Unimal 2.1

Application note 8

Implementing complex string algorithms at compile time

Documentation revision 2.1.1

Techniques:

Representation of pre-defined encoding of strings in predefined charset
Generating tree-like structures and recursive macro expansion
Generating a containing superstring and multi-suffix composite names
Pretty-formatting the output

MacroExpressions
http://www.macroexpressions.com

Table of contents

Foreword

This Application Note deals with non-trivial string-related algorithms which, given constant data, can be executed at compile time.

First, we consider how to encode a string into a pre-defined character map, such as provided by the character generator built in a device. One can find similarities between this and a technique used in Application Note 5 to convert a string-valued macro parameter to an upper-case string.

Second, we'll develop a technique of generating output with better formatting.

Third, we will generate a simple tree representation of a collection of strings. This will be an excuse to elaborate on recursive macro expansion in Unimal.

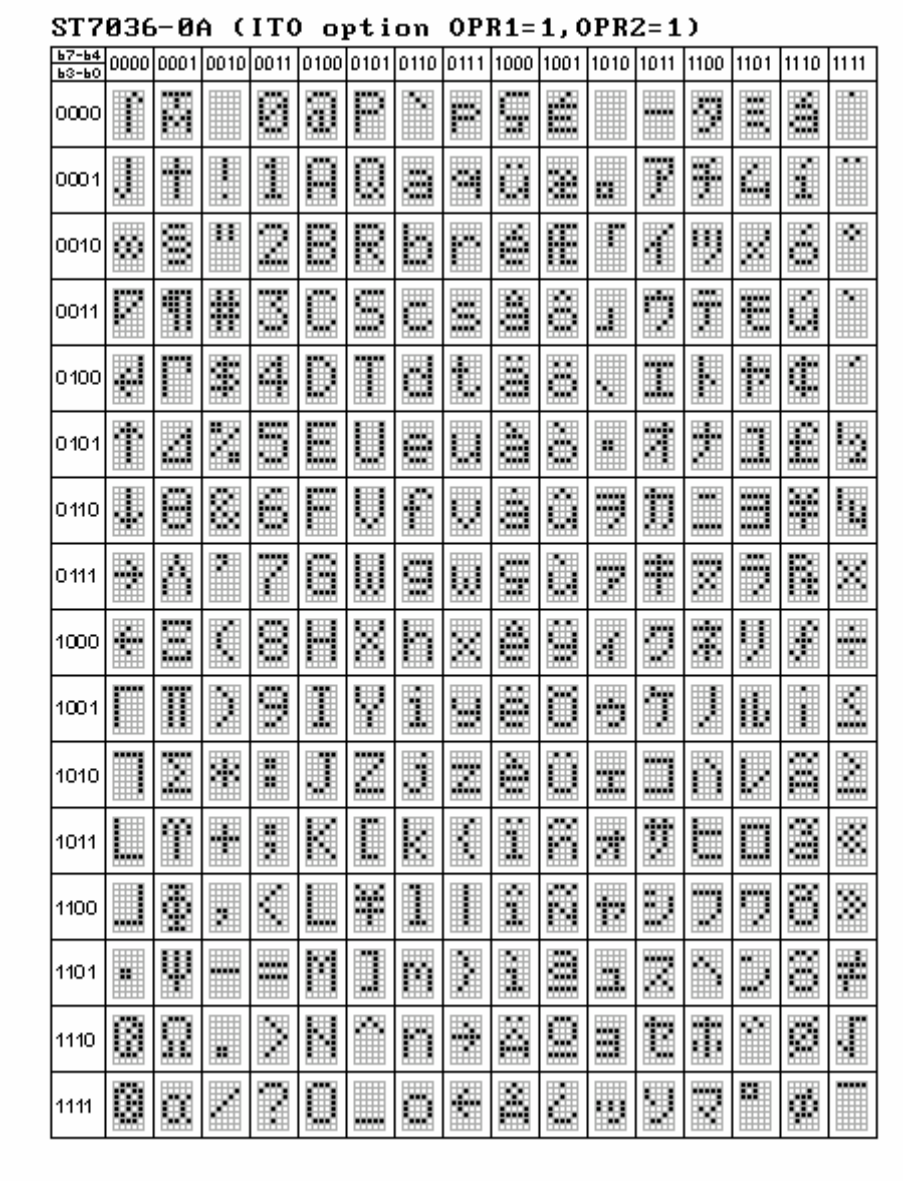
Finally, we'll undertake a rather formidable task of building a containing superstring for a collection of strings using a variation of a commonly used greedy algorithm. This will provide a practical illustration of the use of composite names with more than one suffix.

Examples for this Note are in the directory Samples\AppNotes\8.

Representation of pre-defined encoding of strings

Consider a problem of outputting a readable string to a device with its own character generator. You send it a byte and it draws a character coded with this byte. This encoding is likely to include ASCII but it doesn't have to; it may include other characters as well.

An example of such a device is Sitronix ST7036 LCD controller (http://www.sitronix.com.tw/sitronix/SASpecDoc.nsf/FileDownload/ST70361542382/$FILE/ST7036-V1_6.pdf) with one of its several character maps shown below just for illustration.



It contains the printable subset of ASCII, Kana characters, some Western European and Greek characters and some math and special characters.

How do we define a string with this encoding?

We begin with defining the 256 single-character strings, so that we can concatenate them later to make arbitrary strings.

