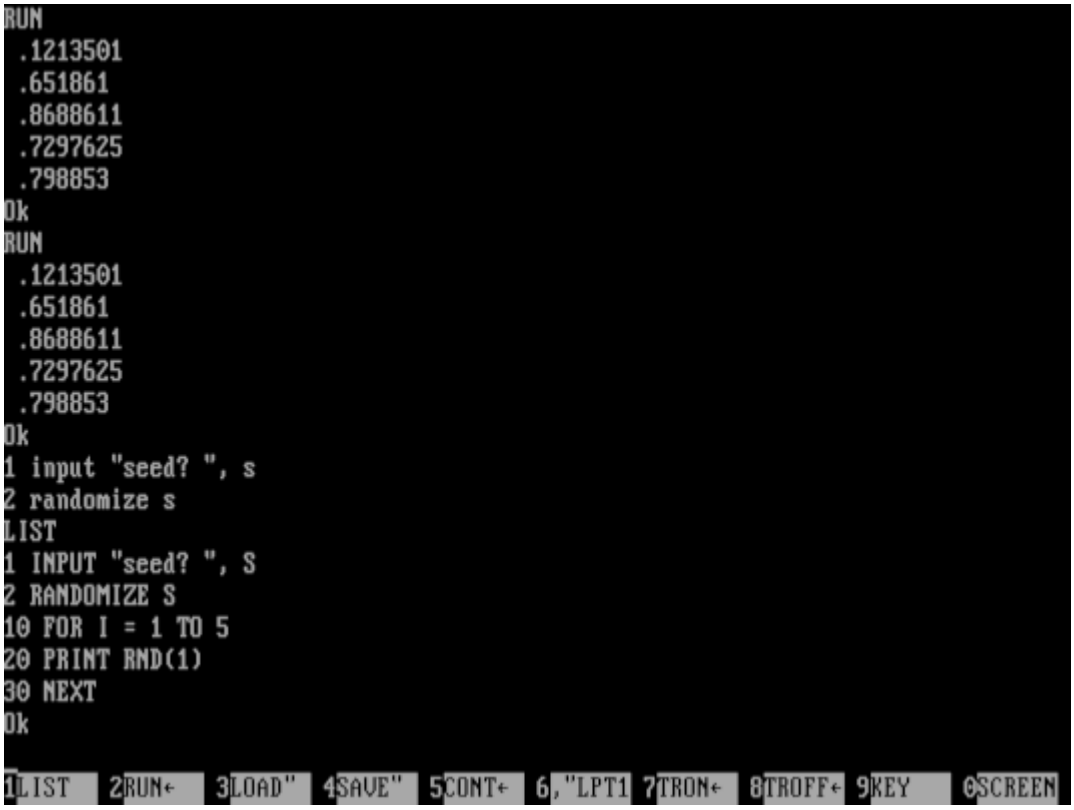
Do you see those highlighted words at the bottom of the screen? Those are keyboard shortcuts that you can access using the "F" keys (or *function* keys) on your keyboard. For example, `F1` will insert the word `LIST` into the GW-BASIC interpreter. The "left arrow" indicates that the shortcut will hit `Enter` for you, so `F2` will enter the `RUN` command and immediately execute it. Let's run the program a few times to see what happens.



The two lists of random numbers are the same
(Jim Hall, CC-BY SA 4.0)

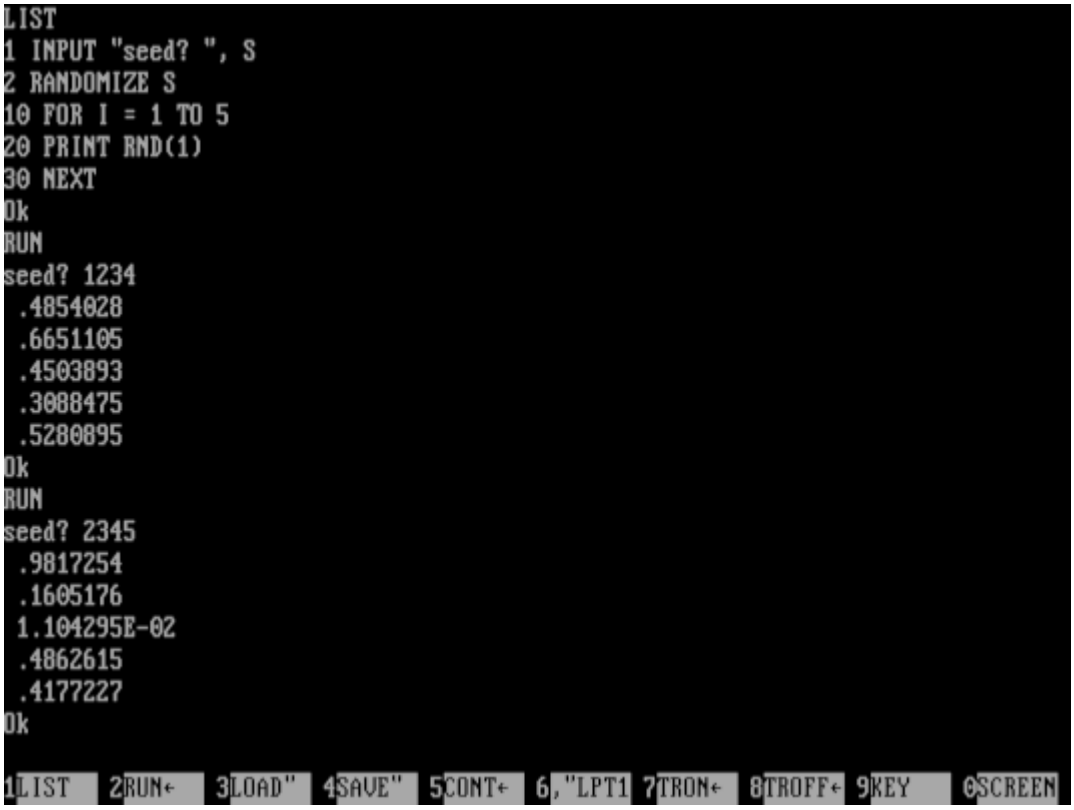
Interestingly, the list of random numbers is the same every time we run the BASIC program. That's because the GW-BASIC random number generator resets every time you execute a BASIC program.

To generate new random numbers every time, we need to "seed" the random number generator with a value. One way to do this is by prompting the user to enter their own seed, then use that value with the `RANDOMIZE` instruction. We can insert those two statements at the top of the program using line numbers 1 and 2. GW-BASIC will automatically add those statements before line 10.



Updating the program
(Jim Hall, CC-BY SA 4.0)

With the random number generator using a new seed, we get a different list of random numbers every time we run our program.



Now the lists of random numbers are different
(Jim Hall, CC-BY SA 4.0)

"Guess the number" game in GW-BASIC

Whenever I start learning a new programming language, I focus on defining variables, writing a statement, and evaluating expressions. Once I have a general understanding of those concepts, I can usually figure out the rest on my own. Most programming languages have some similarities, so once you know one programming language, learning the next one is a matter of figuring out the unique details and recognizing the differences.

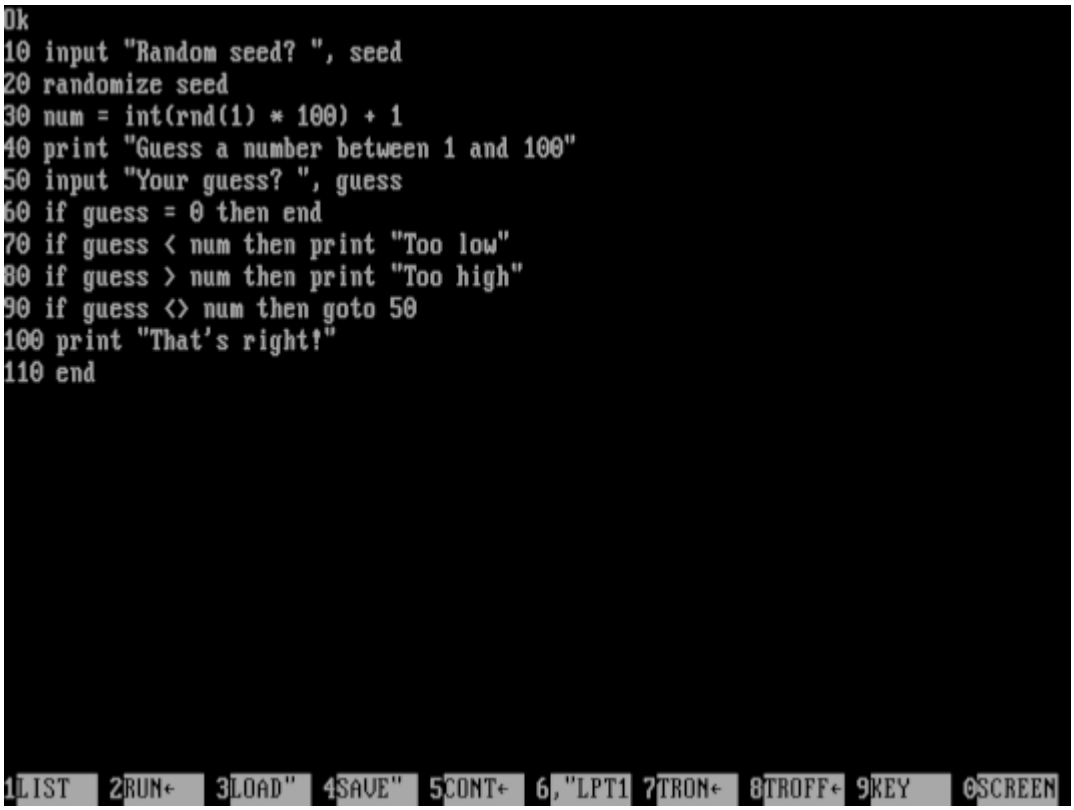
To help me practice a new programming language, I like to write a few test programs. One sample program I often write is a simple "guess the number" game, where the computer picks a number between one and 100 and asks me to guess it. The program loops until I guess correctly.

Let's write a version of this "guess the number" game in GW-BASIC. To start, enter the `NEW` instruction to tell GW-BASIC to forget the previous program and start a new one.

