by Peter Collinson, Hillside Systems

PAUL SCHULENBURG

Working with the Dark Side

Inevitably, I spend a lot of time dealing with files that emanate from other systems. In my case, they mostly start life on a box that's running a product from Mr. Gates' organization. I do also sometimes have to deal with files that are created on a Macintosh, although there are many less such systems in Europe. Apple Computer Inc. stupidly charged much more for early models here than it did in the United States, and Macs became expensive machines running a funny operating system on a little screen. The keyword is "expensive" and the operating system was generally ignored. Apple blew its chance of being the worldwide supplier of the first true personal computer.

Incidentally, I am not even thinking about the ever-changing undocumented internal formats used by various programs. People send me files containing text in a wide variety of formats, and pragmatically I have to deal with them. There are intentional versioning problems with many of the formats used by these applications. Application authors seem to work hard to prevent you from moving files from one application to another, in what can only be defined as a vindictive manner. However, people are mostly sending me just text, and I can usually generate text from their application. How portable is text as a format?

You might think that if you asked someone to create a text file in ASCII and put it on a floppy so you can copy it onto your machine, that text file would be the same everywhere. Sadly, this is not the case. For a start, each of the three system families use different conventions to mean "end of line" for text files.

In the ASCII character set, it was necessary to send two characters to the output device to make it start a new line: "line feed" moved the paper up one notch and "carriage return" moved the print head back to the start of the line. The designers of UNIX decided to use a single character to mean end of line in disk text files and chose line feed for the job. It became the task of the device driver that sent characters to the output device to convert a single end-of-line character into the necessary pair. Macs use only the carriage return character to mean the end of a line. MS-DOS and its descendants use the character pair, carriage return followed by line feed.

The effect of these different design decisions is that when we copy a file image from one system to another, we need to perform conversions to ensure that each line on the local machine is terminated by a character (or sequence) the local system understands.

If you don't do any conversion, then things may still work, but it depends on the application. If you process a native MS-DOS file on a UNIX system, then all the applications will be looking for line feed to end the line and will count the carriage return as part of the text on the line. UNIX editors will show you

that the line terminates in carriage return-line feed by displaying ^M (Control-M, the code for line feed) at the end of each line.

I've found that some Windows applications don't mind that the carriage return is missing from the file. The Windows C compiler, for example, is happy to deal with UNIX files. However, a Mac file can be very bad news for UNIX applications that are not presented with an end-of-line character and are asked to deal with a file that is effectively one single line of text.

Sun helpfully provides a pair of conversion programs intended to translate from UNIX to MS-DOS and back again.

Incidentally, if you are copying the text file over a TCP/IP network, then you can use the FTP protocol in ASCII mode. The protocol insists that the sending machine must change any text file that it sends into "network native" form, where each line is terminated by a carriage return-line feed pair. Your local FTP client will store such files using the native text file representation.

Also, if someone sends you a text file as a MIME attachment to a mail message using the *text/plain* format, then the text file will end up with the correct line-termination characters on your local machine.

Text files can be moved easily and transparently where there is extant conversion software embedded in the utility being used to transport the file. Problems only arise when the file is copied as an image, where we move the bytes in the file directly from one machine to another. I do this quite a bit, either saving text directly from a Windows application on my UNIX disks that are accessible to my Windows machines, taking a file from a floppy, or unpacking a ZIP file (more on this later). So I am stuck with a file in a foreign format that I need to convert so my UNIX programs can deal with it appropriately.

Conversion Programs

As a first hack, you can convert DOS text files to UNIX by simply removing the carriage return characters using `tr`:

```
$ tr -d '\015' < dos > unix
```

The `\015` is the value of the code for the carriage return character expressed in octal. The `-d` flag tells the `tr` command to delete any characters from its input file that match the list in its argument. The command above simply deletes any carriage return characters from its standard input and writes the resulting data to its standard output.

For Mac files, a similar technique is available:

```
$ tr '\015' '\012' < mac > unix
```

Here, we replace all carriage returns with newline characters.

Sun Microsystems Inc. helpfully provides a pair of conversion programs intended to translate from UNIX to MS-DOS and back again. These are unsurprisingly named `unix2dos` and `dos2unix`. I've wondered for some time why these programs are not scripts–they must be doing something that needs slightly more processing than can be done easily in a script.

One reason for the extra processing is that removing carriage returns is not sufficient. Many early DOS programs signalled the end of a text file by appending a Control-Z character to the end of the file. You'll find that `dos2unix` deletes this end-of-file indicator, removing Control-Z only when it is the last character in the file.

It also appears that `dos2unix` deals with character codes that are greater than 128. Before I can discuss why this is desirable, you need some background.

Character Codes

UNIX started life using the ASCII coding sequence for characters. Characters fit into eight bits, but only seven of these were used in ASCII, allowing a character set of 128 characters. The code was intended to be sent down serial lines, which carry a sequence of pulses representing the ones and zeros in the character code. There was a need to check that this serial data had been received properly in early devices and ASCII allowed for this by using the eighth bit to check the remaining data.

