Managing constant data tables in ROMable applications

Class #347 Ark. Khasin (Ark.Khasin@macroexpressions.com)

Introduction

Managing and maintaining software projects is a difficult job by itself but it becomes much more difficult when you have a project with multiple configurations. Consistent reuse of tried-and-true code becomes a necessity, if for no other reason then because of the limited resources that can be dedicated to development and, most importantly, to debugging of new code.

Code reuse, however, seldom comes for free. This may have an adverse effect on the other end of the embedded project: the always-scarce RAM, ROM and execution time. So much the more interesting is to explore cases where well maintainable code can be produced with no runtime penalty whatsoever. In this presentation, we focus on an important special case where this happens to be possible: constant data tables.

Namely, we turn our attention to the tables (arrays) of data that are:

- constant within a given build of the project but
- changing during development cycle
- changing among project's twin variants
- changing from year to year in a product line environment

Here are a few examples of such "varying constant" tables:

- tabulated functions (common thing among ROMable systems)
- recognized communications datagrams (a.k.a. "message lists")
- lookup tables of all sorts
- custom keypad input translation tables

Applications where such tables can appear are many. Almost any embedded application that is mass-produced and/or belongs to a "product line" can serve as an example: consumer electronics, automotive controllers, home appliances, medical instruments, to name a few.

The plan of this presentation is as follows.

We will define requirements for constant tables' reusability and maintainability and then consider an example of a simple function that needs to be tabulated. On this example, we will see what minimum features the programming language must have to meet the requirements we will have adopted. We will conclude with dismay that high-level languages do not serve the purpose, and will resort to a generic macroassembler language in a machine-independent fashion, to which we will try to give a common ground description. Then we will demonstrate a solution to the problem of tabulating a function and discuss generalizations of the problem. Our next example will be sparse tables. We will set the maintainability goals and show the techniques, which make those goals achievable. The next point of interest will be a lookup table for a constant table of data objects. The main maintainability goal here will be generating the lookup tables automatically at compile time. Achieving this bold goal will produce code requiring no maintenance at all; this is the main result of the presentation. Then we will turn our attention to ROM-saving techniques for automatically generated sparse lookup tables. The full solution is too tedious to be presented here but we will present a sketch of the implementation. Finally, we will go back to high-level languages (and weaker assemblers alike) and discuss an option of using Unified macro language (Unimal), which provides language-independent macro extensions sufficient for solving the problems we will have discussed.

Reuse and maintainability requirements

High cost of a software bug and time-to-market considerations dictate emphasis on code reuse and maintainability. In our setting of multiple similar projects, this means, in the first place, that the code must be easy-to-configure by a project engineer who is not necessarily an expert in inner workings of any particular component of the code.

Parameterized projects

An almost obvious solution is to treat an individual project as a member of a parameterized family of projects, whether actual or envisioned. In this paradigm, a real project (the one generating executable code) is an instance of an abstract project (equipped with varying parameters) for a fixed set of the parameters.

Whether the parameters are "discrete switches" for conditional compilation or "continuous" calibration parameters, this approach has the following benefits:

- Doing all the development for an abstract project promotes code reuse
- A concrete project is instantiated by "simply" fixing the parameter set.
- New and unforeseen variations are added incrementally thus making the whole thing manageable.

This is just fine. The problem is, however, that instantiation can be far from trivial, even if there are only scalar parameters. The example below illustrates the problem. (In fact, this sort of challenges initiated my interest in constant tables.)

Example: Tabulating a function

Consider a function that is rather hard to calculate in real time. For our example, let's take

$$\texttt{myfunc(x) = 10000*x*/(1+x}^2\texttt{), 0}\leq\texttt{x}\leq\texttt{1,}$$

with integer precision. A common way of coping with this situation is to tabulate such a function.

Assume that the only parameter varying among projects is the number N+1 of equidistant points of interpolation. I.e., we want to create an array **`Myfunc`** of N+1 elements, whose t-th element is

$$\texttt{10000*(t*N/(t*t+N*N)), t=0,…,N.}$$

The **`Myfunc`** table consists of constant elements once N is fixed. This **Myfunc** array must be created automatically, otherwise it is very difficult and error-prone to maintain. The choices are:

- Calculate the array values at runtime, during system initialization
- Calculate the array values at compile time.