My "guess the number" program first prompts the user to enter a random number seed, then generates a random number between 1 and 100. The `RND(1)` function actually generates a

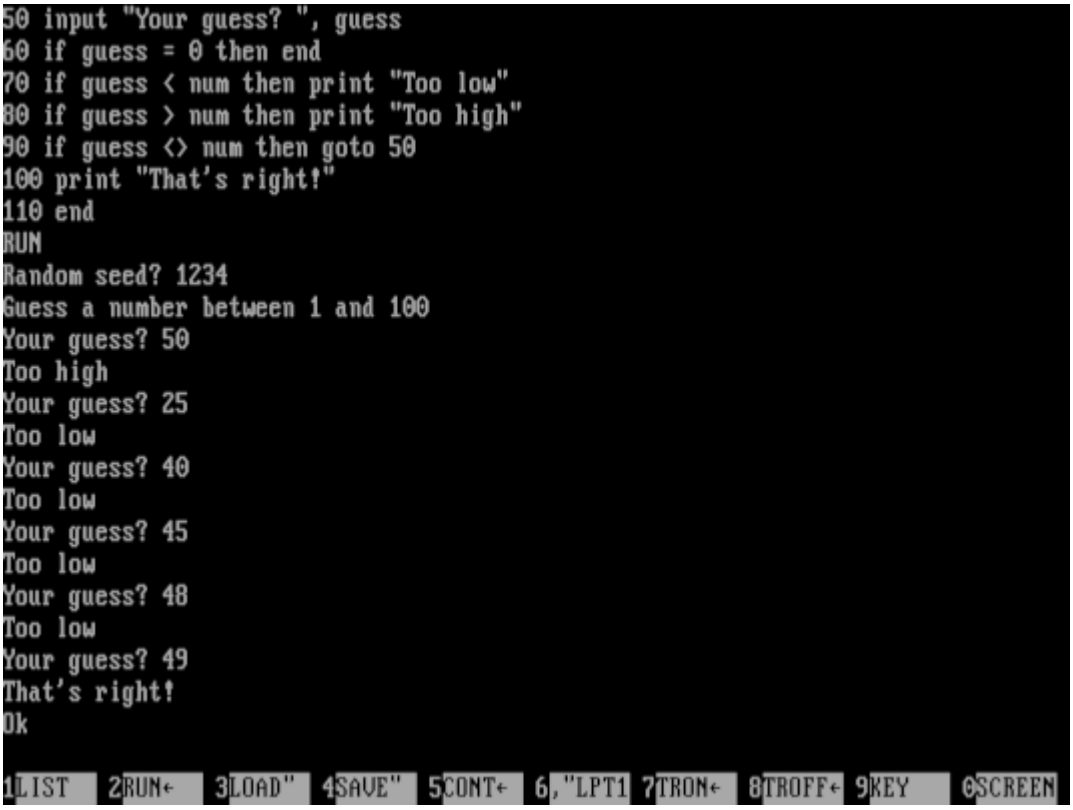
random value between 0 and 1 (actually 0.9999...) so I first multiply `RND(1)` by 100 to get a value between 0 and 99.9999..., then I turn that into an integer (remove everything after the decimal point). Adding 1 gives a number that's between 1 and 100.

The program then enters a simple loop where it prompts the user for a guess. If the guess is too low or too high, the program lets the user know to adjust their guess. The loop continues as long as the user's guess is *not* the same as the random number picked earlier.



Entering a "guess the number" program
(Jim Hall, CC-BY SA 4.0)

We can run the program by tapping the F2 key. Using a random seed of 1234 generates a completely new random number. It took me six guesses to figure out the secret number was 49.



Guessing the secret number

(Jim Hall, CC-BY SA 4.0)

And that's your first introduction to GW-BASIC programming! Thanks to Microsoft for releasing this great piece of history as open source software, and thanks also to the many open source developers who assembled GW-BASIC so we can run it.

One more thing before I go—It's not obvious how to exit GW-BASIC. The interpreter had a special instruction for that—to quit, enter `SYSTEM` and GW-BASIC will exit back to DOS.

Enter SYSTEM to quit GW-BASIC

(Jim Hall, CC-BY SA 4.0)

How to program in C on FreeDOS

By Jim Hall

When I first started using DOS, I enjoyed writing games and other interesting programs using BASIC, which DOS included. Much later, I learned the C programming language.

I immediately loved working in C! It was a straightforward programming language that gave me a ton of flexibility for writing useful programs. In fact, much of the FreeDOS core utilities are written in C and Assembly.

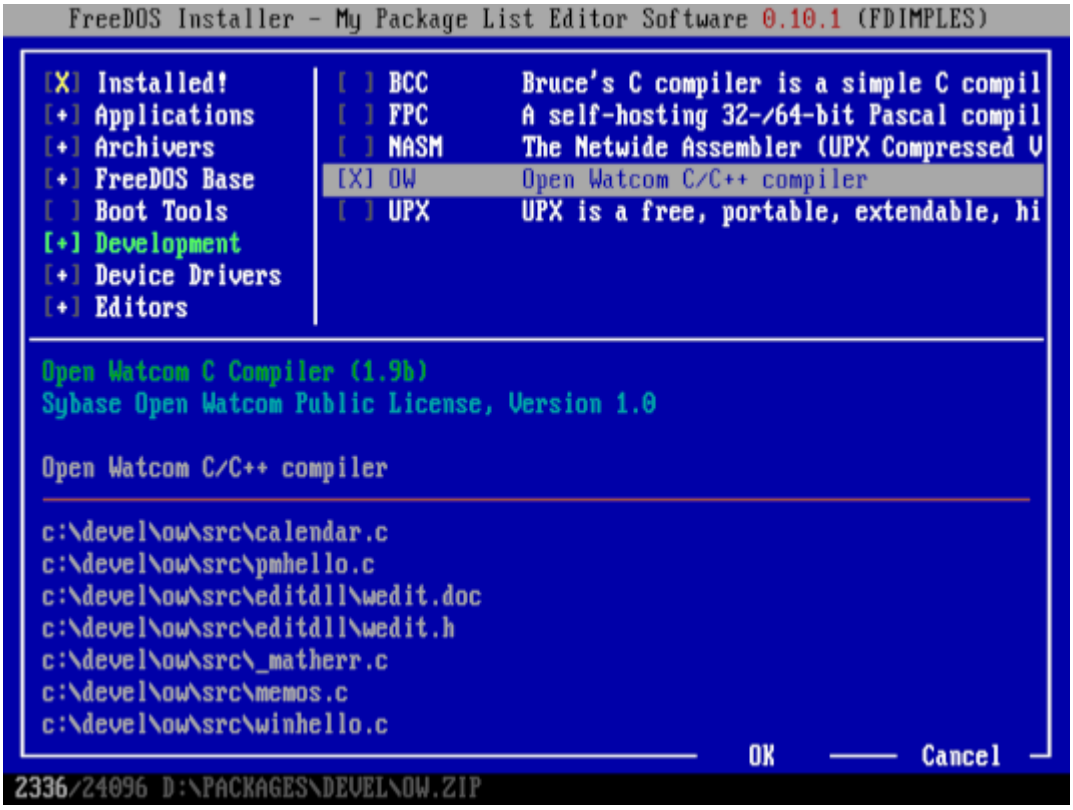
So it's probably not surprising that FreeDOS includes a C compiler—along with other programming languages. The FreeDOS LiveCD includes two C compilers—Bruce's C compiler (a simple C compiler) and the OpenWatcom C compiler. On the Bonus CD, you can also find DJGPP (a 32-bit C compiler based on GNU GCC) and the IA-16 port of GCC (requires a '386 or better CPU to compile, but the generated programs can run on low-end systems).

Programming in C on FreeDOS is basically the same as C programming on Linux, with two exceptions:

1. **You need to remain aware of how much memory you use.** Linux allows programs to use lots of memory, but FreeDOS is more limited. Thus, DOS programs used one of four memory models (large, medium, compact, and small) depending on how much memory they needed.
2. **You can directly access the console.** On Linux, you can create *text-mode* mode programs that draw to the terminal screen using a library like *ncurses*. But DOS allows programs to access the console and video hardware. This provides a great deal of flexibility in writing more interesting programs.

I like to write my C programs in the IA-16 port of GCC, or OpenWatcom, depending on what program I am working on. The OpenWatcom C compiler is easier to install since it's only a single package. That's why we provide OpenWatcom on the FreeDOS LiveCD, so you can install it automatically if you choose to do a "Full installation including applications and games"

when you install FreeDOS. If you opted to install a "Plain DOS system," then you'll need to install the OpenWatcom C compiler afterward, using the FDIMPLES package manager.



opensource.com

DOS C programming

You can find documentation and library guides on the OpenWatcom project website to learn all about the unique DOS C programming libraries provided by the OpenWatcom C compiler. To briefly describe a few of the most useful functions:

From `conio.h`:

- `int getch(void)`—Get a single keystroke from the keyboard
- `int getche(void)`—Get a single keystroke from the keyboard, and echo it

From `graph.h`:

- `_settextcolor(short color)`—Sets the color when printing text
- `_setbkcolor(short color)`—Sets the background color when printing text
- `_settextposition(short y, short x)`—Move the cursor to row `y` and column `x`

- `_outtext(char _FAR *string)`—Print a string directly to the screen, starting at the current cursor location

DOS only supports sixteen text colors and eight background colors. You can use the values 0 (Black) to 15 (Bright White) to specify the text colors, and 0 (Black) to 7 (White) for the background colors:

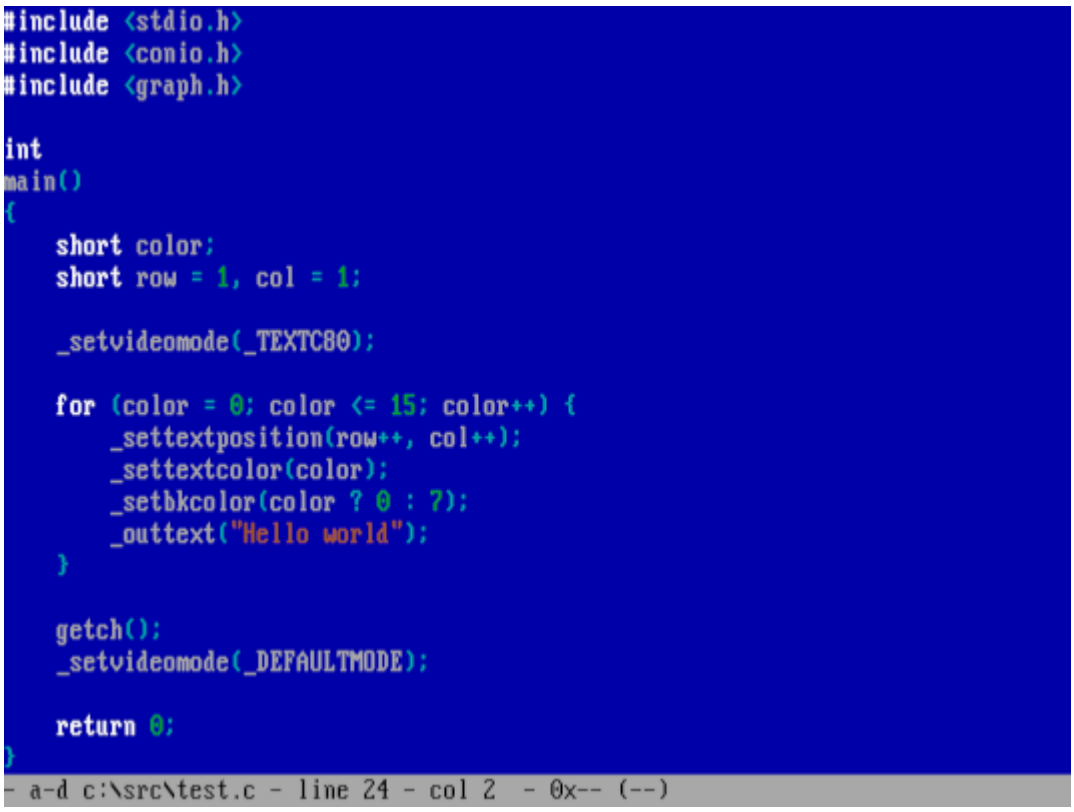
- **0**—Black
- **1**—Blue
- **2**—Green
- **3**—Cyan
- **4**—Red
- **5**—Magenta
- **6**—Brown
- **7**—White
- **8**—Bright Black
- **9**—Bright Blue
- **10**—Bright Green
- **11**—Bright Cyan
- **12**—Bright Red
- **13**—Bright Magenta
- **14**—Yellow
- **15**—Bright White

A fancy "Hello world" program

The first program many new developers learn to write is a program that just prints "Hello world" to the user. We can use the DOS "conio" and "graphics" libraries to make this a more interesting program and print "Hello world" in a rainbow of colors.

In this case, we'll iterate through each of the text colors, from 0 (Black) to 15 (Bright White). As we print each line, we'll indent the next line by one space. When we're done, we'll wait for the user to press any key, then we'll reset the screen and exit.

You can use any text editor to write your C source code. I like using a few different editors, including FreeDOS Edit and Freemacs, but more recently I've been using the FED editor because it provides *syntax highlighting*, making it easier to see keywords, strings, and variables in my program source code.



Before you can compile using OpenWatcom, you'll need to set up the DOS environment variables so OpenWatcom can find its support files. The OpenWatcom C compiler package includes a setup batch file that does this for you, as `\DEVEL\OW\OWSETENV.BAT`. Run this batch file to automatically set up your environment for OpenWatcom.

Once your environment is ready, you can use the OpenWatcom compiler to compile this "Hello world" program. I've saved my C source file as `TEST.C`, so I can type `WCL TEST.C` to compile and link the program into a DOS executable, called `TEST.EXE`. In the output messages from OpenWatcom, you can see that `WCL` actually calls the OpenWatcom C Compiler (`WCC`) to compile, and the OpenWatcom Linker (`WLINK`) to perform the object linking stage:

```
C:\SRC>wcl test.c
Open Watcom C/C++16 Compile and Link Utility Version 1.9
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc TEST.C
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
Open Watcom C16 Optimizing Compiler Version 1.9
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
TEST.C: 24 lines, included 1375, 0 warnings, 0 errors
Code size: 96
        wlink @__wcl__.lnk
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
Open Watcom Linker Version 1.9
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a DOS executable
C:\SRC>_
```

OpenWatcom prints some extraneous output that may make it difficult to spot errors or warnings. To tell the compiler to suppress most of these extra messages, use the `/Q` ("Quiet") option when compiling:

```
C:\SRC>wcl /q test.c
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
C:\SRC>_
```

If you don't see any error messages when compiling the C source file, you can now run your DOS program. This "Hello world" example is `TEST.EXE`. Enter `TEST` on the DOS command line to run the new program, and you should see this very pretty output:

C is a very efficient programming language that works well for writing programs on limited-resource systems like DOS. There's lots more that you can do by programming in C on DOS. If you're new to the C language, you can learn C yourself by following along in our Writing FreeDOS Programs in C self-paced ebook on the FreeDOS website, and the accompanying "how-to" video series on the FreeDOS YouTube channel.

Get started programming with conio

By Jim Hall

One of the reasons so many DOS applications sported a text user interface (or TUI) is because it was so easy to do. The standard way to control **con**sole **i**nput and **o**utput (**conio**) was with the `conio` library for many C programmers. This is a de-facto standard library on DOS, which gained popularity as implemented by Borland's proprietary C compiler as `conio.h`. You can also find a similar `conio` implementation in TK Chia's IA-16 DOS port of the GNU C Compiler in the `libi86` library of non-standard routines. The library includes implementations of `conio.h` functions that mimic Borland Turbo C++ to set video modes, display colored text, move the cursor, and so on.

For years, FreeDOS included the OpenWatcom C Compiler in the standard distributions. OpenWatcom supports its own version of `conio`, implemented in `conio.h` for particular console input and output functions, and in `graph.h` to set colors and perform other manipulation. Because the OpenWatcom C Compiler has been used for a long time by many developers, this `conio` implementation is also quite popular. Let's get started with the OpenWatcom `conio` functions.

Setting the video mode

Everything you do is immediately displayed on-screen via hardware. This is different from the `ncurses` library on Linux, where everything is displayed through terminal emulation. On DOS, everything is running on hardware. And that means DOS `conio` programs can easily access video modes and leverage screen regions in ways that are difficult using Linux `ncurses`.

To start, you need to set the *video mode*. On OpenWatcom, you do this with the `_setvideomode` function. This function takes one of several possible values, but for most programs that run in color mode in a standard 80x25 screen, use `_TEXTC80` as the mode.

```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  …
```

When you're done with your program and ready to exit back to DOS, you should reset the video mode back to whatever values it had before. For that, you can use `_DEFAULTMODE` as the mode.

```
 _setvideomode(_DEFAULTMODE);
  return 0;
}
```

Setting the colors

Every PC built after 1981's Color/Graphics Adapter supports 16 text colors and 8 background colors. Background colors are addressed with color indices 0 through 7, and text colors can be any value from 0 to 15:

0 Black	8 Bright Black
1 Blue	9 Bright Blue
2 Green	10 Bright Green
3 Cyan	11 Bright Cyan
4 Red	12 Bright Red
5 Magenta	13 Bright Magenta
6 Brown	14 Yellow
7 White	15 Bright White

You can set both the text color and the color behind it. Use the `_settextcolor` function to set the text "foreground" color and `_setbkcolor` to set the text "background" color. For example, to set the colors to yellow text on a red background, you would use this pair of functions:

```
 _settextcolor(14);
 _setbkcolor(4);
```

Positioning text

In `conio`, screen coordinates are always *row*,*col* and start with 1,1 in the upper-left corner. For a standard 80-column display with 25 lines, the bottom-right corner is 25,80.

Use the `_settextposition` function to move the cursor to a specific screen coordinate, then use `_outtext` to print the text you want to display. If you've set the colors, your text will use the colors you last defined, regardless of what's already on the screen.

For example, to print the text "FreeDOS" at line 12 and column 36 (which is more or less centered on the screen) use these two functions:

```
  _settextposition(12, 36);
  _outtext("FreeDOS");
```

Here's a small example program:

```
#include <conio.h>
#include <graph.h>
int
main()
{
    _setvideomode(_TEXTC80);
    _settextcolor(14);
    _setbkcolor(4);
    _settextposition(12, 36);
    _outtext("FreeDOS");
    getch();
    _setvideomode(_DEFAULTMODE);
    return 0;
}
```

Compile and run the program to see this output:

(CC BY-SA Jim Hall)

Text windows

The trick to unleashing the power of `conio` is to leverage a feature of the PC video display where a program can control the video hardware by region. These are called text windows and are a really cool feature of `conio`.

A text window is just an area of the screen, defined as a rectangle starting at a particular *row*,*col* and ending at a different *row*,*col*. These regions can take up the whole screen or be as small as a single line. Once you define a window, you can clear it with a background color and position text in it.

To define a text window starting at row 5 and column 10, and extending to row 15 and column 70, you use the `_settextwindow` function like this:

```
  _settextwindow(5, 10, 15, 70);
```

Now that you've defined the window, any text you draw in it uses 1,1 as the upper-left corner of the text window. Placing text at 1,1 will actually position that text at row 5 and column 10, where the window starts on the screen.

