Unimal 2.0

Application note 5

Common string conversions

Documentation revision 2.00

Techniques:

Conversions between numbers and strings
Replacing characters in strings (e.g., converting to uppercase)
Generating an error message algorithmically

MacroExpressions
http://www.macroexpressions.com

Table of contents

Foreword

This Application Note illustrates matters that are simple but might not be immediately obvious.

First, we consider how to convert back and forth between numeric and string macro parameters. Then, using a similar technique, we'll convert a string-valued macro parameter to an upper-case string. This second example can serve as a generic template for character-based string transformations.

It is worth noting that the techniques illustrated here do not depend on character encoding in Unimal strings.

Examples for this Note are in the directory Samples\AppNotes\5.

Converting a number to a string

A simple problem we'll be solving here is to convert a number, such as 12345, to a string, such as "12345".

We'll assume there is a macro parameter that has a numeric value of interest, e.g.
`#MP number = 12345`

We know that `%dnumber` is a composite name resolving to, in our case, the name 12345. Of course this is not a valid name to be used literally, but it is a valid name which we can access as a composite name.

Next, we know that, given a `name`, `{name}` is a string expression whose value is `name`.

So, `{%dnumber}` is a string with the value of string (decimal) representation of the `number`.

Quite similarly, `{%08Xanother_number}` is a string representation of `another_number` as an 8-character hex number, perhaps padded with leading zeros.

The file num2str.u that illustrates this technique using a simple macro to render the result:

```
#MP Macro print ;(name)
#mp%n#1# = #mp%d#1# (as numeric) = #mp%s#1# (as string)
#MP Endm
```

It simply prints the name of the argument and its numeric and string values.

Its first example is

```
#MP number = 12345
#MP Setstr number = {%dnumber}
#MP Expand print(number)
```

This produces the output as expected:

```
number = 12345 (as numeric) = 12345 (as string)
```

The second example tries (as we'll see, erroneously) to use a composite name `another_number_%dnumber`, which of course resolves to `another_number_12345`:

```
#MP another_number_%dnumber = 0x9ABCDEF
#MP Setstr another_number_%dnumber =
{%08xanother_number_%dnumber}
#MP Expand print(another_number_%dnumber)
```

Unimal prints

```
MP:S2011:num2str.u:10 Undefined parameter another_number_; default assumed
another_number_12345 = 162254319 (as numeric) = 0000000012345 (as string)
```

What went wrong here?

Well, our intention in `%08xanother_number_%dnumber` was to render `another_number_%dnumber` with the format `%08x`. But that is not what we wrote: `%08xanother_number_%dnumber` is a composite name with an empty base and two suffixes, `another_number_` and `number`, rendered with formats `%08x` and `%d` respectively. When Unimal resolves this name, it discovers that the first prefix, `another_number_`, is not defined and generates the error.

So, we need to use a simple name in this type of conversion; we can always do that via an intermediate assignment as the last example in num2str.u illustrates:

```
#MP another_number = another_number_%dnumber
#MP Setstr another_number_%dnumber = {%08xanother_number}
#MP Expand print(another_number_%dnumber)
```

This prints the correct output:
another_number_12345 = 162254319 (as numeric) = 09abcdef (as string)

Converting a string to a number

Let's tackle a task of converting a string representing a hexadecimal number to numeric format. For instance, we want to convert a string "AbCd" to the number 43981 (which is ABCD hex).

We'll use a common algorithm:

1. Initialize result to 0
2. Extract one character of the string at a time going left to right
3. Multiply the previous result by 16 and add the hex digit represented by the character
4. Continue to step 2 until all characters in the string are used up.

What we also expect to happen is to generate an error if:

- There is an overflow in computing the result
- The string contains a character that is not a hex digit

We will implement step 2 by extracting a one-character string from a string passed as the first argument to a macro:

```
#MP     Setstr digit_string = {uSubstr, #1#, n, n+1}
```

Here, `n` is a zero-based number of the current position in the argument string.