For characters in the Unimal names set (_, A-Z, a-z, 0-9), a solution is to code the encoding directly, like

```
#MP  LETTER_k = 0x6B
#MP  LETTER_K = 0x4B
```

(Note that the equal sign is optional.) Recall that the [] in this context makes a single-byte string with the byte encoded by the least significant byte of the number in brackets.
Here are some more:

```
#MP  LETTER_S = 0x53
```

```
#MP  LETTER_T = 0x54
#MP  LETTER_0 = 0x30
#MP  LETTER_3 = 0x33
#MP  LETTER_6 = 0x36
#MP  LETTER_7 = 0x37
```

Now, incidentally, we are in a position to encode the string "ST7036":

```
#MP Setstr s = "ST7036"
#MP Setstr result = ""
#MP For i=0, Ustrlen(s)-1
#MP   Setstr t = {uSubstr,s,i,i+1} ;`character' #i of original string
#MP   Setstr result = result + [LETTER_%st]
#MP Endfor
```

This simple example is in the file encode1.u which prints `s` and `result` as strings using

```
s="#mp%ss"
result="#mp%sresult"
```

If your system contains ASCII encoding, the printed result is

```
s="ST7036"
result="ST7036"
```

We've built `result` byte by byte by replacing the desktop encoding (ASCII, EBCDIC or anything else) to the ST7036 controller encoding.

But what about characters that are illegal in Unimal names, like e.g. a space or parentheses? We cannot have something like

```
#MP LETTER_( = 0x28 ;wrong!
```

Recall that any characters are valid in a Unimal name; just not all characters may appear graphically like in the example below. So, we can use a two-step solution, like so:

```
#MP Setstr _LPAREN_ "("
#MP LETTER_%s_LPAREN_ = 0x28
#MP Setstr _SPACE_ " "
#MP LETTER_%s_SPACE_ = 0x20
```

Now, what about other characters which may not even be in the character set of your programmer's editor (arrows, Kana, Greek etc.)? To approach this problem, we have to decide how the original strings with those characters could be presented.

Perhaps the most natural way is to use symbolic names for those characters and use C-style literal string concatenation like so:

```
#MP Setstr say = _sqroot_ " is math, " _theta_ " Greek, " _TSU_ " Japanese"
```

If we adopt this method of presentation, it doesn't really matter how exactly the symbolic names are defined; we can define them for instance with the target encoding:

```
#MP Setstr _sqroot_ = [254] ;a decimal for a change
#MP Setstr _theta_ = [0x16]
#MP Setstr _TSU_ = [0xC2]
```

Now we can define the letters:

```
#MP Setstr LETTER_%s_sqroot_ = [254] ;a decimal for a change
```

```
#MP Setstr LETTER_%s_theta_ = [0x16]
#MP Setstr LETTER_%s_TSU_ = [0xC2]
```

It begins looking tautological but there is nothing wrong with this.

Many (but not all) of the 256 letter definitions of this kind can be found in the file `st.inc`

Now that we can convert a string to a target encoding, how do we output it to the preprocessing result? It makes sense to output a string as a sequence (array) of encoding bytes for the target language to have an object ready for output.

The following macro (see `encode2.u`) does the job for C:

```
1. #MP Macro PrintStringAsHex ;(encoded-string)
2. #MP For i=0,Ustrlen(#1#)-1
3. #MP   Setstr letter = {uSubstr, #1#, i, i+1}
4. #MP   letter = LETTER_%sletter
5.   0x#mp%02xletter,
6. #MP Endfor
7. #MP Endm
```

Line 5 prints a C-style initialization with a hex code defined by a single-character substring.

By the way, `encode2.u` also contains a generic macro to encode a string in a fashion we encoded "ST7036" earlier in this section:

```
1. #MP Macro EncodeString
2. #MP Setstr result = ""
3. #MP For i=0, Ustrlen(#1#)-1
4. #MP   Setstr t = {uSubstr,#1#,i,i+1} ;`character' #i of the original
5. #MP   Setstr result = result + [LETTER_%st]
6. #MP Endfor
7. #MP Endm
```

A test looks like this:

```
const unsigned char one[]=
{
#MP Encode("ST7036")
#MP PrintStringAsHex(result)
};
const unsigned char two[]=
{
#MP Setstr S _sqroot_ "4 = 2"
#MP EncodeString(S)
#MP PrintStringAsHex(result)
};
```

Here is the test output which has the `ST7036` the encoding regardless of the character encoding in a programmer's editor:

```
const unsigned char one[]=
{
  0x53,
  0x54,
  0x37,
  0x30,
```

```
  0x33,
  0x36,
};
const unsigned char two[]=
{
  0xfe,
  0x34,
  0x20,
  0x3d,
  0x20,
  0x32,
};
```

One could notice that there is no point to encode an input string only to tear the `result` apart again to produce the output. That is true, and the two macros can be combined; we'll keep them separate because we will soon manipulate the strings before any output is produced.

Pretty printing

When a string is long, the kind of output we produced is ugly. Consider replacing the string S in `encode2.u` like this:

```
const unsigned char two[]=
{
#MP Setstr S _sqroot_ "4 = 2. There is no limit to the power of math!"
#MP EncodeString(S)
#MP PrintStringAsHex(result)
};
```

The output contains 47 lines, one line per character. Things get unreadable very quickly. We need a prettier output mechanism to print several bytes on a line.

