Parameter expansion

Introduction

One core functionality of Bash is to manage **parameters**. A parameter is an entity that stores values and is referenced by a **name**, a **number** or a **special symbol**.

- parameters referenced by a name are called **variables** (this also applies to arrays)
- parameters referenced by a number are called **positional parameters** and reflect the arguments given to a shell
- parameters referenced by a **special symbol** are auto-set parameters that have different special meanings and uses

Parameter expansion is the procedure to get the value from the referenced entity, like expanding a variable to print its value. On expansion time you can do very nasty things with the parameter or its value. These things are described here.

If you saw some parameter expansion syntax somewhere, and need to check what it can be, try the overview section below!

Arrays can be special cases for parameter expansion, every applicable description mentions arrays below. Please also see the article about arrays.

For a more technical view what a parameter is and which types exist, see the dictionary entry for "parameter".

Overview

Looking for a specific syntax you saw, without knowing the name?

- Simple usage
  - `$PARAMETER`
  - `${PARAMETER}`
- Indirection
  - `${!PARAMETER}`
- Case modification
  - `${PARAMETER^}`
  - `${PARAMETER^^}`
  - `${PARAMETER,}`
  - `${PARAMETER,,}`
  - `${PARAMETER~}`
  - `${PARAMETER~~}`
- Variable name expansion
  - `${!PREFIX*}`
  - `${!PREFIX@}`
- Substring removal (also for **filename manipulation**!)
  - `${PARAMETER#PATTERN}`
  - `${PARAMETER##PATTERN}`
  - `${PARAMETER%PATTERN}`
  - `${PARAMETER%%PATTERN}`
- Search and replace
  - `${PARAMETER/PATTERN/STRING}`
  - `${PARAMETER//PATTERN/STRING}`
  - `${PARAMETER/PATTERN}`
  - `${PARAMETER//PATTERN}`
- String length
  - `${#PARAMETER}`
- Substring expansion
  - `${PARAMETER:OFFSET}`
  - `${PARAMETER:OFFSET:LENGTH}`
- Use a default value
  - `${PARAMETER:-WORD}`
  - `${PARAMETER-WORD}`
- Assign a default value
  - `${PARAMETER:=WORD}`
  - `${PARAMETER=WORD}`
- Use an alternate value

    - ${PARAMETER:+WORD}
    - ${PARAMETER+WORD}
- Display error if null or unset
    - ${PARAMETER:?WORD}
    - ${PARAMETER?WORD}

Simple usage

$PARAMETER

${PARAMETER}

The easiest form is to just use a parameter's name within braces. This is identical to using $FOO like you see it everywhere, but has the advantage that it can be immediately followed by characters that would be interpreted as part of the parameter name otherwise. Compare these two expressions (WORD="car" for example), where we want to print a word with a trailing "s":

```
echo "The plural of $WORD is most likely $WORDs"
echo "The plural of $WORD is most likely ${WORD}s"
```

Why does the first one fail? It prints nothing, because a parameter (variable) named "WORDs" is undefined and thus printed as "" (*nothing*). Without using braces for parameter expansion, Bash will interpret the sequence of all valid characters from the introducing "$" up to the last valid character as name of the parameter. When using braces you just force Bash to **only interpret the name inside your braces**.

Also, please remember, that **parameter names are** (like nearly everything in UNIX®) **case sentitive!**

The second form with the curly braces is also needed to access positional parameters (arguments to a script) beyond $9:

```
echo "Argument  1 is: $1"
echo "Argument 10 is: ${10}"
```

Simple usage: Arrays

See also the article about general array syntax

For arrays you always need the braces. The arrays are expanded by individual indexes or mass arguments. An individual index behaves like a normal parameter, for the mass expansion, please read the article about arrays linked above.

