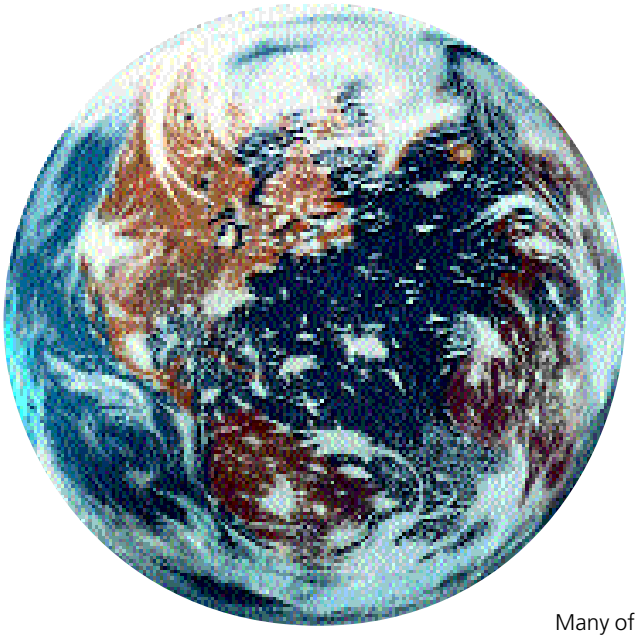
Part 1: Principles of BASH

# HELLO WORLD

BY MIRKO DÖLLE



**Have you ever sat in front of your computer and been irritated by having to perform some annoyingly repetitive task? Many such tasks can be automated by writing a simple program, and in the first feature we'll give you just an introducton into the world of programming.**

Many of you will be asking yourselves why you should even have to start writing programs. After all, there seems to be a tool or utility for Linux that does just about anything you can imagine. The problem is, of course, that while lots of programs and utilities work fine in theory, some are just too complicated or don't meet your exact needs. Learning to program will solve these problems by allowing you to create customised tools and utilities. These can be created either from scratch or by modifying something someone else has written. Since this feature is aimed at beginners, we'll stick to the very basics and limit ourselves to quite simple programs. In doing so, our aim is to give you enough of a glimpse into the wonderful world of programming to encourage you to delve much deeper.

## BASH programming language

The Bourne-Again Shell – BASH for short, has established itself as the standard shell for most Linux distributions, so you'll almost certainly find it ready and waiting for you on your Linux system. But BASH is more than just a command line for starting programs. In fact, it is almost a complete programming language. For this reason we'll use BASH as the programming basis of this feature. We'll also touch on one or two other useful programs in passing. And while the examples we'll be giving are BASH-specific, the programming techniques have been kept as universal as possible, helping you get to grips with other languages should you want to progress to more advanced programming tools and techniques.

## First steps

BASH programs are also called "scripts". Scripts should always begin by describing the shell to be used, then go on to list commands to be executed (which you could also enter manually outside the program). Traditionally, the first program new programmers write is one that prints "Hello world" on screen. Here's how a BASH script to do just that (twice) looks:

```
#!/bin/bash
echo Hello world
echo Hello   world
```

The first line is strictly speaking a commentary line, since it begins with a hash "#". Generally the rule is

### Version conflict

*The examples shown here in this feature all relate to Version 2 of BASH, and are at best only partly applicable with the old Version 1. Although Version 2 has now been in use for over two years and has so far shown no significant problems, some distribution manufacturers, such as Red Hat and Caldera, are still installing the old Version 1.4 under /bin/bash as well as Version 2 under /bin/bash2.*

*For this reason, the first thing you must do if you have any problems with entering our code examples is to check which version you are addressing via /bin/bash by entering the following command:*

```
/bin/bash -c 'echo $BASH_VERSION'
```

*If the result is Version 1, you must hunt down Version 2 – it will probably be lying around in bin and be called bash2. Enter the following command to find out:*

```
ls /bin/bash2
```

*If ls reports that it is unable to find /bin/bash2, you should look on the installation CD of your distribution and if necessary install the package from there. If the ls command does find /bin/bash2, then when entering any of our program examples you must then always use /bin/bash2 instead of /bin/bash.*
*As an alternative, you could completely convert your system to BASH 2. To do so, first copy /bin/bash into /bin/bash1. Next copy /bin/bash2 to /bin/bash. In fact renaming is also sufficient, but then programs which use /bin/bash2 would no longer function, which is why /bin/bash2 should be retained. The real problem is that you cannot overwrite or move a file which is in use – and as you are using normal /bin/bash as root, before you can follow our instructions you must first release /bin/bash using a handy little trick.*
*First log in as root and again ensure that there really is a /bin/bash2. If there isn't then you must not perform the following steps under any circumstances unless you want to run the risk of never being able to log in again as root! Whatever the case, though, you should launch an (extra) text console and log in there as root. We will refer to this console for the sake of simplicity as "emergency console". At first you won't need to enter anything on this emergency console, so you should now change back to your (normal) work console. Here we need to alter the default shell for the user root using the following command:*

```
chsh -s /bin/bash2 root
```

