Unimal 2.0

Application note 4

Automating sparse and lookup tables

Documentation revision 2.00

Techniques:
Examples of developing rather complex algorithms and implementing them at compile time:
- storage and maintenance optimization for sparse tables;
- zero-maintenance lookup tables for accessing constant data

MacroExpressions
http://www.macroexpressions.com

Table of contents

Foreword

This application is one of the more complex and involved. However, it is of high practical importance because it results in zero maintainability efforts for lookup tables and associated access functions. In fact, the need to automate lookup tables was one of the main reasons why Unimal was born.

Examples of constant objects that require a lookup include such disparate cases as

- Translation between character encodings (e.g., between Korean KSC and Unicode)
- Incoming message qualification based on the message header

Whenever a table of objects of some kind is constant, so, conceptually, is the corresponding lookup table (or a "perfect hash" table, if you will). It makes a lot of sense to generate such stuff at compile time so that the code consumes less memory and starts faster. It is also important to eliminate project maintenance efforts by updating the support table automatically when the table of objects does change.

In this application note, we will consider several problems of increasing complexity; our objective will always be reducing the project maintenance efforts. In the examples we use, the target programming language will be C; not that it is essential, but it makes example fixed syntax of the output we will want to generate.

Sparse tables

A problem and the first solution

Consider the task of static initialization of a sparse table, i.e., a one with many zeros or "don't care" elements. In our example, the table will have 32 elements with significant entries at positions 9, 11, 24 and 27 with values 1, 2, 3, 4 respectively. All other elements are zeros (or "don't care"). This is what it might look like in C:

```
const unsigned SparseTable[32] = {
  0,0,0,0,0,0,0,0,0,1,0,2,0,0,0,0,0,
  0,0,0,0,0,0,0,3,0,0,4,0,0,0,0
};
```

You may notice that such a table is extremely difficult and error-prone to maintain, no matter how much of nice formatting and commenting you add. Maintainability would improve greatly if we could enter only significant elements in an easily maintainable way. This problem will occupy us for quite some time, with different levels of complexity. For now, we suggest the following Unimal solution (see Samples\AppNotes\4\sparse1.u):

```
#MP Set Entry9 = 1   ;value at offset 9
#MP Set Entry11 = 2  ;value at offset 11
#MP Set Entry24 = 3  ;value at offset 24
#MP Set Entry27 = 4  ;value at offset 27

const unsigned SparseTable[32] = {
#MP For Index = 0, 31
     #MP Ifdef Entry%uIndex
      #mp%dEntry%uIndex, /* Offset=#mp%uIndex */
     #MP Else
      0, /* dummy */
     #MP Endif
#MP Endfor
};
```

Here is how it works:

First, we define an `Entry9`…`Entry27` variables, one for each significant entry's offset. We assign them corresponding values the table's entries should take (in our example, 1,2,3,4 respectively).

Then, the `For`/`Endfor` loop creates the table's 32 entries. If the `Index` (loop variable) is such that `Entry%uIndex` is defined, then its value is assigned in the beginning, and we just render it to the output file. If, however, `Entry%uIndex` is not defined, it means (by our construction) that there is no significant entry at this offset in the table, and we render a zero, for the sake of certainty. We also add target-language comments indicating whether an entry is dummy or significant.

The output is shown below.

```
const unsigned SparseTable[32] = {
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          1, /* Offset=9 */
          0, /* dummy */
          2, /* Offset=11 */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          3, /* Offset=24 */
          0, /* dummy */
          0, /* dummy */
          4, /* Offset=27 */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
          0, /* dummy */
};
```

A nicer solution

Now, we want our solution to have the standardized look of a table definition, as follows:

```
#MP Expand SparseTableBegin(SparseTable, 32) ;(name, size)
   #MP Expand DefineEntry(9, 1)     ;value 1 at offset 9
   #MP Expand DefineEntry(11, 2)    ;value 2 at offset 11
   #MP Expand DefineEntry(27, 4)    ;value 4 at offset 27
   #MP Expand DefineEntry(24, 3)    ;value 3 at offset 24
```

```
#MP Expand SparseTableEnd()
```

Note that, as a favor to the application programmer and to the maintainer, we want to allow the entries to be defined in any order.

Here are the macros that achieve our objective (see Samples\AppNotes\4\sparse2.u):

```
#MP Macro SparseTableBegin ;(name, size)
const unsigned #mp%n#1#[#mp%u#2#] = {
#MP     tsize = #2# ;save the size
#MP Endm
#MP Macro DefineEntry ;(position, value)
     #MP Set Entry%u#1# = #2#
#MP Endm
#MP Macro SparseTableEnd ;()
   #MP For Index = 0, tsize-1 ;use the saved size
     #MP Ifdef Entry%uIndex
       #mp%uEntry%uIndex, /* Offset=#mp%uIndex */
     #MP Else
       0, /* dummy */
     #MP Endif
   #MP Endfor
};
#MP Endm
```

`SparseTableBegin` simply renders the C array definition line and saves the array size. `DefineEntry` merely defines the entry for Unimal: the entry indexed by "position" is defined only for the positions we want; in this case the value of the entry is what we supplied in a macro invocation.
The bulk of the work is done in `SparseTableEnd`. Note that when we arrive there, the definitions of `Entry%uIndex` do not depend on the order of invocations of `DefineEntry`. `SparseTableEnd` renders in a loop the values that were previously defined or "don't care" entries. Finally, it closes the C initialization statement.