The seemingly un-Unimal part is in step 3 is to find the numeric value corresponding to `digit_string`. Do accomplish this, we simply define a few macro parameters such that a composite name `xdigit_%sdigit_string` resolves to the numeric value of `digit_string`, or is undefined if `digit_string` is not in fact a name of a hex digit.

Here we go (see also the file str2num.u):

```
#MP xdigit_0 = 0
#MP xdigit_1 = 1
#MP xdigit_2 = 2
#MP xdigit_3 = 3
#MP xdigit_4 = 4
#MP xdigit_5 = 5
#MP xdigit_6 = 6
#MP xdigit_7 = 7
#MP xdigit_8 = 8
#MP xdigit_9 = 9
#MP xdigit_a = 10
#MP xdigit_A = 10
#MP xdigit_b = 11
#MP xdigit_B = 11
#MP xdigit_c = 12
#MP xdigit_C = 12
#MP xdigit_d = 13
#MP xdigit_D = 13
#MP xdigit_e = 14
#MP xdigit_E = 14
```

```
#MP xdigit_f = 15
#MP xdigit_F = 15
```

Now we are in a position to define the string-to-hex conversion macro:

```
#MP Macro str2num ;(string)
#MP result = 0
#MP strlen = Ustrlen(#1#)   ;get the number of digits
#MP For n=0,strlen - 1
#MP     Setstr digit_string = {uSubstr, #1#, n, n+1}
#MP     result = 16*result + xdigit_%sdigit_string
#MP Endfor
#MP Undef n {NUM}            ;cleanup
#MP Undef strlen {NUM}       ;cleanup
#MP Undef digit_string{STR}  ;cleanup
#MP Endm
```

This macro can be easily modified for any radix: the multiplier will change (it is the radix) and the definitions of one-character strings mapping to their numeric values will have to be defined for the radix. That's all there is to it.

Let's run a few examples:

```
#MP Expand str2num("00000000000000000000002006")
result = #mp%xresult
```

This will print
result = 2006

```
#MP Expand str2num("FDA")
result = #mp%xresult
```

This will print
result = fda

```
#MP Expand str2num("DOD")
result = #mp%xresult
```

This will print something like
MP:S2011:str2num.u:29 Undefined parameter xdigit_O; default assumed
result = d0d

```
#MP Expand str2num("7fffffff")
result = #mp%xresult
```

This will print
result = 7fffffff

```
#MP Expand str2num("2006000000")
result = #mp%xresult
```

This will print something like
MP:M3502:str2num.u:29 Multiplication overflow; result 2147483647 assumed

MP:M3502:str2num.u:29 Multiplication overflow; result 2147483647 assumed
result = 7fffffff

So, we've got a conversion macro which detects invalid characters (using the way we decided to convert one-character strings) and arithmetic overflows (using Unimal's built-in range check).

What may be considered odd is a failure of the following test:

```
#MP Expand str2num("80000000")
result = #mp%xresult
```

The output will be something like

MP:M3502:str2num.u:29 Multiplication overflow; result 2147483647 assumed
result = 7fffffff

One can argue that the behavior is correct: 80000000 is outside the range of 32-bit signed numbers Unimal uses. However, hexadecimal numbers are often and routinely used as bitmaps in bitwise logic operations, so for the radix 16 (unlike any other radix), we'd like to make an exception. That exception results in a separate macro below:

```
#MP Macro str2hex ;(string)
#MP result = 0
#MP strlen = Ustrlen(#1#)   ;get the number of digits
#MP For n=0,strlen - 1
#MP     Setstr digit_string = {uSubstr, #1#, n, n+1}
#MP     If ((result<<4)>>4) != result                ;(1)
#MP         Error "result overflow"                  ;(2)
#MP     Endif                                        ;(3)
#MP     result = (result<<4) | xdigit_%sdigit_string ;(4)
#MP Endfor
#MP Undef n {NUM}            ;cleanup
#MP Undef strlen {NUM}       ;cleanup
#MP Undef digit_string{STR}  ;cleanup
#MP Endm
```

Compared to the macro `str2num`, the new macro `str2hex` has line (4) changed and lines (1), (2), (3) added. The line (4) is changed to reflect our bitwise logic inspiration and also to eliminate overflow errors.