- ${array[5]}
- ${array[*]}
- ${array[@]}

Indirection

${!PARAMETER}

Everywhere you can name a parameter to expand, like for example

```
${PARAMETER}

${PARAMETER:0:3}
```

you can use the form

```
${!PARAMETER}
```

which will enter a level of indirection: The referenced parameter is not PARAMETER itself, but the parameter named by the value of it. If your parameter PARAMETER has the value "TEMP", then ${!PARAMETER} really references TEMP:

```
read -p "Which variable do you want to inspect? " look_var

echo "The value of \"$look_var\" is: \"${!look_var}\""
```

Of course the indirection also works with special variables:

```
# set some fake positional parameters
set one two three four

# get the LAST argument ("#" stores the number of arguments, so "!#" will reference the LAST argument)
echo ${!#}
```

This is also known as "variable variables" or "indirect reference". **Indirect references to array names are not possible (as of Bash 4.1).**

Case modification

`${PARAMETER^}`

`${PARAMETER^^}`

`${PARAMETER,}`

`${PARAMETER,,}`

`${PARAMETER~}`

`${PARAMETER~~}`

These expansion operators modify the case of the letters in the expanded text.

The `^` operator modifies the first character to uppercase, the `,` operator to lowercase. When using the double-form (`^^` and `,,`), all characters are converted.

The (**currently undocumented**) operators `~` and `~~` reverse the case of the given text (in `PARAMETER`).`~` reverses the case of first letter of words in the variable while `~~` reverses case for all.Thanks to `Bushmills` and `geirha` on the Freenode IRC channel for this finding.

Example: Rename all `*.txt` filenames to lowercase

```
for file in *.txt; do
  mv "$file" "${file,,}"
done
```

Note: The feature worked word-wise in Bash 4 RC1 (a modification of a parameter containing `hello world` ended up in `Hello World`, not `Hello world`). In the final Bash 4 version it works on the whole parameter, regardless of something like "words". IMHO a technically cleaner implementation. Thanks to Chet.

Case modification: Arrays

For array expansion, the case modification applies to **every expanded element, no matter if you expand an individual index or mass-expand** the whole array using `@` or `*` subscripts. Some examples:

Assume: `array=(This is some Text)`

- `echo "${array[@],}"`
  - ⇒ `this is some text`
- `echo "${array[@],,}"`
  - ⇒ `this is some text`
- `echo "${array[@]^}"`
  - ⇒ `This Is Some Text`
- `echo "${array[@]^^}"`
  - ⇒ `THIS IS SOME TEXT`
- `echo "${array[2]^^}"`
  - ⇒ `TEXT`

Variable name expansion

`${!PREFIX*}`

`${!PREFIX@}`

This expands to a list of all set **variable names** beginning with the string `PREFIX`. The elements of the list are separated by the first character in the `IFS`-variable (<space> by default).

This will show all defined variable names (not values!) beginning with "BASH":

```
$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_LINENO BASH_SOURCE BASH_SUBSHELL BASH_VERSINFO BASH_VERSION
```

This list will also include array names.

Substring removal

${PARAMETER#PATTERN}

${PARAMETER##PATTERN}

${PARAMETER%PATTERN}

${PARAMETER%%PATTERN}

This one can **expand only a part** of a parameter's value, **given a pattern to describe what to remove** from the string. The pattern is interpreted just like a pattern to describe a filename to match (globbing). See Pattern matching for more.

Example string (*just a quote from a big man*):

```
MYSTRING="Be liberal in what you accept, and conservative in what you send"
```

From the beginning

${PARAMETER#PATTERN} and ${PARAMETER##PATTERN}

This form is to remove the described pattern trying to **match it from the beginning of the string**. The operator "#" will try to remove the shortest text matching the pattern, while "##" tries to do it with the longest text matching. Look at the following examples to get the idea (matched text ~~marked striked~~, remember it will be removed!):

Syntax	Result
`${MYSTRING#* }`	~~Be~~ liberal in what you accept, and conservative in what you send
`${MYSTRING##* }`	~~Be liberal in what you accept, and conservative in what you~~ send

From the end

${PARAMETER%PATTERN} and ${PARAMETER%%PATTERN}

In the second form everything will be the same, except that Bash now tries to match the pattern from the end of the string:

Syntax	Result
`${MYSTRING% *}`	Be liberal in what you accept, and conservative in what you ~~send~~
`${MYSTRING%% *}`	Be ~~liberal in what you accept, and conservative in what you send~~

Common use

How the heck does that help to make my life easier?