The output of sparse2.u, not surprisingly, is the same as the output of sparse1.u.

Analysis and an improvement: Access function and auto-sizing

There are two issues with our solution.

One is that we do not have a protection against an out-of-range index. This is easy to fix with an access function, like

```
unsigned myaccess(unsigned index)
{
     if(index>=32) return 0; /* don't care value */
     return SparseTable[index];
}
```

The other issue is an out-of-range definition, like the one in Samples\AppNotes\4\sparse3.u:

```
#MP Expand SparseTableBegin(SparseTable, 32) ;(name, size)
   #MP Expand DefineEntry(39, 1) ;value 1 at an out-of-range 39
   #MP Expand DefineEntry(11, 2)    ;value 2 at offset 11
   #MP Expand DefineEntry(27, 4)    ;value 4 at offset 27
   #MP Expand DefineEntry(24, 3)    ;value 3 at offset 24
#MP Expand SparseTableEnd()
```

We can see by running

Unimal sparse3.u

or simply by inspecting the `SparseTableEnd` macro that an out-of-range definition is silently thrown away, without any indication of an error.

Of course, we could trap this error in the `DefineEntry` macro, but it is not the point. The real issue here is that in order to reduce maintenance, we want the table size to adjust automatically according to actual entries definitions.

To do so, we need to keep track of the maximum offset of an entry. We can do this with the help of a simple macro computing the running maximum (Samples\AppNotes\4\sparse4.u):

```
#MP Macro Maximum ;(new_entry, running_max)
     #MP Ifdef #2#
          #MP If #1#>#2#
               #MP #2# = #1# ;Set to current maximum
          #MP Endif
     #MP Else
          #MP ;Maximum of one element equals to that element
          #MP #2# = #1#
     #MP Endif
#MP Endm
```

We need to redefine the table generating macros.

First, we no longer supply the table size to `SparseTableBegin`; since the C syntax allows omitting the size of an initialized array, it becomes very simple:

```
#MP Macro SparseTableBegin ;(name)
const unsigned #mp%n#1#[] = {
#MP Endm
```

`DefineEntry` will now keep track of the max index encountered thus far; by the time we reach `SparseTableEnd`, `max_index` will contain the actual max index of the array.

```
#MP Macro DefineEntry ;(position, value)
     #MP Set Entry%u#1# = #2#
    #MP Expand Maximum(#1#, max_index)
#MP Endm
```

Since the table size is one greater than `max_index`, `SparseTableEnd` is modified accordingly:

```
#MP Macro SparseTableEnd ;()
   #MP For Index = 0, max_index
     #MP Ifdef Entry%uIndex
       #mp%uEntry%uIndex, /* Offset=#mp%uIndex */
     #MP Else
       0, /* dummy */
     #MP Endif
   #MP Endfor
};
#MP Endm
```

(The only modification is the upper limit of the loop in the `For` statement.)

Finally, we need to modify our access function to take into account the actual max index to the table:

```
unsigned myaccess(unsigned index)
{
     if(index>#mp%umax_index) return 0; /* don't care value */
     return SparseTable[index];
}
```

We can now run

```
Unimal sparse4.u
```

and see that everything is sized correctly:

```
const unsigned SparseTable[] = {
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       2, /* Offset=11 */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
```

```
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        3, /* Offset=24 */
        0, /* dummy */
        0, /* dummy */
        4, /* Offset=27 */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        1, /* Offset=39 */
};

unsigned myaccess(unsigned index)
{
        if(index>39) return 0; /* don't care value */
        return SparseTable[index];
}
```

Size optimization: Chopping off the leading dummy entries

Let's return to our original table entries (Samples\AppNotes\4\sample5.u):

```
#MP Expand SparseTableBegin(SparseTable) ;(name)
   #MP Expand DefineEntry(9, 1)     ;value 1 at offset 9
   #MP Expand DefineEntry(11, 2)    ;value 2 at offset 11
   #MP Expand DefineEntry(27, 4)    ;value 4 at offset 27
   #MP Expand DefineEntry(24, 3)    ;value 3 at offset 24
#MP Expand SparseTableEnd()
```

If we run

```
Unimal sparse5.u
```

we can observe that the table lost its trailing zeros:

```
const unsigned SparseTable[] = {
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
```

```
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        1, /* Offset=9 */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        0, /* dummy */
        3, /* Offset=24 */
        0, /* dummy */
        0, /* dummy */
        4, /* Offset=27 */
};

unsigned myaccess(unsigned index)
{
        if(index>27) return 0; /* don't care value */
        return SparseTable[index];
}
```

This is good news: we save on the table size. (Of course, we could anticipate this given the way we auto-size the array.)

But this also prompts us to think whether we can drop the nine leading "don't care" entries. The answer is yes, but the index correction must be made in the access function.

