Managing constant data tables in ROMable apps

Scope of our interest for 1.5 hours

Data tables (arrays)

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

Examples of such tables:

- tabulated functions
- recognized communications datagrams
- various lookup tables

Typical applications:

- Consumer electronics
- Automotive controllers
- Home appliances
- Most mass-produced devices
- You name it!

Managing constant data tables in ROMable apps

Focus on reuse and maintainability

1. High cost of a software bug
2. Time-to-market considerations
3. Thus, emphasis on reuse of tried-and-true code
4. Code must be easy-to-configure by a project engineer

Variations across and within “model year”:

1. Impede software reuse
2. May require additional code development and therefore
3. Introduce new bugs

Solution:

Treat a project as a member of a parameterized family of projects, whether actual or envisioned.

In other words, a real software project (i.e., generating production code) is an instance of an abstract project (equipped with varying parameters) for a fixed set of parameters.

Managing constant data tables in ROMable apps

Parameterized projects

- 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 — on paper. Even if there are only scalar parameters, instantiation can be far from simple.

Example: Tabulating a function

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

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

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

$$10000*(t*N/(t*t+N*N)),\ t=0,\ldots,N.$$

Requirements

The Myfunc table consists of constant elements once N is fixed.

The **`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

Runtime vs. compile time (static) initialization

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

Conclusion: runtime initialization is uniformly worse than compile-time initialization.

The goal should be static initialization of the table.

Limitations of high-level languages

Our example's goal is therefore to reproduce the functionality of the C statement

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

at compile time.

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 don't know of any HLL with *any* of these facilities, let alone all of them.

Hypothetical Macroassembler (Hypoassembler)

Let's list some useful language features

- commonly available in various macro assemblers but
- for inexplicable reasons absent from high-level languages.

Assemblers

- are machine-dependent, but
- their macro facilities are not

Syntactical differences in macro languages are just vendors' quirks.

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

Macro Definitions and Invocations

Macro is a language construct that allows

- to define a parameterized piece of source code (*macro definition*) and
- to insert 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*.

Differences between a macro and a function (or subroutine):

- A function call passes actual parameters and control to a separate piece of code
- A macro invocation produces the necessary code on the spot by cloning its definition.
- 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.

Macro Definitions and Invocations (cont'd)

Macros can be used for:

- merely defining constants, but also
- 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.

Symbolic names

Symbolic names (identifiers) inside the macro body can be a concatenation of different parts; '&' 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.

Early evaluation of parameters

To handle this problem, there is a less common feature, so-called "early evaluation". 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

The following construct is a means of arranging a compile-time loop:

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.

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*!

Important addition:

EXITR – breaks the **REPT**/**ENDR** loop.

Assignments

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

<name> **SET** <expression>

Important: the <expression> can use the <name>, so it is possible to arrange a compile-time counter, e.g.,

S SET S+1

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>

Controlling the location counter (a.k.a. program counter)

The current value of the 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>

Tabulated function myfunc in Hypoassembler

```
Myfunc:
t   SET   0
    REPT  N+1 ;remember, N is constant
    DC 10000*N*t/(N*N+t*t)
t   SET   t+1
    ENDR
```

Managing constant data tables in ROMable apps

Making it reusable

1. Let's wrap in a macro the implementation of the function to be tabulated

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

2. 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
```

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

```
FuncTable N, myfunc
```

Discussion

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

Maintainability of sparse tables

Consider, for the example, 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 something like this:

C:

```
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};
```

Hypoassembler:

```
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!

Managing constant data tables in ROMable apps

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.
3. Reduce maintenance to editing (adding, removing) significant entries
4. Save ROM by chopping off leading and trailing "don't care" entries of the table.

First implementation in Hypoassembler

```
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 LC past the table
```

Goals 1 and 2 and 3 are met (sort of).

Managing constant data tables in ROMable apps

Cosmetic improvement:

For a nicer look, let's use the following macros:

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

Defines default table and locates the beginning of the table

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

Locates a _value at _offset from the current default table

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

This is much better.
Still,

- need to position the location counter past the table.
- Also, no ROM savings yet

Managing constant data tables in ROMable apps

Second implementation

```
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
```

What we did:

1. Overlapped the 9 leading “don’t care” entries with preceding code or data.
2. Overlapped the 4 trailing “don’t care” entries with subsequent code or data.

Automating the second implementation

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)

Targeting two-pass implementation:

Pass 1. Calculate min and max offsets of significant entries

Pass 2. Locate the table.

Managing constant data tables in ROMable apps

Target implementation:

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

Notes:

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

Managing constant data tables in ROMable apps

Building blocks

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

Helper macro calculating current minimum. **Current** must be pre-initialized to a large number.

Maximum is similar.