You can also clear the window with a background color. The `_clearscreen` function does double duty to clear either the full screen or just the window that's currently defined. To clear the entire screen, give the value `_GCLEARSCREEN` to the function. To clear just the window, use `_GWINDOW`. With either usage, you'll fill that region with whatever background color you last set. For example, to clear the whole screen with cyan (color 3) and a smaller text window with blue (color 1) you could use this code:

```
  _setbkcolor(3);
  _clearscreen(_GCLEARSCREEN);
  _settextwindow(5, 10, 15, 70);
  _setbkcolor(1);
  _clearscreen(_GWINDOW);
```

This makes it really easy to fill in certain areas of the screen. In fact, defining a window and filling it with color is such a common thing to do that I often create a function to do both at once. Many of my `conio` programs include some variation of these two functions to clear the screen or window:

```
#include <conio.h>
#include <graph.h>
void
clear_color(int fg, int bg)
{  _settextcolor(fg);
  _setbkcolor(bg);
  _clearscreen(_GCLEARSCREEN);
}
void
textwindow_color(int top, int left, int bottom, int right, int fg, int bg)
{
  _settextwindow(top, left, bottom, right);
  _settextcolor(fg);
  _setbkcolor(bg);
  _clearscreen(_GWINDOW);
}
```

A text window can be any size, even a single line. This is handy to define a title bar at the top of the screen or a status line at the bottom of the screen. Again, I find this to be such a useful addition to my programs that I'll frequently write functions to do it for me:

```
#include <conio.h>
#include <graph.h>
#include <string.h>                    /* for strlen */
```

```
void
clear_color(int fg, int bg)
{
  …
}
void
textwindow_color(int top, int left, int bottom, int right, int fg, int bg)
{
  …
}
void
print_header(int fg, int bg, const char *text)
{
  textwindow_color(1, 1, 1, 80, fg, bg);
  _settextposition(1, 40 - (strlen(text) / 2));
  _outtext(text);
}
void
print_status(int fg, int bg, const char *text)
{
  textwindow_color(25, 1, 25, 80, fg, bg);
  _settextposition(1, 1);
  _outtext(text);
}
```

Putting it all together

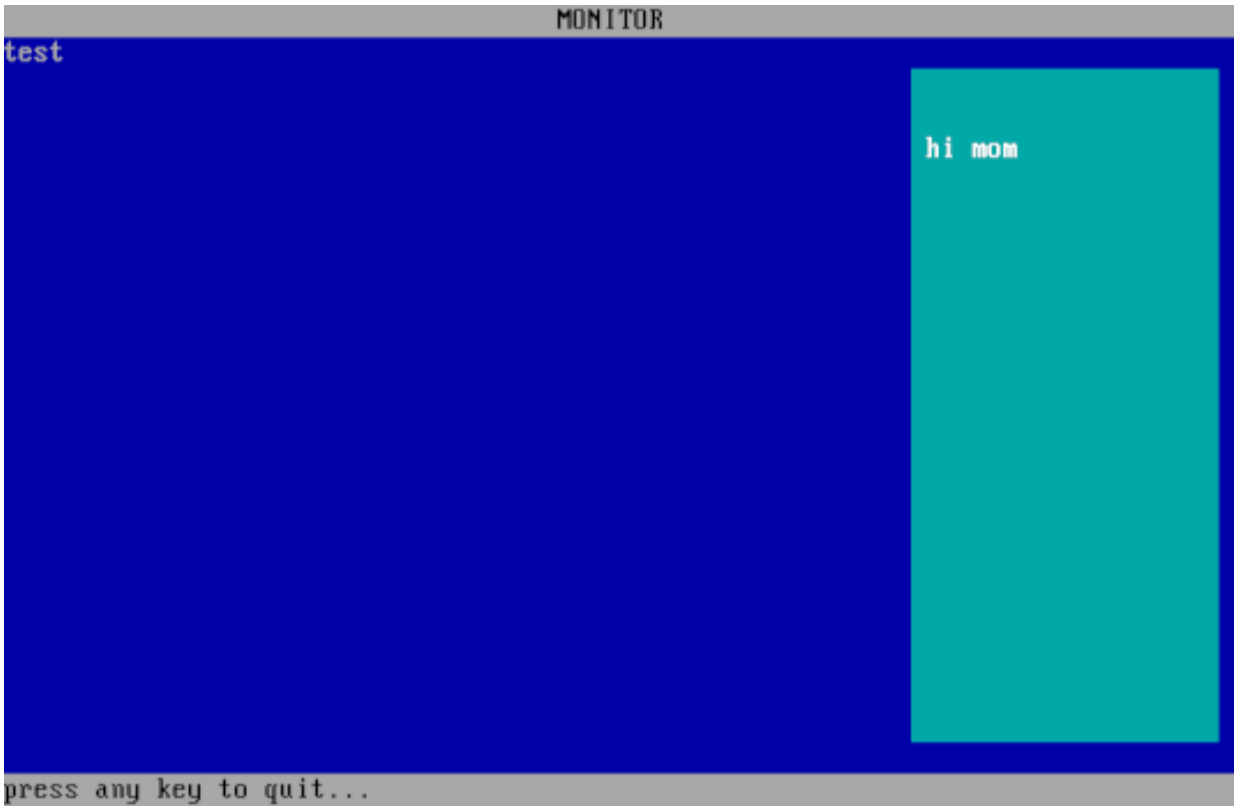
With this introduction to `conio`, and with the set of functions we've defined above, you can create the outlines of almost any program. Let's write a quick example that demonstrates how text windows work with `conio`. We'll clear the screen with a color, then print some sample text on the second line. That leaves room to put a title line at the top of the screen. We'll also print a status line at the bottom of the screen.

This is the basics of many kinds of applications. Placing a text window towards the right of the screen could be useful if you were writing a "monitor" program, such as part of a control system, like this:

```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  clear_color(7, 1);                    /* white on blue */
  _settextposition(2, 1);
```

```
  _outtext("test");
  print_header(0, 7, "MONITOR");       /* black on white */
  textwindow_color(3, 60, 23, 79, 15, 3);       /* br white on cyan */
  _settextposition(3, 2);
  _outtext("hi mom");
  print_status(0, 7, "press any key to quit...");       /* black on white */
  getch();
  _setvideomode(_DEFAULTMODE);
  return 0;
}
```

Having already written our own window functions to do most of the repetitive work, this program becomes very straightforward: clear the screen with a blue background, then print "test" on the second line. There's a header line and a status line, but the interesting part is in the middle where the program defines a text window near the right edge of the screen and prints some sample text. The `getch()` function waits for the user to press a key on the keyboard, useful when you need to wait until the user is ready:



Conio mon

We can change only a few values to completely change the look and function of this program. By setting the background to green and red text on a white window, we have the start of a solitaire card game:

```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  clear_color(7, 2);                  /* white on green */
  _settextposition(2, 1);
  _outtext("test");
  print_header(14, 4, "SOLITAIRE");    /* br yellow on red */
  textwindow_color(10, 10, 17, 22, 4, 7);       /* red on white */
  _settextposition(3, 2);
  _outtext("hi mom");
  print_status(7, 6, "press any key to quit...");       /* white on brown */
  getch();
  _setvideomode(_DEFAULTMODE);
  return 0;
}
```

You could add other code to this sample program to print card values and suits, place cards on top of other cards, and other functionality to create a complete game. But for this demo, we'll just draw a single "card" displaying some text:

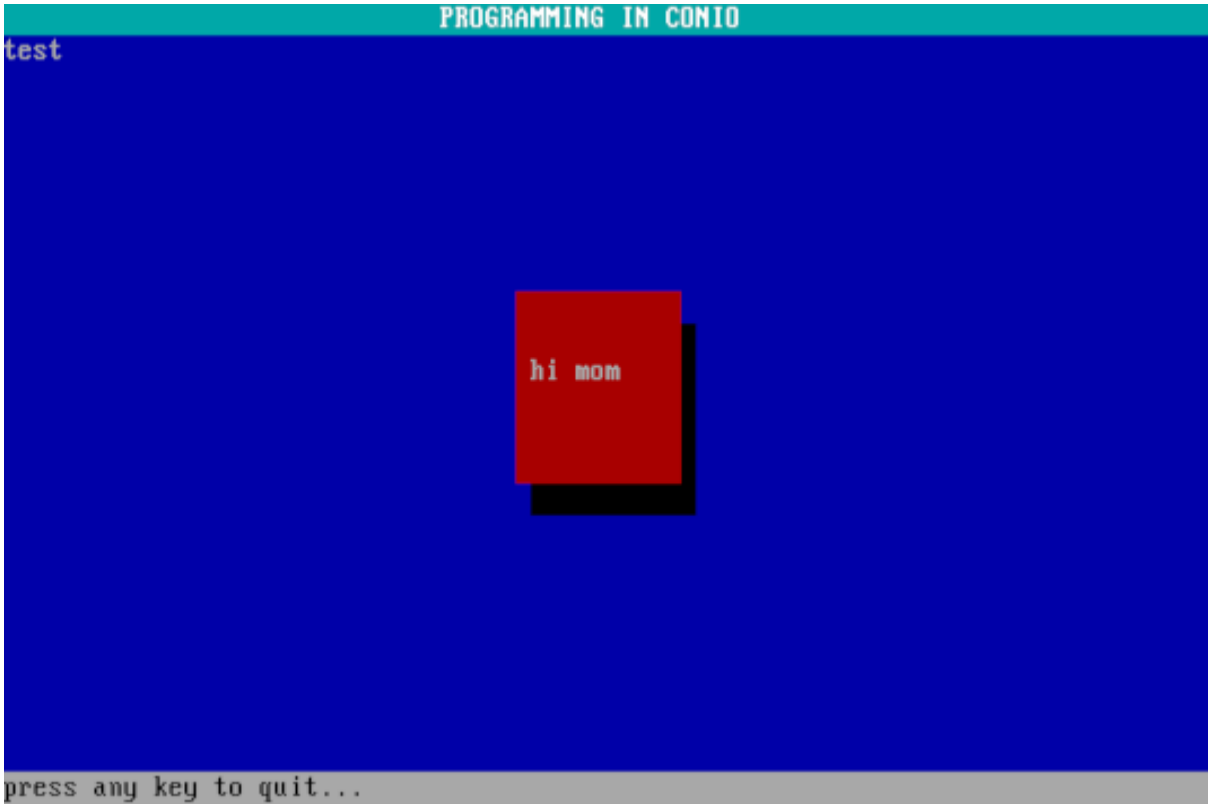
Conio solitaire

You can create other effects using text windows. For example, before drawing a message window, you could first draw a black window that's offset by one row and one column. The text window will appear to create a shadow over that area of the screen to the user. And we can do it all by changing only a few values in our sample program:

```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  clear_color(7, 1);                    /* white on blue */
  _settextposition(2, 1);
  _outtext("test");
  print_header(15, 3, "PROGRAMMING IN CONIO");  /* br white on cyan */
  textwindow_color(11, 36, 16, 46, 7, 0);       /* shadow */
  textwindow_color(10, 35, 15, 45, 7, 4);       /* white on red */
  _settextposition(3, 2);
  _outtext("hi mom");
  print_status(0, 7, "press any key to quit...");       /* black on white */
  getch();
```

```
  _setvideomode(_DEFAULTMODE);
  return 0;
}
```

You often see this "shadow" effect used in DOS programs as a way to add some visual flair:



Conio Window with shadow

The DOS `conio` functions can do much more than I've shown here, but with this introduction to `conio` programming, you can create various practical and exciting applications. Direct screen access means your programs can be more interactive than a simple command-line utility that scrolls text from the bottom of the screen. Leverage the flexibility of `conio` programming and make your next DOS program a great one.