Comparing runtime vs. compile time (static) initialization is rather straightforward:

- Runtime initialization requires to link in some math support, thus increasing the code size (=ROM)
- Runtime initialization slows down system initialization
- With runtime initialization the table ends up being in RAM even though it is constant and logically belongs to less scarce ROM.

Thus, it is not hard to arrive at the following conclusion: runtime initialization is uniformly worse than compile-time initialization, so the goal should be static initialization of the table.

The problem with achieving this goal is the lack of appropriate programming tools. We will see shortly that high-level languages are not suited for this static initialization task. Macroassemblers are not designed with this task in mind either. However they possess necessary language features, which in an unusual combination can produce the solution we are seeking.

Limitations of high-level languages

The goal in our unsophisticated example is therefore to find a compile-time equivalent of the functionality of the C statement

```
for (t=0; t<=N; t++) Myfunc[t] = 10000*(t*N/(t*t+N*N));
```

To achieve this, static initialization of **`Myfunc`** table requires a source code sprinkled with compiler directives controlling the compiler in some special ways. Namely,

- We need some kind of a repetition mechanism, or loop, to force the compiler to re-scan a piece of source code repeatedly
- We need a way to arrange an incrementing compile-time counter (t) to calculate the values of the table elements.

The availability of these facilities depends solely on the programming language used. I am not aware of any HLL with *any* of these facilities, let alone all of them. More complex problems may require additional features missing in high-level languages, such as calculated names.

With the assumption that my perception is correct and that the HLLs indeed do not serve the cause of static (compile-time) initialization, we are left with two choices:
1. Use an add-on utility as a helper in static initialization or
2. Write static initialization in a macroassembler.

We will briefly discuss the first option in the end of this presentation; the center of our interest will be the second option.

Hypothetical Macro Assembler

Let's list the language features commonly available in various macro assemblers but for inexplicable reasons absent from high-level languages.

Assemblers are, of course, machine-dependent, but their macro facilities don't have to be. Still, there are syntactical differences among assemblers from different vendors even for the same machine. The truth appears to be that a macro language built in an Assembler is for the most part a thing independent of the target processor. Thus, the macro language is essentially a high-level language not usually recognized as such. The main difficulty with macroassemblers is lack of any standards of macro facilities.

To concentrate on the general techniques rather than peculiarities, we will use a hypothetical macro assembler (Hypoassembler, for short) whose macro (and related) syntax is described below.

Macro Definitions and Invocations

Macro is a language construct that allows defining a parameterized piece of source code (macro definition) and inserting this piece of code with parameters resolved to their actual values anywhere in the source *code (macro invocation*). The process and the result of substitution of a macro invocation with the appropriately resolved macro definition are often called *macro expansion*.

Most people would say that the difference between a macro and a function (or subroutine) is that a function call passes actual parameters and control to a separate piece of code, whereas macro invocation produces the necessary code on the spot by cloning its definition. While it is true, there is more to it, while obvious, often underestimated: Macro expansions are produced at compile (read: assembly) time, and function calls are made in run time. Therefore, functions are completely useless in defining constant (ROMable) items, because they (constants) must be resolved at compile time. Macros, on the other hand, can be used not only for merely defining constants, but also for giving those definitions that glossy look that is commonly expected from high-level languages.

Macro definitions in Hypoassembler have commonly accepted syntax

```
<macro_name> MACRO <comma_separated_list_of_formal_parameters>
      <macro body>
      ENDM
```

Macro is invoked by its name with comma-separated list of actual parameters. For our purposes, an actual parameter can be an arithmetic expression (calculated for us by assembler) or an alphanumeric string. Hypoassembler translates macro call by replacing it with the macro body with formal parameters replaced by actual parameters.

Name concatenation and early evaluation

Symbolic names (identifiers) inside the macro body can be a concatenation of different parts. The ampersand '&' serves as concatenation operator. For instance, if macro body contains an identifier `x&arg1&arg2` in it, and if actual parameter `arg1` is `123` and `arg2` is `abc`, then the name in the macro expansion will be `x123abc`. However, if `arg1` were `120+3`, then the name's expansion would be `x120+3abc`, which is syntactically incorrect and is not what's normally intended.

To handle this problem, there is a less common feature, so-called "early evaluation"; we will note each case of its use. If an actual parameter is an expression, it can be prefixed by a '%' and then the Hypoassembler evaluates it and passes a numeric (say, decimal) result to the macro instead. In the previous example, if `arg1` were `%120+3` then our example name would correctly expand to `x123abc`.