We want to replace the macro `PrintStringAsHex` with some `PrettyPrintStringAsHex` which can be given the number of bytes per line (see `encode3.u`). The plan is to pre-assemble an output line and then just dump it with the %s format. Here is an implementation:

```
1. #MP Macro PrettyPrintStringAsHex ;(encoded-string, bytes-per-line)
2. #MP Set line 0
3. #MP Repeat
4. #MP     index = line*#2#
5. #MP     Setstr line = ""
6. #MP     For b = 0, #2#-1
7. #MP         ix = index + b
8. #MP         Setstr let = {uSubstr, #1#, ix, ix+1}
9. #MP         If Ustrlen(let) == 0
10.   #MP             b = #2# ;break the loop
11.   #MP         Else
12.   #MP             let = LETTER_%slet
13.   #MP             Setstr line = line + "0x" + {%02xlet} + ","
14.   #MP         Endif
15.   #MP     Endfor
```

```
16.   /*[#mp%03uindex]*/ #mp%sline
17.   #MP     Set line line+1
18.   #MP While Ustrlen(let) != 0
19.   #MP Endm
```

In this macro, `line` has a numeric value – the line number – and a string value – the content of the output line. The parameter `index` is the index of the first byte on the `line` (see line 4 above).
In the code above, line 16 outputs this `index` in a C comment (this goes to the category of a cherry on top), followed by the assembled `line` string.
This output occurs while the character `let` extracted from the encoded string is non-empty (see lines 2, 3, 17, 18). When `let` is empty, the input string is exhausted and the output is completed.
Lines 5-15 assemble an output `line` by extracting and processing non-empty characters (`let`) of the input string one by one. The assembly terminates when enough bytes are added (line 15) or when the input string ends (line 10). The actual assembly happens in line 13; `%02xlet` is a composite name representing the two hex digits of `let`, and enclosing a name in braces converts it to a string with the value of the name. This explains line 13 (see also Application Note 5).

Now let's process the code in the beginning of this section with a printing replacement (encode3.u):

```
const unsigned char two[]=
{
#MP Setstr S _sqroot_ "4 = 2. There is no limit to the power of math!"
#MP EncodeString(S)
#MP PrettyPrintStringAsHex(result, 10) ;10 bytes per line
};
```

Here is the output, which is much more compact:

```
const unsigned char two[]=
{
    /*[000]*/ 0xfe,0x34,0x20,0x3d,0x20,0x32,0x2e,0x20,0x54,0x68,
    /*[010]*/ 0x65,0x72,0x65,0x20,0x69,0x73,0x20,0x6e,0x6f,0x20,
    /*[020]*/ 0x6c,0x69,0x6d,0x69,0x74,0x20,0x74,0x6f,0x20,0x74,
    /*[030]*/ 0x68,0x65,0x20,0x70,0x6f,0x77,0x65,0x72,0x20,0x6f,
    /*[040]*/ 0x66,0x20,0x6d,0x61,0x74,0x68,0x21,
};
```

Trees and Recursive Macro Expansions

There are many text processing algorithms based on tree-like data structures. The most elegant ways of building trees of various sorts are based on recursive calls. The purpose of this section is to implement a simple compile-time tree-building algorithm in Unimal and to discuss how recursion is treated in Unimal.

When the text is constant, it has a lot of merits to construct the relevant data structures at compile time. This section illustrates the ways of doing this job with Unimal.

We will consider a very simple tree data structure: a node is tagged with a character and references its parent node; the root node has an empty reference.

Such a tree can store a collection of strings with a help of an extra array of references to the tree nodes: a string is a sequence of tags encountered in walking the tree from the referenced node to the root.

Our task will be to construct such a tree representing a given collection of strings.

The data design is as follows:

- The tree itself will be represented by a pair of arrays:
  `const unsigned char tree_letters[]`, and
  `const unsigned short tree_parents[]`.
  They are indexed by the same index; `tree_letters[i]` is the letter tag of the node and `tree_parents[i]` is the index of the parent node. By convention, index 0 corresponds to the root of the tree.
- Strings in the collection we build the tree from are enumerated in the order of appearance. They produce an array,
  `const unsigned short name_index[]`,
  indexed by the string number. The value of `name_index[n]` is the index (in `tree_letters` and `tree_parents`) of the node corresponding to (the first letter of) the string `n`.
- For readability, C99-style designated initializers and sensible comments will be generated as appropriate

Let's assume that Unimal somehow got the following parameters computed:

- `Lindex` – the number of strings in the collection
- `Tindex` – the number of nodes in the tree
- For each `t` = 0, …, `Tindex` – 1,
  `NODELETTER_%dt` has a string value of the character tag of the node, and a numeric value of the encoding of the character;
  `PARENT_%dt` has a numeric value of the parent index of the node t, and a string value of the (suffix) string resulting from walking up the tree from the parent node (for comments only)

Then generating the tree representation is a standard fare (see `tree.u`)

```
const unsigned short name_index[#mp%dLindex] =
{
#MP For i=0, Lindex-1
    [#mp%di] = #mp%dLINDEX_%di, //link to "#mp%sLINDEX_%di"
#MP Endfor
};

const unsigned char tree_letters[#mp%dTindex] = {
#MP For t=0, Tindex-1
    [#mp%dt] = #mp%dNODELETTER_%dt, //letter "#mp%sNODELETTER_%dt"
#MP Endfor
};

const unsigned short tree_parents[#mp%dTindex] = {
#MP For t=0, Tindex-1
    [#mp%dt] = #mp%dPARENT_%dt, //link to "#mp%sPARENT_%dt"
#MP Endfor
};
```

The strings are defined using the yet-to-be-defined macro AddString, e.g.

```
#MP AddString("good")
#MP AddString("mood")
#MP AddString("bad")
#MP AddString("fad")
#MP AddString("good")
```

The macro AddString might take the form (see the file tree.inc)

```
1. #MP Macro AddString ;(string)
2. #MP AddSuffix(#1#)
3. #MP LINDEX_%dLindex = TINDEX_%s#1#
4. #MP Setstr LINDEX_%dLindex = #1# ;for generated comments
5. #MP Lindex = Lindex+1
6. #MP Endm
```

The macro AddSuffix in line 2 (again, yet to be defined) the actual work, and lines 3-5 do the housekeeping. The numeric value of TINDEX_%s<string> is the tree node index corresponding to the <string>. It should come out from AddSuffix.