To do so, we need to keep track of the minimum offset of a defined entry. We can do this with the help of a macro computing the running minimum (Samples\AppNotes\4\sparse6.u), which (naturally) is the same as `Maximum`, except for the comparison sign:

```
#MP Macro Minimum ;(new_entry, running_min)
     #MP Ifdef #2#
          #MP If #1#<#2#
               #MP #2# = #1# ;Set to current minimum
          #MP Endif
     #MP Else
          #MP ;Minimum of one element equals to that element
          #MP #2# = #1#
     #MP Endif
#MP Endm
```

We will now modify to also keep track of the min defined index:

```
#MP Macro DefineEntry ;(position, value)
    #MP Set Entry%u#1# = #2#
    #MP Expand Maximum(#1#, max_index)
    #MP Expand Minimum(#1#, min_index)
#MP Endm
```

By the time we reach `SparseTableEnd`, `min_index` will contain the actual min index of a defined entry of the array.

`SparseTableEnd` needs a slight modification: the `For` loop must now begin with `min_index`, so as not to output the leading dummy entries:

```
#MP Macro SparseTableEnd ;()
   #MP For Index = min_index, max_index
     #MP Ifdef Entry%uIndex
       #mp%uEntry%uIndex, /* Offset=#mp%uIndex */
     #MP Else
       0, /* dummy */
     #MP Endif
   #MP Endfor
};
#MP Endm
```

Finally, we need to modify the access function. The test for out-of-range index is now two-sided; also the actual index to the array must be adjusted by the number of omitted dummy entries:

```
unsigned myaccess(unsigned index)
{
    if(index>#mp%umax_index || index<#mp%umin_index) {
        return 0; /* don't care value */
    }
    return SparseTable[index - #mp%umin_index];
}
```

If we run

```
Unimal sparse6.u
```

we can observe that the table lost its trailing zeros:

```
const unsigned SparseTable[] = {
       1, /* Offset=9 */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       3, /* Offset=24 */
       0, /* dummy */
       0, /* dummy */
       4, /* Offset=27 */
};

unsigned myaccess(unsigned index)
{
    if(index>27 || index<9) {
        return 0; /* don't care value */
    }
    return SparseTable[index - 9];
}
```

Compressing a sparse table: a plan

Let's reflect on the result we got in the previous section. We have now a compressed representation of the array (we removed leading and trailing dummy entries). The price we paid for it is that now we absolutely need a related access function.

But if so, a natural question is: If we are willing to make our access function a tad more complex, can we further compress the table representation?

We will explore the following idea:

1. Split the sparse table into several smaller (partial) tables.
2. Merge those partial tables in a single common table, each partial table starting at its own offset. The offsets of partial tables are selected in such a way that no two significant

entries of any two partial tables have the same offset in the common table (i.e., do not collide).

As an illustration of what we are going to do, consider splitting the access index into two pieces: a quotient and a remainder for some split divisor (in our example, 4)

Index	3-bit digest (quotient, q)	2-bit offset (remainder, r)	Value (object number)
9	2	1	1
11	2	3	2
24	6	0	3
27	6	3	4

Now we can read this table as follows:
Given an index, compute the quotient and the remainder; for each quotient, there is a small sparse table yielding the value. All quotients can be arranged in a (primary) small lookup table to reference the (secondary) tables indexed by remainders of the original index.
We can concatenate all tables together so that the primary lookup table contains indices to the same array. Of course, while doing so, we can remove leading and trailing dummy entries from all tables.

In our example, the combined table should be like this:

```
const unsigned SparseTable[] = {
/* begin (primary) lookup table */
     /*0 omitted */
     /*0 omitted */
       4, /* base index for q=2 */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       8, /* base index for q=6 */
     /*0 omitted */
/* end of (primary) lookup table */
/* begin index table for quotient 2 */
     /*0 omitted */
       1, /* q=2 r=1 */
       0, /* dummy */
       2, /* q=2 r=3 */
/* end index table for quotient 2 */
/* begin index table for quotient 6 */
       3, /* q=6 r=0 */
       0, /* dummy */
       0, /* dummy */
       4, /* q=6 r=3 */
/* end index table for quotient 6 */
};
```

The table is supposed to work like this:

If for instance the index is 11 (quotient q=2, remainder r=3), we get an internal index by subtracting the number of omitted dummy entries of the primary table (2) from the quotient (2). The result is 0, so we read the index from `SparseTable[0]`. It is a base of the secondary lookup table with any omitted entries taken into account. `SparseTable[0]` is 4; we add the remainder (3) to it to get 7; `SparseTable[7]` is the result; it is 2 as it should be for the index 11.

Likewise, if the index is 10, we'll arrive at the result at `SparseTable[6]`; it is 0 ("don't care") as it should be.

However, we get in trouble with the index 8: following the same calculation, we find the result at `SparseTable[4]`; it is supposed to be "don't care" since the value at index 8 is not defined. Yet it is 8 and, the way we constructed the table, actually happens to be an entry in the primary (quotient) table.

This mishap could in fact be anticipated. Indeed, since we did remove leading and trailing "don't care" entries and did *not* add index range checking for the secondary (remainder) table, all the sub-tables in our combined table are overlapping, so a significant entry in one table can occupy a place where a "don't care" of another sub-table is expected.