Download the conio cheat sheet

As you explore programming with `conio`, it's helpful to have a list of common functions close at hand. I've created a double-sided cheat sheet with all the basics of `conio`, so **download it** and use it on your next `conio` project.

Why FreeDOS has 16 colors

By Jim Hall

If you've looked carefully at FreeDOS, you've probably noticed that text only comes in a limited range of colors—sixteen text colors, and eight background colors. This is similar to how Linux displays text color—you might be able to change *what text colors are used* in a Linux terminal, but you're still stuck with just sixteen text colors and eight background colors.

DOS text comes in 16 colors and 8 background colors
(Jim Hall, CC-BY SA 4.0)

Why does text only come in this limited palette, and why does FreeDOS use *those* colors and shades, instead of some other colors? The answer, like many things in technology, is because of *history*.

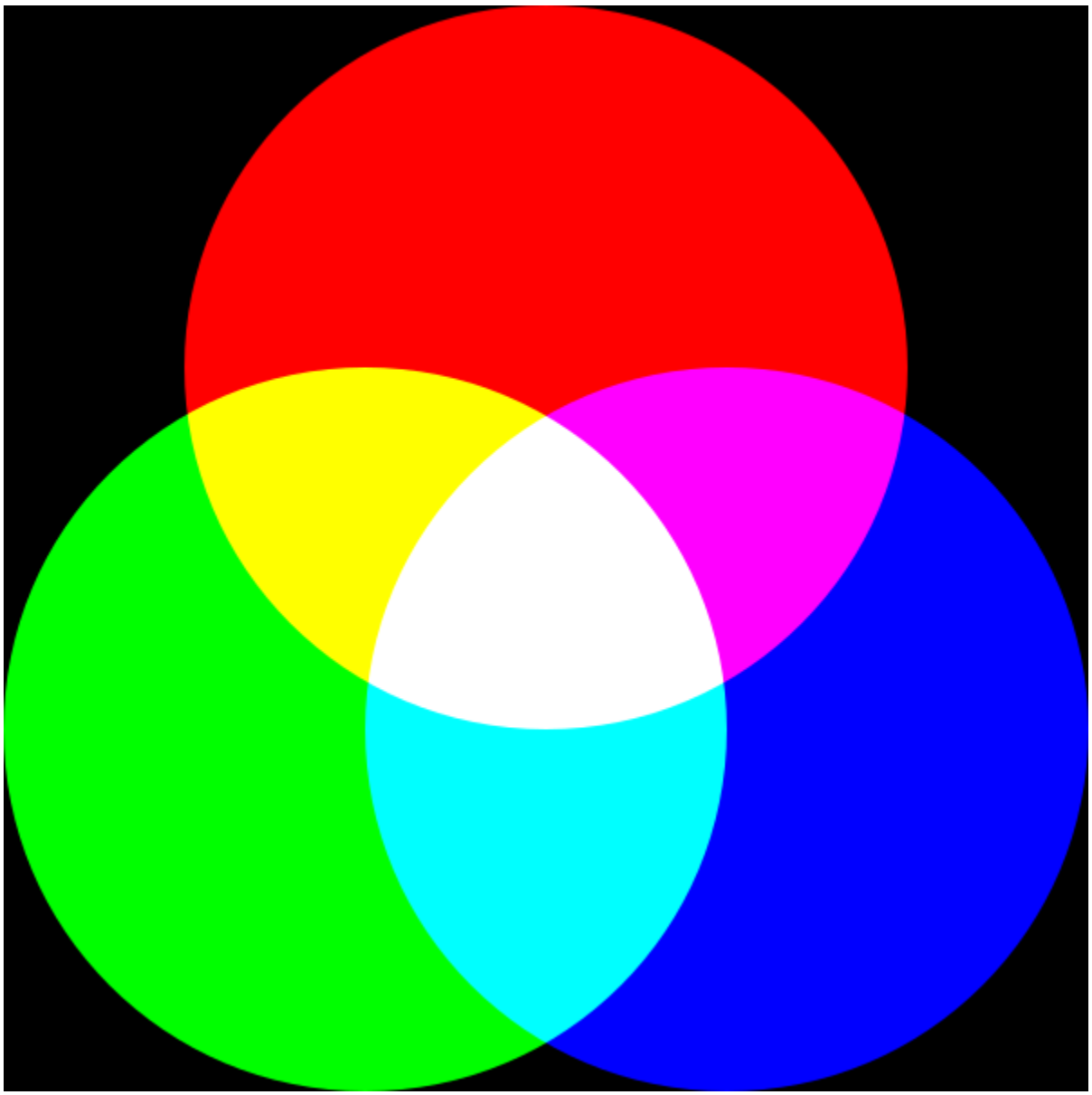
The origins of PC color

To explain why text only comes in sixteen colors, let me tell you a story about the first IBM Personal Computer. Parts of this story may be somewhat apocryphal, but the basics are close enough.

IBM released the Personal Computer 5150 (the "IBM PC") in 1981. The PC used a simple monitor screen that displayed text in green. Because this display only worked with one color, it was dubbed *monochrome* (the "IBM 5151 monochrome display," with the IBM Monochrome Display Adapter card, or "MDA").

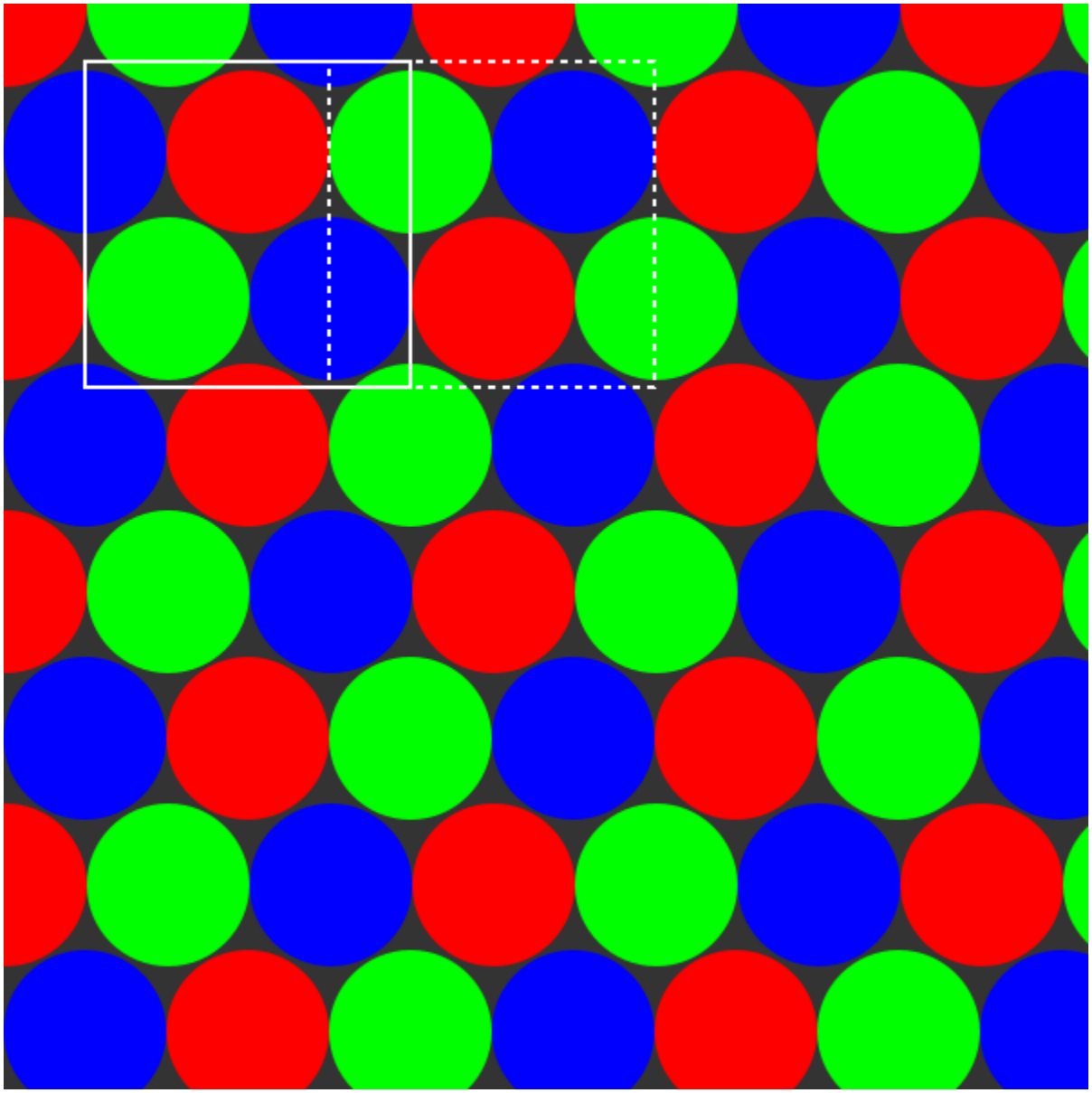
That same year, IBM released an updated version of the PC that sported an amazing technical achievement—color! The new IBM 5153 color display relied on a new IBM Color Graphics Adapter, or "CGA." And it is because of this original CGA that all DOS text inherited their colors.

But before we go there, we first need to understand something about color. When we talk about colors on a computer screen, we're talking about mixing different values of the three *primary* light colors—red, green, and blue. You can mix together different levels (or "brightnesses") of red, green, and blue light to create almost any color. Mix just red and blue light, and you get magenta. Mix blue and green, and you get cyan or aqua. Mix all colors equally, and you get white. Without any light colors, you see black (an absence of color).