Of course, we need to initialize the root node values and the string counter. This is done (quite sloppily) in the beginning of tree.inc, so it is executed every time the file is included:

```
#MP Lindex = 0 ;String index
#MP ;Manually put in the empty string
#MP Tindex = 0
#MP TINDEX_ = 0 ;index of the empty string
```

Now, AddSuffix:

```
1. #MP Macro AddSuffix ;[string]
2. #MP If !Defined( TINDEX_%s#1#  )
3. #MP     Setstr Remainder = {uSubstr, #1#, 1, Ustrlen(#1#)}
4. #MP     If Ustrlen(Remainder) != 0
5. #MP         AddSuffix[{%sRemainder}]
6. #MP         Setstr Remainder = {uSubstr, #1#, 1, Ustrlen(#1#)}
7. #MP     Endif
8. #MP     PARENT_%dTindex = TINDEX_%sRemainder
9. #MP     Setstr PARENT_%dTindex = Remainder ;for comments only
10.   #MP     Setstr let = {uSubstr, #1#, 0, 1}
11.   #MP     Setstr NODELETTER_%dTindex = let
12.   #MP     NODELETTER_%dTindex = LETTER_%slet
13.   #MP     TINDEX_%s#1# = Tindex
14.   #MP     Tindex = Tindex + 1
15.   #MP Endif
16.   #MP Endm
```

It is to discuss this macro that this section has been added.
First of all, line 2 checks whether the string to add is already in the tree (as a previously added string or as a suffix of a previously added string). If it is so, there is nothing to do; otherwise, lines 3-14 do the processing.

Line 3 computes the remainder (suffix) of the string. If it is non-empty (line 4) then line 5 adds the remainder (recursively!) and line 6 re-computes the remainder since it will have been corrupted in recursive expansion of `AddSuffix`.
Lines 8-14 do compute the parameters we defined previously, with first letter extraction and translation done much like in the previous section.

Let's discuss lines 5-6 in more detail.

If we run `tree.u`, the output is verifiably correct:

```
const unsigned short name_index[5] =
{
    [0] = 3, //link to "good"
    [1] = 4, //link to "mood"
    [2] = 6, //link to "bad"
    [3] = 7, //link to "fad"
    [4] = 3, //link to "good"
};

const unsigned char tree_letters[8] = {
    [0] = 100, //letter "d"
    [1] = 111, //letter "o"
    [2] = 111, //letter "o"
    [3] = 103, //letter "g"
    [4] = 109, //letter "m"
    [5] = 97, //letter "a"
    [6] = 98, //letter "b"
    [7] = 102, //letter "f"
};

const unsigned short tree_parents[8] = {
    [0] = 0, //link to ""
    [1] = 0, //link to "d"
    [2] = 1, //link to "od"
    [3] = 2, //link to "ood"
    [4] = 2, //link to "ood"
    [5] = 0, //link to "d"
    [6] = 5, //link to "ad"
    [7] = 5, //link to "ad"
};
```

If we replace line 5 with

```
#MP          AddSuffix({%sRemainder})
```

(The only difference is that the argument is passed in parentheses instead of in brackets), the output is wrong and contains error messages like

```
MP:S2022:tree.inc:10 Recursive use of macro AddSuffix; ignored (use [])
MP:S2011:tree.inc:13 Undefined parameter TINDEX_ood; default assumed
MP:S2011:tree.inc:13 Undefined parameter TINDEX_ad; default assumed
```

The first error message is of interest: Unimal refused to expand the macro. Other errors are induced since the algorithm didn't execute correctly. So, what's the deal here?

Unimal macros may have unbalanced block elements (`If`/`Else`/`Endif`, `For`/`Endfor`, `Repeat`/`While`); part of a block may be in one macro, another part in another macro or outside any macro at all. The implication is that Unimal must expand a macro even in a false block, in an attempt to locate an unbalanced block keyword. This causes any recursive macro call to result in infinite recursion. Unimal would chug along until it runs out of memory, and so it was prior to version 2.1. Beginning with version 2.1, Unimal detects recursion, whether direct or mutual, and refuses to expand the offending macro.
By passing the argument(s) in square brackets, the code tells Unimal that the macro is balanced and should not be expanded in a false block. This allows correct recursion to be implemented, as we saw in the example. If you lied to Unimal and in fact the macro is not balanced, the unbalanced keywords there will not be found and the corresponding error messages will be produced.

Now, what is the argument that is passed to `AddSuffix` in the recursive call in line 5? It is a string expression whose value is that of `Remainder` – a copy of `Remainder`, in other words. But why do we need to pass a copy instead of `Remainder` itself? The answer is simple: The next (recursive, i.e. nested) instance of `AddSuffix` expansion will corrupt `Remainder` before it was used as the passed argument. This also explains line 6 where `Remainder` is computed again on return from recursive expansion.

Constructing a containing string with Unimal

The goal of this section is to demonstrate how Unimal can be useful in implementing a computationally complex algorithm. It becomes necessary to pay attention to the efficiency of the preprocessing algorithm.

We want to save some space for storing a set of constant strings by creating a constant (super)string containing all strings in the set as substrings. A string in the set is defined then by its index into the superstring and its length.

Outline of the Algorithm

Computing the shortest superstring is at least NP-hard and thus computationally intractable. However, the greedy algorithm is known to produce a superstring at most twice as long as optimal, and usually a lot better than that. It is considered a good practical technique.