*Next, switch to yet another text console and log in there again as root. If everything has gone well, echo $BASH_VERSION will now report that it is Version 2 on this console. If you are unable to log in, something has gone wrong and you absolutely must now restore the former status. To do so, change to the emergency console and enter:*

```
chsh -s /bin/bash root
```

*This will set the log-in shell back to the initial value.*
*If the new log-in worked, treat this as a new emergency console and log out of the old one and out of all other consoles. It may also become necessary to close KDE or GNOME. All that should now remain is the new emergency console.*
*The next step is to convert from BASH 1 to BASH 2. To do this, log in again to a new console as root and copy bash to bash1 and then bash2 to bash as we outlined when we started using the following commands:*

```
cp /bin/bash /bin/bash1
cp /bin/bash2 /bin/bash
```

*The last step is then to reset the log-in shell from root :*

```
chsh -l /bin/bash root
```

*If you can log in again on another console as root and see Version 2 of BASH displayed, the conversion has worked, and Version 2 is now your standard shell. Congratulations!*

***Meta character:*** *Character interpreted in a special way.*
***Escape character:*** *Cancels the effects of meta-, control- and escape-characters.*
***Quotes:*** *Double quotes, single quotes and reverted single quotes. These ensure that most meta and control characters are no longer interpreted as such. These must always be used in pairs.*
***Script:*** *Generally used to refer to shell and sometimes also Perl programs. The script is always a text file that can be displayed directly.*

■

that only comments should stand between the hash sign and the end of that line. You can have comment lines without any commentary if you wish – to break apart code sections, for example – but blank lines can also be used for this purpose and are a better choice.

The first line in our example tells you which program this script should be processed with (BASH in this case) though it could be Perl or TCL/TK for instance, which is why this line is so important.

The next two lines both cause "Hello world" to be outputed onto the screen, followed by a line break.

## Meta-characters and Escapes

Despite the variation in the text following the *echo* command, the result of BASH processing lines two and three are identical. This is because BASH interprets what are known as control symbols (also known as **meta-characters** and **control characters**), in a special way. The spaces character is one such control symbol, and is used as a separator between parameters of a command, where a sequence of space characters is interpreded as one parameter separation. The command in question here is *echo*, which simply prints all parameters (the things following the *echo* command) in sequence, each one separated by a space. In our example both line two and line three therefore have the same effect. *echo* simply sees two parameters "Hello" and "world".

**Table 1: Control- and special characters in BASH**

| Character | Function |
|---|---|
| Space | Separator between program parameters |
| Tabulator (tab) | Separator between program parameters |
| Enter (newline) | Enter command |
| \ (backslash) | Escape character |
| \| (pipe) | Concatenation of input/output of several programs |
| & (ampersand) | Start program as background process, input-/output redirect |
| ; (semicolon) | Separator between two program calls |
| ( ) (braces) | Grouping, calculation |
| < (input redirect) | |
| > (output redirect) | |
| \|\| (logical or) | Link two commands with "OR" |
| && (logical and) | Link two commands with "AND" |
| ;; | End of a *case* |

**Table 2: BASH Operators**

| Operator | Assignment Operator | Function |
|---|---|---|
| +, -, *, / | +=, -=, *=, /= | basic types of arithmetic |
| % | %= | Remainder from whole number division (5%2=1) |
| ! | | Logic negation (!1 = 0, !0 = 1) |
| && | | Logic AND (a and b) |
| \|\| | | Logic OR (a or b) |
| ==, != | | equality, inequality |
| <=, >=, <, > | | comparison larger/smaller |
| ~ | | Binary inversion (~1101 = 0010) |
| & | &= | Binary AND (1011 & 1101 = 1001) |
| \| | \|= | Binary OR (1100 \| 0101 = 1101) |

There are quite a few of these special characters, and the most important ones are listed in Table 1. To get more than one space between "Hello" and "world" we have to use another special character, *escape*. This character informs BASH that the next character isn't a special character – a true space and not a separator between parameters in our case. In BASH, and many other tools and languages, the *escape* symbol is represented by the backslash "\" character. So, to print "Hello", three spaces, then "world" we'd have to use the following command.

```
echo Hello\ \ \ world
```

Of course there will be times when we actually want to print a backslash. To do so, we have to "escape the escape" by typing "\\". Alternatively, you can use single or double **quotes** to demote spaces and most other special characters into simple text, such as in the following example:

```
echo "Hello world"
```

The use of *quotes* in this way is, however, best employed infrequently. They are far more effective when used for other purposes, as we'll see later on.

## Variables

It would be boring to only be able to print fixed, unchanging text. This is where things known as variables come into the equation. These are simply named containers for text or numbers.