Conditional Assembly

Syntax:

```
IF <expression>
<body>
ENDIF
```

Hypoassembler evaluates constant `<expression>`; text between `IF` and `ENDIF` lines is discarded or literally included if the result is zero or non-zero, respectively. This construct can be used within macro definitions. Such macro definitions produce different macro expansions depending on actual parameters and / or place of invocation.

Repeated scanning of the source code

We can say that IF repeats the <body> zero or one time. The following construct is a generalization:

```
REPT <expression>
<body>
ENDR
```

Hypoassembler evaluates `<expression>` and includes as many copies of `<body>` in the source file. The net effect is that `<body>` is scanned the `<expression>` number of times. In fact, assembler programmers routinely use this facility, for instance, to include multiple-line comments: all ASCII text between `REPT 0` and `ENDR` is ignored!

The `REPT` construct is extremely powerful when combined with conditional assembly (whether folded in macros or not). The key is that combining assembler directives with conditional assembly in a `REPT` loop, we can write an Assembler source file that is at the same time *a sophisticated program to control the behavior of the Assembler*!

Assignments

The following syntax allows assigning a value of an expression to a symbolic variable:

```
<name>      SET <expression>
```

A name assigned with `SET` can be re-assigned (with another `SET`).

Allocating memory for constant data

For simplicity, we will assume only one data type good for holding integers and addresses; the directive to allocate it at the current program counter is `DC`. Syntax:

```
DC <expression>
```

Current value of the program counter (a.k.a. location counter) is available to the programmer as '**$**'. We can change the position of the program counter at will using the `ORG` directive. Syntax:

```
ORG <expression>
```

It is the humble `ORG` that is the heart of the solution. For instance, if I need to place something at offset 17 from the current location, I don't need any placeholders, counters or such. I simply command "`ORG $+17`"! Please, email me about any high-level language capable of this!

Maintainable implementation of a tabulated function

First implementation

Returning to the example of **Myfunc** in Hypoassembler, recall that our task was to find a compile-time equivalent of the C statement

```
for (t=0; t<=N; t++) Myfunc[t] = 10000*(t*N/(t*t+N*N));
```

Here it is:

```
Myfunc:
t     SET   0   ;loop initialization
      REPT  N+1 ;loop engine; remember, N is constant
      DC    10000*N*t/(N*N+t*t) ;allocating a tabulated value
t     SET   t+1 ;increment loop counter
      ENDR      ;end of loop body
```

This can be instantiated for any given N. For instance, if N=6, here is a (slightly doctored) output of an actual Assembler:

```
       HEX
000000           Myfunc:
000000 0000          DC     10000*N*t/(N*N+t*t) ;allocating a tabulated value
  00000001       t   SET     t+1 ;increment loop counter
000001 0655          DC     10000*N*t/(N*N+t*t) ;allocating a tabulated value
  00000002       t   SET     t+1 ;increment loop counter
000002 0BB8          DC     10000*N*t/(N*N+t*t) ;allocating a tabulated value
  00000003       t   SET     t+1 ;increment loop counter
000003 0FA0          DC     10000*N*t/(N*N+t*t) ;allocating a tabulated value
  00000004       t   SET     t+1 ;increment loop counter
000004 1207          DC     10000*N*t/(N*N+t*t) ;allocating a tabulated value
  00000005       t   SET     t+1 ;increment loop counter
000005 1336          DC     10000*N*t/(N*N+t*t) ;allocating a tabulated value
  00000006       t   SET     t+1 ;increment loop counter
000006 1388          DC     10000*N*t/(N*N+t*t) ;allocating a tabulated value
  00000007       t   SET     t+1 ;increment loop counter
```

Improving maintainability

First, let's wrap the implementation of the function to be tabulated in a macro:

```
myfunc MACRO t, Size
      DC    10000*Size*t/(Size*Size+t*t)
      ENDM
```

Second, let's wrap the generator of the table in a macro

```
FuncTable MACRO Size, Func
   _&Func&_tab:
   t     SET   0
         REPT  Size+1 ;remember, Size is constant
         &Func& t, Size
   t     SET   t+1
         ENDR
         ENDM
```

Now, we have a reusable component. E.g., the original myfunc is generated as the table `_myfunc_tab` by

```
FuncTable N, myfunc
```

Discussion

- The solution doesn't have terrifying look of assembly code
- Maintenance is straightforward on two levels:
  - The person maintaining the project simply specifies the **N**, and
  - The person maintaining the algorithm component using **myfunc** function modifies the **myfunc** macro as needed.
- These two persons may or may not be one.