So, the combined table above:

- Correctly finds entries that were originally defined
- May incorrectly find a phantom value for a "don't care" (undefined) entry.

A way to correct this problem is to have an "oracle" which would tell us whether the value we found is a phantom or an actually defined value.

There is an economical way to design such an oracle: just a table of pairs <index, value> such as:

```
const struct values_t {
    unsigned index;
    unsigned value;
} values[] = {
    {9, 1},
    {11, 2},
    {24, 3},
    {27, 4},
};
```

The secondary sub-tables of `SparseTable` above should contain, instead of the value, an index to the `values` array. Then, an index we pull out of `SparseTable` is good if it is less than the number of entries in the `values` array and the index field of the entry pointed to is in fact the index we started with.

Here is a redesigned table appropriately renamed `LookupTable` with changes shown in boldface:

```
const unsigned LookupTable[] = {
/* begin (primary) lookup table */
     /*0 omitted */
```

```
     /*0 omitted */
       4, /* index of the index table for quotient 2 */
       0, /* dummy */
       0, /* dummy */
       0, /* dummy */
       8, /* index of the index table for quotient 6 */
     /*0 omitted */
/* end of (primary) lookup table */
/* begin index table for quotient 2 */
     /*0 omitted */
       0, /* q=2 r=1 */
       0, /* dummy */
       1, /* q=2 r=3 */
/* end index table for quotient 2 */
/* begin index table for quotient 6 */
       2, /* q=6 r=0 */
       0, /* dummy */
       0, /* dummy */
       3, /* q=6 r=3 */
/* end index table for quotient 6 */
};
```

Now we are in a position to finalize a design of the access function:

```
unsigned myaccess(unsigned index)
{
    unsigned Object; //reference to the object in values table
    unsigned ObIndex; //index to the object reference
    unsigned base; //base index of the secondary sub-table
    unsigned quotient = index/4;
    unsigned remainder = index%4;

    quotient = quotient-2; //back by leading 0's in quotients
    if(quotient>=12) {
        return 0; /* index out of range */
    }
    base = LookupTable[quotient]; //start of sub-table
    ObIndex = base+remainder; //object reference index
    if(ObIndex>=12) {
        return 0; /* index out of range */
    }
    Object = LookupTable[base+remainder]; //object reference
    if(Object>=4) return 0; //out of range
    if(values[Object].index != index) {
        return 0; //check failed
    }
    return Object + 1; //1-based index
}
```

This function works as follows:
It computes the index to the primary (quotient) sub-table and rolls it back by the number of leading dummy entries (in this case, 2). If the result is greater than 11, the max index to `LookupTable`, the index is out of range and the function returns 0 for "don't care". Otherwise, it retrieves the base index of the sub-table for the given quotient and computes the (secondary) index `ObIndex` by adding the remainder. If `ObIndex` is out of range, the function returns 0 for "don't care". Otherwise, the function retrieves `Object`, the index to the `values` array. If the index comparison passes, the corresponding index is returned; otherwise, 0.

It's worth noting that by looking at the entries of `LookupTable`, we cannot tell anymore whether a 0 is an index to `values[0]` or a dummy entry. However, for a dummy entry the index match test in `myaccess` will fail.

This design deserves some discussion that follows.

A generalization: From sparse tables to lookup tables

A plan of compressing sparse tables which we designed in the previous section has some curious properties.

First, the `LookupTable` contains only indices (to itself or to `values`). Those indices can very well be small numbers and can be stored in a C data type smaller than unsigned. We note this as a potential space saver but will not explore it any further because this is almost trivial.

Second, an additional space can be saved if we subtract the minimum value of a valid index first. Indeed, if valid indices are in the range 10000000 to 10000009, splitting the index value will probably produce a single-entry quotient table and the remainder table for it would be as long as the original index table before the split. Normalizing the index range to zero is a good idea. We, however, will pass it up for simplicity; it is not difficult to add but it would contaminate the oh-so-great clarity of this presentation.

Third, we never used the fact that the `value` field of the elements of the `values` array is numeric. In fact, it can be any type whatsoever, so long as the `values_t` type is defined appropriately. In our Unimal implementation in the next section we will assume that the `value` field of has a type `ob_type` defined elsewhere; we'll use Ob1, Ob2, Ob3, Ob4 *names* instead of the *values* 1, 2, 3, 4.

Fourth, the design of the function `myaccess` takes into consideration that the quotient and remainder tables can overlap in the combined table; it makes no assumption on *how exactly* they overlap. This opens an opportunity to compress the table further; we'll explore it later.

Now, let's implement the design of the previous section in Unimal.