Well, maybe the most common use for it is to **extract parts of a filename**. Just look at the following list with examples:

- § $ # | @ **[illegible]hout extension**
  - ${FILENAME%.*}
  - ⇒ bash_hackers~~.txt~~
- **Get extension**
  - ${FILENAME##*.}
  - ⇒ ~~bash_hackers.~~txt
- **Get directory name**
  - ${PATHNAME%/*}
  - ⇒ /home/bash~~/bash_hackers.txt~~
- **Get filename**
  - ${PATHNAME##*/}
  - ⇒ ~~/home/bash/~~bash_hackers.txt

These are the syntaxes for filenames with a single extension. Depending on your needs, you might need to adjust shortest/longest match.

Substring removal: Arrays

As for most parameter expansion features, working on arrays **will handle each expanded element**, for individual expansion and also for mass expansion.

Simple example, removing a trailing is from all array elements (on expansion):

Assume: `array=(This is a text)`

- `echo "${array[@]%is}"`
  - ⇒ `Th a text`
  - (it was: `Th~~is~~ ~~is~~ a text`)

All other variants of this expansion behave the same.

Search and replace

`${PARAMETER/PATTERN/STRING}`

`${PARAMETER//PATTERN/STRING}`

`${PARAMETER/PATTERN}`

`${PARAMETER//PATTERN}`

This one can substitute (*replace*) a substring matched by a pattern, on expansion time. The matched substring will be entirely removed and the given string will be inserted. Again some example string for the tests:

```
MYSTRING="Be liberal in what you accept, and conservative in what you send"
```

The two main forms only differ in **the number of slashes** after the parameter name: `${PARAMETER/PATTERN/STRING}` and `${PARAMETER//PATTERN/STRING}`

The first one (*one slash*) is to only substitute **the first occurrence** of the given pattern, the second one (*two slashes*) is to substitute **all occurrences** of the pattern.

First, let's try to say "happy" instead of "conservative" in our example string:

```
${MYSTRING//conservative/happy}
```

⇒ `Be liberal in what you accept, and ~~conservative~~happy in what you send`

Since there is only one "conservative" in that example, it really doesn't matter which of the two forms we use.

Let's play with the word "in", I don't know if it makes any sense, but let's substitute it with "by".

First form: Substitute first occurrence

```
${MYSTRING/in/by}
```

⇒ `Be liberal ~~in~~by what you accept, and conservative in what you send`

Second form: Substitute all occurrences

```
${MYSTRING//in/by}
```

⇒ `Be liberal ~~in~~by what you accept, and conservative ~~in~~by what you send`

Anchoring Additionally you can "anchor" an expression: A # (hashmark) will indicate that your expression is matched against the beginning portion of the string, a % (percent-sign) will do it for the end portion.

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING/#x/y}  # RESULT: yxxxxxxxxx
echo ${MYSTRING/%x/y}  # RESULT: xxxxxxxxxy
```

If the replacement part is completely omitted, like, the matches are replaced by the nullstring, i.e. they are removed. This is equivalent to specifying an empty replacement:

```
echo ${MYSTRING//conservative/}
# is equivalent to
echo ${MYSTRING//conservative}
```

Search and replace: Arrays

This parameter expansion type applied to arrays **applies to all expanded elements**, no matter if an individual element is expanded, or all elements using the mass expansion syntaxes.

A simple example, changing the (lowercase) letter `t` to `d`:

Assume: `array=(This is a text)`

- echo "${array[@]/t/d}"
    - ⇒ This is a dext
- echo "${array[@]//t/d}"
    - ⇒ This is a dexd

String length

${#PARAMETER}

When you use this form, the length of the parameter's value is expanded. Again, a quote from a big man, to have a test text:

```
MYSTRING="Be liberal in what you accept, and conservative in what you send"
```

Using echo ${#MYSTRING}...

⇒ 64

There's not much to say about it, mh?

(String) length: Arrays

For arrays, this expansion type has two meanings:

- For **individual** elements, it reports the string length of the element (as for every "normal" parameter)
- For the **mass subscripts** @ and * it reports the number of set elements in the array

Example:

Assume: array=(This is a text)

- echo ${#array[1]}
    - ⇒ 2 (the word "is" has a length of 2)
- echo ${#array[@]}
    - ⇒ 4 (the array contains 4 elements)

Attention: The number of used elements does not need to conform to the highest index. Sparse arrays are possible in Bash, that means you can have 4 elements, but with indexes 1, 7, 20, 31. **You can't loop through such an array with a counter loop based on the number of elements!**

Substring expansion

${PARAMETER:OFFSET}

${PARAMETER:OFFSET:LENGTH}

This one can expand only a **part** of a parameter's value, given a **position to start** and maybe a **length**. If LENGTH is omitted, the parameter will be expanded up to the end of the string. If LENGTH is negative, it's taken as a second offset into the string, counting from the end of the string.

OFFSET and LENGTH can be **any** arithmetic expression. **Take care:** The OFFSET starts at 0, not at 1!

Example string (a quote from a big man): MYSTRING="Be liberal in what you accept, and conservative in what you send"

Using only Offset

In the first form, the expansion is used without a length value, note that the offset 0 is the first character:

```
echo ${MYSTRING:34}
```

⇒ ~~Be liberal in what you accept, and~~ conservative in what you send

Using Offset and Length

In the second form we also give a length value:

```
echo ${MYSTRING:34:13}
```

⇒ ~~Be liberal in what you accept, and~~ conservative~~ in what you send~~

Negative Offset Value

If the given offset is negative, it's counted from the end of the string, i.e. an offset of -1 is the last character. In that case, the length still counts forward, of course. One special thing is to do when using a negative offset: You need to separate the (negative) number from the colon:

```
${MYSTRING: -10:5}
${MYSTRING:(-10):5}
```

Why? Because it's interpreted as the parameter expansion syntax to use a default value.

Negative Length Value

If the `LENGTH` value is negative, it's used as offset from the end of the string. The expansion happens from the first to the second offset then:

```
echo "${MYSTRING:11:-17}"
```

⇒ ~~Be liberal~~ in what you accept, and conservative~~ in what you send~~

This works since Bash 4.2-alpha, see also Bash changes.

Substring/Element expansion: Arrays

For arrays, this expansion type has again 2 meanings:

- For **individual** elements, it expands to the specified substring (as for every “normal” parameter)
- For the **mass subscripts** `@` and `*` it mass-expands individual array elements denoted by the 2 numbers given (*starting element*, *number of elements*)

Example:

Assume: `array=(This is a text)`

- `echo ${array[0]:2:2}`
  - ⇒ `is` (the "is" in "This", array element 0)
- `echo ${array[@]:1:2}`
  - ⇒ `is a` (from element 1 inclusive, 2 elements are expanded, i.e. element 1 and 2)

Use a default value

`${PARAMETER:-WORD}`

`${PARAMETER-WORD}`

If the parameter `PARAMETER` is unset (never was defined) or null (empty), this one expands to `WORD`, otherwise it expands to the value of `PARAMETER`, as if it just was `${PARAMETER}`. If you omit the `:` (colon), like shown in the second form, the default value is only used when the parameter was **unset**, not when it was empty.

```
echo "Your home directory is: ${HOME:-/home/$USER}."
echo "${HOME:-/home/$USER} will be used to store your personal data."
```

If `HOME` is unset or empty, everytime you want to print something useful, you need to put that parameter syntax in.

```
#!/bin/bash

read -p "Enter your gender (just press ENTER to not tell us): " GENDER
echo "Your gender is ${GENDER:-a secret}."
```

It will print "Your gender is a secret." when you don't enter the gender. Note that the default value is **used on expansion time**, it is **not assigned to the parameter**.

Use a default value: Arrays

For arrays, the behaviour is very similar. Again, you have to make a difference between expanding an individual element by a given index and mass-expanding the array using the `@` and `*` subscripts.

- For individual elements, it's the very same: If the expanded element is NULL or unset (watch the `:-` and `-` variants), the default text is expanded
- For mass-expansion syntax, the default text is expanded if the array
  - contains no element or is unset (the `:-` and `-` variants mean the **same** here)
  - contains only elements that are the nullstring (the `:-` variant)

In other words: The basic meaning of this expansion type is applied as consistent as possible to arrays.

Example code (please try the example cases yourself):

```
####
# Example cases for unset/empty arrays and nullstring elements
####


### CASE 1: Unset array (no array)

# make sure we have no array at all
unset array

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}

### CASE 2: Set but empty array (no elements)

# declare an empty array
array=()

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}


### CASE 3: An array with only one element, a nullstring
array=("")

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}


### CASE 4: An array with only two elements, a nullstring and a normal word
array=("" word)

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}
```

Assign a default value

`${PARAMETER:=WORD}`

`${PARAMETER=WORD}`

This one works like the using default values, but the default text you give is not only expanded, but also **assigned** to the parameter, if it was unset or null. Equivalent to using a default value, when you omit the `:` (colon), as shown in the second form, the default value will only be assigned when the parameter was **unset**.