Maintainability of sparse tables

To demonstrate the problem and a solution to it, it is best to follow a small-size example. For the example, let's take a sparse table of 32 entries, where significant entries are references to A1…A4:

A1 at offset 9
A2 at offset 11
A3 at offset 24
A4 at offset 27

All other entries are "don't care."

Using NULL or 0 for "don't care" entries, we can write in C something like this:

```
const ob_type * const Table[] =
{NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,&A1,NULL,
&A2,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL,&A3,NULL,NULL,&A4,NULL,NULL,NULL,NULL};
```

In straight Hypoassembler, it looks equally bad:

```
Table:
  DC 0,0,0,0,0,0,0,0,0
  DC A1,0,A2,0,0,0,0,0,0
  DC 0,0,0,0,0,0,A3,0,0,A4,0,0,0,0
```

Unreadable, not maintainable and error-prone! It doesn't matter how much nice formatting and how many comments you add – any modification is treacherous.

Inspecting the horrible code above and considering the maintenance tasks that may be needed for such a table, we can formulate the following maintainability goals:

1. Supply only significant entries of the table. Don't care about "don't care" entries.
2. Allow listing significant entries in arbitrary order.

This would reduce maintenance to editing (adding, removing) significant entries.

The additional goal is to save ROM by chopping off leading and trailing "don't care" entries of the table. This must be done without compromising the maintainability of the resulting implementation.

First implementation in Hypoassembler

The following code meets the first two goals in a crude way:

```
Table:
      ORG Table+11
       DC A2
      ORG Table+9
       DC A1
      ORG Table+27
       DC A4
      ORG Table+24
       DC A3
      ORG Table+32 ;position the Location Counter past the table
```

This implementation lists a significant entry as a pair of lines, one positioning the location and another defining the entry's value. The key here is the ORG directive controlling the location counter. Notice that we need to position the location counter past the actual table.

Cosmetic improvement

For a nicer look, let's use the following wrapper macros.
The first macro locates a `_value` at `_offset` from the current default table:

```
LocateElement MACRO _offset, _value
      ORG __DefaultTableName + _offset
      DC  _value
      ENDM
```

Our second macro is a companion to the first one: it defines the default table and locates the beginning of the table:

```
StartLocateTable MACRO _table
_table:
__DefaultTableName SET _table
                ENDM
```

Using these simple macros, we can rewrite our implementation of the table as follows:

```
       StartLocateTable Table
       LocateElement 11, A2
       LocateElement  9, A1
       LocateElement 27, A4
       LocateElement 24, A3
       ORG Table+32
```

This last implementation not only looks much better, it is also almost portable. Still, we need to position the location counter past the table. Also, we did not gain any ROM savings yet.

Second implementation

Our second implementation builds upon the first one. To save ROM, we offset the start of the table to overlap with preceding code or constant data thus taking advantage of leading "don't" care entries. In a similar way, we position the location counter past the last *significant* entry of the table, so the trailing "don't care" entries vanish overlapping the subsequent code or data.

```
Table: ORG $-9
ORG Table+11
       DC A2
      ORG Table+9
       DC A1
      ORG Table+27
       DC A4
      ORG Table+24
       DC A3
      ORG Table+27+1 ;position the LC past the last
                     ; significant entry
```

Reformulating, here is what we did:

1. Overlapped the 9 leading "don't care" entries with preceding code or data. Notice that 9 is the minimum offset of a significant entry of the table.
2. Overlapped the 4 trailing "don't care" entries with subsequent code or data. Notice that 27 (=(32−1)−4) is the maximum offset of a significant entry of the table.

Our task is now to make the necessary calculations automatically.

Automating the second implementation

So, we need to know in advance:

1. The number of leading "don't care" entries (= min offset of a significant entry)
2. The position of the location counter past the last significant entry (1 greater than max offset of a significant entry)

Therefore, we are targeting two-pass implementation:

Pass 1. Calculate min and max offsets of significant entries
Pass 2. Locate the table.