Implementing a compressed lookup table

We still want to get the standard look of the table; but since we changed the paradigm, more appropriate names are used below:

```
#MP Expand ObjectTableBegin(ObjectTable) ;(name)
   #MP Expand DefineEntry(9, Ob1)   ;Ob1 with index 9
   #MP Expand DefineEntry(11, Ob2)  ;Ob2 with index 11
   #MP Expand DefineEntry(27, Ob4)  ;Ob4 with index 27
   #MP Expand DefineEntry(24, Ob3)  ;Ob3 with index 24
#MP Expand ObjectTableEnd()
```

Note that we also want to allow out-of-order definitions of entries as shown above. The example file for this section is Samples\AppNotes\4\sparse7.u

Our implementation will have two passes over the object definitions: one for the (sparse) lookup table and another for the table of the objects (`values`). So, this time it will have a loop where the `For` is wrapped in `ObjectTableBegin` and `Endfor` in `ObjectTableEnd`.

The macro `ObjectTableBegin` starts the loop and renders pass-specific C definition statements. It also initializes count to count the number of entries:

```
#MP Macro ObjectTableBegin ;(name)
#MP For pass=0,1
#MP     count = 0
#MP     If pass == 0
const unsigned LookupTable[] = {
#MP     Endif
#MP     If pass == 1
const struct values_t {
    unsigned index;
    unsigned value;
} values[] = {
#MP     Endif
#MP Endm
```

The macro `DefineEntry` (see below) computes the quotient q and the remainder r and maintains (in pass 0) running minimum and maximum of the quotients (`min_q` and `max_q`) and, for each quotient q, running minimum and maximum of the corresponding remainders. These minimums and maximums will be used to trim "don't care" zeros from the combined table.

```
#MP Macro DefineEntry ;(index, ob_name)
#MP     q = #1#/divisor
#MP     r = #1#% divisor
#MP     If pass == 0
    #MP     Expand Maximum(q, max_q)
    #MP     Expand Minimum(q, min_q)
    #MP     Expand Maximum(r, max_r_%uq)
    #MP     Expand Minimum(r, min_r_%uq)
    #MP     Entry%u#1# = count
#MP     Endif
#MP     If pass == 1
    {#mp%u#1#, #mp%n#2#},
#MP     Endif
#MP     count = count+1
#MP Endm
```

Like a previous implementation (sparse6.u), Entry<index> is defined; this time its value is `count` which is the index to the oracle (`values`) table of objects made in the next pass. In pass 1, the macro simply renders the line of the `values` array as a pair {index, object}. Finally, the running number of defined entries (`count`) is incremented.

Of course, divisor must be defined before use; in our example it is
`#MPSet divisor = 4`

The macro `ObjectTableEnd` is conceptually very simple:

```
#MP Macro ObjectTableEnd ;()
#MP     If pass == 0
#MP         Expand FillLookupTable()
#MP     Endif ;pass
};
#MP Endfor
#MP Endm
```

In pass 0 it expands the macro `FillLookupTable` which renders the data of the combined lookup table. We discuss `FillLookupTable` in detail later. Then it closes the C definition statement with "`};`". In pass 0 this ends the lookup table, and in pass 1 the table of objects (`values`). Finally, it ends the loop body that began in `ObjectTableBegin`.

The access function is rendered strictly according to the design of the previous section:

```
unsigned myaccess(unsigned index)
{
    unsigned Object; //reference to the object in values table
    unsigned ObIndex; //index to the object reference
    unsigned base; //base index of the secondary sub-table
    unsigned quotient = index/#mp%udivisor;
    unsigned remainder = index%#mp%udivisor;

    //back by leading 0's in quotients
    quotient = quotient-#mp%umin_q;

    if(quotient>=#mp%uTableSize) {
        return 0; /* index out of range */
    }
    base = LookupTable[quotient]; //start of sub-table
    ObIndex = base+remainder; //object reference index
    if(ObIndex>=#mp%uTableSize) {
        return 0; /* index out of range */
    }
    Object = LookupTable[base+remainder]; //object reference
    if(Object>=#mp%ucount) return 0; //out of range
    if(values[Object].index != index) {
        return 0; //check failed
    }
    return Object + 1; //1-based index
}
```

The parameter `TableSize` used for out-of-bounds checks is produced by the macro `FillLookupTable` which is described next.

```
#MP Macro FillLookupTable ;() - works with predefined names
#MP     curr_base = max_q - min_q + 1                ;(1)
#MP     For q = min_q, max_q
```

```
#MP         Ifdef min_r_%uq                          ;(2)
#MP             base_%uq = curr_base - min_r_%uq     ;(3)
#MP             curr_base = base_%uq + max_r_%uq + 1
    #mp%ubase_%uq, /* base index for q=#mp%uq */     ;(4)
#MP         Else
    0, /* dummy */                                   ;(5)
#MP         Endif
#MP     Endfor
    /* end of (primary) lookup table */

#MP     For q = min_q, max_q                         ;(6)
#MP         Ifdef min_r_%uq                          ;(7)
    /* begin index table for quotient #mp%uq */
#MP             For r = min_r_%uq, max_r_%uq         ;(8)
#MP                 Index = q*divisor + r ;recover index
#MP                 Ifdef Entry%uIndex
    #mp%uEntry%uIndex, /* q=#mp%uq r=#mp%ur */
#MP                 Else
    0, /* dummy */
#MP                 Endif
#MP             Endfor                               ;(9)
    /* end index table for quotient #mp%uq */
#MP         Endif
#MP     Endfor
#MP     TableSize = curr_base                        ;(10)
#MP Endm
```

In this macro, `curr_base` is used to keep the index in the combined lookup table where the next trimmed secondary (remainder) sub-table begins; the first one begins right after the trimmed primary (quotient) sub-table; line (1).

