Improving embedded software quality using an external preprocessor

Ark Khasin

MacroExpressions

White paper

MacroExpressions

http://www.macroexpressions.com

A smaller and faster code also gets to the target and starts faster. All this is crucial for SoC and code delivered over slow networks. Surprisingly big help comes from evaluating conceptually constant data at compile time. To do this, an external tool is required, as programming languages are not expressive enough. A good preprocessor utility may be the tool of choice, and provide additional perks.

A simple example shows how and why improving maintainability and optimizing constant data naturally leads to a preprocessor

Example: displaying status messages on the LCD

As a very simple yet realistic illustration, consider a task of displaying pre-defined up-to-twenty-character "status" messages on the LCD according to some status bit array. A message is displayed for, say, 5 seconds, provided that the corresponding bit is set, and then is replaced by the next (modulo the number of status bits) message with a status bit set.

A naïve implementation plan and its drawbacks

A "naïve" implementation say, in C, would probably define a const array of pointers to const strings. The ordinal number of the status bit would also be the index into the array of pointers, so the corresponding message string is can be accessed.

There are two problems with this idea.

The first is memory consumption. Assuming four-byte pointers, we've got five bytes of overhead per string (a pointer and the terminating null character). For a twenty-byte payload, it's 25%. If most messages are shorter than 20 characters, it's even more. E.g. if the average length of the messages is 10 characters, we have 50% overhead. That's not counting any data alignment overhead. Dummy pointers corresponding to undefined status bits have not been counted either.

The second problem is maintainability. If, for some reason, a bit corresponding to some status had its number changed from 15 to 4, the array of pointers has to be modified accordingly. If a previously undefined bit gets defined, or a previously defined bit is no longer, then again, the array of pointers needs to be updated.

Data optimization and growing maintainability problems

To address the first problem, we can choose a different data structure.

Let's have a (large) string comprising all message strings concatenated together (and even without the terminating null).

Let's further have an array of indices into this large string such that the *n*th element of the array is the index to the beginning of the *n*th message string. The last (extra) index in the array is the

length of our large string. The length of the message n to display is the difference between indices *n*+1 and *n*: no information is lost.

Typically, two bytes would be enough to hold an index. Since the large string has no terminating nulls, the overhead of this data structure is 10% (down from 25%) or, with average counting, 20% (down from 50%), plus a fixed two-byte expense on the last index.

Great. However, on the maintainability front things just got much worse: Maintaining the array of indices is very error-prone. Even if it was not, it would still be yet another thing to maintain, thank you very much.

Preprocessor to the rescue

A zero-maintenance solution to both the original and the newly-created maintainability problem is to define a status bit number and the corresponding text message in a single statement, like so:

```
BeginStatus
DefineStatus(3, “my fault”)
DefineStatus(8, “mea culpa”)
.........................
EndStatus
```

We want these statements to execute at compile time and produce the source code with the bit array definition and the constant data structure we invented previously.

To achieve this goal, we are willing to do an extra work (once!) and describe, to some conversion tool, how to execute those statements and produce the C source snippets that we want.

Guess what? We are talking about some preprocessor and about writing macros for it.

To reiterate: we naturally identified a need in a preprocessor in our effort to reduce (to zero if possible) error-prone maintenance work, especially in cases of optimized data structures.

A programming language may already have a built-in preprocessor of its own, as is the case with C and C++. If such a preprocessor exists and is expressive enough for the tasks, that’s wonderful. Otherwise, we’ve got to use an external preprocessor.

There are many project automation problems that a preprocessor can solve

Some tasks for the preprocessor to do

Here are some of the tasks where a good preprocessor can be of great help.

Tabulated functions

A hard-to-compute function can be tabulated for faster performance. Tabulating at compile time removes the table generating code from the final build. Additionally, the resulting table resides in ROM, which saves precious RAM and, in some applications, the need to test its integrity.

Preprocessed data

More generally, any data set may call for a processing algorithm that requires one-time preprocessing of the data set.

Some of the examples include lookup tables, perfect hashes, dictionary trees of all sorts etc.

When the data set is constant for the project, so is its associated preprocessed (derived) data. In this case, the derived data can be pre-computed at compile time. As with tabulated functions, the challenge is to find a tool capable of sufficiently complex compile-time processing.

Loop unrolling

A decision to unroll a time-critical loop should not be left to the compiler's heuristics: they have no knowledge of time criticality in your application. Unrolling a loop manually eliminates a runtime variable – the loop counter – but creates a maintainability challenge (and an implied constant parameter, the number of repetitions of the loop body).

Project configuration management

In a context of a project family, a good architecture for software project configuration management is project-independent code processing project definition *data*, the latter being of course constant for a given project.