We will implement the following flavor of it:

1. (Preliminary step) Eliminate strings from the set such that they are already substrings of some string(s) still in the set.
2. (Build the overlap info). For each (ordered) pair of strings in the set with non-empty overlap, record the length of overlap.
3. Starting with the longest overlap, repeatedly merge the overlapping pairs into one string.

Notice that steps 1-2 make the complexity of the algorithm quadratic on the number of strings. Generally, the number of overlaps recorded in step 2 is also quadratic on the number of strings, so step 3 has a quadratic number of merges. Strictly speaking, for each merge in step 3, we need to repeat steps 1-2 for the pairs that include the new (merged) string. This is linear on the (current) number of strings, which makes the overall merge process of step 3 of cubic complexity. For a set of strings of any practical size, this is probably too much for any interpreted language, including any preprocessor and thus including Unimal.

To make step 3 of quadratic complexity on the number of strings, we need to make an individual merge operation of constant complexity. To do so, we make a shortcut: The new string inherits overlap info from the left string and the right string that where merged. More precisely, if we merged pair <i,j>, then

- The new string gets an index i so that all overlap info for pairs <k,i> remain valid, and
- All overlap info for pairs <i,k> is replaced with overlap info for pairs <j,k>.

To accomplish the second requirement, we will maintain MergeBase[i] for each string index i. Initially, MergeBase[i] = i; if we merge the pair <i,j> we assign MergeBase[i]=j.

There are (rare) cases where this simplification of the original greedy algorithm misses a newly developed merge opportunity thus yielding a longer superstring than could be achieved. In practice, however, the losses, if any, are small.

It is very cumbersome to keep track of where in the superstring a given string ends up being. It is simpler to just find the string's location once the superstring has been built.

Step 1: Eliminating the substrings

Like in the previous section where we were building a tree, the strings are defined using the yet-to-be-defined macro AddString, e.g.

```
#MP AddString("stuff")
#MP AddString("foo")
#MP AddString("bar")
#MP AddString("foobar")
#MP AddString("baroque")
#MP AddString("queue")
#MP AddString("garb")
#MP AddString("stuff")
```

Our goal is to eliminate "foo" and "bar" as substrings of "foobar" and to eliminate one copy of "stuff".

A new variant of AddString is simply collecting the data; it produces no output (see super.inc):

```
1. #MP Macro AddString ;(string)
2. #MP Setstr STR_%dLindex = #1#
3. #MP Setstr STRx_%dLindex = #1#
4. #MP len = Ustrlen(#1#)
5. #MP If maxlen < len
6. #MP     maxlen = len
7. #MP Endif
8. #MP Total = Total + len
9. #MP Lindex = Lindex+1
10.   #MP Endm
```

Lines 2-3 make two copies of the supplied string indexed in the order of appearance: one for manipulations related to building a superstring, the other for finding the index of the string in the superstring.
Lines 4-6 keep track of the longest encountered string length; we will use it to estimate the longest possible length of string overlaps.

Line 8 counts the total length of all strings (for statistics only).
Line 9 keeps track of the number of strings.

While eliminating substrings, we'll also re-number them by moving the highest-numbered string to the number of the removed strings. This will generally make the number of remaining strings smaller and thus the algorithm to run faster.

The following lengthy macro does the job:

```
1.  #MP Macro RemoveFluff ;()
2.  #MP MaxIndex = Lindex - 1
3.  #MP For i=0, MaxIndex
4.  #MP    For j=0, MaxIndex
5.  #MP      If i!=j
6.  #MP        Setstr dummy = {uSplit, STR_%uj, STR_%ui}
7.  #MP        If uSplit>=0
8.  #MP          If i != MaxIndex ;not last string: compact the array
9.  #MP            Setstr STR_%ui = STR_%uMaxIndex
10. #MP            i = i - 1 ;force re-examining the same string index
11. #MP          Endif
12. #MP          Undef STR_%uMaxIndex
13. #MP          MaxIndex = MaxIndex - 1
14. #MP          j = Lindex ;break the inner loop
15. #MP        Endif
16. #MP      Endif
17. #MP    Endfor
18. #MP    If i>=MaxIndex
19. #MP      i = Lindex ;(manually) break the (reduced) loop
20. #MP    Endif
21. #MP Endfor
22. #MP Endm
```

Line 2: `MaxIndex` is the last index of a string; it will grow down when a string is removed.
Lines 3-21 examine whether string `i` must be deleted (and do the deletion). Since `MaxIndex` can grow down, the normal loop termination is not sufficient (recall that the `For` operator doesn't re-evaluate the loop limit); so lines 18-20 provide manual end-of-loop condition check. Note that in line 18 `i` indeed can be greater than `MaxIndex`, and that is if when at line 8, i was equal to MaxIndex.

(This was a subtle and hard-to-detect bug in the previous version; a better implementation altogether would be to remove lines 8 and 11. The worst that would happen is a string copied to itself in line 9. No harm.)

Lines 4-17 do the work for the string i by examining each string j. Of course, a string is not checked against itself (line 5).
Lines 6-7 check if string i is a substring of j (recall that `uSplit` gets a numeric value which is negative if and only if the split didn't occur, i.e. string j doesn't contain string i.

If i *is* contained in j then lines 8-11 replace string i with the last string (provided that i itself was not the last). Note that since string i changed, it must be evaluated again in the next pass of the outer loop; line 10 ensures that.

Line 12 removes the last string: it was either removed (if `i=MaxIndex`) or copied to `i` and became redundant. Correspondingly, line 13 decrements the max string index.
Once a containing string `j` has been found, there is no point in continuing search for it; so line 14 terminates the inner loop.