The following is our target implementation:

```
      PrepareLocateTable
      REPT 2
      StartLocateTable Table
      LocateElement 11, A2
      LocateElement  9, A1
      LocateElement 27, A4
      LocateElement 24, A3
      ENDR
      EndLocateTable
```

Notes:

1. REPT and ENDR statements cannot be portably folded in macros, so they remain exposed.
2. The project engineer can treat new elements of the table definition (shown in larger font) as fixed incantations.
3. The target implementation meets the goals we set: maintainability and ROM savings.

We begin our implementation with simple macros serving as building blocks. The first one is a helper macro calculating current minimum. The actual argument for `Current` must be pre-initialized to a large number.

```
Minimum MACRO Current, New
 IF New<Current
Current     SET New
 ENDIF
        ENDM
```

Another necessary macro, `Maximum`, is similar and is therefore omitted here. The next two macros simply encapsulate the initialization of the parameters we use (**`PrepareLocateTable`**) and the positioning of the location counter (**`EndLocateTable`**).

```
PrepareLocateTable MACRO
__Pass      SET 0
MinOffset   SET 10000 ;very large number
MaxOffset   SET 0
                  ENDM
```

```
 EndLocateTable    MACRO
      ORG __DefaultTableName+MaxOffset+1
                  ENDM
```

Now, we are ready to redefine the macros **`StartLocateTable`** and **`LocateElement`** for our two-pass strategy. Newly added code is shown in the larger font.

```
StartLocateTable MACRO _table
__Pass    SET  __Pass+1
 IF __Pass=2
          ORG  $-MinOffset
_table:
__DefaultTableName SET _table
 ENDIF
          ENDM
```

```
LocateElement MACRO _offset, _value
 IF __Pass=1
     Minimum MinOffset, _offset
     Maximum MaxOffset, _offset
 ENDIF
 IF __Pass=2
     ORG __DefaultTableName + _offset
     DC  _value
 ENDIF
          ENDM
```

This set of macros completes the implementation of the code we targeted in the beginning of this subsection.

Generating lookup tables automatically

Let's consider now a table of objects of more or less any nature. Assume that to perform a search on such a table by a numeric key, we want to supplement it with the corresponding lookup table. The lookup table maps a valid key to the reference to an object; entries at positions not being valid keys are "don't care."

An immediate observation one can make is that if the table of objects is constant, so is its lookup table, which is therefore a perfect candidate to go to ROM. Thus, we want to generate it at compile time. Additionally, the lookup table is not an independent entity: it can be calculated from the table of objects and so its generation can be automated. Hence our goal: To generate lookup tables for tables of any objects completely automatically, to require no maintenance at all.

Framework

We assume that an object is defined by a macro that looks like this:

`DefineOb` key, *<arguments>*

where `key` is a unique numeric identifier of the object, and other arguments are whatever the application calls for: parameters, indices, function pointers and so on. (There can be a generalization of this format, where an object's key is not a part of the definition, but rather can be calculated at assembly time based on object's content, i.e., *<arguments>*. Let's just note that it can be done and go on with our simpler representation.)

For the example demonstrating the techniques, let's take the following table:

```
ObTable:
A2:    DefineOb 11, <arguments1>
A1:    DefineOb  9, <arguments3>
A4:    DefineOb 27, <arguments2>
A3:    DefineOb 24, <arguments4>
```

Labels A1…A4 are not needed other than for our reference.

Observe that the sparse table from our previous example is also the lookup table for the table of objects **ObTable**. But now it contains only data that can be calculated from the **ObTable**, so it potentially can be hidden from the application programmer. When the **ObTable** changes, so does the lookup table, but it will be transparent to the user!

An approach to implementation

One can observe that in the **ObTable**, the invocations of **DefineOb** macro are in one-to-one correspondence with the invocations of the **LocateElement** macros of our implementation of the standalone sparse table. The obvious reason for this is that significant elements of the lookup table are in one-to-one correspondence with the objects they reference. So, our natural plan is to reuse the general appearance of the sparse (lookup) table allocation.

To do so, we plan to create an "extended" **DefineOb** macro, **ExtDefineOb,** to combine macros **DefineOb** and **LocateElement**, i.e., to define both an object and an entry of the lookup table.

Below is our target implementation:

```
ObTable:
      PrepareLocateTable
      REPT 2
      StartLocateTable LookupForObTable

      ExtDefineOb 11, <arguments1>
      ExtDefineOb 27, <arguments2>
      ExtDefineOb  9, <arguments3>
      ExtDefineOb 24, <arguments4>

      ENDR
      EndLocateTable
```

To generate lookup table (named here **LookupForObTable**), we may use the same prefix and suffix code (underlined) as before, for the stand-alone sparse table.