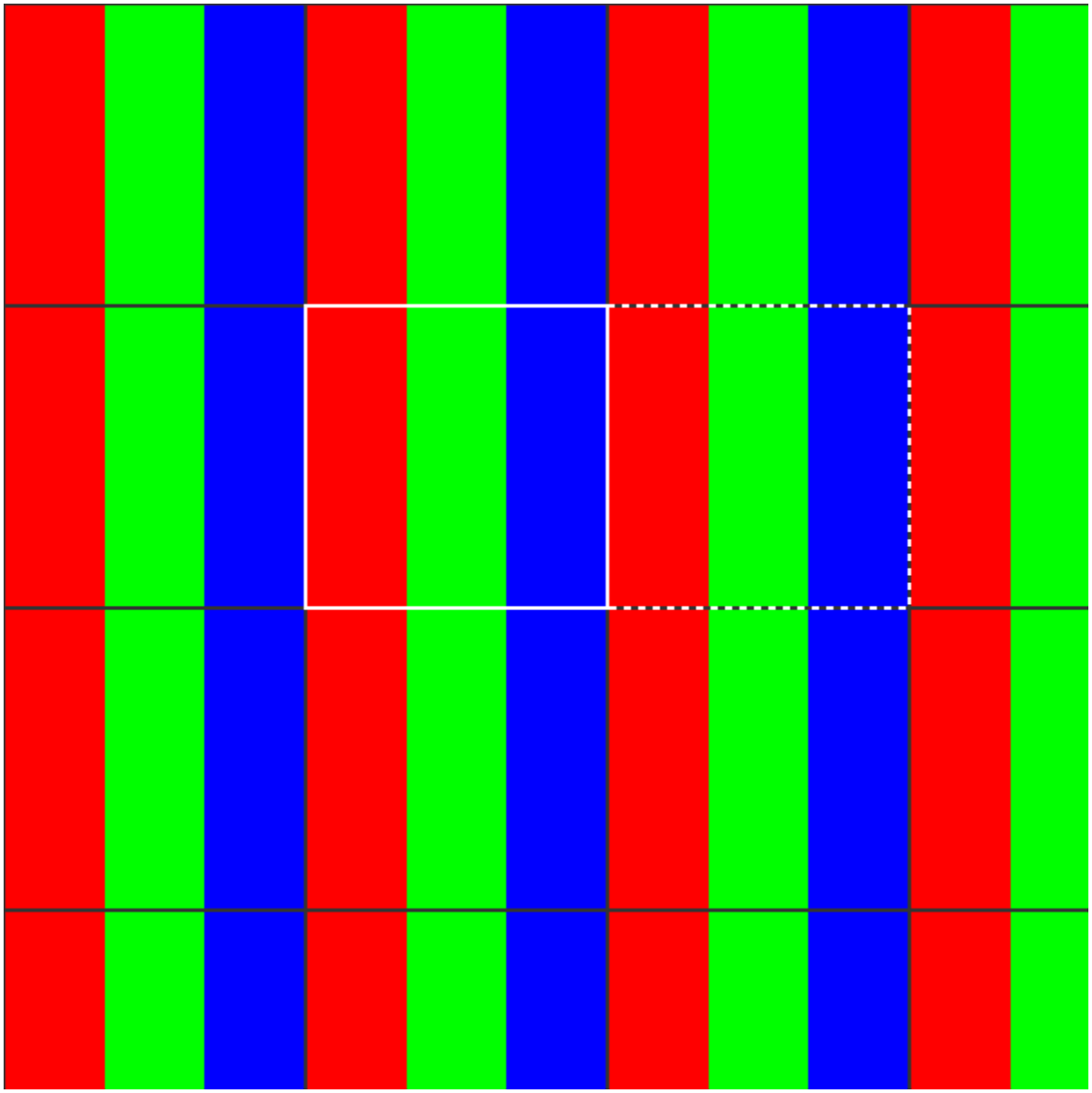
Mix red, green, and blue light to get different colors
(Jim Hall, CC-BY SA 4.0)

The IBM 5153 color display presented color to the user by lighting up tiny red, green, and blue phosphor dots on a cathode ray tube (a "CRT"). These tiny dots were arranged very close together and in a pattern where a triad of red, green, and blue dots would form a "pixel." By controlling which phosphor dots were lit at one time, the IBM 5153 color display could show different colored pixels.



Each red, green, and blue triad is a single pixel
(Jim Hall, CC-BY SA 4.0)

By the way, even modern displays use this combination of red, green, and blue dots to represent colors. The difference in modern computers is that instead of tiny phosphor dots, each pixel uses a triad of red, green, and blue LED lights—usually arranged side by side. The computer can turn each LED light on or off to mix the red, green, and blue colors in each pixel.



Each red, green, and blue triad is a single pixel
(Jim Hall, CC-BY SA 4.0)

Defining CGA colors

The IBM engineers realized they could display several colors by mixing each red, green, and blue pixels. In the simplest case, you could assume each red, green, and blue dot in a single-pixel was either "on" or "off." And as any computer programmer will tell you, you can represent "on" and "off" as binary—ones (1=on) and zeroes (0=off).

Representing red, green, and blue with ones or zeroes means you can combine up to eight colors, from 000 (red, green, and blue are all off) to 111 (red, green, and blue are all on). Note

that the bit pattern goes like "RGB," so RGB=001 is blue (only blue is on) and RGB=011 is cyan (both green and blue are on):

000 Black
001 Blue
010 Green
011 Cyan
100 Red
101 Magenta
110 Yellow
111 White

But that's just the simplest case. A particularly clever IBM engineer realized you could double the number of colors from eight to sixteen simply by adding another bit. So instead of a bit pattern like RGB, we can use a bit pattern like iRGB. We'll call this extra "i" bit the "intensity" bit because if we set the "intensity" bit to 1 (on), then we'll light up the red, green, and blue phosphor dots at full brightness; if the "intensity" bit is 0 (off) we can use some mid-level brightness.

And with that simple fix, now CGA could display sixteen colors! For the sake of simplicity, the IBM engineers referred to the high-intensity colors as the "bright" versions of the regular color names. So "red" pairs with "bright red," and "magenta" pairs with "bright magenta."

0000 Black
0001 Blue
0010 Green
0011 Cyan
0100 Red
0101 Magenta
0110 Yellow
0111 White

1000 Bright Black
1001 Bright Blue
1010 Bright Green
1011 Bright Cyan
1100 Bright Red
1101 Bright Magenta
1110 Bright Yellow
1111 Bright White

Oh no! But wait! This isn't actually sixteen colors. If you notice iRGB=0000 (black) and iRGB=1000 (bright black), they are both the same *black*. There's no color to make "bright," so they are just both regular black. This means we only have fifteen colors, not the sixteen we were hoping for.

But IBM has clever engineers working for them, and they realized how to fix this to get sixteen colors. Rather than implement a straight RGB to iRGB, IBM actually implemented a *modified* iRGB scheme. With this change, IBM set four levels of brightness for each

phosphor dot: completely off, one-third brightness, two-thirds brightness, and full brightness. If the "intensity" bit was turned off, then each red, green, and blue phosphor dot would light up at two-thirds brightness. If you set the "intensity" bit on, any zeroes in the RGB colors would be lit at one-third brightness, and any ones in the RGB colors would be lit at full brightness.

Let me describe this to you another way, using web color code representation. If you are familiar with the HTML colorspace, you probably know that you can represent colors using #RGB, where RGB represents a combination of red, green, and blue values, each between the hexadecimal values 0 through F. So using IBM's modified iRGB definition, iRGB=0001 is #00a (blue) and iRGB=1001 is #55f (bright blue) because with high-intensity colors, all zeroes in RGB=001 are lit at one-third brightness (around "5" on the 0 to F scale) and all ones in RGB=001 are lit at two-third brightness (about "A" on the 0 to F scale).

0000 Black	1000 Bright Black
0001 Blue	1001 Bright Blue
0010 Green	1010 Bright Green
0011 Cyan	1011 Bright Cyan
0100 Red	1100 Bright Red
0101 Magenta	1101 Bright Magenta
0110 Yellow	1110 Bright Yellow
0111 White	1111 Bright White

And with those colors, we are finally done! We have a full spectrum of colors from iRGB=0000 (black) to iRGB=1111 (bright white) and every color in between. Like a rainbow of colors, this is beautiful.

Except, no. Wait. Something's wrong here. We can't actually replicate all of the colors of the rainbow yet. The handy mnemonic we learned in grade school was ROYGBIV, to help us remember that a rainbow has colors from red, orange, yellow, green, blue, indigo, and violet. Our modified iRGB color scheme includes red, yellow, green, and blue—and we can "fake" it for indigo and "violet." But we're missing orange. Oh no!

A beautiful rainbow - which unfortunately contains orange
(Paweł Fijałkowski, public domain)

To fix this, the smart IBM engineers made one final fix for RGB=110. The high-intensity color (iRGB=1110) lit up the red and green phosphor dots at full brightness to make yellow. But at the low-intensity color (iRGB=0110), they lit the red at two-thirds brightness and the green at one-third brightness. This turned iRGB=0110 into an orange color—although it was later dubbed "brown" because IBM had to mess up the standard names somewhere.

0000 Black	1000 Bright Black
0001 Blue	1001 Bright Blue
0010 Green	1010 Bright Green
0011 Cyan	1011 Bright Cyan
0100 Red	1100 Bright Red
0101 Magenta	1101 Bright Magenta
0110 Brown	1110 Yellow
0111 White	1111 Bright White

And that's how CGA—and by extension, DOS—got the sixteen colors! And in case you're curious, that's also why there's a "bright black" color, even though it's just a shade of gray.

Representing colors (bits and bytes)

But you may wonder: why can DOS only display eight background colors if it can display sixteen text colors? For that, we need to take a quick diversion into how computers passed color information to the CGA card.

In brief, the CGA card expected each character's text color and background color to be encoded in a single byte packet. That's eight bits. So where do the eight bits come from?

We just learned how iRGB (four bits) generates the sixteen colors. Text color uses iRGB, or four bits. The background color is limited to the eight low-intensity colors (RGB, or three bits). Together, that makes only seven bits. Where is the missing eighth bit?

The final bit was reserved for perhaps the DOS era's most important user interface element—blinking text. While the blinking text might be annoying today, throughout the early 1980s, blinking text was the friendly way to represent critical information such as error messages.

Adding this "blink" bit to the three background color bits (RGB) and the four text color bits (iRGB) makes eight bits or a byte! Computers like to count in full bytes, making this a convenient way to package color (and blink) information to the computer.

Thus, the full byte to represent color (and blink) was `Bbbbffff`, where `ffff` is the iRGB bit pattern for the text color (from 0 to 15), `bbb` is the RGB bit pattern for the low-intensity background color (from 0 to 7), and `B` is the "blink" bit.