To test this macro, we can output the remaining strings (see `remove.u`):

```
#MP RemoveFluff
#MP For i=0,MaxIndex
[#mp%ui] #mp%sSTR_%ui
#MP Endfor
```

The result is

```
[0] stuff
[1] garb
[2] queue
[3] foobar
[4] baroque
```

As you can see, the result is correct but the strings got rearranged.

Step 2: Computing the strings overlap information

First, let's address an algorithmic issue: How do we compute the overlap of strings x and y, i.e. the (longest) suffix of x which is also a prefix of y?
We could try all suffixes of x to fit a prefix of y, starting from the longest suffix. But that's quite slow.

A faster way is to find the first occurrence in x of the first letter of y. If such an occurrence exists, the suffix of x starting at this occurrence is a candidate to try as a prefix of y. If it fits, we are done; otherwise, replace x with its suffix just beyond the failed occurrence and repeat. If no occurrence found, there is no overlap.

Returning to notation we developed so far, let `FirstLetter` be the first-letter substring of `STR_%ui`:

```
#MP   Setstr FirstLetter = {uSubstr, STR_%ui, 0, 1}
```

Here is a skeleton of Unimal code finding the longest overlap of `str1` and `str2=STR_%ui`:

```
      #MP Macro ComputeOverlap
1.    #MP Repeat
2.    #MP  Setstr dummy = {uSplit, str1, FirstLetter}
3.    #MP  If uSplit>=0 ;match found
4.    #MP   Setstr str1 = {uSubstr, str1, str1-1, Total} ;match to test
5.    #MP   matchlen = Ustrlen(str1)
6.    #MP   Setstr dummy = {uSplit, str2, str1}
7.    #MP   If uSplit>=0 && matchlen == str2
8.    #MP      ProcessOverlap[]
9.    #MP   Else
10.   #MP      Setstr str1 = {uSubstr, str1, 1, Total}
11.   #MP      uSplit = -1 ; continue
12.   #MP   Endif
13.   #MP      uSplit = 1 ; break
14.   #MP  Endif
```

```
15.   #MP While uSplit>=0
      #MP Endm
```

Here, lines 1, 12 provide a loop over tries of different suffixes of str1.
Lines 2-3 check if `FirstLetter` (of `str2`) is found in `str1`; if so, the suffix is extracted and tried.
At line 4, str1 has a numeric value of the length of the character of `str1` just after the first occurrence of `FirstLetter`, so line 4 extracts the suffix starting with `FirstLetter`.
Line 5 computes `matchlen` as the length of the extracted suffix (now `str1`).
Line 6 splits str2 by the suffix (`str1`). `str2` gets the numeric value of the index in the string `str2` just after the first occurrence of `str1`, provided `uSplit` got a non-negative value.

We have an overlap found if `uSPlit` succeeds *and* `matchlen` equals to the index after split (meaning that the match occurred at the beginning of `str2`, i.e. that `str1` is a prefix of `str2`).
Line 7 checks if we have an overlap and if so line 8 does some processing of it. Otherwise, line 10 removes the first (matching) character from `str1` making it ready for the next try.

Lines 11 and 13 fix the numeric value of `uSplit` to continue the loop if the overlap was *not* found (verify that it's correct!).

Note that the original str1 is destroyed during this process

To test this framework, we'll execute it in a loop over all pairs of strings which result from Step 1 (see `overlap.u`):

```
#MP For i=0, MaxIndex
#MP   Setstr str2 = STR_%ui
#MP   Setstr FirstLetter = {uSubstr, STR_%ui, 0, 1}
#MP   For j=0, MaxIndex
#MP     If i!=j
#MP         Setstr str1 = STR_%uj
#MP         ComputeOverlap[]
#MP     Endif
#MP   Endfor
#MP Endfor
```

For the required macro `ProcessOverlap` we'll just print the left string, the length of the overlap in parentheses and the right string:

```
#MP Macro ProcessOverlap
"#mp%sSTR_%uj" (#mp%umatchlen) "#mp%sSTR_%ui"
#MP Endm
```

Here is the test run of `overlap.u`:

```
"baroque" (3) "queue"
"stuff" (1) "foobar"
"garb" (1) "baroque"
"foobar" (3) "baroque"
```

Our next step is to define the macro `ProcessOverlap` to store the overlap info so as to facilitate the greedy merge of the overlapped strings. It should be done together with the way we accomplish the greedy merge, so we go on to

Step 3: Merging the strings into a superstring

First, we need to define some parameters (variables) that will assist us in the merge process.

Recall that according to the algorithm outline we want to merge `STR_%ui` and `STR_%uj` in a new `STR_%ui` which has to inherit right-side merge information from `STR_%uj`. For that, we'll maintain `MergeBase%ui` for all `i` as a right-side "merge as" index.
Initially, `MergeBase%ui = i` (merge information comes from the overlaps computations of the previous step); when we merge `STR_%ui` and `STR_%uj`, we assign `MergeBase%ui = MergeBase%uj`.
To disqualify string `j` from being on the right side of a merge, we simply undefine `STR_%uj`.

Now, greedy merge requires that we merge strings with longest overlap first, so we need to maintain the merge information somehow sorted by the overlap length.
There are many ways of doing it; we'll choose the idea of "limited-height heap" best suited for relatively short overlaps.
We will maintain, for each string `i` and each overlap length `m`,

- `count_%08Xm%08Xi` – the number of overlaps of length `m` with the string `i` on the left. (We will not define these variables unless they are non-zero.)
- Assuming `count_%08Xm%08Xi` is defined (and thus non-zero), for each `co` between 0 and `count_%08Xm%08Xi-1`, we'll define `match%08Xm%08Xi%uco` as the `co`-th string found to overlap with string `i` as the right-hand side, and with the overlap length `m`.