The project-dependent data have to be shared across disparate languages (e.g., to a C source and to the linker command file)

If a general-purpose preprocessor can do the job, it is preferred to a dedicated code generator

Dedicated code generators vs. preprocessors

An extreme case of a preprocessor is a dedicated tool working for a specific data set. For instance, the macros in our example of status messages can take the form

```
3 “my fault”
8 “mea culpa”
.............
```

All the smarts of converting this to the C source we want are in the tool itself; the data definition has no trace of what needs to be done with it.

This approach is (or may be) better than none at all but is best avoided if a suitable preprocessor is available. The first reason for that is that a dedicated code generating tool (whether written in C++ or Perl or anything) requires maintenance of its own, or else the data design becomes unjustifiably rigid.

Secondly, there can be (and probably is, right in your project) more than one data definition of this kind, which is to produce an entirely different output, according to an entirely different data design. It would therefore require an entirely different code generator; this is very difficult to justify unless all data designs are extremely stable.

Thirdly, it is highly desirable that our macros can be plugged in an otherwise normal source file. This has to do with aesthetics not to underestimate: the source code sprinkled with preprocessor statements still preserves the look and feel of the target programming language. Even more importantly, it has to do with visibility (and linkage) of the generated output. Writing a code generator supporting this feature is no small feat.

Of course, a solution to all these problems is to split the code generator into two pieces: a conceptually simple yet flexible common language to describe how we process our definitions, and a common tool that recognizes and processes these description statements in a perhaps otherwise normal source file. This (of course) means a normal preprocessor.

A preprocessor should have certain usability and power properties

What to look for in a preprocessor

When choosing a preprocessor, you may want to consider the following criteria:

The language style of the preprocessor

If preprocessor statements are planted into the source file, do they really, really stand out (like C/C++ preprocessor and *unlike* m4)?

Can reusable constructs be wrapped in macros and tucked away in an include file, so that they can be invoked on as-needed basis?

Can the same preprocessor language be used for different target programming (and description) languages?

Error handling

If the preprocessing results in an error, is there a guarantee that the generated source will not compile?

Is there a place where all errors are conveniently collected, even

if multiple files are generated?

Flexibility and expressive power of the language

Does the preprocessor language meet your realistic needs? For instance, how easy is it or is it possible to tabulate a trig function? How easy is it or is it possible to create a lookup table automatically?

A basic criterion, is it possible to arrange a re-scan of (a compile-time loop over) a segment of the source code?

Another basic criterion, does the language provide sufficient arithmetic capabilities?

Integration into the development environment

How easy is it to include the preprocessor in your Integrated Development Environment, provided it supports inclusion of third-party tools?

Can the preprocessor search specified include directories?

Can the preprocessor output include file dependencies for make-driven build process?

Ability to output multiple files

In our example with status messages, the status bit array and the message table may have (depending on coding policy) to go to different files.

Data sharing across different target languages simply *requires* to output several files.

Can the preprocessor do it?

Unimal 2.0 is an advanced preprocessor meeting all the requirements

Unimal – a solution from MacroExpressions

MacroExpressions has developed a preprocessor to meet all the needs identified in this paper. To my best knowledge, it is, presently, the only preprocessor of this kind.

Unimal (http://www.macroexpressions.com/unimal.html) is an advanced preprocessor independent of the target programming languages. Industry-tested and with its claims proved, Unimal is currently at version 2.0.

Unimal features

- an ability to output more than one file, program-controlled, for sharing data among different source files of the target programming language(s)
- an ability to output dependencies on the included files, for build process integration
- 32-bit arithmetic and math functions, for easy

manipulation of constant data

- string operations common to preprocessing
- an ability to scan a segment of input repeatedly, for calculating values that cannot be evaluated in a single pass
- a well-developed macro facility, for encapsulating common preprocessing patterns
- a comprehensive error detection and reporting facility

Unimal syntax is very simple, due to clean separation of Unimal statements and the target language.

Unimal statements are line-based and begin with a signature #MP; all other lines are considered target language to be copied to the output.

The target language lines may contain a special markup called Unimal target language interface; this markup is replaced with the corresponding Unimal parameters.

Conclusion

Maintaining and managing optimized code across a family of projects requires serious attention to the data structures that are constant within a given project build. It is advantageous to use a preprocessor to pre-compute any derived data and to share data among different languages.

MacroExpressions offers a solution with Unimal, an advanced preprocessor which allows simple solutions for simple problems and makes complex solutions possible. It seamlessly integrates with established make-style build processes and it is easy to integrate Unimal into the Integrated Development Environments supporting third-party tools. For more on Unimal, visit http://www.macroexpressions.com/unimal.html.