The eighth bit was used in early devices as a "parity" bit. When the sending hardware transmitted the character as a sequence of ones and zeros, it had the ability to force the extra bit to a one or a zero, ensuring that the total number of ones was odd for "odd parity" or even for "even parity." Choices. Choices. The installer had to choose the type of parity to be used on the line. Let's choose odd parity. In this case, the receiving hardware checked for an odd number of ones in the character that it was sent. If the total number of ones that the receiver had seen was not odd, then the receiver knew the data was corrupt.

As UNIX grew, so did semiconductor technology. Serial line hardware improved, and the need for parity checking disappeared. We were able to use the top bit to mean something different. Some machines allowed the user to set it by pressing the Alt key. It was also used to code a set of accented characters, allowing most European languages to be written correctly. We ended up with a standard that's often referred to as Latin-1, which is the original ASCII character set plus 128 accented characters. It's an ISO standard, ISO 8859. This coding became the de facto standard for the Web.

In MS-DOS-land along One Microsoft Way, the top 128 characters were used early on in applications for their

own multilingual purposes and MS-DOS uses a different mapping of codes onto characters to enable accented characters, codes for fractions and some other special characters. It seems Sun's `dos2unix` command understands this mapping and will convert these characters into the appropriate Latin-1 codes.

However, MS-DOS text format is not used these days when people save text from Microsoft Word. In Word, you can specify that a file is to be saved as "MS-DOS TEXT," which uses this coding. But when most people save a file from Word as text they save it as "TEXT," which uses a mapping that's an extended Latin-1. The extensions exist from codes (decimal) 127 to (decimal) 159 and are used to express some characters that don't exist in Latin-1, and also to preserve things like smart quotes and other internal codings. So, it may be that `dos2unix` should no longer be used on text files that come from modern versions of Windows.

Working Out What's Happening

Dealing with all these character codes is confusing; when moving files between machines and operating systems it can become unclear what system is doing what conversion on the file. The box below shows a tiny Perl program you can run to generate a file that can be read by an application using different settings to investigate the character set.

```
for ($i = 0; $i < 256; $i++) {
    printf("--%c-- %d 0x%x 0%o\r\n",
               $i, $i, $i, $i);
}
```

It prints a character code in between some minus signs and then the value of that code in decimal, hexadecimal and octal. I've made it terminate the line in MS-DOS fashion. To run the program, put it into a file, say, `cgen` and type

```
$ perl cgen > file.txt
```

If you don't have Perl, you can render this program in `awk`. Take the code above, remove the dollar characters and make sure the `printf` and the line that follows it are joined on one source line. Then wrap the whole thing in

```
BEGIN {
    program
}
```

and place it into a file, `cgen.aw`. To run it, type

```
$ awk -f cgen.aw /dev/null
```

Try looking at this output file using a UNIX tool like `cat` or `more`. I suspect that you will see the Latin-1 character set (I cannot be sure because my machine is set up to use Latin-1 and yours may not be). You can then read the file into various Windows utilities with various settings and see what character set is being used.

By reading this file into Word, you can see the mapping that is used for Word's internal special characters. I've used this intelligence to create another small Perl program, `win2dos`, that converts Windows text file output from Word. This program, and its friend `unix2win`, are available from my Web site (see below for details). It's much too complicated to describe here.

Incidentally, if you take data from a Web form driven by a CGI script, you must expect people to cut and paste from programs like Word into the form. Your script will undoubtedly see the special character codings Word uses for smart quotes and the like, so this problem of private coding has more knock-on effects.

Packages

Most systems have some way of packaging several files into one. UNIX traditionally uses `tar` or `cpio`. The Windows world uses the ZIP format and has various freeware and shareware applications that are available on the Internet to pack and unpack files. Some of these versions will read `tar` files.

However, early versions of these programs didn't understand the need to preserve directory hierarchy, and it can sometimes be confusing to work out how to move a complete tree from one machine to another.

From a UNIX perspective, ZIP files are supported by open-source software, `zip` and `unzip`. Incidentally, don't confuse these programs with `gzip` and `gunzip`, which form a compression and decompression utility. I find `zip` and `unzip` easy to use, and because you can run them from the command line, they can be embedded easily into scripts.

Mac users are prone to send you email containing files in `binhex` format. There are a small number of shareware programs for the Windows world that can unpack files in this format. But for UNIX, I used to be somewhat stumped until I found a small program

by Dave Johnson, Brown University Computer Science. Again, this code is open source, a rubric in the code says: "May be used but not sold." I don't remember exactly where I found the program–but I had to do some digging, I think. The program unpacks the files, and then sometimes you have to work out exactly what is what because the Mac transmits files along with internal meta information on how to use them.