Then we have a loop over the valid range of quotients; its job is to render the primary (quotient) sub-table. Line (2) checks if any valid remainder exists for the given quotient; in this, and only in this case the minimum of those remainders is defined (in pass 0 over all `DefineEntry` macros).

If, according to line (2), the remainders exist, the corresponding sub-table will be added in order. The index to this sub-table (base_<q>, line (3)) is, according to our design, to its untrimmed beginning, which is the number of trimmed leading zeros less than `curr_base`. The next line updates the `curr_base` by the trimmed size of the encountered secondary sub-table.

Line (3) deserves a little discussion: base_<q> may end up being negative, and we use unsigned indices. (The same observation, by the way, applies to conditioning `quotient` in the beginning of `myaccess` function.) Both cases take advantage of modulo arithmetic – in Unimal unsigned rendering in the first case and in C arithmetic in the second case. Mathematically negative numbers will show as huge positive numbers, and this is OK. When used in `myaccess` for index computation, they will produce a positive result or, for a trimmed-out index, perhaps another huge number which surely will be out of range.

Line (4) renders the base index to the remainders sub-table for the quotient q, along with a nice comment.

If, however, the test in line (2) failed, a dummy 0 is rendered in line (5).

Line (6) begins a loop that renders the remainders sub-tables. The test in line (7) is identical to that in line (2) and checks if the corresponding remainders sub-table exists at all. If it does, the nested loop between lines (8) and (9) renders the trimmed sub-table, sandwiched, again, between nice comment lines. In this nested loop, the original index is recovered from the quotient and the remainder; if the corresponding Entry<index> is defined (in one of `DefineEntry` macros) then its value is the index to the `values` table of objects, and it is rendered with a corresponding comment. Otherwise, a dummy 0 is rendered.

Finally, line (10) exposes the next available index in the table (`curr_base`) as `TableSize`.

Putting everything together (sparse7.u) produces the following result of Unimal processing:

```
const unsigned LookupTable[] = {
    4, /* base index for q=2 */
    0, /* dummy */
    0, /* dummy */
    0, /* dummy */
    8, /* base index for q=6 */
    /* end of (primary) lookup table */

    /* begin index table for quotient 2 */
    0, /* q=2 r=1 */
    0, /* dummy */
    1, /* q=2 r=3 */
    /* end index table for quotient 2 */
    /* begin index table for quotient 6 */
    3, /* q=6 r=0 */
    0, /* dummy */
    0, /* dummy */
    2, /* q=6 r=3 */
    /* end index table for quotient 6 */
};
const struct values_t {
    unsigned index;
    unsigned value;
} values[] = {
    {9, 0b1},
    {11, 0b2},
    {27, 0b4},
    {24, 0b3},
};

unsigned myaccess(unsigned index)
{
```

```
    unsigned Object; //reference to the object in values table
    unsigned ObIndex; //index to the object reference
    unsigned base; //base index of the secondary sub-table
    unsigned quotient = index/4;
    unsigned remainder = index%4;

    quotient = quotient-2; //back by leading 0's in quotients
    if(quotient>=12) {
        return 0; /* index out of range */
    }
    base = LookupTable[quotient]; //start of sub-table
    ObIndex = base+remainder; //object reference index
    if(ObIndex>=12) {
        return 0; /* index out of range */
    }
    Object = LookupTable[base+remainder]; //object reference
    if(Object>=4) return 0; //out of range
    if(values[Object].index != index) {
        return 0; //check failed
    }
    return Object + 1; //1-based index
}
```

Note that the tables are slightly different than in the design prototype since in the beginning of this section we intentionally changed the order in which the objects are defined.

A better compression of the lookup table

Recall that the implementation we are through with so much effort is about the plan stated before:

1. Split the table into several smaller (partial) tables.
2. Merge those partial tables in a single common table, each partial table starting at its own offset. The offsets of partial tables are selected in such a way that no two significant entries of any two partial tables have the same offset in the common table (i.e., do not collide).

What we did, in a way, is we trimmed each sub-table by chopping off leading and trailing "don't care" entries and concatenated all the sub-tables in a single (lookup) table. In our access function, we had to implement index validation because a significant entry of one sub-table could pose as a "don't care" entry of another sub-table.

This means that our method of producing the access function does not depend on *how exactly* the sub-tables overlap, and this opens an opportunity for a better compression of the sub-tables in a combined table. Specifically, we want to impose the following additional requirement:

3. Implement step 2 in such a way that the sub-tables are intertwined. This means that a significant entry of a partial table can be placed in the common merged table between two significant entries of another partial table.