```
echo "Your home directory is: ${HOME:=/home/$USER}."
echo "$HOME will be used to store your personal data."
```

After the first expansion here (`${HOME:=/home/$USER}`), `HOME` is set and usable.

Let's change our code example from above:

```
#!/bin/bash

read -p "Enter your gender (just press ENTER to not tell us): " GENDER
echo "Your gender is ${GENDER:=a secret}."
echo "Ah, in case you forgot, your gender is really: $GENDER"
```

Assign a default value: Arrays

For arrays this expansion type is limited. For an individual index, it behaves like for a "normal" parameter, the default value is assigned to this one element. The mass-expansion subscripts `@` and `*` **can not be used here** because it's not possible to assign to them!

Use an alternate value

`${PARAMETER:+WORD}`

`${PARAMETER+WORD}`

This form expands to nothing if the parameter is unset or empty. If it is set, it does not expand to the parameter's value, **but to some text you can specify**:

```
echo "The Java application was installed and can be started.${JAVAPATH:+ NOTE: JAVAPATH seems to be set}"
```

The above code will simply add a warning if `JAVAPATH` is set (because it could influence the startup behaviour of that imaginary application).

Some more unrealistic example... Ask for some flags (for whatever reason), and then, if they were set, print a warning and also print the flags:

```
#!/bin/bash

read -p "If you want to use special flags, enter them now: " SPECIAL_FLAGS
echo "The installation of the application is finished${SPECIAL_FLAGS:+ (NOTE: there are special flags set: $SPECIAL_FLAGS)}."
```

If you omit the colon, as shown in the second form (`${PARAMETER+WORD}`), the alternate value will be used if the parameter is set (and it can be empty)! You can use it, for example, to complain if variables you need (and that can be empty) are undefined:

```
# test that with the three stages:

# unset foo
# foo=""
# foo="something"

if [[ ${foo+isset} = isset ]]; then
  echo "foo is set..."
else
  echo "foo is not set..."
fi
```

Use an alternate value: Arrays

Similar to the cases for arrays to expand to a default value, this expansion behaves like for a "normal" parameter when using individual array elements by index, but reacts differently when using the mass-expansion subscripts `@` and `*`:

- For individual elements, it's the very same: If the expanded element is **not** NULL or unset (watch the :+ and + variants), the alternate text is expanded
- For mass-expansion syntax, the alternate text is expanded if the array
    - contains elements where min. one element is **not** a nullstring (the :+ and + variants mean the same here)
    - contains **only** elements that are **not** the nullstring (the :+ variant)

For some cases to play with, please see the code examples in the description for using a default value.

Display error if null or unset

`${PARAMETER:?WORD}`

`${PARAMETER?WORD}`

If the parameter `PARAMETER` is set/non-null, this form will simply expand it. Otherwise, the expansion of `WORD` will be used as appendix for an error message:

```
$ echo "The unset parameter is: ${p_unset?not set}"
bash: p_unset: not set
```

After printing this message,

- an interactive shell has $? to a non-zero value
- a non-interactive shell exits with a non-zero exit code

The meaning of the colon (`:`) is the same as for the other parameter expansion syntaxes: It specifies if

- only unset or
- unset and empty parameters

are taken into account.

Code examples

Substring removal

Removing the first 6 characters from a text string:

```
STRING="Hello world"

# only print it
echo "${STRING#??????}"

# store it
STRING=${STRING#??????}
```

See also

- Internal: Introduction to expansion and substitution
- Internal: Arrays
- Dictionary, internal: Parameter

Discussion

Stan R., 2010/06/20 01:48

Found an error under the "Search and replace" sub-heading, near the beginning of the "Anchoring" section, where there is an example snippet:

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING//#x/y}  # RESULT: yxxxxxxxxx
echo ${MYSTRING//%x/y}  # RESULT: xxxxxxxxxy
```

This should be:

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING/#x/y}  # RESULT: yxxxxxxxxx
echo ${MYSTRING/%x/y}  # RESULT: xxxxxxxxxy
```

The difference is MYSTRING/ instead of MYSTRING//, since you cannot have both anchoring and .

Jan Schampera, 2010/06/20 14:56

Thanks for this finding. A late night typo, maybe :)

Stan R., 2010/06/21 03:11

Glad I could help.

Shmerl, 2010/10/20 18:37

I found an error. You write:

${PARAMETER:-WORD} ${PARAMETER-WORD} If the parameter PARAMETER is unset (never was defined) or null (empty), this one expands to WORD, otherwise it expands to the value of PARAMETER, as if it just was ${PARAMETER}. ***If you omit the : (colon), like shown in the second form, the default value is only used when the parameter was unset, not when it was empty.***

Running a simple test

`str=""; echo ${str-test_result}`

Produces:

test_result

Now running:

`unset str; echo ${str-test_result}`

Produces nothing. So the above is just the opposite. It should be: If you omit the : (colon), like shown in the second form, the default value is only used when ***the parameter was empty, not when it was usnet.***

Jan Schampera, 2010/10/20 23:13

It works (and always worked) for me:

```
bonsai@core:~$ str=""; echo ${str-test_result}

bonsai@core:~$ unset str; echo ${str-test_result}
test_result
bonsai@core:~$
```

I'm not sure what could be the reason for your tests to do the exact opposite. Is this really a Bash? Though, this behaviour is specified by POSIX(R), too: ***"In the parameter expansions [shown previously], use of the <colon> in the format shall result in a test for a parameter that is unset or null; omission of the <colon> shall result in a test for a parameter that is only unset."***

Do you have any way to find out where your (or my) problem is here?

Shmerl, 2010/10/21 17:59

Sorry, I really meant the case with plus.

Run this test:

str=""; echo ${str+'test_result'}

produces: ***test_result***

unset str; echo ${str+'test_result'}

produces: (nothing)

I was wrong about other cases, this difference only applies to + (alternative value) case.

Shmerl, 2010/10/21 18:03

Make sure there is a plus there - the preview ate it.

Jan Schampera, 2010/10/23 11:36

It does exactly what it should do. + produces an alternate value, means it expands to something ***instead*** *of the real value of the parameter, if the parameter is "set" (+) or "set or null" (:+).*

Shmerl, 2010/10/24 06:20

Yes, but your text in the wiki above is misleading. It writes:

If you omit the colon, as shown in the second form (${PARAMETER+WORD}), the alternate value will not be used when the parameter is empty, only if it is ***unset!***

While it should say:

If you omit the colon, as shown in the second form (${PARAMETER+WORD}), the alternate value will not be used when the parameter is ***unset****, only if it is* ***empty****!*

Jan Schampera, 2010/10/24 08:09

Oh, yes! Sorry, it took a while to see what you mean :(

I rephrased it a bit. If you're not okay with it, feel free to change it. Thanks for pointing it out.

Shmerl, 2010/10/20 18:38

Actually the previous note relates to all similar examples with :-, :=, etc.

my, 2011/01/25 11:55

the ${ ^^ } and friends have a pattern parameter. But it seems anything beyond [ ] does not work.

```
TEST="abcABCxyz"

echo ${TEST^[abc]} # change first char if it is a||b||c
echo ${TEST^![abc]} # change first char if it is NOT a||b||c

TEST="hello world"
echo ${TEST^^[aeiou]}    # change ANY char that is in aeiou
```

```
AbcABCxyz
abcABCxyz
hEllO wOrld
```

Jan Schampera, 2011/02/01 06:30

I don't see what doesn't work here. Did you mean

```
${TEST^[!abc]}
# (instead of ${TEST^![abc]})
```

here?

Frederick Grose, 2011/02/11 23:43

In GNU bash, 4.1.7(1)-release (x86_64-redhat-linux-gnu),

the 'Negative Length Value' syntax,

```
echo "${MYSTRING:11:-17}"
```

results in the following:

```
bash: -17: substring expression < 0
```

LENGTH seems to behave like the array element COUNT.

Jan Schampera, 2011/02/12 09:44

Hello Frederick,

you're absolutely right, this is a Bash 4.2 feature. I added a note.