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

Loop initialization (new macro)

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

Positioning the location counter past the table (new macro)

Managing constant data tables in ROMable apps

Class #347

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

Redefined **StartLocateTable** for the two-pass strategy

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

Redefined **LocateElement** for the two-pass strategy

Generating the lookup table automatically

Goal: To generate lookup tables for tables of any objects completely automatically, to require no maintenance at all.
Framework: Objects are defined by existing macro **`DefineOb`** taking the key and "other arguments" as arguments.

Example:

```
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 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!

Approach to implementation

"Extended" DefineOb macro, ExtDefineOb, combines DefineOb and LocateElement macros.

```
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

Unique label is *calculated* for each object. The method requires name catenation.

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

The labels are ObPosition_9, etc.

In Pass 1, objects are allocated and MinOffset and MaxOffset are calculated.
In Pass 2, the lookup table is generated.

Implementation: Method 2

Instead of labeling each object individually, let's reference them by ordinal number of their occurrence, ItemNum.

We must add

```
ItemNum SET 0
```

to the macro **StartLocateTable**.

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

Lookup table is generated to contain indices (rather than pointers) to the objects.

Managing constant data tables in ROMable apps

Test drive

Let's compare the manually created lookup table against the automatically generated one.

```
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
```

The original table of objects and the corresponding lookup table, created manually

Managing constant data tables in ROMable apps

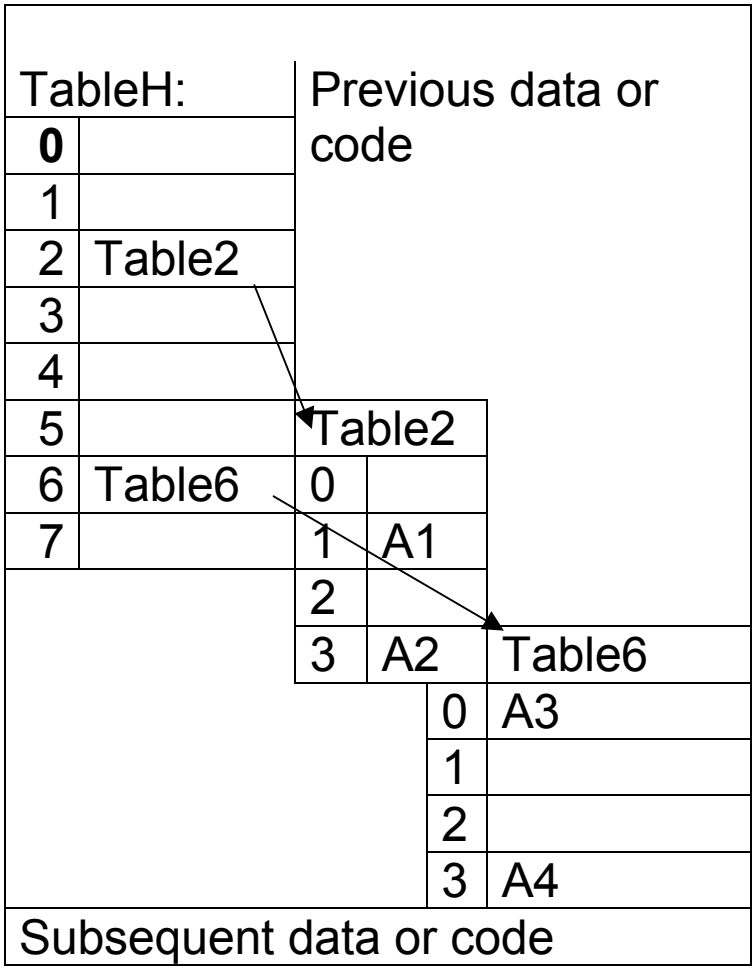

```
ObTable:
ObPosition_11:    DefineOb 11, <arguments1>
ObPosition_9:     DefineOb  9, <arguments3>
ObPosition_27:    DefineOb 27, <arguments2>
ObPosition_24:    DefineOb 24, <arguments4>

          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
```

Automatically generated lookup table using method 1

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.

Further improvements – splitting the key



Splitting the key into two parts

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

TableH = {0, 0, Table2, 0, 0, 0, Table6, 0}

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

Managing constant data tables in ROMable apps

Further improvements – merging separate tables

TableH:				Previous data or code	
0					
1		Table2			
2	Table2	0			
3		1	A1	Table6	
4		2		0	A3
5		3	A2	1	
6	Table6			2	
7				3	A4
Subsequent data or code					

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

Managing constant data tables in ROMable apps

Target implementation

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

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

   ENDR
   EndLocateTable
```

A guessed large repetition number is required to provide for iterative process. Other than that, the application programmer sees no differences in her table of objects.

Merging the tables – a sketch of implementation

A two-pass process followed by an iterative process.

- The two-pass process allocates the table of objects and the "hash" (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 hash 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 hash table more than once (since "digests" do repeat).

- 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 beginning and marked as allocated. If no more tables remain, end, otherwise, repeat the process.

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.

Managing constant data tables in ROMable apps

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)