Unlike more sophisticated languages, BASH does not differentiate between different types of variable, so you can store whole numbers (in principle decimal numbers are not allowed), letters, words or whole sentences in a BASH variable without first having to tell it which of these you want the variable to store. What's more – again unlike some other programming languages – a variable does not have to be declared ("registered") before you can use it; it simply comes into existence automatically as the result of the first value assignment.

Something well worth watching out for is the fact that variable names are case sensitive. In the following example we'll give the same variable five completely different values using the "=" operator, whose use should be pretty much self-evident:

```
#!/bin/bash
var=2
var=a
var=Hello
var=Hello\ world
var="Hello world"
```

To find out a variable's contents (also known as its value), you simply use its name preceded by a dollar symbol – "$" (another one of these special characters). You can also optionally enclose the name of the variable in curly brackets. This variant is used when a letter immediately follows the variable

name. To better understand what we're talking about here, have a look at the following example:

```
Cost=100
echo $Cost Euros
echo ${Cost}Euros
```

The second line would print "100 Euros", while the third prints "100Euros" without a space between the amount and the unit of currency. Without curly brackets in the third line, we would have got the content of the (non-existent) variable *$CostEuros*. Non-existent variables always have no value, they are simply empty, and *echo $CostEuros* would therefore print out a blank line.

BASH, by the way, replaces the variable names at almost every point by the value of the variable, a process called *variable resolution*. The only exception to this rule occurs when inverted commas " ' " are used. Anything between these is never replaced with a value. To use the name of a variable, including the dollar sign, for printing (or for other functions), we can also escape the "$" as an alternative. Here are some examples of what we mean:

```
Cost=100
Cost="$Cost Euros"
echo Content of the variable '$Cost': $Cost
echo Content of the variable \$Cost: $Cost
```

Note that in the second line we have assigned *Cost* a character string in which the variable itself occurs. In this case, BASH first evaluates the part *to the right* of the equals sign, therefore substituting "100" for *$Cost*. To BASH, then, the right hand side of the equals sign is therefore "100 Euros". Once the right-hand side of the equation has been evaluated (to form what's known as the **r-value**), it then assigns it to the variable on the left-hand side (the **l-value**). Don't worry if you don't understand this yet, though, as we'll be coming back to this in another example later on.

## Arithmetical operations and zero-function

Assigning values to variables is all very well, but they only come into their own when manipulated, compared or used in computations. BASH provides quite a few operators to do this, including arithmetic, logic and binary ones. When using these, though, with the exception of logical operators (which can be used with variables containing absolutely anything) you must make sure that you work with variables containing only numbers. Remember, BASH can store both text and numbers in a variable. It is your job to make sure you aren't trying to calculate with letters at any time.

To understand this better, take another look at the first two lines of our last example. In the first line there was still just a number in *Cost* (100), but in the second line we added on "Euros". In other words the Cost variable has suddenly turned into a character **string** rather than a number. This means we can no longer do calculations with the *Cost* variable. If you try to do so, BASH will abort your program with an error message.

There are two notations for calculations; the instruction is either enclosed in square brackets or in double rounded brackets. In both cases it is preceded by a dollar sign. Both notations can be seen in the next example:

```
Value=$[$Value+1]
Value=$(($Value+1))
: $[Value+=1]
: $((Value+=1))
```

In the end all four lines do exactly the same thing, which is to increase the content of *Value* by one. Lines one and two are the easiest to understand here, as usual the R-Value is evaluated first and then assigned to the L-Value. Lines three and four work on the same principle, because the operation "a+=b" is defined as "a=$a+b" – internally, BASH converts the short notation into the full one.

What is unusual here is the colon before the calculation instruction. This is a zero function, a command that does nothing. What matters to us here is that BASH evaluates the parameter after it in exactly the same way as it would when calling up any other command, such as *echo*, thus calculating the result for us. The colon is necessary because the calculation operations are always replaced by the result, namely *$Value*, and without the colon this would be interpreted as a command for BASH, which it would then try to call up and undoubtedly fail.

As we've already said, as well as basic types of arithmetic, BASH also offers logic and binary operators, though these are only used rarely. Whatever operators are used, though, additional assignment operators are usually also available – such as the "+=" we used in our example and which can make reading programs a great deal simpler. You can find an overview of the operators in Table 2.

## Until next time

This brings us to the end of the first installment; next time we'll tell you about processing character strings, introduce you to arrays and explain their use by means of a few more examples. ■

***r-value:*** *The result of the instructions to the right of the equals sign*
***l-value:*** *Variable to the left of the equals sign, in which the r-value is stored*

■

| Quick glossary | |
|---|---|
| **echo** | Outputs all parameters separated from each other by a space. |
| **: (colon)** | Zero-function, has no direct effect. Is sometimes used for arithmetical operations or variable manipulations. The actual operations are stated as parameters of the ":" function. |
| **$[..]** | Calculates the arithmetical expression in brackets and delivers the result. |
| **$((..))** | Calculates the arithmetical expression in brackets and delivers the result. |