Lines (1)-(3) are added in the assumption that we do want to detect overflow beyond (unsigned) 32 bits. Line (1) checks if left shift of result by four bits would shift out any non-zero bits. Line (2) generates an error in that case.

Generating an error message algorithmically

It is worth discussing in some detail the line (2) above. Unimal doesn't have an Error statement, so how do we report an error with a sensible text? When Unimal chokes on the syntax, it reports a syntax error with a text about the offending token, like
Bad syntax near <token>.
In line (2), Unimal finds two tokens, `Error`, which to Unimal is a name of a macro parameter, and `"result overflow"` which is a string literal. There is bad syntax with a promising beginning, so the offending token is the string literal, and it will be printed in an error message.

For an example, let's run

```
#MP Expand str2hex("2006000000")
result = #mp%xresult
```

The output is something like

```
MP:S2001:str2num.u:57 Bad syntax near "result overflow"; statement ignored
result = 6000000
```

This is a generic way to report an error with a sensible message; you can always use it for error conditions detected algorithmically.

Converting a string to UPPERCASE

In this section, we will devise a simple macro to convert all lower-case characters in a string to upper case (see `toupper.u`). We will use the same technique as in the previous section, but this time `result` will be a string.

First, we define a bunch of uppercase one-character strings to map the lowercase characters:

```
#MP Setstr toupper_a = "A"
#MP Setstr toupper_b = "B"
#MP Setstr toupper_c = "C"
#MP Setstr toupper_d = "D"
#MP Setstr toupper_e = "E"
#MP Setstr toupper_f = "F"
#MP Setstr toupper_g = "G"
#MP Setstr toupper_h = "H"
#MP Setstr toupper_i = "I"
#MP Setstr toupper_j = "J"
#MP Setstr toupper_k = "K"
#MP Setstr toupper_l = "L"
#MP Setstr toupper_m = "M"
#MP Setstr toupper_n = "N"
#MP Setstr toupper_o = "O"
#MP Setstr toupper_p = "P"
#MP Setstr toupper_q = "Q"
#MP Setstr toupper_r = "R"
#MP Setstr toupper_s = "S"
#MP Setstr toupper_t = "T"
#MP Setstr toupper_u = "U"
#MP Setstr toupper_v = "V"
#MP Setstr toupper_w = "W"
#MP Setstr toupper_x = "X"
#MP Setstr toupper_y = "Y"
#MP Setstr toupper_z = "Z"
```

Here is the macro:

```
#MP Macro toupper ;(string)
#MP Setstr result = ""
#MP strlen = Ustrlen(#1#)
#MP For n=0,strlen - 1
#MP     Setstr letter_string = {uSubstr, #1#, n, n+1}
#MP     Ifdef toupper_%sletter_string {STR}                ;(1)
#MP         Setstr letter_string = toupper_%sletter_string ;(2)
#MP     Endif
#MP     Setstr result = {uJoin, result, letter_string}     ;(3)
#MP Endfor
```

```
#MP Endm
```

It works very similarly to `str2num` by iterating over all characters in the argument string (and extracting them into `letter_string`). The cleanup statements are omitted for brevity.

By our construction, the one-character string `letter_string` holds a lowercase character if and only if the macro parameter with a composite name `toupper_%sletter_string` is defined as a string. If `letter_string` holds a lowercase character, as tested in line (1), we replace it with its uppercase counterpart, line (2). Whatever the final content of `letter_string`, we append it to the current `result` in line (3). (We tacitly assumed that `uJoin` is not defined, and thus that no "spacer" substrings will be added. It may be prudent to undefine `uJoin` in the beginning of the macro.)

Here is a sample use of this new macro:

```
#MP Expand toupper("year 2006")
result = #mp%sresult
#MP Expand toupper("Unimal can do things!")
result = #mp%sresult
#MP Expand toupper(";!@#$aaabbb%^&*")
result = #mp%sresult
```

The output is as expected:

```
result = YEAR 2006
result = UNIMAL CAN DO THINGS!
result = ;!@#$AAABBB%^&*
```