The limit of sixteen text colors and eight background colors continues to this day. Certainly, DOS is stuck with this color palette, but even Linux terminal emulators like GNOME Terminal remain constrained to sixteen text colors and eight background colors. Sure, a Linux terminal might let you change the specific colors used, but you're still limited to sixteen text colors and eight background colors. And for that, you can thank DOS and the original IBM PC. You're welcome!

Print a holiday greeting with ASCII art on Linux

By Jim Hall

Full-color ASCII art used to be quite popular on DOS, which could leverage the extended ASCII character set and its collection of drawing elements. You can add a little visual interest to your next FreeDOS program by adding ASCII art as a cool “welcome” screen or as a colorful “exit” screen with more information about the program.

But this style of ASCII art isn’t limited just to FreeDOS applications. You can use the same method in a Linux terminal-mode program. While Linux uses ncurses to control the screen instead of DOS's conio, the related concepts apply well to Linux programs. This article looks at how to generate colorful ASCII art from a C program.

An ASCII art file

You can use a variety of tools to draw your ASCII art. For this example, I used an old DOS application called TheDraw, but you can find modern open source ASCII art programs on Linux, such as Moebius (Apache license) or PabloDraw (MIT license). It doesn’t matter what tool you use as long as you know what the saved data looks like.

Here’s part of a sample ASCII art file, saved as C source code. Note that the code snippet defines a few values: `IMAGEDATA_WIDTH` and `IMAGEDATA_DEPTH` define the number of columns and rows on the screen. In this case, it’s an 80x25 ASCII art “image.” `IMAGEDATA_LENGTH` defines the number of entries in the `IMAGEDATA` array. Each character in the ASCII art screen can be represented by two bytes of data: The character to display and a color attribute containing both the foreground and background colors for the character. For an 80x25 screen, where each character is paired with an attribute, the array contains 4000 entries (that’s 80 * 25 * 2 = 4000).

```
#define IMAGEDATA_WIDTH 80
#define IMAGEDATA_DEPTH 25
#define IMAGEDATA_LENGTH 4000
unsigned char IMAGEDATA [] = {
    '.', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
    ' ', 0x08,  ' ', 0x08,  '.', 0x0F,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
    ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  '.', 0x0F,
    ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
    ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
```

and so on for the rest of the array.

To display this ASCII art to the screen, you need to write a small program to read the array and print each character with the right colors.

Setting a color attribute

The color attribute in this ASCII art file defines both the background and foreground color in a single byte, represented by hexadecimal values like `0x08` or `0x6E`. Hexadecimal turns out to be a compact way to express a color "pair" like this.

Character mode systems like ncurses on Linux or conio on DOS can display only sixteen colors. That's sixteen possible text colors and eight background colors. Counting sixteen values (from 0 to 15) in binary requires only four bits:

- `1111` is 16 in binary

And conveniently, hexadecimal can represent 0 to 15 with a single character: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. So the value `F` in hexadecimal is the number 15, or `1111` in binary.

With color pairs, you can encode both the background and foreground colors in a single byte of eight bits. That's four bits for the text color (0 to 15 or 0 to F in hexadecimal) and three bits for the background color (0 to 7 or 0 to E in hexadecimal). The leftover bit in the byte is not used here, so we can ignore it.

To convert the color pair or attribute into color values that your program can use, you'll need to use a bit mask to specify only the bits used for the text color or background color. Using the OpenWatcom C Compiler on FreeDOS, you can write this function to set the colors appropriately from the color attribute:

```
void
textattr(int newattr)
{
```

```
  _settextcolor(newattr & 15);         /* 0000xxxx */
  _setbkcolor((newattr >> 4) & 7);     /* 0xxx0000 */
}
```

The `_settextcolor` function sets just the text color, and the `_setbkcolor` function sets the background color. Both are defined in `graph.h`. Note that because the color attribute included both the background color and the foreground color in a single byte value, the `textattr` function uses & (binary AND) to set a bit mask that isolates only the last four bits in the attribute. That's where the color pair stores the values 0 to 15 for the foreground color.

To get the background color, the function first performs a bit shift to "push" the bits to the right. This puts the "upper" bits into the "lower" bit range, so any bits like `0xxx0000` become `00000xxx` instead. We can use another bit mask with 7 (binary `0111`) to pick out the background color value.

Displaying ASCII art

The `IMAGEDATA` array contains the entire ASCII art screen and the color values for each character. To display the ASCII art to the screen, your program needs to scan the array, set the color attribute, then show the screen one character at a time.

Let's leave room at the bottom of the screen for a separate message or prompt to the user. That means instead of displaying all 25 lines of an 80-column ASCII screen, I only want to show the first 24 lines.

```
  /* print one line less than the 80x25 that's in there:
     80 x 24 x 2 = 3840 */

  for (pos = 0; pos < 3840; pos += 2) {
...
  }
```

Inside the `for` loop, we need to set the colors, then print the character. The OpenWatcom C Compiler provides a function `_outtext` to display text with the current color values. However, this requires passing a string and would be inefficient if we need to process each character one at a time, in case each character on a line requires a different color.

Instead, OpenWatcom has a similar function called `_outmem` that allows you to indicate how many characters to display. For one character at a time, we can provide a pointer to a

character value in the `IMAGEDATA` array and tell `_outtext` to show just one character. That will display the character using the current color attributes, which is what we need.

```
  for (pos = 0; pos < 3840; pos += 2) {
    ch = &IMAGEDATA[pos];              /* pointer assignment */
    attr = IMAGEDATA[pos + 1];

    textattr(attr);
    _outmem(ch, 1);
  }
```

This updated `for` loop sets the character `ch` by assigning a pointer into the `IMAGEDATA` array. Next, the loop sets the text attributes, and then displays the character with `_outmem`.

Putting it all together

With the `textattr` function and the `for` loop to process the array, we can write a full program to display the contents of an ASCII art file. For this example, save the ASCII art as `imgdata.inc` and include it in the source file with an `#include` statement.

```
#include <stdio.h>
#include <conio.h>
#include <graph.h>
#include "imgdata.inc"
void
textattr(int newattr)
{
  _settextcolor(newattr & 15);         /* 0000xxxx */
  _setbkcolor((newattr >> 4) & 7);     /* 0xxx0000 */
}
int
main()
{
  char *ch;
  int attr;
  int pos;
  if (_setvideomode(_TEXTC80) == 0) {
    fputs("Error setting video mode", stderr);
    return 1;
  }
  /* draw the array */
  _settextposition(1, 1);              /* top left */
  /* print one line less than the 80x25 that's in there:
     80 x 24 x 2 = 3840 */
```

```
  for (pos = 0; pos < 3840; pos += 2) {
    ch = &IMAGEDATA[pos];              /* pointer assignment */
    attr = IMAGEDATA[pos + 1];
    textattr(attr);
    _outmem(ch, 1);
  }
  /* done */
  _settextposition(25, 1);             /* bottom left */
  textattr(0x0f);
  _outtext("Press any key to quit");
  getch();
  textattr(0x00);
  return 0;
}
```

Compile the program using the OpenWatcom C Compiler on FreeDOS, and you'll get a new program that displays this holiday message:

Happy Halloween

Appendix: Get started with batch files in FreeDOS

By Kevin O'Brien

On Linux, it's common to create *shell scripts* to automate repetitive tasks. Similarly, on FreeDOS, the open source implementation of old DOS operating systems, you can create a *batch file* containing several FreeDOS commands. Then you can run your batch file to execute each command in order.

You create a batch file by using an ASCII text editor, such as the FreeDOS Edit application. Once you create a batch file, you save it with a file name and the extension `.bat`. The file name should be unique. If you use a FreeDOS command name as your own file name, the FreeDOS command probably will execute instead of your batch file.

Virtually all internal and external FreeDOS commands can be used in a batch file. When you create a batch file, you are essentially writing a program. FreeDOS batch files may not have the power of a structured programming language, but they can be very handy for quick but repetitive tasks.

Commenting your code

The No. 1 good habit for any programmer to learn is to put comments in a program to explain what the code is doing. This is a very good thing to do, but you need to be careful not to fool the operating system into executing your comments. The way to avoid this is to place `REM` (short for "remark") at the beginning of a comment line.

FreeDOS ignores lines starting with `REM`. But anyone who looks at the source code (the text you've written in your batch file) can read your comments and understand what it's doing. This is also a way to temporarily disable a command without deleting it. Just open your batch file for editing, place `REM` at the beginning of the line you want to disable, and save it. When you

want to re-enable that command, just open the file for editing and remove `REM`. This technique is sometimes referred to as "commenting out" a command.

Get set up

Before you start writing your own batch files, I suggest creating a temporary directory in FreeDOS. This can be a safe space for you to play around with batch files without accidentally deleting, moving, or renaming important system files or directories. On FreeDOS, you create a directory with the `MD` command:

```
C:\>MD TEMP
C:\>CD TEMP
C:\TEMP>
```

The `ECHO` FreeDOS command controls what is shown on the screen when you run a batch file. For instance, here is a simple one-line batch file:

```
ECHO Hello world
```

If you create this file and run it, you will see the sentence displayed on the screen. The quickest way to do this is from the command line: Use the `COPY` command to take the input from your keyboard (`CON`) and place it into the file `TEST1.BAT`. Then press **Ctrl**+**Z** to stop the copy process, and press Return or Enter on your keyboard to return to a prompt.

Try creating this file as `TEST1.BAT` in your temporary directory, and then run it:

```
C:\TEMP>COPY CON TEST1.BAT
CON => TEST1.BAT
ECHO Hello world
^Z
C:\TEMP>TEST1
Hello world
```

This can be useful when you want to display a piece of text. For instance, you might see a message on your screen telling you to wait while a program finishes its task, or in a networked environment, you might see a login message.

What if you want to display a blank line? You might think that the `ECHO` command all by itself would do the trick, but the `ECHO` command alone asks FreeDOS to respond whether `ECHO` is on or off:

```
C:\TEMP>ECHO
ECHO is on
```

The way to get a blank line is to use a + sign immediately after `ECHO`:

```
C:\TEMP>ECHO+
C:\TEMP>
```

Batch file variables

A variable is a holding place for information you need your batch file to remember temporarily. This is a vital function of programming because you don't always know what data you want your batch file to use. Here's a simple example to demonstrate.