This is to say that for our example data definition,

```
#MP Expand ObjectTableBegin(ObjectTable) ;(name)
   #MP Expand DefineEntry(9, Ob1)   ;Ob1 with index 9
   #MP Expand DefineEntry(11, Ob2)  ;Ob2 with index 11
   #MP Expand DefineEntry(27, Ob4)  ;Ob4 with index 27
   #MP Expand DefineEntry(24, Ob3)  ;Ob3 with index 24
#MP Expand ObjectTableEnd()
```

we want the combined table to look like this:

```
const unsigned LookupTable[] = {
    0, /* base index for q=2 */
    0, /* q=2 r=1 */
    3, /* q=6 r=0 */
    1, /* q=2 r=3 */
    2, /* base index for q=6 */
    2, /* q=6 r=3 */
};
```

Of course, the access function will have different index limit to check against:

```
unsigned myaccess(unsigned index)
{
    unsigned Object; //reference to the object in values table
    unsigned ObIndex; //index to the object reference
    unsigned base; //base index of the secondary sub-table
    unsigned quotient = index/4;
    unsigned remainder = index%4;

    quotient = quotient-2; //back by leading 0's in quotients
    if(quotient>=6) {
        return 0; /* index out of range */
    }
    base = LookupTable[quotient]; //start of sub-table
    ObIndex = base+remainder; //object reference index
    if(ObIndex>=6) {
        return 0; /* index out of range */
    }
    Object = LookupTable[base+remainder]; //object reference
    if(Object>=4) return 0; //out of range
    if(values[Object].index != index) {
        return 0; //check failed
    }
    return Object + 1; //1-based index
}
```

(The lines changed from the previous implementation are shown in bold.)

Our new lookup table happens to be twice as short as the previous one; it even happens not to have dummy entries (for this example). But we need to work a bit harder to generate it automatically. The implementation we are pursuing is found in

Samples\AppNotes\4\sparse8.u; its goal is to leave the previous implementation (sparse7.u) intact, except the macro `FillLookupTable`.

To achieve the new interleave of sub-tables, we will control `curr_base` differently: instead of leaping to the point just beyond the last merged sub-table, we'll increment it by 1 and see if *any* of the remaining sub-tables can be placed at this base. If such a sub-table exists, we'll merge it in the current combined table and remove it from the list of remaining sub-tables.

This plan requires iterative processing of sub-tables (until all sub-tables are merged in); it also requires more housekeeping:

- We need to keep track of the current size of the combined table (it may or may not grow when a new sub-table is merged in); we'll use a parameter `curr_end` for this purpose.
- When we place the primary (quotient) sub-table, we know where its significant entries are but we don't know yet what their values are (they are the base indices of the secondary sub-tables which are known only when a sub-table is merged in). To simplify the matters, we will not render the combined lookup table on the go; instead, we will maintain a set of parameters `lt_<x>` where x is an index in the lookup table. The (final) numeric value of `lt_<x>` will be an index (to the secondary sub-table or to the table of objects values) per the design; the string value will be a comment line we want to render. At any time, if `lt_<x>` is defined, it means that the place at index x in the combined table is occupied.

With this in mind, the macro `FillLookupTable` can *end* with the following macro `RenderLookupTable`:

```
#MP Macro RenderLookupTable ;()
#MP     For temp = 0, curr_end
#MP         Ifdef lt_%utemp
    #mp%ult_%utemp, /* #mp%slt_%utemp */
#MP             Undef lt_%utemp
#MP         Else
    0, /* dummy */
#MP         Endif
#MP     Endfor
#MP Endm
```

This macro fills all entries of the combined table in a loop: if an `lt_<n>` is defined, it is rendered as an index, along with the corresponding comment, and then undefined for the sake of cleaning up; otherwise, a dummy entry is rendered.

Everything preceding is needed only to define parameters `lt_<n>` and `curr_end` appropriately.

We begin with the macro `ReservePrimaryPlaces` which only marks the indices corresponding to the primary sub-table as significant entries; the values don't matter:

```
#MP Macro ReservePrimaryPlaces ;()
#MP     For q = min_q, max_q        ;(1)
```

```
#MP          Ifdef min_r_%uq          ;(2)
#MP              temp = q - min_q     ;(3)
#MP              lt_%utemp = 2006 ;(4) just define a placeholder
#MP          Endif
#MP      Endfor
#MP
#MP      curr_base = 1
#MP      curr_end = max_q-min_q
#MP Endm
```

Line (1) begins a loop over valid range of quotients; the test in line (2) checks if the corresponding secondary sub-table exists at all (we've already used this trick before). If the test passes, temp in line (3) is the index of its base. We don't know this base yet, so we use an arbitrary number for now in line (4).

Finally, we set `curr_base` to 1 (place 0 is guaranteed to be occupied) and `curr_end` to the length of the trimmed primary table.

Now, consider an attempt to place the sub-table corresponding to quotient q at `curr_base`. A remainder r corresponds to the original `Index = q*divisor + r`; if `Entry<Index>` is defined (in a `DefineEntry` macro), the r's position in the combined table would be

```
temp = curr_base + r - min_r_<q>
```

So, if `lt_<temp>` is already defined, we have a collision and the sub-table cannot be placed at `curr_base`. However, if we inspected all valid values of r and didn't detect a collision, the sub-table can be placed at `curr_base`.