The following macro does greedy merging of all strings with overlap length `matchlen`:

```
1.    #MP Macro MergeStrings
2.    #MP For i = 0, MaxIndex
3.    #MP   Ifdef STR_%ui
4.    #MP     bm = MergeBase%ui
5.    #MP     If Defined(count_%08Xmatchlen%08Xbm)
6.    #MP       For co = 0, count_%08Xmatchlen%08Xbm-1
7.    #MP         j = match%08Xmatchlen%08Xbm%uco
8.    #MP         If j != i
9.    #MP           Ifdef STR_%uj ;then it's our best match: merge to i
          i.  #MP     Setstr tail {uSubstr, STR_%uj, matchlen, Total}
         ii.  #MP     Setstr STR_%ui = STR_%ui + tail ;new string
        iii.  #MP     MergeBase%ui = MergeBase%uj
         iv.  #MP     Undef STR_%uj
          v.  #MP     co = 0x7fffffff ;break the inner loop
         vi.  #MP     i = i-1 ;repeat with merged string
10.   #MP           Endif
11.   #MP         Endif
12.   #MP       Endfor
13.   #MP     Endif
14.   #MP   Endif
15.   #MP Endfor
```

```
16.   #MP Endm
```

Lines 2,3,15 arrange scanning of all strings that are still defined.
Line 4 gets the right context of the potential merge as described above.

Line 5 checks if there are any overlaps recorded and if so lines 6-9 find the first index j such that `STR_%uj` is available for merging.

If a suitable string `j` is found, line (i) extracts its `tail` beyond the overlap, and line (ii) appends it to the string `i`, which completes the merge. Line (iii) transfers the right-side merge context information from string `j` to string `i`, and line (iv) undefines string `j` thus removing it from further consideration.
If we did the merge, we have nothing to search for in the old string `i` right-side merge context, so line (v) forces that loop to terminate. String `i` has changed, so it must be evaluated again. That is ensured by line (vi) since line 15 will increment `i`.

The following simple macro wraps to make a greedy merge form longest to shortest overlap:

```
#MP Macro GreedyMerge
#MP For xx = 0, maxmatchlen-1
#MP     matchlen = maxmatchlen - xx ;make it a descending order
#MP     MergeStrings[]
#MP Endfor
#MP Endm
```

Now that we know how to do the greedy merge, we can go back to our debt to the previous section and create the macro `ProcessOverlap` responsible for recording the overlap information.
Recall that it is invoked whenever an overlap of length `matchlen` is found between string `j` on the left and string `i` on the right.

```
1.    #MP Macro ProcessOverlap
2.    #MP        count = 0
3.    #MP        Ifdef count_%08Xmatchlen%08Xj
4.    #MP          count = count_%08Xmatchlen%08Xj
5.    #MP        Endif
6.    #MP        match%08Xmatchlen%08Xj%ucount = i;
7.    #MP        count_%08Xmatchlen%08Xj = count + 1
8.    #MP        If matchlen > maxmatchlen
9.    #MP            maxmatchlen = matchlen
10.   #MP        Endif
11.   #MP Endm
```

Lines 2-5 set count to the number of previously recorded overlaps of length `matchlen` with the string j on the left. As usual, count is also the next available index.
Line 6 records the new overlap as being with string `i`.
Line 7 updates the number of recorded overlaps.
Lines 8-10 keep track of the maximum length of a recorded overlap; it is used in `GreedyMerge` above to iterate over the range of overlap lengths.

To put it all together, we need to initialize `maxmatchlen` and all `MergeBase%ui`; and complete the wrapper around `ComputeOverlap` to cover all strings. We'll do so by stealing

the example code from `overlap.u` (see also the Step 2 section) and put the missing initialization there:

```
1.    #MP Macro ComputeAllOverlaps
2.    #MP maxmatchlen = 0
3.    #MP For i=0, MaxIndex
4.    #MP   MergeBase%ui = i ;no redirection yet
5.    #MP   Setstr str2 = STR_%ui
6.    #MP   Setstr FirstLetter = {uSubstr, STR_%ui, 0, 1}
7.    #MP   For j=0, MaxIndex
8.    #MP     If i!=j
9.    #MP        Setstr str1 = STR_%uj
10.   #MP        ComputeOverlap[]
11.   #MP     Endif
12.   #MP   Endfor
13.   #MP Endfor
14.   #MP Endm
```

In this straightforward macro, lines 2 and 4 provide the missing initialization; the rest of the code is copied.

To see the results of what we've got, we add the following code to string definitions (see `merge.u`)

```
#MP RemoveFluff
#MP ComputeAllOverlaps
#MP GreedyMerge
#MP For i=0,MaxIndex-1
#MP     Ifdef STR_%ui
STR_#mp%ui is "#mp%sSTR_%ui"
#MP     Else
STR_#mp%ui is no longer defined
#MP     Endif
#MP Endfor
```

This will print the values of the strings remaining after the merge, or stubs for undefined strings:

```
STR_0 is "stuffoobaroqueue"
STR_1 is "garb"
STR_2 is no longer defined
STR_3 is no longer defined
```

To complete the superstring construction, we need to concatenate together all strings remaining after `GreedyMerge`:

```
1.    #MP Macro Concatenate
2.    #MP Setstr SuperString = ""
3.    #MP For i=0, MaxIndex
4.    #MP     Ifdef STR_%ui
5.    #MP         Setstr SuperString = SuperString + STR_%ui
6.    #MP     Endif
7.    #MP Endfor
```

```
8.    #MP SuperString = Ustrlen(SuperString)
9.    #MP Endm
```

Note that we could (and probably should) undefine `STR_%ui` after line 5 if we wanted to be memory-conscious. Also, line 8 prepares the length of the superstring for future convenience.