When reading files from a Mac into a Windows box, you may need to coerce the filename suffix into something that your version of Windows understands and can find a suitable application to process the file. Windows uses the suffix to determine the file type, but the Mac doesn't. For example, if you are moving EPS files from a Mac to Windows, you need to rename the files to have a suitable suffix (usually `.EPS`) for the Windows applications to read the information correctly.

Filenames

All of the package unpacking programs will generate files on your machine whose names obey the filename rules on the machine from which the files emanate. There is a range of incompatibilities with file naming conventions across the various platforms. UNIX allows filenames to contain any character, except `/`, and have infinite length, where infinity is set to 255 characters. Older UNIX systems set infinity to be 16 characters, which was never enough. Filenames are case-dependent, so files named `Fred`, `FRED` and `fred` can all coexist in the same directory.

These days, Windows of all flavors allows filenames to contain any character except the Control characters and `\`, `/`, `:`, `?`, `"`, `<`, `<` and `|`. Filenames have infinite length, where infinity seems to be set around 255 characters too. Windows has a problem with backwards compatibility into MS-DOS, where the names are eight characters with a three-letter extension. Names on Windows are case-independent. Case-independence is maintained by the software on Windows NT, and is not implemented in the file system itself. When you create a file on NT, it's really created using the case that the application uses. If you type `FRED`, it will be created as `FRED`, but will be shown to you in the GUI as `Fred`.

Problems with the case of filenames can be pretty boring to resolve when moving files from a Windows machine to a UNIX system. I typically see this when I'm sent pages in HTML, where the source talks about `picture.gif` but the Windows system has created a file called `picture.GIF`. These are the same files on Windows, but are different on a UNIX system.

Listing 1 shows a small "template" file that I hack to create the edit I need in order to combat the tedium and inaccuracy of repetitive edits.

The `grep` command is inside backquotes so that its output is read and processed by the shell. It is used to create a list of arguments to the `for` statement. The `-l` switch tells `grep` to list the filenames it finds that contain the string, so the output from the command is a list of files.

The `for` statement scans through this list of files presenting their names one at a time to the loop. It opens each file using `ed`, which takes a sequence of instructions from the "here" document starting at the first `EOT` and finishing at the second. The backslash before the first `EOT` serves to quote it, so the shell does not try to expand any dollars or stars in the document itself.

The `ed` commands perform a global

Listing 1. Template File

```
#!/bin/sh
for name in `grep -l picture.GIF *.htm`
do
ed - $name <<\EOT
g/picture.GIF/s,,picture.gif,g
w
q
EOT
done
```

search for some text and replace any found with the new contents. I've used commas as the separator for the substitute (s) command instead of the usual /, so that I can place pathnames more easily in the command. Finally, I write the file back in place. Beware! It's perfectly possible to lose all your data in this way. If you are uncertain whether the edit will work as expected, make a backup copy of the original data first, then use `diff` to check the changes.

Spaces

The other problem we have on UNIX is caused by the ability of Windows and Mac users to use spaces in filenames. UNIX does support spaces in a filename, but spaces can cause problems for shells because they expect spaces to be used to separate files. UNIX operates with the tacit assumption that there are no spaces in filenames. For example, a common loop like

```
for name in `find . -print`
do
    echo $name
done
```

where the `echo` command can be replaced by several operations, assumes that the `find` command will generate a space-separated list (where the space is really a newline) and the `for` loop will process that list one name at a time. This assumption breaks down when a file is found whose name contains a space.

I tend to rename filenames that contain spaces, changing the space to underscore. Here's a script that works in the current directory:

```
for name in *
do
   na=`echo "$name"|tr ' ' '_'`
   if [ "$na" != "$name" ]
   then
        echo mv "$name" "$na"
        mv "$name" "$na"
   fi
done
```

We are using the shell here directly (*) to expand the list of names in the `for` loop, so we know that a name with a space will be passed into the name variable as a single entity. We are careful to use quotes around the name variable so that the shell will treat names with embedded spaces as one string.

First, we need to check whether a name has a space or not, and the simplest way to do this is to change any spaces in the name to an underscore, which we want to do anyway. This is done using the `tr` command we discussed earlier. We now have a string that possibly contains underscores and a string that possibly contains spaces. We know that the `tr` command will only have altered names that contain spaces, and if the original string differs from the massaged one, then `tr` will have made the change to the name. In this case, we rename the old file name to the new, taking care to quote the original name.

Where to Get Things

You can find my Perl 5 version of `win2unix` and `unix2win` on my Web site at `ftp://ftp.hillside.co.uk/sunexpert/darkside`. I'll also make `hexbin.c` available, because I have forgotten where I obtained it. Alternately, the programs can be found at `ftp://ftp.expert.com/pub/UNIXBasics`. You can get `zip` and `unzip` from `ftp://ftp.uu.net/archiving/zip`, where you will find the source and also many precompiled binaries for different platforms.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.