This explains how the following macro works:

```
#MP Macro DetectCollision ;()
#MP      collision = 0 ;no collision found yet
#MP      For r = min_r_%uq, max_r_%uq
#MP          Index = q*divisor + r
#MP          Ifdef Entry%uIndex
#MP              temp = curr_base + r - min_r_%uq
#MP              Ifdef lt_%utemp ;collision?
#MP                  collision = 1 ;mark a collision
#MP                  r = max_r_%uq ;break the loop (for speed)
#MP              Endif
#MP          Endif
#MP      Endfor
#MP Endm
```

Now, when the time comes to merge a sub-table, the following macro does the job:

```
#MP Macro MergeTable ;()
#MP      For r = min_r_%uq, max_r_%uq
#MP          Index = q*divisor + r
```

```
#MP         Ifdef Entry%uIndex
#MP             temp = curr_base + r - min_r_%uq
#MP             lt_%utemp = Entry%uIndex
#MP             Setstr lt_%utemp={uJoin,"q=",{%uq},"  r=",{%ur}}
#MP             Undef Entry%uIndex
#MP         Endif
#MP     Endfor
#MP     temp = q - min_q
#MP     ;index to secondary table replaces the placeholder:
#MP     lt_%utemp = curr_base - min_r_%uq
#MP     Setstr lt_%utemp = {uJoin, "base index for q=", {%uq} }
#MP
#MP     Expand Maximum(curr_base+max_r_%uq-min_r_%uq, curr_end)
#MP     Undef min_r_%uq
#MP     Undef max_r_%uq
#MP Endm
```

First, we define, in a loop over remainders corresponding to defined object indices, `lt_<x>` as indices to the table of objects. We also define corresponding comment text; consult the manual for how the string expressions work. For cleanup purpose, we undefine the object index; it will no longer participate in detecting a collision in `DetectCollision`.

Outside the loop, we repair the tentative definition of the sub-table index to point to the (untrimmed) base of the currently merged sub-table. We also define the comment text we like.

The remaining things done there:

- Update the `curr_end` (in case the currently merged sub-table increased the total size of the combined table)
- Undefine the valid range of the indices of the sub-table. It is not only a cleanup operation; most importantly, we'll not try to merge this table again and it will not participate in detecting collisions.

Now we are in a position to put together the macro `FillLookupTable`:

```
#MP Macro FillLookupTable ;() - works with predefined names
#MP     Expand ReservePrimaryPlaces()
#MP
#MP     For merge_pass = 0, 0     ;(1)
#MP         merged_now = 0 ;flag  ;(2)
#MP         For q = min_q, max_q  ;(3)
#MP             Ifdef min_r_%uq   ;(4) q-table is not merged yet
#MP                 Expand DetectCollision()
#MP                 If collision == 0 ;found mergeable sub-table
#MP                     Expand MergeTable()
#MP                     not_merged = not_merged-1 ;(5)
#MP                     merged_now = 1 ;set the flag
#MP                     q = max_q+1 ;break the loop (for speed)
#MP                 Endif
```

```
#MP             Endif
#MP         Endfor
#MP
#MP         If not_merged > 0   ;(6)
#MP             merge_pass = -1 ;force the loop continue
#MP             ; If nothing merged, increment curr_base
#MP             If !merged_now  ;(7)
#MP                 curr_base = curr_base + 1
#MP             Endif
#MP         Endif
#MP
#MP     Endfor ;merge_pass loop
#MP
#MP     Expand RenderLookupTable()
#MP Endm
```

After preliminarily defining the `lt_<x>` parameters in `ReservePrimaryPlaces`, we start the loop of iterations in line (1). The end-of-loop test is on line (6): if the number of sub-tables remaining to merge is non-zero, we force the loop counter to -1. At `Endfor` it will be incremented to 0 and the loop will continue. If we failed to merge a sub-table in this merge pass (line (7)), we increment `curr_base` to try the next base.

Within the body of the loop, we first clear the indicator of having merged a table (line (2)), and start a loop in line (3) which attempts to find and merge a sub-table. If `min_r_%uq` is defined for the quotient q, it means that it was defined (in `DefineEntry`) and was not undefined (in `MergeTable`), that is, the sub-table for the quotient q exists and is not merged yet. For such a table, we check if placing it at would cause a collision; and if not, we

- Merge the table by using `MergeTable`
- Decrement the remaining number of sub-tables to merge (line (5))
- Set the `merged_now` flag in line 5 not to increment `curr_base` (because another sub-table may fit at the same base)

To test this implementation, we can run

```
Unimal sparse8.u
```

to see that our results are exactly what we were targeting.

Discussion

The macro code presented in this Note grew increasingly complex to match the more ambitious goals we set. Implementation described in the previous section is far from being *very* simple. Whether or not you had enough patience to follow it, there are a few points to keep in mind:

1. Based on original definitions of objects of any nature, the macros automatically generate a highly compressed lookup table and a guaranteed two-step access (search) method to the objects.
2. Since the compressed lookup table and the access method are generated automatically, they require zero maintenance effort, no matter how the original table of objects changes. The objects can be entered in the table of objects in any order.
3. All configuration work is done at compile time, saving runtime resources.

And finally, this Note illustrates how Unimal scales up with the complexity of the task at hand.