A test of it is in `superstr.u` where we added

```
#MP RemoveFluff
#MP ComputeAllOverlaps
#MP GreedyMerge
#MP Concatenate
Superstring of length #mp%uSuperString:
"#mp%sSuperString"
```

The output is

```
Superstring of length 20:
"stuffoobaroqueuegarb"
```

Step 4: Generating the C tables

The remaining part is quite straightforward; we'll borrow the technique from the earlier examples (`tree.u` and `encode3.u`)

Recall that the macro `AddString` collected two copies of a supplied string: `STR_%ui` which are now gone, and `STRx_%ui` which are still available:

```
1.    #MP Macro CLength
2.    const unsigned char length[#mp%uLindex] =
3.    {
4.    #MP For i=0,Lindex-1
5.    #MP   len = Ustrlen(STRx_%ui)
6.    [#mp%ui] = #mp{%ulen}U, //length of "#mp%sSTRx_%ui"
7.    #MP  Endfor
8.    };
9.    #MP Endm
```

This macro generates the array of string lengths. Line 6 provides C99-style designated initialization for better readability, but it's not necessary.

```
1.    #MP Macro CIndex
2.    const unsigned short index[#mp%uLindex] =
3.    {
4.    #MP For i=0,Lindex-1
5.    #MP    Setstr dummy = {uSplit, SuperString, STRx_%ui}
6.    #MP    ix = SuperString - Ustrlen(STRx_%ui)
7.        [#mp%ui] = #mp{%uix}U, //index of "#mp%sSTRx_%ui"
8.    #MP  Endfor
9.    };
10.   #MP Endm
```

This macro generates the array of indices of the original strings into `SuperString`. After line 5 the numeric value of `SuperString` is just after the first occurrence of string `i` in the *string* `SuperString`. Subtracting the length of string `i` in line 6 yields an index to the beginning of string i in `SuperString`.

Finally, we need to encode and pretty-print `SuperString`:

```
#MP Macro CSuper
const unsigned char SuperString[] =
{
#MP EncodeString(SuperString)
#MP PrettyPrintStringAsHex(result, 10)
};
#MP Endm
```

The macros invoked in `CSuper` do all the work; they have been covered earlier in this Application Note.

Now we can wrap all the processing in a macro so that gory details do not disturb an application programmer:

```
#MP Macro EndNames
#MP RemoveFluff
#MP ComputeAllOverlaps
#MP GreedyMerge
#MP Concatenate
#MP CLength
#MP CIndex
#MP CSuper
#MP Endm
```

We can test the whole thing now (see `super.u` which is also shown below in all its glory)

```
#MP Include "st.inc"
#MP Include "super.inc"
#MP BeginNames
#MP AddString("stuff")
#MP AddString("foo")
#MP AddString("bar")
#MP AddString("foobar")
#MP AddString("baroque")
#MP AddString("queue")
#MP AddString("garb")
#MP AddString("stuff")
#MP EndNames
```

The result of the run:

```
const unsigned char length[8] =
{
  [0] = 5U, //length of "stuff"
  [1] = 3U, //length of "foo"
  [2] = 3U, //length of "bar"
```

```
  [3] = 6U, //length of "foobar"
  [4] = 7U, //length of "baroque"
  [5] = 5U, //length of "queue"
  [6] = 4U, //length of "garb"
  [7] = 5U, //length of "stuff"
};
const unsigned short index[8] =
{
  [0] = 0U, //index of "stuff"
  [1] = 4U, //index of "foo"
  [2] = 7U, //index of "bar"
  [3] = 4U, //index of "foobar"
  [4] = 7U, //index of "baroque"
  [5] = 11U, //index of "queue"
  [6] = 16U, //index of "garb"
  [7] = 0U, //index of "stuff"
};
const unsigned char SuperString[] =
{
    /*[000]*/ 0x73,0x74,0x75,0x66,0x66,0x6f,0x6f,0x62,0x61,0x72,
    /*[010]*/ 0x6f,0x71,0x75,0x65,0x75,0x65,0x67,0x61,0x72,0x62,
    /*[020]*/
};
```

Note. If we have many strings, the processing would take some time. In part, it's the nature of a quadratic-complexity algorithm, in part, it's the memory consumption: requesting more memory slows the system down. To save some memory, we can change `AddString` so that it doesn't collect additional copies of strings `STRx_%i`.

To gain access to the user-supplied strings in `EndNames`, we'll scan the `AddString` definitions twice (see `superb.inc` and `superb.u`). In the end of the first scan we'll make the superstring calculations, and in the end of the second scan, use the strings acquired again to generate the C tables.

Here are the modified macros:

```
#MP Macro BeginNames
#MP maxlen = 0
#MP For pass =0,1
#MP Lindex = 0
#MP Total = 0
#MP Endm

#MP Macro EndNames
#MP If pass == 0
#MP     RemoveFluff
#MP     ComputeAllOverlaps
#MP     GreedyMerge
#MP     Concatenate
#MP Else
#MP     CLength
#MP     CIndex
#MP     CSuper
#MP Endif
```

```
#MP Endfor
#MP Endm
```

`AddString` is different only in that it no longer collects `STRx_` copies, and `CLength` and `CIndex` macros use name bases `STR_` instead of `STRx_`.
The output of `superb.u` matches that of `super.u`.