Implementation: Method 1

ExtDefineOb must allocate the object and make its address known. This, of course, needs to be done only once, and the first pass is a natural place to do it. Then, the already defined addresses of objects are used in **LocateElement** invoked in both passes. To uniquely name an object's starting point, we can use its key concatenated with a standard base name. Now we are in a position to define our **ExtDefineOb** macro:

```
ExtDefineOb MACRO key, <arguments>
      IF __Pass=1
ObPosition_&key:
           DefineOb key, <arguments>
      ENDIF
           LocateElement key,
ObPosition_&key
            ENDM
```

(The labels of the objects are **ObPosition_9**, etc.)
Thus, in Pass 1, **DefineOb** allocates objects and **LocateElement** calculates **MinOffset** and **MaxOffset**. In Pass 2, **LocateElement** generates the lookup table.

Test drive

Let's compare the manually created lookup table against the automatically generated one. The original table of objects and the corresponding lookup table created manually are shown on the left and the automatically generated lookup table, on the right.

```
ObTable:
A2:    DefineOb 11, <arguments1>
A1:    DefineOb  9, <arguments3>
A4:    DefineOb 27, <arguments2>
A3:    DefineOb 24, <arguments4>

LookupForObTable:
  DC 0,0,0,0,0,0,0,0,0
  DC A1,0,A2,0,0,0,0,0,0
  DC 0,0,0,0,0,0,A3,0,0,A4,0,0,0,0
```

```
ObTable:
ObPosition_11:  DefineOb 11, <arguments1>
ObPosition_9:   DefineOb  9, <arguments3>
ObPosition_27:  DefineOb 27, <arguments2>
ObPosition_24:  DefineOb 24, <arguments4>

                  ORG $-9
LookupForObTable:
            ORG LookupForObTable+11
            DC ObPosition_11
            ORG LookupForObTable+9
            DC ObPosition_9
            ORG LookupForObTable+27
            DC ObPosition_27
            ORG LookupForObTable+24
            DC ObPosition_24
            ORG LookupForObTable+27+1
```

Implementation: Method 2

Let's consider a variation of the method just described; then we will compare the two.

Instead of labeling every object individually, we will reference an object by the ordinal number of its appearance in the table of objects. To do so, we enter lookup table elements as before. If **`ItemNum`** counts the current number of objects, starting with 0 on each of the two passes then the following macro is a solution.

```
ExtDefineOb MACRO key, <arguments>
   IF __Pass=1
     DefineOb key, <arguments>
   ENDIF
     LocateElement key, ItemNum
ItemNum   SET ItemNum + 1 ;count current number
          ENDM
```

Of course, for proper counting, we need to add the line

```
      ItemNum SET 0
```

to the macro **`StartLocateTable`**.

In this version, lookup table entries reference actions by index in `ObTable` rather than by pointer.

Comparison of method 1 and method 2

Method 1 requires a fairly advanced macro Assembler: It must support string catenation and perhaps (depending on application) early evaluation of arguments. None of this is required by method 2, so if it solves the problem, it should generally be the choice.

There is a case, though, where method 2 doesn't work, and method 1 does. This is a pretty odd case of objects of variable length (e.g., objects containing a text string). Indeed, the assumption behind method 2 is that an object can be referenced by its number, and it implies constant object length. Since method 1 references an object by its address, it is free from this limitation.

Summary of our achievements

1. We do not enter the lookup table manually; the Assembler builds it for us automatically.
2. Automatically built lookup table is more ROM-efficient than the one entered manually.
3. Item 1 not only saves us typing; it saves much more in code maintenance.
4. The macros we came up with depend very little on the example at hand; they can be reused almost 1:1 in different circumstances.

5. These macros do not depend on the target machine instruction set, so portability issues have to do only with differences in macro languages among assemblers.