Create `TEST3.BAT`:

```
@MD BACKUPS
COPY %1 BACKUPS\%1
```

Variables are signified by the use of the percentage symbol followed by a number, so this batch file creates a `BACKUPS` subdirectory in your current directory and then copies a variable `%1` into a `BACKUPS` folder. What is this variable? That's up to you to decide when you run the batch file:

```
C:\TEMP>TEST3 TEMP1.BAT
TEST1.BAT => BACKUPS\TEST1.BAT
```

Your batch file has copied `TEST1.BAT` into a subdirectory called `BACKUPS` because you identified that file as an argument when running your batch file. Your batch file substituted `TEST1.BAT` for `%1`.

Variables are positional. The variable `%1` is the first argument you provide to your command, while `%2` is the second, and so on. Suppose you create a batch file to list the contents of a directory:

```
DIR %1
```

Try running it:

```
C:\TEMP>TEST4.BAT C:\HOME

```

```
ARTICLES
BIN
CHEATSHEETS
GAMES
DND
```

That works as expected. But this fails:

```
C:\TEMP>TEST4.BAT C:\HOME C:\DOCS
ARTICLES
BIN
CHEATSHEETS
GAMES
DND
```

If you try that, you get the listing of the first argument (`C:\HOME`) but not of the second (`C:\DOCS`). This is because your batch file is only looking for one variable (`%1`), and besides, the `DIR` command can take only one directory.

Also, you don't really need to specify a batch file's extension when you run it—unless you are unlucky enough to pick a name for the batch file that matches one of the FreeDOS external commands or something similar. When FreeDOS executes commands, it goes in the following order:

1. Internal commands
2. External commands with the *.COM extension
3. External commands with the *.EXE extension
4. Batch files

Multiple arguments

OK, now rewrite the `TEST4.BAT` file to use a command that takes two arguments so that you can see how this works. First, create a simple text file called `FILE1.TXT` using the `EDIT` application. Put a sentence of some kind inside (e.g., "Hello world"), and save the file in your `TEMP` working directory.

Next, use `EDIT` to change your `TEST4.BAT` file:

```
COPY %1 %2
DIR
```

Save it, then execute the command:

```
C:\TEMP\>TEST4 FILE1.TXT FILE2.TXT
```

Upon running your batch file, you see a directory listing of your `TEMP` directory. Among the files listed, you have `FILE1.TXT` and `FILE2.TXT`, which were created by your batch file.

Nested batch files

Another feature of batch files is that they can be "nested," meaning that one batch file can be called and run inside another batch file. To see how this works, start with a simple pair of batch files.

The first file is called `NBATCH1.BAT`:

```
@ECHO OFF
ECHO Hello
CALL NBATCH2.BAT
ECHO world
```

The first line (`@ECHO OFF`) quietly tells the batch file to show only the output of the commands (not the commands themselves) when you run it. You probably noticed in previous examples that there was a lot of feedback about what the batch file was doing; in this case, you're permitting your batch file to display only the results.

The second batch file is called NBATCH2.BAT:

```
echo from FreeDOS
```

Create both of these files using `EDIT`, and save them in your TEMP subdirectory. Run `NBATCH1.BAT` to see what happens:

```
C:\TEMP\>NBATCH1.BAT
Hello
from FreeDOS
world
```

Your second batch file was executed from within the first by the `CALL` command, which provided the string "from FreeDOS" in the middle of your "Hello world" message.

FreeDOS scripting

Batch files are a great way to write your own simple programs and automate tasks that normally require lots of typing. The more you use FreeDOS, the more familiar you'll become with its commands, and once you know the commands, it's just a matter of listing them in a batch file to make your FreeDOS system make your life easier. Give it a try!

Programming with conio Cheat Sheet

The OpenWatcom conio.h and graph.h functions give you the flexibility to print text on different areas of the screen, create windows, and apply text formatting.

Set text mode when your program starts, and reset it before your program exits.			
`Short _setvideomode(short mode);`			
`_TEXTC80`	80x25 color	`_TEXTBW80`	80x25 black and white
`_TEXTC40`	40x25 color	`_TEXTBW40`	40x25 black and white
`_TEXTMONO`	80x25 mono	`_DEFAULTMODE`	Mode before running your program

Text windows

Screen coordinates are `row,col` and start 1,1 in the upper-left corner:	
`void _FAR _settextwindow(short top, short left, short bottom, short right);`	Define a text window from `top,left` to `bottom,right`
`void _FAR _clearscreen(short area);`	Clear screen or window
`_GCLEARSCREEN`	Clear the whole screen
`_GWINDOW`	Clear the defined window

Printing text

Screen coordinates are `row,col` and start 1,1 in the upper-left corner:	
`struct rccoord _FAR _settextposition(short row, short col);`	Move the cursor to `row,col`
`short _FAR _settextcolor(short color);`	Set the text foreground color (0-15)
`long _FAR _setbkcolor(long color);`	Set the text background color (0-7)
`void _FAR _outtext(char _FAR *text);`	Print text at cursor position in current color

	0. Black		8. Bright Black		4. Red		12. Bright Red
	1. Blue		9. Bright Blue		5. Magenta		13. Bright Magenta
	2. Green		10. Bright Green		6. Brown		14. Yellow
	3. Cyan		11. Bright Cyan		7. White		15. Bright White

Programming with conio Cheat Sheet

Video

`struct videoconfig * _getvideoconfig(struct videoconfig *cfg);`	Probe the video capabilities
short `humtextrows`	The number of text rows (such as 25)
short `numtextcols`	The number of text columns (usually 80 or 40)
short `numcolors`	The number of available colors
short `mode`	The current video mode
short `adapter`	The video adapter connected to the system
short `monitor`	The display or monitor attached to the system

Common adapter values		Common monitor values	
`_SVGA`	Super VGA adapter	`_ENHANCED`	Enhanced color display
`_VGA`	Standard VGA adapter	`_COLOR`	Regular color display
`_EGA`	Older EGA adapter	`_MONO`	Monochrome display
`_CGA`	Older CGA adapter		

Create a text window

```
Void
textwindow_color(int top,int left,int bottom,int right,int fg,int bg) {
  _settextwindow(top, left, bottom, right);
  _settextcolor(fg);
  _setbkcolor(bg);
  _clearscreen(_GWINDOW);
}
```

Print a status line at the bottom of the screen

```
void print_status(int fg, int bg, const char *text) {
  textwindow_color(25, 1, 25, 80, fg, bg);

  _settextposition(1, 1);
  _outtext(text); }
```

FreeDOS is a complete, free, DOS-compatible operating system.
Use this cheat sheet to help you with the most common commands.

WHAT DO YOU WANT TO DO?	HOW TO DO IT ON FREEDOS:	SIMILAR COMMAND ON LINUX:
List directory contents	**DIR**	**ls**
-in the directory "above"	**DIR ..**	**ls ..**
-in a different directory	**DIR C:\FDOS\BIN**	**ls /usr/bin**
Change the current drive	**D:**	
Change the current directory	**CD \FDOS\BIN**	**cd /usr/bin**
-"up" one directory	**CD ..**	**cd ..**
Display the contents of a file	**TYPE FILE.TXT**	**cat file.txt**
-one screen at a time	**MORE FILE.TXT**	**less file.txt**
Copy a file	**COPY FILE.TXT NEW.TXT**	**cp file.txt new.txt**
Delete a file	**DEL FILE.TXT**	**rm file.txt**
Copy a directory and its contents	**XCOPY DIR NEWDIR**	**cp -r dir newdir**
Delete a directory and its contents	**DELTREE MYFILES**	**rm -rf myfiles**
Create a new directory	**MKDIR NEWDIR**	**mkdir newdir**
Remove an empty directory	**RMDIR MYFILES**	**rmdir myfiles**
Rename a file or directory	**REN FILE.TXT FILE.OLD**	**mv file.txt file.old**
Show all lines that contain "Hello"	**FIND "Hello" FILE.TXT**	**grep "Hello" file.txt**
-without regard to case	**FIND /I "Hello" FILE.TXT**	**grep -i "Hello" file.txt**
Clear the screen	**CLS**	**clear**
Edit a text file	**EDIT FILE.TXT**	**vi file.txt**
View and set the system date	**DATE**	**date**
View and set the system time	**TIME**	**date**
Show the usage for a program	**DIR /?** (for most programs)	**ls --help**
Get more help	**HELP**	**info**
Show the command history	**HISTORY**	**history**
Show the DOS version	**VER**	**uname**

Batch scripts

Reference normal batch script variables by enclosing the variable name with **%**, such as **%PATH%**

WHAT DO YOU WANT TO DO?	HOW TO DO IT IN A BATCH SCRIPT:
Execute another batch script from within a script	**CALL SCRIPT.BAT**
Run a command for each file in a list	**FOR %%F IN (*.TXT) DO EDIT %%F** or at the command line: **FOR %F IN (*.TXT) DO EDIT %F** The loop variable name can only be one character
Print output	**ECHO Hello world**
Jump to a label in a batch file	**:LOOP** **GOTO LOOP**
Test the value of a string	**IF %VAR%==1 ECHO One**
Test if a file exists	**IF EXIST TEMP.DAT DEL TEMP.DAT**
Test the return value of the previous command	**IF ERRORLEVEL 0 ECHO Success**
Test the opposite of something (works for all **IF**s)	**IF NOT ERRORLEVEL 0 ECHO Fail**
Set the shell's search path for programs	**PATH C:\FDOS\BIN;C:\MY\BIN** or to reference the existing path: **PATH %PATH%;C:\MY\BIN** Use **;** to separate paths
A comment in a batch script	**REM This is a comment**
Set a variable	**SET TEMPFILE=TEMP.DAT**
Shift the command line options to a batch script	**SHIFT** or **SHIFT 1** or any n Reference command line options as **%1**, **%2**, and so on

A few things to remember

NOTES:	FOR EXAMPLE:
DOS commands can be upper or lowercase	**DIR** is the same as **dir**
Pipes (**|**) are the same on DOS as on Linux	**TYPE FILE.TXT	MORE**
Output redirection (**>**) is the same too	**FIND "X" FILE.TXT > X.TXT**
. and **..** are the same on DOS as on Linux	**CD ..** moves "up" one directory
The directory separator is ****	**C:** or **C:\FDOS** or **C:\FDOS\BIN**
File names can only be 8.3 characters	**FILENAME.EXT**
DOS uses letters for each drive	**C:** is the first hard drive
A full path is a drive letter and a directory path	**C:** or **C:\FDOS** or **D:\GAMES**