ROM saving improvements – splitting the key

There is a way to save significant amount of ROM by replacing single-step search using lookup table by a two-step search. The idea is to split the search key into two parts of about equal length, called primary and secondary keys respectively. E.g., for our example,

9 = 01001B à (010B, 01B) = (2,1)
11=01011B à (010B, 11B) = (2,3)
24=11000B à (110B, 00B) = (6,0)
27=11011B à (110B, 11B) = (6,3)

Then lookup table search is done on the first (primary) part of the key using the (exponentially) smaller lookup table, which we will call *primary* lookup table. The latter references (exponentially) smaller secondary lookup tables, one per one *existing* primary key. The secondary tables reference the original objects. In our example, the primary table is

TableP = {0, 0, Table2, 0, 0, 0, Table6, 0}

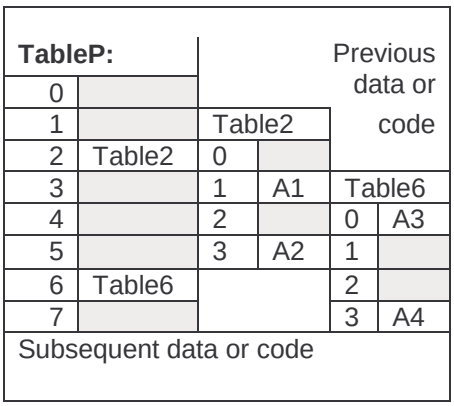
referencing the following secondary tables for primary keys 2 and 6:

Table2 = {0, A1, 0, A2}
Table6 = {A3, 0, 0, A4}.

Merging separate tables

Separate tables can be intertwined so that significant entries of one table fall in the "holes" of other tables (grayed in the illustration below). This can be achieved by moving tables' origins and allocating one table at a time when possible. The assembler must be good at memory/swap file management, so there are certain requirements to the host computer platform. Assembly may take noticeable time. Macroassembler implementation produces (without print controls) a tremendous listing file.

Target implementation is shown to the right. A guessed large repetition number is required to provide for iterative process of moving tables' origins. Other than that, the application programmer sees no differences in her table of objects.




```
ObTable:
        PrepareLocateTable
        REPT 100000 ;really large number
        StartLocateTable LookupForObTable

        ExtDefineOb 11, <arguments1>
        ExtDefineOb 27, <arguments2>
        ExtDefineOb  9, <arguments3>
        ExtDefineOb 24, <arguments4>

        ENDR
        EndLocateTable
```

A sketch of implementation

A possible implementation consists of a two-pass process followed by an iterative process.
The two-pass process allocates the table of objects and the primary lookup table ala simple single-level lookup table generation. In addition, min and max offsets of all secondary lookup tables are calculated.

Note 1: since the primary table contains references to the lookup tables that are not yet allocated, the assembler must support forward references, so single-pass assemblers will not do. (A different implementation is possible though.)

Note 2: care must be taken not to attempt to allocate the same entry of the primary table more than once (since they appear multiple times during a scan of the message table).

The iterative process:
1. shifts the next prospective allocation position of remaining secondary tables
2. for each secondary table checks whether it has collisions with already allocated tables
3. if a "good" secondary table is found, it is allocated at the current shift offset from the common origin and marked as allocated. If no more tables remain, end, otherwise, repeat the process.

Some caveats:
1. For the iterative process, the REPT statement must have sufficiently large number. Experimentation is required as well as error control (not all tables allocated after all assembler passes are over).
2. To avoid empty passes (after all tables are allocated) use EXITR or equivalent (whenever available) to exit the re-scanning. Otherwise it translates to increased assembly time.

An option for high-level languages and simpler assemblers

Tricks with macros shown here are a serendipitous by-product of macro facilities of good assemblers. The resulting code produces enormously long listing file with very spare occurrences of code-generating lines. Usually, thorough listing control statements are advised.

Consistent solutions for high-level languages are available with a tool called **Unimal** (for UNIfied MAcro Language). It handles the static initialization tasks independently of the target language. (Please, visit www.macroexpressions.com.) It allows to:
- Perform complex compile-time configuration
- Reduce maintenance complexity of your code
- Put in ROM what you had to configure in runtime before.
- Reduce memory requirements of your project

Additionally, it allows to:
- Do more sophisticated arithmetic on parameters at compile time
- Export and share parameters between different languages (e.g., between C, FORTRAN, Assembler and the make utility)

Conclusion

In this paper we discussed just one problem of static initialization of constant tables. We found a readable and maintainable ways of coding them. We were able to save some ROM along the way.

While doing so we found that high-level languages lack features we needed. We therefore resorted to an assembler, but used only portable features of macro languages.

And finally:
1. Our macros depend very little on the example at hand; they can be reused almost 1:1 in different circumstances.
2. These macros do not depend on the target machine instruction set, so portability issues have to do only with differences in macro languages among assemblers.